École Polytechnique Fédérale de Lausanne

Swiss Post E-Voting

by Ella Kummer

Master Semester Project Report

Prof. Bryan Ford
Project Advisor

Louis-Henri Merino
Project Supervisor

EPFL IC IINFCOM DEIDS
BC 210 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 6, 2022

Contents

Chapter 1

Introduction

E-voting in Switzerland is currently put in place with a collaboration between the Confederation and the cantons, established by the Swiss E-Government. At the beginning of 2019, e-voting was provided in ten cantons out of twenty-six. In some cases it was offered to all resident voters, however in some cantons it was restricted to voters living abroad. The cantons had the possibility to select one of the two systems available. The two systems were the system offered by the canton of Geneva, and the system offered by Swiss Post. In June 2019, it was announced that the system offered by the canton of Geneva would no longer be offered. Not long after, on July 2019, it was Swiss Post's turn to announce that their system would not be available anymore either. As a result, no e-voting system is currently available in Switzerland[1]. However, Swiss Post is since developing a system with complete verifiability[2].

Swiss Post has long been a strong proponent of E-Voting. Currently, it is developing an E-Voting system to be utilized in Swiss elections. As part of their commitment to transparency, Swiss Post releases their source code as well as documentation and conduct intrusion tests. During an intrusion test in 2019, researchers discovered an implementation issue that would have allowed an attacker to change the outcome of an election[3]. Since then, Swiss Post has rectified the issue, among others, and made available to the public the source code, specifications and additional documents. They launched in parallel a bug bounty program to continuously improve the security of the Swiss Post e-voting system.

The goal of my project was to take part in this bug bounty program and review the documentation as well as the source code. This covers implementation issues or vulnerabilities as well as possible attacks under different scenarios and threats models. These researches are explained in more detail in the Methodology chapter 4.

This report first retraces the historical background of the Swiss post e-voting system in the

[1] https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html
[2] https://evoting-community.post.ch/en/about-e-voting
[3] https://www.post.ch/en/about-us/news/2019/swiss-post-temporarily-suspends-its-e-voting-system

background section 2 and states where it is now. After this, we give a description of the system and its properties in chapter 3 to be able to later explain the analysis provided in chapter 5.

Chapter 2

Background

This section gives an overview of the Swiss post e-voting system's history and what the current state of the system is.

2.1 Timeline

At the beginning of 2019, e-voting was, under some restrictions, offered in ten cantons of Switzerland with the possibility of using the system offered by Swiss Post[1].

Swiss Post conducted a public intrusion test on its new, universally verifiable e-voting system between 25 February and 24 March 2019. In addition to the intrusion test, it published the source code for its e-voting system on 7 February. Feedback on the published source code revealed critical errors such as an issue that would have allowed an attacker to change the outcome of an election. Following this, Swiss Post announced that their individually verifiable system would no longer be offered to the cantons[2] and the Federal Chancellery mandated a group of experts to work on a source code analysis of the e-voting system . As a result on July 2019 they released a paper called "Analysis of the Cryptographic Implementation of the Swiss Post Voting Protocol"[10].

Since 2020, Swiss post has been developing its new system with complete verifiability at its IT site in Neuchâtel, providing an e-voting system from Switzerland and for Switzerland.

At the start of 2021, Swiss Post began to publish the components and relevant documentation from its e-voting system as part of a community program. Following the publication of the verification software, all significant parts of the beta version of the Swiss Post e-voting system are now available.

[1] https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html
[2] https://www.post.ch/en/about-us/news/2019/swiss-post-temporarily-suspends-its-e-voting-system

Swiss post plans to make this system available to the cantons for the trial operation once the relevant legal framework conditions are in place and the development and evaluation of the system has been completed.

2.2 Current findings

Confirmed findings are categorized according to their severity. On 23 December 2021, 117 confirmed findings were reported, of which 3 were qualified as "high severity". An overview of all reports can be found on the GitLab[3].

[3] https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/issues

Chapter 3

Description of the System

In order to understand the analysis, an overview of the swiss post e-voting system is provided in this section.

3.1 Security Objective

Swiss Post e-voting system must comply with the Federal Chancellery's Ordinance on Electronic Voting[1]. Essentially, it must display three properties: individual verifiability, universal verifiability, and vote secrecy[15].

3.1.1 Individual verifiability

Individual verifiability allows the voter to check that the vote was correctly transmitted and registered by the server.

The voter compares a verification code that they receive with their voting documents to the verification code displayed online when they go to the ballot box. Faithfully executing the two-round return code scheme guarantees to the voter that the system's trustworthy part registered the intended vote. This implies that even a powerful adversary, controlling the voting client and most of the server infrastructure, cannot alter or drop a vote without being detected by a diligent voter or auditor.

[1] https://www.post.ch/en/business-solutions/e-voting/legal-basis

3.1.2 Universal verifiability

Universal verifiability allows voters or auditors to check that the election outcome corresponds to the registered votes.

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

3.1.3 Vote secrecy

Vote secrecy preserves the privacy of the voter and does not reveal a voter's vote to anyone. By extension, vote secrecy ensures that no component learns the election results before the final decryption step. The Swiss Post Voting System protects vote secrecy by encrypting votes end-to-end and splitting the decryption key among multiple entities.

3.2 Parties of the system

The Swiss Post Voting System includes multiple components and involves different parties. We divide them in two categories: the actors and the building blocks.

3.2.1 Actors

We consider as actors parties of the system that are not part of the implementation[14].

Voter

The Voter authenticates to the voting server, selects voting options, and confirms their vote by verifying return codes. We assume that the voter can collude with the adversary. For example, this can be done by revealing their codes or even trying to impersonate another voter. By consequence, we assume that a significant part of the voter in untrustworthy.

Print Office

The Print Office combines the Control Component's contributions, generates the codes, and prints the code sheets. It runs in a controlled and offline environment on the canton's premises. All operations in the print office are subject to strict four-eyes principles and are executed on hardened laptops with special access rights. Therefore, the print office in considered as trustworthy.

Electoral Board

The Electoral Board performs the final decryption of the votes. The electoral board's secret key is securely stored on a smart card and is distributed to the electoral members. This requires that at least one of the electoral board members is trustworthy.

Electoral Administrators

Through the Electoral Administrators, the canton sets up the ballot by configuring the e-voting platform, called Voting Server, with voting options and assigns the authorisations for electronic voting.

Auditors

The Auditors verify that the parties faithfully executed their operations within the system using a software called the Verifier which runs various verifications. This requires that at least one trustworthy auditor with their verifier checks an election event.

3.2.2 Building Blocks

The e-voting solution uses functional decomposition to separate responsibilities. It is composed of several components which represent applications or services, and libraries which encapsulate specific functionalities[17]. This decomposition matches the structure of the Gitlab repositories[2].

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

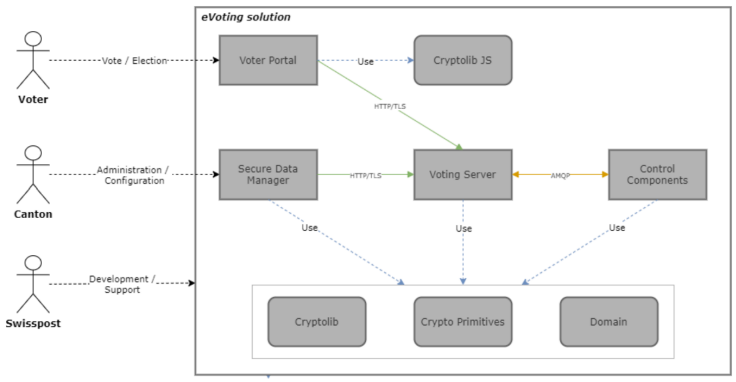

Figure 3.1: Building blocks view - Level 1 - e-voting solution
[17]

Voter Portal

The Voter Portal is a web application to perform the full voting process from the voter's perspective.

Voting Server

The Voting Server is a system composed of several microservices where each is responsible for one part of the voting process and provides an API. Precisely, it authenticates the voter, processes the votes, and stores them. The Federal Chancellery's Ordinance on Electronic Voting imposes the assumption that the Voting Server might be under malicious control and by consequence is considered untrustworthy.

Control Components

The Control Components compose a system in which they work together as a group. They generate the return codes, shuffle the encrypted votes, and decrypt them at the end of the election while guaranteeing the integrity of the voting protocol. There are two types of control components: the Return Codes Control Component (CCR) and the Mixing Control Component (CCM). The Return Codes Control Component compute the Choice Return Codes and the Vote Cast Return Codes in interaction with the setup component (configuration phase) and the voting server (voting phase). The Mixing Control Component mix and partially decrypt ciphertexts in the ballot box. The cryptographic part of the system avoids a single point of vulnerability distributing the server-side cryptographic operations onto four Control Components. The fourth Distributed Mixing Control Component node is found in the offline Secure Data Manager. It is assumed at least one of them must be trustworthy while three of them might be under an

adversaries' control.

Secure Data Manager

The Secure Data Manager is a system that provides the necessary cryptographic functionalities to securely configure and manage an election. It centralizes, in one application, the administration, configuration, and tallying of an election.

Cryptolib

Cryptolib is a library that provides key sharing and encryption capabilities to the voting protocol (eg. El Gamal key pair generation and encryption, X509 certificates, etc.). Its goal is to prevent insecure usage of cryptographic algorithms and providers.

Crypto Primitives

Crypto Primitives is an open-source, server side, robust, and misuse-resistant library which implements cryptographic algorithms used for the voting protocol. An important part of the crypto-primitives library focuses on the implementation of the verifiable mix net and the zero knowledge proof.

Cryptolib JS

Cryptolib JS is a client side javascript library which groups and implements the concepts of Cryptolib and Crypto Primitives.

Domain

Domain is a library that groups domain objects shared by the different voting system components.

Voting Client JS

The voting-client-js builds upon the cryptolib-js and defines a javascript frontend API for the voter-portal. It provides the authentication of the voter, encrypts the votes, calculates the partial Choice Return Codes and encrypts them. It also provides zero-knowledge proofs for the

encrypted vote and the partial Choice Return Codes, the signing of the vote, and the generated confirmation key. In order to provide vote secrecy, the Swiss Post Voting System assumes that the attacker is not controlling the voting client. However, no cryptography can prevent an attacker from spying on the voter's choices on malicious clients.

3.2.3 Verifier

The Verifier is a technical tool used to check that the vote has been conducted properly.

3.3 Phases

From a user's point of view, the Swiss Post Voting system is a return code scheme[15]. Prior to the election, every voter receives a printed code sheet which contains four types of codes that are unique for every voter and every election event:

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

To begin with, the voter enters the Start Voting Key in order to authenticate themself and selects their vote(s). In turn, the system responds with a Choice Return Code for each selected voting option. The voter checks that the Choice Return Codes match the ones printed on the voting card. If the Choice Return Codes match, the submitted vote corresponds to the voter's intention, the voter enters the Ballot Casting Key. In the case of the Choice Return Codes not matching, the voter aborts the process and alerts the election authorities. At the end, the system acknowledges a successful confirmation by sending back the Vote Cast Return Code.

From a "runtime" point of view, the cryptographic protocol divides the Swiss Post Voting System's into three parts: configuration phase, voting phase, and tally phase[16].

3.3.1 Configuration phase

The configuration phase consists of two sub-protocols: SetupVoting and SetupTally.

SetupVoting

The voter's codes are generated in order to be subsequently sent to the voter by postal mail.

Each control component generates a CCRj Choice Return Codes encryption key pair $(pkCCR_j, skCCR_j)$ and the print office combines the generated keys' public part yielding the Choice Return Codes encryption public key $pkCCR$.
The print office generates the verification card key pair (K_{id}, k_{id}) for each voter. It hashes, squares, and encrypts the partial Choice Return Codes pCC_{id} and the Confirmation Key CK_{id} with the setup public key pksetup. The print office also sends the ciphertexts to the Return Codes control components CCR.
The Return Codes control components CCR raise the ciphertexts to the Voter Choice Return Code Generation secret key $k_{j,id}$ and Voter Vote Cast Return Code Generation secret key $kc_{j,id}$, respectively, and return the result to the print office.
The print office generates the short Choice Return Codes cc_{id} and short Vote Cast Return Code VCC_{id}, encrypts them, and maps them to the long Return Codes in the Return Codes Mapping table CMtable.
If the auditors verify the configuration phase successfully, each voter receives a Voting Card VCardid containing the Start Voting Key SVK_{id}, the short Choice Return Codes cc_{id}, the Ballot Casting Key BCK_{id} and the short Vote Cast Return Code VCC_{id}.

SetupTally

During SetupTally, the print office and the mixing control components generate the election public key used for encrypting the votes. Each control component, except the last one, generates a CCMj election key pair and sends its CCMj election public key to the print office. The print office combines the CCMj election public keys' public parts to create the election public key and send it to the voting server. The print office also generates the electoral board key pair for the electoral board.

3.3.2 Voting phase

The voting phase consists of two interactive protocols SendVote and ConfirmVote, where a voter sends and confirms their vote with a voting client and their voting card. At the end of the phase, the auditors execute a verification procedure VerifyVotingPhase to ensure that the ballot box bb is consistent.

SendVote

First, the voter authenticates themself using a Start Voting Key (SVK) and selects their voting options. Then the voting client creates an encrypted vote based on the voter's selection and sends it to the voting server. The Return Codes control components CCR must validate the encrypted vote. Eventually, the control components verify the ballot, decrypt the partial Choice Return Codes pCCid and derive the long Choice Return Codes lCC. Using this long Choice Return Codes lCC, the voting server extracts the short Choice Return Codes ccid from the Return Codes Mapping table CMtable. Finally, the voter verifies the short Choice Return Codes ccid.

ConfirmVote

The voter enters the Ballot Casting Key BCKid and the voting client derives the Confirmation Key CKid to send it to the voting server. Afterwards, the control components derive the long Vote Cast Return Code lVCCid and using the Return Codes Mapping table CMtable the voting server extracts the short Vote Cast Return Code VCCid. To finish, voters verify the short Vote Cast Return Code VCCid.

3.3.3 Tally phase

During the tally phase, the voting server and the mixing control components decrypt the votes and compute the election result. The goal is to protect vote secrecy and guarantee universal verifiability. The phase is composed of two protocols, MixOnline and MixOffline. On top of that, two algorithms, VerifyOnlineTally and VerifyOfflineTally, are used to verify the correcteness of all operations during the tally phase and ensure that the protocol was executed faithfully in all parties. The auditors run VerifyOnlineTally after the MixOnline protocol and in case of success the electoral board releases its key pair to the final CCM to perform MixDecOffline. The auditors also run VerifyOfflineTally after the MixDecOffline protocol to ensure the correctness of the last Mixing control component output.

MixOnline

Initially, the voting server strips all information from the encrypted votes except the ciphertext and discards all unconfirmed votes. The auditors can verify that the voting server did not alter the votes (include or discard votes) and the cleansing is deterministic and reproducible. Next, all but the last Mixing control components shuffle and partially decrypt the list of ciphertext received from the preceding control component, or from the voter server in case of the first control component. In addition, MixOnline proves correct shuffling and partial decryption.

MixOffline

The last mixing and decryption step is executed offline by the cantons and requires the presence of the electoral board. During this last step, the last Mixing control component shuffles and decrypts the permuted ballot box received in order to decode and tally the votes. Furthermore, MixOffline proves correct shuffling and decryption, and returns the election result.

Chapter 4

Methodology

This chapter aims to provide explanation on why some specific parts of the system were chosen to be analyzed. It also describes what is looked for in the analysis.

4.1 Starting point

The strategy was to first read the documentation and understand the whole system and protocol. The first step was to identify who the actors are and what roles do they play in the system. It was also important to know who was trustworthy and who was not as this gives different possibilities to an attacker. Already, this allows us to understand the overall flow of the procedure and how the system will be decomposed. The second point was to understand what is expected of this system and precisely what are the main three security objectives. After this, we started doing research in order to learn the whole architecture of the system and the different components called building blocks (which slightly differ from the actors). This covers mainly their role, their place in the architecture and their own architecture. The final research was about the precise flow of the protocol. The protocol is divided into phases. Each phase uses different components and involves different actors in order to achieve a different goal, from the authentication of the voter to the tally. All these steps are covered in chapter 3. To start looking into the code, we begun by understanding the organisation of the project on Github and then retrieved most of the used libraries in the different building blocks. The idea was to go over and see if any might cause issues. This process did not produce any findings as the software purposefully does not use many outside libraries.

4.2 Where to look at

At the beginning, two different directions were considered: focusing on specific building block(s), or focusing on phases. At first, we decided to focus on building blocks as this approach is more targeted than looking at phases, which might use many building blocks at the same time. The strategy was to find which building blocks are the most interesting to focus on, taking into account what protocol they implement and how widely they are used. This lead us to considering the Control Components, Voting Server, and cryptolib.

After starting linearly with the functions implemented by the control component during the configuration phase 5.2, it quickly became evident that this approach would not yield interesting results. This was because the approach did not target the most interesting functions of a building block, but instead the functioning of the whole. Instead, our approach should target more specific parts of each building block.

The Swiss post e-voting system implements its own libraries and cryptographic protocols. These were considered more useful to look at as these can be tricky to implement and are very critical as an important part of the system's security rely on them. These analyses can be found in section 5.1.

Furthermore, the whole system protocol relies on various Return Choice Codes, which are specific to this implementation. These are used during the voting phase; their functionality is crucial to understand. This lead us to consider the second part of the Voting Phase which can be found in section 5.3.

Finally, during the last phase of the protocol, a lot of different schemes are used during the Shuffling and decryption part. These scheme are complex and the security of the system relies on them. Again, as the system implements the majority of these schemes, this is a place where it would be likely to find implementation errors, breaches, or errors during the combination of several schemes. This corresponds to section 5.4.

4.3 What to look at

The first focus of this work is on the implementation. To start, some research was conducted on what are the most common implementation issues in cryptographic protocols [1] [2] [3] [4] [5] [6] [7] [8].

[1] https://owasp.org/Top10/A02$_2$021 − $Cryptographic_Failures/$
[2] https://cwe.mitre.org/top25/archive/2021/2021$_cwe_top25.html$
[3] https://www.sans.org/top25-software-errors/
[4] https://waverleysoftware.com/blog/top-software-vulnerabilities/
[5] http://www.cs.umd.edu/ jkatz/security/downloads/kohno.pdf
[6] https://www.crypteron.com/blog/the-real-problem-with-encryption/
[7] https://inst.eecs.berkeley.edu/ cs161/sp16/slides/3.8.CryptoMistakes.pdf
[8] https://littlemaninmyhead.wordpress.com/2017/04/22/top-10-developer-crypto-mistakes/

In addition, the work mandated by the Federal Chancellery[10] was used. Looking at what the authors covered gives leads as to what kind of issues there might be and what could be left to be checked. This resulted in the following list of potential issues to check for:

- Verify that the implementation follows the protocol from the documentation
- Verify that the implementation follows the source-code best practices:
  1. it uses Java best practices for naming conventions
  2. the Java doc should associate the Java parameter name with the variable name used in the pseudo code
- Make sure the membership and consistency checks for all algorithm parameters are performed
- Enforce group Operations
- Make sure that the implementation of the hashing provides collision resistance when multiple values are hashed
- Ensure that there is no import of insecure library
- Verify that it does not implement improper restriction of operations within the bounds of a memory Buffer, or authorizes Out-of-bounds Read and Writes (Buffer, string overflows). As the implementation uses java, this can not happen[9]. However, it was important to still mention it.
- Ensure that the system does not encrypt data without signing it or
- accepts signed data without authenticating the signer.
  As in this work we focus on the functions themselves and not the communication in between services, these last two point will not actually be used.

For each function, we also verify that it computes what it is designed for, that the result is correct, and that it provides the security it aims for.

In addition to this, for functions running in untrustworthy environment, we try to examine if an attacker could mount an attack. We consider as an attack any situation where one can learn more information than we are supposed to, considering the security properties. We also consider as an attack any situation where an attacker can influence the outcome of the election.

To finish this chapter, we would like to mention that approaching such a large project is not simple and one can feel lost. Indeed, to know what to look for, or even where to look, is a whole process to learn. This section is also here to help the next person that will work on this project.

[9] https://www.baeldung.com/java-overflow-underflow

Chapter 5

Analysis

This chapter covers the whole analysis done. It starts with the cryptographic protocols investigated and then proceeds by phases.

5.1 Cryptography

In order to have complete faith in the system and provide robust primitives, the Swiss post e-voting system implements its own cryptographic protocols inside the Cryptolib[1] and Cryptomprimitives[2] libraries.

5.1.1 Multi-recipient ElGamal scheme

The ElGamal encryption scheme is an asymmetric public-key encryption scheme IND-CPA secure[20]. IND-CPA security implies that it is not feasible to extract any information about the message from the ciphertext. It also has multiplicative homomorphic properties that can be used to manipulate encryptions.
The whole El Gamal scheme is implemented inside the Cryptoprimitive library.

Parameters generation

The ElGamal[4] encryption scheme is instantiated over a cyclic group of quadratic residues G_q of order q with generator g, where the decisional Diffie-Hellman (DDH) problem is believed to

[1] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/tree/master/cryptolib
[2] https://gitlab.com/swisspost-evoting/crypto-primitives

be hard. This last step is ensured by having p and q as large primes, and the quadratic residuosity implies $p = 2q + 1$.

The Swiss post e-voting system implements a function called $GetEncryptionParameters$ [3] which generates the group modulus p, the group cardinality q such that $p = 2q + 1$, and the group generator g.

The default generator g is designated as the smallest element in Q_p. This is performed by iterating over [2,4] and taking the first element in G_q. This indeed computes the smallest generator g for the set of quadratic residues modulo p: as $p = 2q + 1$ and q is prime, p is considered as a safe prime and then every element of the set of quadratic residues modulo p is a generator for this set. It is not an issue to fix an upper-bound equal to 4 as $4 = 2^2$ and will always be a quadratic residue considering that p will always be $p > 2$ due to it security level. In order to make sure that an element i is in G_q, one can easily verify that $i^q \bmod p = 1$. Indeed, if i is in the set of quadratic residues modulo p, we have $i = x^2 \bmod p$ for an integer x. Thus, $i^q \bmod p = x^{2q} \bmod p = x^{p-1} \bmod p = 1$ by the Little Fermat theorem.
The group cardinality q is the main computation of the function. It is computed through a loop iterating over a counter. It starts by creating an output of 2048 bits (2048 bits is the default value in the e-voting system implementation but it can be extended to 3072) using SHAKE-128 which uses different inputs one of which is the counter. SHAKE-128 will be later explained. The byte <0x01> is prepended to the digest's output and a bitwise right-shift operation is performed to enforce the initial candidate value for q to be in the interval $[2^{|q1|}, 2^{|q|})$. After this, a computation to ensure that q is odd is performed and finally it checks for the primality of q and $p = 2 * q + 1$. If both are prime it jumps to the next parameter generation q, otherwise the loop starts again incrementing the counter used as part of the input for SHAKE-128.
At the end of each loop trying to derive a prime q, a primality test is perform. The implementation uses the function $isProbablePrime$ from the java class BigInteger[4]. This function checks that the probability that the BigInteger is prime exceeds $(1 - 1/2^{certainty})$. In the e-voting implementation the certainty default value is set to 112 but can be extended to 118. Looking at the implementation of the class [5], one can see that the function uses Miller-Rabin primality testing[18] and for numbers greater than 100 bits, Lucas-Lehmer primality testing[9], two well-known and reliable primality tests.
Regarding SHAKE-128, it is an extendable-output function (XOF) and has been standardized by NIST[19] . It is a generalization of a cryptographic hash function: instead of creating a fixed-length digest, it produces outputs of any desired length. A random function whose output length is d bits cannot provide more than $d/2$ bits of security against collision attacks and d bits of security against preimage and second preimage attacks. In the case of the Swiss post e-voting system, we reach the upper-bound of 128 bits of security for SHAKE128 as $d/2 > 128$ in all cases. It is important to note that XOFs are not approved as hash functions. The implementation uses

[3] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/elgamal/EncryptionParameters.java

[4] https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.htmlisProbablePrime(int)

[5] https://hg.openjdk.java.net/jdk/jdk/file/274a0bcce99d/src/java.base/share/classes/java/math/BigInteger.java

the already implemented SHAKEDigest class from the bouncycastle library, with the election name as the seed.
As efficiency is not a concern in the Swiss post e-voting system, this type of loop with no fixed end and no stopping conditions in case of a long runtime is not considered as problematic.
p is directly computed from q as $p = 2q + 1$ and verified to be prime during the computation of q which is also prime.

In conclusion, the algorithm works perfectly and provides the security aimed for. Everything is implemented correctly.

Encryption

The Swiss post e-voting system implements a function called $GetCiphertext$ [6] which computes an ElGamal ciphertext using randomness.

ElGamal encryption scheme[4] encrypts a vector of message $m \in G_q^l$ using a vector of public key $pk \in G_q^k$ such that $pk_i = g^{sk_i}$ and a random $r \in Z_q$. In the implementation, the random exponent r is pre-computed and given as input.
If the message has less elements than the public key ($l < k$), the excess public key elements are multiplied such that $pk_l = pk_l * ... * pk_k$. If the message has more elements than the public key ($l > k$) an error is thrown.
The ciphertext is computed as follow : $c = (c_0, c_1, ..., c_{l-1}) = (g^r, pk_1^r * m_1, ..., pk_l^r * m_l) \in G_q^{l+1}$.
The implementation performs the public key compression relative to the message size and the message is encrypted according to the ElGamal encryption scheme.

All the important checks are made on the input and the scheme is correctly implemented.

Decryption

The Swiss post e-voting system implements a function called $GetMessage$ [7] which retrieves the message from the ciphertext.

The ElGamal scheme decrypts a ciphertext $c \in G_q^{l+1}$ using a vector private key $sk \in Z_q^k$. If the message has less elements than the public key ($l < k$), the excess secret key elements are added such that $sk_l = sk_l + ... + sk_k$, corresponding to the multiplication of the public key using the sk_i as exponents. If ($l > k$) an error is thrown. Then we decrypt the ciphertext and output a vector of message $m \in G_q^l$ such that $m_i = c_i * c_0^{-sk_i} = c_i * g^{-sk_i * r}$. Contrary to the encryption, the randomness is not needed as input as it is exponentiated in the first element of the ciphertext.

[6] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/elgamal/ElGamalMultiRecipientCiphertext.java

[7] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/elgamal/ElGamalMultiRecipientMessage.java

The implementation performs the secret key compression relative to the ciphertext size and the ciphertext is decrypted according to the explained ElGamal encryption scheme. The ElGamal encryption scheme has perfect correctness: for any given pair of keys (sk, pk) generated by algorithm KeyGen, it holds that Dec(Enc(m, pk), sk) = m for all $m \in G_q$.

All the important checks are made on the input and the group operations are corrects. It implements the correct decryption regarding the encryption, and the naming is consistent and follows java best practises.

5.1.2 Digital signatures

Digital signatures ensure authenticity and integrity which guarantees that the adversary cannot modify ciphertexts with no detection. Additionally, the sender signs one-time additional information to prevent replay attacks and to allow the receiver to verify the context's correctness. In the Swiss post e-voting it is used for a multitude of reasons. For example, to sign the messages that the parties exchange during the configuration phase, the election public key during the voting phase or during the tally phase when the auditors must check the signature of each control component's message.

The Swiss Post Voting System claims to use the RSA-PSS signature scheme with 2048-bits key length[16] with SHA-256 as a digest. As for example in [8].

Regarding the security of RSA-PSS, when instantiated with "ordinary" collision-resistant hash functions, RSA-PSS can be tightly related to the hardness of the RSA inversion problem. The proof can be found in Bellare-Rogaway's work[2] and RSA-PSS is standardised in PKCS1 v2.1[8]. Therefore, if one has or wants to use RSA signatures, then RSA-PSS is the right choice of scheme.

The implementation uses the java Signature class which provides applications that include the functionality of a digital signature algorithm. [9]. In particular, they call $Signature.getInstance(SHA256withRSA)$. The oracle documentation states that the signature algorithm with SHA-* and the RSA encryption algorithm are as defined in the OSI Interoperability Workshop, using the padding conventions described in PKCS 1[10]. This seems to use RSASSA-PSS which corresponds to RSA-PSS with appendix.

The implementation correctly uses the java Signature class and by consequence is considered correct. However, as the system does not implements directly the signature, the security of the signature relies on the class used.

Nonetheless, it is interesting to question the choice of an RSA based signature scheme. The

[8] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/return-codes-service/src/main/java/ch/post/it/evoting/controlcomponents/returncodes/securelogger/SecureLogAppender.java

[9] https://docs.oracle.com/javase/7/docs/api/java/security/Signature.html

[10] https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.htmlSignature

whole Swiss post e-voting system relies on the El Gamal scheme for encryption or Schnorr protocol for zero-knowledge proof which both rely on the discrete logarithm problem. It is questionable to chose a scheme based on the RSA inversion problem knowing that these schemes also require bigger security parameters. Schnorr signature could be an option, or some variant of it.

5.1.3 Randomness generation

High entropy for randomness generation is one of the most complicated goals to achieve in cryptography. It is essential in our system as if it fails an adversary could for example predict a secret key and use it to break vote secrecy. Swiss post e-voting system implements its own randomness service in order to provide different types of output such as integers or string [11]. The randomness generation relies on the java class SecureRandom [12], a class which provides a cryptographically strong random number generator.

When instantiating a SecurerRandom object, the system calls the default constructor. It does not give any seed material or provider as input. In this case, the SecureRandom class constructs a secure random number generator (RNG) implementing the default random number algorithm. The list of standard RNG algorithms can be found in the Oracle documentation [13]. The default constructor also traverses the list of registered security Providers, starting with the most preferred Provider. After this, the returned object will be self-seeded. Accepting defaults is usually not the best java practise as in this case all the security parameters rely on the SecureRandom class in terms of choices.

It has to be mentioned that randomness generation is implemented in both Cryptoblib and Cryptoprimitives, sometimes with the exact same code (e.g. for random integer generation). Code duplication should usually be avoided. However, in this case it is done do separate work and avoid dependencies in between Cryptoblib and Cryptoprimitives libraries as a trade-off.

Both randomness services and their functions were checked and everything is well implemented regarding the steps explained in the methodology chapter 4.

5.1.4 Hashing

The Swiss post e-voting system uses the SHA-256 hash function when it needs to hash an input with no other constraint. However, when the system needs to compute the hash value to a given bit length, the system uses SHAKE-256. The system respectively implements two

[11] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/math/RandomService.java

[12] https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

[13] https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.htmlSecureRandom

functions called $RecursiveHash$ and $RecursiveHashOfLength$. SHA-256[5] as well as SHAKE-256[19] are currently considered as secure and satisfy all current security goals and properties of hash functions. Hash function collision resistance is ensured by SHA-256 and SHAKE-256, thus collisions could only come from input collisions: different inputs formatted into equal ones. The crypto-primitives specification prevents hash-collisions across different domains by using type separators, however this has not been implemented yet.

Another point is the system hashes the concatenation of multiple inputs. If inputs were of non-fixed sizes, this could lead to hash-collisions for different inputs which concatenated results into identical outputs. However, this is not the case as all input parameters will be same same size inside a same election (e.g. same key sizes, same group parameters, same number of voting options) and consequently the system is input-collisions safe.

Another thought could be hash replay-attack inside the same elections. Hashing is widely used in zero-knowledge proof so one might want to re-use a hash output from another vote to hide an alteration of a vote. However, a hash output alone is useless as it is never used alone but in functions using other parameters such as plaintext or ciphertexts. In order to be efficiently used, the hash steal should be combined with these other stolen parameters. These cannot be collected together as this is not possible given the trust assumptions (mainly that at least one of the Control Components which proceeds the partial decryption of the votes is trustworthy).

Replay attacks in between elections are also ruled out as non-predictable hashes always contain at least one input parameter specific to the current election.

The function providing hashing can be found here [14]. They were checked and everything was well implemented with the right checks.

5.2 Configuration phase

We focused on the protocols run by the Control Components. This is due to a choice done at the beginning which was to focus on a specific building block. Control Components were chosen due to their important contribution in generating the return codes and participating in the shuffling of encrypted votes and their decryption. The Control Components were also considered as only one of them is considered trustworthy and this leads to possibly more attacks.

[14] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/hashing/HashService.java

5.2.1 SetupVoting

During the SetupVoting phase, two algorithms are run by the Control Components: GenKeysCCRj and GenEncLongCodeShares.

GenKeysCCRj

Each control component generates a CCRj Choice Return Codes encryption key pair (pkCCRj , skCCRj) and a CCRj Return Codes Generation secret key. The first pair is used to encrypt the partial Choice Return Codes during the voting phase. The CCRj Return Codes Generation secret key is used in the next algorithm $GenEncLongCodeShares$ to generate other pair of keys.

4.1.1 GenKeysCCR

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

Algorithm 4.1 $\mathsf{GenKeysCCR_j}$

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election Event ID $\mathsf{ee} \in (\mathbb{A}_{Base16})^{1\text{to}}$
The CCR's index $j \in [1, 4]$
Maximum number of selectable voting options $\varphi \in \mathbb{N}^*$ ▷ See section 3.4.3

Operation:

1: $(\mathbf{pk}_{\text{CCR}_j}, \mathbf{sk}_{\text{CCR}_j}) \leftarrow \mathsf{GenKeyPair}(p, q, g, \varphi)$ ▷ See crypto-primitives specification
2: $\mathtt{k}'_j \leftarrow \mathsf{GenRandomIntegerUpperBounded}(q)$ ▷ See Algorithm 8.5

Output:

CCR_j Choice Return Codes encryption public key $\mathbf{pk}_{\text{CCR}_j} = (\mathtt{pk}_{\text{CCR}_j,0}, \ldots, \mathtt{pk}_{\text{CCR}_j,\varphi-1}) \in \mathbb{G}_q^\varphi$
CCR_j Choice Return Codes encryption secret key $\mathbf{sk}_{\text{CCR}_j} = (\mathtt{sk}_{\text{CCR}_j,0}, \ldots, \mathtt{sk}_{\text{CCR}_j,\varphi-1}) \in \mathbb{Z}_q^\varphi$
CCR_j Return Codes Generation secret key $\mathtt{k}'_j \in \mathbb{Z}_q$

Figure 5.1: GenKeysCCRj algorithm
[16]

$GenRandomIntegerUpperBound$ is not addressed in this section as it is related to section 5.1.3 concerning the randomness generation.

Regarding $GenKeyPair$, the function is actually implemented twice: one time directly inside the e-voting part inside Cryptolib, and one time in the Cryptoprimitive library. This lead to some confusion as to whether they were really similar and if so, why the code was duplicated. Additionally, it raised the questions of what each implementation was used for, and why each implements the same algorithm but differently. After research, no particular reason was found. We conclude this was in order to separate tasks clearly and limit the calls in between each library. However, since the beginning of this work this has changed and been made explicit in the code. It now states that they are equivalent. They also wrote about it and said "Some cryptographic primitives are implemented both in the crypto-primitives and the cryptolib (for

Algorithm 4.2 GenKeyPair: Generate a multi-recipient key pair

Input:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Number of key elements $N \in \mathbb{N}^*$

Operation:

1: **for** $i \in [0, N)$ **do**
2: $sk_i \leftarrow$ GenRandomInteger(q) ▷ See algorithm 3.1
3: $pk_i \leftarrow g^{sk_i} \mod p$
4: **end for**

Output:

A pair of secret and public keys $\{(sk_i, pk_i)\}_{i=0}^{N-1}, sk_i \in \mathbb{Z}_q, pk_i \in \mathbb{G}_q$

In the Swiss Post Voting System, a ciphertext (i.e. an encrypted voter ballot) comprises the following elements:

- g^r, the left-hand side part of a standard ElGamal encryption,
- $pk_0^r \cdot m_0$, the encryption of the main message, under the first part of the system's public key. The message is the product of the choices of the voter,
- $pk_j^r \cdot m_j$ for all $1 \leq j < l$, where m_j is the encoding of a write-in chosen by the voter, or 1.
- When the number of write-ins a voter is eligible to vote for is smaller than the election configuration allows, the remaining keys are compressed by multiplication, so that all keys are used in every case.

The algorithms for handling the key-compression mentioned in the last point are provided below.

Figure 5.2: GenKeyPair algorithm
[16]

instance the ElGamal encryption scheme). The implementations are functionally equivalent. We are continuously replacing the cryptolib implementation with the more robust crypto-primitives one" [15].

The first implementation is interesting to trace up through the system. It starts in the control components return codes service, in the middle of the e-voting system, precisely inside the return code keys repository [16]. There is a function called $addGeneratedKey$ which manages the different CCRj return code keys and the relevant computations as well, such as certification. In order to create the return code keys, it calls the key manager [17], which in turn calls the generator [18]. The generator handles the computation of the CCRj Return Codes Generation secret key, and the CCRj Choice Return Codes encryption key pair, the actual output of the GenKeysCCRj function. In order to compute those keys, the generator calls the ElGamal service inside the Cryptolib library [19], which finally calls the ElGamal key pair generator inside the ElGamal service

[15] https://gitlab.com/swisspost-evoting/e-voting/e-voting
[16] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/return-codes-service/src/main/java/ch/post/it/evoting/controlcomponents/returncodes/service/ReturnCodesKeyRepository.java
[17] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/control-components-commons/src/main/java/ch/post/it/evoting/controlcomponents/commons/keymanagement/KeysManager.java
[18] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/control-components-commons/src/main/java/ch/post/it/evoting/controlcomponents/commons/keymanagement/Generator.java
[19] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/cryptolib/src/main/java/ch/post/it/evoting/cryptolib/elgamal/service/ElGamalService.java

inside Cryptolib [20].
There is a small mismatch in between the documentation and the implementation. The function $generateKeys$ corresponds to $GenKeysCCR$ but also to the computation of the CCRj Return Codes Generation public key, computed as the ElGamal public key corresponding to its secret key and following the $GenKeyPair$ algorithm. However, this is not considered as an issue as it is more practical to generate the key pair together. Otherwise, the function follows exactly the protocol with the exception that it does not call a function called $GenRandomIntegerUpperBound$ to generate the randomness, but an equivalent function named differently. Not naming the functions in the code corresponding to their name in the documentation (GenRandomIntegerUpperBound->getRandomExponentValue, GenKeysCCR->generateKeys) is not best practise, however this has no impact on the security of the system and thus can be ignored. In general, in this function implementation, opposed to the major part of the code, they did not pay attention to the naming of variables matching the documentation.

The second implementation can be found inside the Cryptoprimitive library, inside the ElGamal section [21]. The function is called $genKeyPair$ and corresponds to the $GenKeyPair$ function which implement ElGamal key pairs in the Cryptoprimitive library.

For both implementations, every necessary membership and consistency check is done correctly; some unnecessary checks are done as well. The implementation implement their own checks (eg. for groups comparison) or use $com.google.common.base.Preconditions.checkArgument$ and $com.google.common.base.Preconditions.checkNotNull$ which follow the recommendations [22]. Group operations' consistency are also correct. Regarding outside libraries they only use java and springframework which are trustworthy.

GenEncLongCodeShares

GenEncLongCodeShares algorithm is run by each return codes control component CCR_j to create shares of the long return codes. It is composed of three main functions : $DeriveKey$, $GetCiphertextExponentiation$, $GenExponentiationProof$. The function outputs:

- Vector of Voter Choice Return Code Generation public keys: this key and its corresponding secret key are used in an algorithm called $CreateLCCShare$ during the voting phase in SendVote. We won't cover this algorithm here or the derivations where this key is used.
- Vector of Voter Vote Cast Return Code Generation public keys: its corresponding secret key will be used for generating the Voter Long Vote Cast Return Code Share in the

[20] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/cryptolib/src/main/java/ch/post/it/evoting/cryptolib/elgamal/factory/CryptoElGamalKeyPairGenerator.java
[21] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/elgamal/ElGamalMultiRecipientKeyPair.java
[22] https://guava.dev/releases/19.0/api/docs/com/google/common/base/Preconditions.html

$CreateLVCCShare$ 5.3.4 algorithm during the voting phase.

- Vector of exponentiated, encrypted, hashed partial Choice Return Codes received as input.
- Proofs of correct exponentiation of the partial Choice Return Codes: Schnorr protocol made non-interactive using Fiat-Shamir trick is used zero-knowledge proof (explained in section 5.3.2). It proves that it knows the derived Voter Choice Return Code Generation secret key. This key is used to derive the vector of Voter Choice Return Code Generation public keys and exponentiate the vector of encrypted, hashed partial Choice Return Codes received as input.
- Vector of exponentiated, encrypted, hashed Confirmation Keys received as input.
- Proofs of correct exponentiation of the Confirmation Keys: zero-knowledge proof is used to prove that it knows the derived Voter Vote Cast Return Code Generation secret keys. This key is used to derive the vector of Voter Vote Cast Return Code Generation public keys and exponentiate the vector of encrypted, hashed Confirmation Keys received as input.

We last looked at the function $GenEncLongCodeShares$ on the 4th of november on [23]. However since then, the file was removed and the implementation changed. We can no longer double check what was previously done even though everything was well implemented at the time with the right checks and group consistency.

Luckily, $DeriveKey$ and $GenExponentiationProof$ are both checked and explained later in the function $CreateLVCCSharej$ during the voting phase 5.3.4. $GetCiphertextExponentiation$ is also checked later on during the shuffling part of the tally phase, precisely in $GenVerifiableShuffle$ 5.4.4. This implies that the analysis of the skeleton of $GenEncLongCodeShares$ was no longer valid but all functions called inside separately are and are developed later in this report.

5.2.2 SetupTally

During the SetupTally phase, each mixing control component (CCM_j) generates a key pair corresponding to the election public key $EL_{pk,j} \in G_q^\mu$ and election secret key $EL_{sk,j} \in Z_q^\mu$

SetupTallyCCMj

SetupTallyCCMj generates this key pair for each control component.

[23] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/domain/src/main/java/ch/post/it/evoting/domain/returncodes/ReturnCodeGenerationOutput.java

Theoretically, the algorithm simply calls the previously mentioned $GenKeyPair$ algorithm with inputs: p, q, g, μ. As seen before p is group modulus, q is the group cardinality, and g is the group generator. $\mu \in N^*$ represents the maximum number of write-in options in the election.

Similarly to the first function $GenKeysCCRj$, in practise there is no direct $SetupTallyCCMj$ implementation. The mixing control components have a $CcmjKeyRepository.java$ [24] where there is a function called $addGeneratedKey$. This function is meant to check whether the CCMj election keys were already created or not. If not, it calls $keysManager.createCcmElectionKey$ [25], which in turn calls $generator.generateteCcmElectionKey$ [26]. Finally it is $generateCcmElectionKey$ which implements what seems to be $SetupTallyCCMj$ algorithm. It is done by calling the previously mentioned $generateKeyPair$ from the ElGamal service and signs it.

Java best practises are followed in these different functions. Membership and consistency checks for all algorithm parameters are also well performed as well as all the checks on the parameters inputs.

Inside $createCcmElectionKey$, compared to all other functions seen during this work, a lot of libraries are imported, mainly from java*, javax*, org.slf4j*, and org.springframework*, but none of them seem insecure.

This shows a gap that can happen between the documentation and the implementation. The documentation includes a theoretical implementation of the $SetupTallyCCMj$ function. In practice, the function is actually implemented by different components and not by this theoretical function. At the end, the right functions are called to derive and store the keys and the right checks are performed on the inputs.

5.3 Voting phase

We focused on the second part of the voting phase, called $ConfirmVote$ 3.3.2. This second part is composed of four different algorithms called $CreateConfirmMessage$, $CreateLVCCSharej$, and $ExtractVCC$, respectively.

In order to understand these algorithms, one must understand the cryptographic protocols behind them. This section gives a brief overview of these protocols.The system uses the Schnorr protocol explained in the work by Maurer[11] in order to compute zero-knowledge proof and combines it with the Fiat-Shamir[3] to make it non-interactive.

[24] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/distributed-mixing-service/src/main/java/ch/post/it/evoting/controlcomponents/mixing/service/CcmjKeyRepository.java

[25] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/control-components-commons/src/main/java/ch/post/it/evoting/controlcomponents/commons/keymanagement/KeysManager.java

[26] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/control-components-commons/src/main/java/ch/post/it/evoting/controlcomponents/commons/keymanagement/Generator.java

5.3.1 Schnorr protocol

In the Schnorr protocol, Peggy wants to prove that she knows the value of x without revealing it, to a party having access to h^x. The Schnorr protocol can be found on Figure 5.3. Vic can check

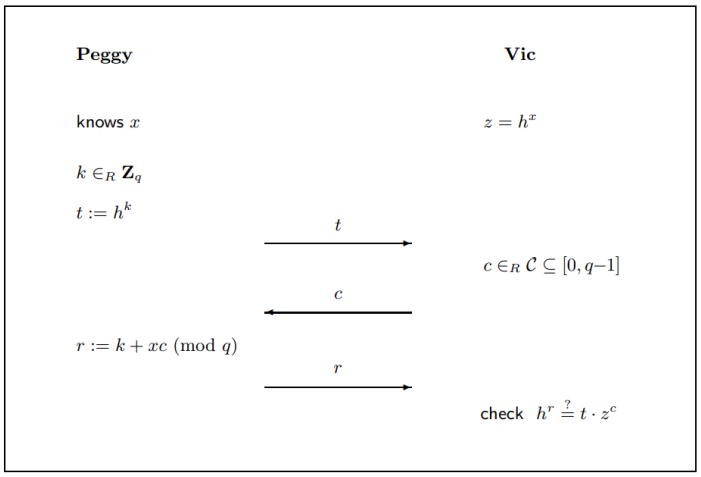

Fig. 1. The Schnorr protocol

Figure 5.3: The Schnorr protocol
[11]

that Peggy knows x by computing $h^r = t * z^c$. Indeed, first $r = k + x * c$ can only be computed with the knowledge of all exponents. Second, $h^r = h^{k+x*c} = h^k * hx * c = h^k * (h^x)^c$ which can be verified with having only the knowledge of h^x. The proof can be found in the Maurer work[11].

5.3.2 Fiat-Shamir trick

The Fiat-Shamir trick[3] removes any interactions from the above protocol in order to switch to a non-interactive setting. Each value sent from the challenger, in this case the value c, is replaced by the output of a cryptographic hash function. The hash function takes as inputs the previously sent data from the prover (Peggy) to the challenger (Vic). Context variables can also be added in some cases as well as auxiliary information.

5.3.3 CreateConfirmMessage

The voter enters the Ballot Casting Key in the voting client. The voting client processes the Ballot Casting Key using the Verification card secret key and yields the confirmation key. The algorithm can be found on Figure 5.4. In the documentation, the algorithm first hashes and squares the Ballot casting key and then exponentiates the result with the Verification card secret

5.2.1 CreateConfirmMessage

The voter enters the Ballot Casting Key $\texttt{BCK}_{\texttt{id}}$ in the voting client, which executes the CreateConfirmMessage algorithm.

Algorithm 5.8 CreateConfirmMessage

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\texttt{ee} \in (\mathbb{A}_{Base16})^{1_{\texttt{ID}}}$
Character length of the Ballot Casting Key $1_{\texttt{BCK}} = 9$

Input:

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

Operation:

1: $\texttt{hBCK}_{\texttt{id}} \leftarrow \mathsf{HashAndSquare}(\mathsf{StringToInteger}(\texttt{BCK}_{\texttt{id}}))$ ▷ See crypto primitives specification
2: $\texttt{CK}_{\texttt{id}} \leftarrow \texttt{hBCK}_{\texttt{id}}^{\texttt{k}_{\texttt{id}}} \bmod p$

Output:

Confirmation Key $\texttt{CK}_{\texttt{id}} \in \mathbb{G}_q$

Figure 5.4: CreateConfirmMessage algorithm
[16]

key. However, in the code [27] the Ballot casting key is not hashed before being squared. This is a mismatch but is not considered as a security issue. If the Ballot casting key is known, the computation is feasible with or without the hashing considering that the hash function SHA-256 is public. The threat is the recovery of the Ballot casting key from the Confirmation key. Even if once an attacker has recovered $hBCK_{id}$, reversing the hashing to find the corresponding Ballot Casting key adds difficulty, the exponentiation performed on $hBCK_{id}$ with the Verification card secret key modulus p is enough to avoid any inversion.

Otherwise, there is nothing else to report regarding the code or the implementation.

5.3.4 CreateLVCCSharej

The CreateLVCCSharej algorithm is executed by the control components [28]. It uses the Voter Vote Cast Return Code Generation secret key as we want to prevent any adversary from using the CreateLVCCSharej algorithm as an oracle to learn Choice Return Codes, and allows up to five executions of CreateLVCCSharej in case the voter entered an invalid ballot casting key. The algorithm gets as input :

- Confirmation key: derived in the previous step.
- Verification Card id: for each election, the system groups each voter id into a verification card set vcs and a voting card set vcds and assign the voter a verification card id and a voting card id.

[27] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/voting-client-js/src/protocol/confirmation-key.js
[28] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/return-codes-service/src/main/java/ch/post/it/evoting/controlcomponents/returncodes/service/CreateLVCCShareService.java

- CCRj Return Codes Generations secret key: computed in the GenKeysCCRj algorithm during the configuration phase 5.2.1.

The system updates the log with the new confirm vote and outputs:

- Hashed, squared Confirmation Key
- CCRj's long Vote Cast Return Code share
- Voter Vote Cast Return Code Generation public key
- Exponentiation proof: non-interactive zero-knwoledge proof of the exponentiation. Implicitly, it proves the knowledge of the Voter Vote Cast Return Code Generation secret key used to generate the corresponding public key and the CCRj's long Vote Cast Return Code share.
- Confirmation attempt number: the number of attempts used by the voter to confirm his vote.

Let's first understand the key hierarchy. During the configuration phase, the control components derive the CCRj Return Codes Generations secret key in the $GenKeysCCRj$ algorithm 5.2.1. This later is used to generate the Voter Choice Return Code Generation key pair and Voter Vote Cast Return Code Generation key pair in the $GenEncLongCodeSharesj$ algorithm 5.2.1. The Voter Vote Cast Return Code Generation secret key is then used for generating the Voter Long Vote Cast Return Code Share which has corresponding entry in the Return Codes Mapping table CMtable.
The Return Codes Mapping table allows the control components and the voting server to retrieve the short Choice Return Codes and access yet to be or confirmed votes. The CMtable is ordered by the hash of the Long Return Codes to break the correlation between the voting option and the order of insertion.

Let's focus on what the $CreateLVCCSharej$ does.
In the $CreateLVCCSharej$ algorithm, the voter derives what corresponds to the Voter Vote Cast Return Code Generation key pair derived in the $GenEncLongCodeShares$ algorithm. These two derivations need to match as we later need to retrieve information from the CMtable.
Let's notice that the derivation of the Voter Vote Cast Return Code Generation secret key differs in the system specification[16] (Algorithm 4.5) and the Swiss Post Voting Protocol Computational proof[15] (Algorithm 12.1.1.5) regarding to the string concatenated before the secret key derivation. As it still matches in both functions $GenEncLongCodeShares$ and $CreateLVCCSharej$, this is not an issue and the System specification documentation[16] is followed.
From this last derived key, it computes the CCRj's long Vote Cast Return Code share which will be used in the next algorithm in section 5.3.5 and computes the exponentiation proof using the $GenExponentiationProof$ algorithm explained below.

Considering the implementation, there is nothing to report. Every check needed (non-null inputs, cross group checks, maximum attempts, insurance that the verification card id was previously used to vote, etc.) are correctly performed and keys are correctly derived. Moreover, each access to the database is performed java prepared statements for SQL eliminates most risks of SQL injections.

We worked on this section the first week of December. Since then, the function $CreateLVCCShare_j$ was removed from the Swiss Post Voting Protocol Computational proof[15], where there used to be some mismatch with System specification[16] (auxiliary information missing in some inputs for deriving keys and names not matching).

GenExponentiationProof

The $GenExponentiationProof$[29] shown in Figure 5.5 implements the Schnorr protocol 5.3.1 combined with the Fiat-Shamir trick 5.3.2 to prove the knowledge of the Voter Vote Cast Return Code Generation secret key.
The witness corresponds to the Voter Vote Cast Return Code Generation secret keys.

Algorithm 6.5 GenExponentiationProof: Generate a proof of validity for the provided exponentiation

Context:

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

Input:

A vector of bases $\mathbf{g} = (g_0, \dots, g_{n-1}) \in {\mathbb{G}_q}^n$ s.t. $n \in \mathbb{N}^*$
The witness – a secret exponent $x \in \mathbb{Z}_q$
The statement – a vector of exponentiations $\mathbf{y} = (y_0, \dots, y_{n-1}) \in {\mathbb{G}_q}^n$ s.t. $y_i = g_i^x$
An array of optional additional information $\mathbf{i}_{\text{aux}} \in (\mathbb{A}_{UCS^*})^*$

Operation:

1: $b \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 3.1
2: $\mathbf{c} \leftarrow \mathsf{ComputePhiExponentiation}(b, \mathbf{g})$ ▷ See algorithm 6.4
3: $\mathbf{f} \leftarrow (p, q, \mathbf{g})$
4: $\mathbf{h}_{\text{aux}} \leftarrow (\texttt{"ExponentiationProof"}, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
5: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\text{aux}}))$ ▷ See algorithms 2.8 and 3.6
6: $z \leftarrow b + ex$

Output:

Proof $(e, z) \in \mathbb{Z}_q \times \mathbb{Z}_q$

Figure 5.5: GenExponentiationProof algorithm
[13]

The base is the generator and the hashed and squared confirmation key.
The statement is composed of the Voter Vote Cast Return Code Generation public key and the Long Vote Cast Return Code Generation share. These are equal to both elements of the base exponentiated to the witness. They correspond to the knowledge given to a challenger that needs to be convinced and does not know the exponent (witness) in the Schnorr algorithm.

[29] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/zeroknowledgeproofs/ExponentiationProofService.java

The Schnorr protocol needs to fulfill some conditions. First, it requires that the challenge space leans in the subset of [0, q1]. This is ensured by the hash generation.
The value q must be prime, which is ensured when it is generated in the Swiss post e-voting.
Maurer work[11] states that "a protocol consisting of s rounds is a proof of knowledge if $1/|C|^s$ is negligible and it is zero-knowledge if $|C|$ is polynomially bounded". The Swiss post e-voting performs only one round and considers that 1/q is negligible.
Since we consider only one round, the rest of the conditions are irrelevant. (Section 6.1 and Theorem 3).

5.3.5 ExtractVCC

In the $ExtractVCC$ algorithm[30], the voting server extracts the short Vote Cast Return Code from the Return Codes Mapping table CMtable. The voting server then stores the Verification card ID and the encrypted vote in the list of confirmed vote. The whole algorithm is shown on Figure 5.6.

From the documentation[16], the algorithm receives as input:

- long Vote Cast Return Code shares: received from the previous algorithm.
- Verification card ID
- Return Codes Mapping table
- Encrypted vote: encrypted in $CreateVote$ algorithm during the SendingVote phase.

The implementation also takes as input the Election Event ID and the Tenant ID.

The aim of the algorithm is to output a Vector of short Vote Cast Return Codes to allow the voter to check that the short Vote Cast Return Code id on their voting client is identical to the short Vote Cast Return Code printed on their voting card.

First, the long Vote Cast Return Code shares are multiplied. This corresponds to the multiplication of the hashed Confirmation key exponentiated with the Voter Vote Cast secret key. The result is then hashed and used to retrieve the codes from the CMtable.
The computation's result must be equal to the one used to construct the CMtable. Let's focus on the computation when generating the CMtable in order to compare.
In the $GenEncLongShares$ algorithms during the configuration phase 5.2.1, the Voter Vote Cast secret key is derived similarly as in the previous algorithm $CreateLVCCShare$. The matrix of exponentiated, encrypted, hashed Confirmation Keys is then computed using the Voter Vote Cast secret key. Using this matrix, the algorithm $CombineEncLongCodeShares$ computes the

[30] https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/voting-server/vote-verification/src/main/java/ch/post/it/evoting/votingserver/voteverification/service/ExtractVCCService.java

Algorithm 5.10 ExtractVCC

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\mathsf{ee} \in (\mathbb{A}_{Base16})^{1_{\mathrm{ID}}}$
Character length of the Base64 encoded hash output 1_{HB64} ▷ 44 for a 256-bit hash function
Voting Server's list of confirmed votes $\mathsf{L}_{\mathtt{confirmedVotes}}$

Input:

CCR long Vote Cast Return Code shares $(\mathtt{lVCC}_{\mathtt{id},1}, \mathtt{lVCC}_{\mathtt{id},2}, \mathtt{lVCC}_{\mathtt{id},3}, \mathtt{lVCC}_{\mathtt{id},4}) \in {\mathbb{G}_q}^4$
Verification card ID $\mathtt{vc}_{\mathtt{id}} \in (\mathbb{A}_{Base16})^{1_{\mathrm{ID}}}$
Return Codes Mapping table $\mathsf{CMtable} \in \left({\mathbb{A}_{Base64}}^{1_{\mathrm{HB64}}} \times {\mathbb{A}_{Base64}}^{*}\right)^{\mathtt{N_E}\cdot(n+1)}$
Encrypted vote $\mathtt{E1}_{\mathtt{id}} \in \mathbb{G}_q \times \mathbb{G}_q$

Operation:

1: $\mathtt{pVCC}_{\mathtt{id}} \leftarrow \prod_{j=1}^{4} \mathtt{lVCC}_{\mathtt{id},j} \mod p$
2: $\mathtt{lVCC}_{\mathtt{id}} \leftarrow \mathsf{RecursiveHash}(\mathtt{pVCC}_{\mathtt{id}}, \mathtt{vc}_{\mathtt{id}}, \mathsf{ee})$ ▷ See crypto primitives specification
3: **if** $\mathsf{Base64Encode}(\mathsf{RecursiveHash}(\mathtt{lVCC}_{\mathtt{id}})) \notin \mathsf{CMtable}$ **then**
4: **return** $\perp$
5: **else**
6: Look up $\mathsf{Base64Encode}(\mathsf{RecursiveHash}(\mathtt{lVCC}_{\mathtt{id}}))$ and retrieve $\mathtt{ctVCC}_{\mathtt{id}}$ in CMtable
7: $\mathtt{skvcc}_{\mathtt{id}} \leftarrow \mathsf{KeyDerivationFunction}(\mathtt{lVCC}_{\mathtt{id}}, 16)$ ▷ See Algorithm 8.7
8: $\mathtt{VCC}_{\mathtt{id}} \leftarrow \mathsf{GetMessageSymmetric}(\mathtt{ctVCC}_{\mathtt{id}}, \mathtt{skvcc}_{\mathtt{id}})$ ▷ See Algorithm 8.10
9: $\mathtt{VCC}_{\mathtt{id}} \leftarrow \mathsf{ByteArrayToString}(\mathtt{VCC}_{\mathtt{id}})$ ▷ See crypto primitives specification
10: $\mathsf{L}_{\mathtt{confirmedVotes}} \leftarrow \mathsf{L}_{\mathtt{confirmedVotes}} \cup (\mathtt{vc}_{\mathtt{id}}, \mathtt{E1}_{\mathtt{id}})$
11: **end if**

Output:

Short Vote Cast Return Code $\mathtt{VCC}_{\mathtt{id}} \in (\mathbb{A}_{10})^8$

Figure 5.6: ExtractVCC algorithm
[16]

vector of encrypted pre-Vote Cast Return Codes which corresponds to the multiplication of the exponentiated, encrypted, hashed Confirmation Keys for each eligible voter. Finally, this is decrypted by the $GenCMTable$ algorithm and hashed. This results in the multiplication of the exponentiated, hashed Confirmation Keys.
In conclusion, the computation is different but leads to the same result: the long Vote Cast Return Code.

This last computation is hashed and allows to retrieve from CMtable potentials for the right tenantId, electionEventId, verificationCardId. This is done the exact same way as when the potentials were pushed inside the CMtable.

However, the entries of the table are symmetrically encrypted using AES 128 in Galois/Counter Mode (GCM) mode. The implementation uses the javax GCMParameterSpec class [31]. For this reason, a new key is derived from the long Vote Cast Return Code applying the mask generation function MGF1 according to RFC8017. This key allows the decryption of the information retrieved from the table which corresponds to the Short Vote Cast Return Code. During the $GenCMTable$ algorithm, the input (Short Vote Cast Return Code) was indeed encrypted using the same key derived from the long Vote Cast Return Code.

In conclusion, the protocol performs the right computation to confirm a vote and there are no security breaches. To start with, only the knowledge of all keys allows the confirmation of

[31] https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/GCMParameterSpec.html

the votes. Indeed, in order to recover the right Short Vote Cast Return Code, the secret keys are needed, which are known only by the Control Component and at least one of them is trustworthy. This means that it won't share the keys to help an attacker or try to modify the protocol. Then, as explained in section 3.2, the print office is trustworthy and it is the voter's job to compare the returned code and to make sure that they keep secret the personal code received on the printing sheet. It is finally up to the voter to make sure of the comparison to detect any attack that would have been detected by a modified output.

Considering the implementation, there is nothing to report. Every check needed is correctly performed, each access to the database is performed, java prepared statements for SQL eliminate most risks of SQL injections, and the code follows the protocol specifications.

5.4 Tally phase

During the tally phase, the voting server and the mixing control component decrypt the votes and compute the election result running two algorithms: $MixOnline$[32] and$MixOffline$[33]. They also ensure that all the parties executed the protocol faithfully running $VerifyOnlineTally$ and $VerifyOfflineTally$. More details about the tally phase can be found in subsection 3.3.3.

During the tally phase, the Swiss post e-voting system uses the Pedersen scheme and the Bayer-Groth mixnets. For this reason we give an overview of them in the next two subsections.

5.4.1 Pedersen scheme

A cryptographic commitment allows a party to commit to a secret value and to keep it hidden from others to then reveal it later. A commitment scheme must be binding and hiding, these properties are explained later.

In a non-interactive commitment scheme, the commitment is computed and made public or it is sent. Later in order to reveal the value, the value is sent along with the randomness used when creating the commitment. With these parameters, anyone can run the commitment protocol to check that indeed this was the value committed to.

The Swiss post e-voting system uses the Pedersen commitment scheme[12] with a commitment key $ck = (h, g_1, ..., g_v)$ generated in a verifiable manner.

The Pedersen commitment scheme satisfies three properties that the Bayer-Groth mix net

[32]https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/control-components/distributed-mixing-service/src/main/java/ch/post/it/evoting/controlcomponents/mixing/service/MixDecryptOnlineService.java

[33]https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/blob/master/secure-data-manager/secure-data-manager-backend/services/src/main/java/ch/post/it/evoting/sdm/application/service/MixDecryptService.java

requires. The scheme is perfectly hiding, computationally binding, and homomorphic.
Perfectly hiding means that one cannot retrieve the committed value before it is revealed or guess it. Indeed, the commitment is uniformly distributed in G_q.
Computationally binding indicates that the committed value cannot be modified after the commitment. This is ensured by the fact that it is computationally infeasible to find two different values producing the same commitment in a setting where the discrete logarithm problem is hard and when the commitment keys are generated independently and verifiably at random.
Finally, as the scheme is homomorphic, it holds that GetCommitment(a + b, r + s) = GetCommitment(a, r)*GetCommitment(b, s) for messages a and b, commitment key ck and random values r and s.

The Swiss post e-voting system instantiates the Pedersen commitment scheme over the group of quadratic residues $Q_p \subset Z_p$. Since p and q are large primes, quadratic residuosity implies p = $2q$ + 1. The Pedersen commitment scheme works with the generators $G_1, ..., G_n \in Q_p$. This allows the system to set up parameters where the discrete logarithm problem is hard and to ensure the computationally binding property holds.

There is an important generalization of the Pedersen commitment scheme that makes it possible to commit to multiple values at once[6]. The public key consists of $m+1$ group elements $\gamma_1, ..., \gamma_m, h$ and we compute a commitment to $m = (m_1, ..., m_m) \in Z_p$ as $c = h^t \prod_{i=1}^{m} \gamma_i^{m_i}$ using the randomness $t \in Z_q$.

5.4.2 Bayer-Groth mixnet

Verifiable mix nets are used in modern e-voting schemes as they hide the relationship between encrypted votes possibly linked to a voter, and decrypted votes[7]. A re-encryption mix net is made of a sequence of mixers where each mixer receives a list of ciphertext and outputs a list containing the shuffled and re-encrypted ciphertexts. However, as the ciphertexts are re-encrypted, it is not possible to verify directly whether the shuffle and encryption operations were done correctly or not. Thus, we need to compute a zero-knowledge argument that makes it possible to verify that the shuffle was done correctly and proves its knowledge, but without revealing anything about the permutation and the randomizers used. The verifier can then verify the proofs and guarantee that no mixer added, deleted, or modified a vote.

Bayer-Groth[1] propose an honest verifier zero-knowledge argument for the correctness of a shuffle of homomorphic encryptions. It constructs a shuffle of $C_1, ..., C_N$ by selecting a permutation π and randomizers $\rho_1, ...\rho_N$, and calculating $C'_1 = C_{\pi(1)} E_p k(1; \rho_1), ..., C'_N = C_{\pi(N)} E_p k(1; \rho_N)$.

5.4.3 MixDecOnline and MixDecOffline

The MixDecOnline and MixDecOffline algorithms are composed of two main functions, $GenVerifiableShuffle$ and $GenVerifiableDecryptions$.
In order to understand the shuffling combined with re-encryption and partial decryption of the votes, it is important to understand how the election key is derived. We have seen that each mixing control component generates a CCM election key pair $(EL_{pk,j} = g^{EL_{sk,j}}, EL_{sk,j})$. The election public key EL_{pk} is then derived by computing the product of all CCM elections public keys: $\prod_{j=1}^{4} EL_{pk,j} \pmod p = \prod_{j=1}^{4} g^{EL_{sk,j}} p = g^{EL_{sk,1}+EL_{sk,2}+EL_{sk,3}+EL_{sk,4}} \pmod p$.
One by one, each control component shuffles and re-encrypts the ciphertexts using the election public key, and consecutively partially decrypts them using its personal election secret key. Each time a control component is finished with running the algorithm, it removes its part from the election public key. For example, for the thirst control components, the election public key is $g^{EL_{sk,3}+EL_{sk,4}} \pmod p$.
The $MixDecOnline$ algorithm can be found on Figure 5.7, and $MixDecOffline$ only differs by its last step as it corresponds to the last decryption outputing the plaintexts.

Algorithm 6.2 MixDecOnline$_j$

Context:
- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$
- Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{1..32}$
- Ballot box ID $\mathtt{bb} \in (\mathbb{A}_{Base16})^{1..32}$
- Control component index $j \in [1, 3]$
- Number of allowed write-ins + 1 for this specific ballot box $\hat{\delta} \in \mathbb{N}^*$
- List of shuffled and decrypted ballot boxes $\mathtt{L}_{\mathtt{bb},j}$

Input:
- Partially decrypted votes $\mathbf{c}_{\mathtt{dec},j-1} \in (\mathbb{H}_l)^{\mathtt{N_c}}$
- Remaining election public key $\overline{\mathtt{EL}}_{\mathtt{pk},j-1} \in \mathbb{G}_q^{\delta}$ ▷ $= \mathtt{EL}_{\mathtt{pk}}$, if $j = 1$
- CCM$_j$ election key pair $(\mathtt{EL}_{\mathtt{pk},j}, \mathtt{EL}_{\mathtt{sk},j}) \in \mathbb{G}_q^{\mu} \times \mathbb{Z}_q^{\mu}$

Ensure: $\mathtt{N_c} \geq 1$ ▷ The algorithm runs with at least one vote
Ensure: $l = \hat{\delta}$
Ensure: $0 < l \leq \delta \leq \mu$
Ensure: $\mathtt{bb} \notin \mathtt{L}_{\mathtt{bb},j}$

Operation:
1: $\mathbf{i}_{\mathtt{aux}} \leftarrow (\mathtt{ee}, \mathtt{bb}, \text{"MixDecOnline"}, \mathsf{IntegerToString}(j))$
2: **if** $\mathtt{N_c} > 1$ **then** ▷ Shuffling requires at least 2 votes
3: $(\mathbf{c}_{\mathtt{mix},j}, \pi_{\mathtt{mix},j}) \leftarrow \mathsf{GenVerifiableShuffle}(\mathbf{c}_{\mathtt{dec},j-1}, \overline{\mathtt{EL}}_{\mathtt{pk},j-1})$ ▷ See crypto primitives specification
4: $(\mathbf{c}_{\mathtt{dec},j}, \boldsymbol{\pi}_{\mathtt{dec},j}) \leftarrow \mathsf{GenVerifiableDecryptions}(\mathbf{c}_{\mathtt{mix},j}, (\mathtt{EL}_{\mathtt{pk},j}, \mathtt{EL}_{\mathtt{sk},j}), \mathbf{i}_{\mathtt{aux}})$ ▷ See crypto primitives specification
5: **else** ▷ If there is only 1 vote in the ballot box
6: $(\mathbf{c}_{\mathtt{dec},j}, \boldsymbol{\pi}_{\mathtt{dec},j}) \leftarrow \mathsf{GenVerifiableDecryptions}(\mathbf{c}_{\mathtt{dec},j-1}, (\mathtt{EL}_{\mathtt{pk},j}, \mathtt{EL}_{\mathtt{sk},j}), \mathbf{i}_{\mathtt{aux}})$
7: **end if**
8: $\mathtt{EL}'_{\mathtt{pk},j} \leftarrow \mathsf{CompressPublicKey}(\mathtt{EL}_{\mathtt{pk},j}, \delta)$ ▷ See crypto primitives specification
9: $\overline{\mathtt{EL}}_{\mathtt{pk},j} \leftarrow \frac{\overline{\mathtt{EL}}_{\mathtt{pk},j-1}}{\mathtt{EL}'_{\mathtt{pk},j}} \mod p$
10: $\mathtt{L}_{\mathtt{bb},j} \leftarrow \mathtt{L}_{\mathtt{bb},j} \cup \mathtt{bb}$

Output:
- Shuffled votes $\mathbf{c}_{\mathtt{mix},j} \in (\mathbb{H}_l^{\mathtt{N_c}})$
- Shuffle proof $\pi_{\mathtt{mix},j}$ ▷ See the domain of the Shuffle proof. Empty if $\mathtt{N_c} = 1$.
- Partially decrypted votes $\mathbf{c}_{\mathtt{dec},j} \in (\mathbb{H}_l)^{\mathtt{N_c}}$
- Decryption proofs $\boldsymbol{\pi}_{\mathtt{dec},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q^l)^{\mathtt{N_c}}$
- Remaining election public key $\overline{\mathtt{EL}}_{\mathtt{pk},j} \in \mathbb{G}_q^{\delta}$

Figure 5.7: MixDecOnline algorithm
[16]

We notice that the algorithm accepts the case where only one vote is submitted to decryption (line 5 on Figure 5.7). This could be considered as a security issue as this case does not preserve the voter anonymity. Indeed, as there is only one vote, no shuffling can be performed to break the link between the ciphertexts and the plaintexts and thus the voter. This opens up the possibility of linking a voter to their vote without any interference. It should be enforced to receive at least two ciphertexts to ensure 2-anonymity and have only 50% of chance of linking the voter to their vote.
Thankfully, as the verifier is trustworthy, it is not possible for an active attacker to try to reduce the list of ciphertexts in order to use this condition as an oracle to ask for the decryption of a specific ciphertext. As during the shuffling (and decryption) proof, the original vector of unshuffled ciphertexts is used as input, a difference between the lists' sizes would be directly detected by verifier.
This is a special case not likely to happen as during a vote there is a very low chance of having only one voter. However, for the sake of security it should still be modified or they should acknowledge that voter anonymity is not ensured in this case.

5.4.4 GenVerifiableShuffle

In the $GenVerifiableShuffle$ algorithm[34] shown on Figure 5.8, a control component shuffles and re-encrypts the received ciphertexts. It also provides a Bayer-Groth proof of the shuffle. The

Algorithm 5.1 `GenVerifiableShuffle`: Shuffle (including re-encryption), and provide a Bayer-Groth proof of the shuffle

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

Input:
A vector of ciphertexts $\mathbf{C} \in (\mathbb{H}_l)^N$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$ ▷ This public key is passed as context to all sub-arguments

Require: $0 < l \leq k$
Require: $2 \leq N \leq q - 3$

Operation:
1: $(\mathbf{C}', \pi, \mathbf{r}) \leftarrow \mathsf{GenShuffle}(\mathbf{C}, \mathbf{pk})$ ▷ See algorithm 5.3
2: $(m, n) \leftarrow \mathsf{GetMatrixDimensions}(N)$ ▷ See algorithm 5.5
3: $\mathbf{ck} \leftarrow \mathsf{GetVerifiableCommitmentKey}(n)$ ▷ See algorithm 5.6 ▷ This commitment key is passed as context to all sub-arguments
4: $\mathsf{shuffleStatement} \leftarrow (\mathbf{C}, \mathbf{C}')$
5: $\mathsf{shuffleWitness} \leftarrow (\pi, \mathbf{r})$
6: $\mathsf{shuffleArgument} \leftarrow \mathsf{GetShuffleArgument}(\mathsf{shuffleStatement}, \mathsf{shuffleWitness}, m, n)$ ▷ See algorithm 5.11

Output:
$\mathbf{C}' \in (\mathbb{H}_l)^N$
shuffleArgument

Figure 5.8: GenVerifiableShuffle algorithm [13]

[34] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/MixnetService.java

algorithm first calls the $GenShuffle$ function[35] with the list of ciphertexts and the multi-recipient public keys as inputs.

GenShuffle

The function shuffles and re-encrypts a list of ciphertexts $C_1, ..., C_N$ with the given multi-recipient public key pk. The given multi-recipient public key corresponds to the election key.
Hence, the first control component re-encrypts the list of ciphertexts using the election public key. Then, each next control component needs to re-encrypt the list of ciphertexts using the remaining election public key. As explained previously, the remaining public key corresponds to the election public key without its part coming from control components which already decrypted the ciphertexts.
Specifically, the re-encryption of ciphertexts $C_1, ...C_N$ works as follows: it selects a permutation π and randomizers $\rho_1, ...\rho_N$, and computes $C'_1 = C_{\pi(1)}E_{EL_{pk}}(1;\rho_1), ..., C'_N = C_{\pi(N)}E_{EL_{pk}}(1;\rho_N)$. For each i, this corresponds to $C'_i = C_{\pi(i)} * EL^{\rho_i}_{pk}$, which in turn can be written as $C'_i = C_{\pi(i)} * g^{(sk1_{\pi(i)}+sk2_{\pi(i)}+sk3_{\pi(i)}+sk4_{\pi(i)})*\rho_i}$. Let's write the sum of the different secret keys ski from all control components as sk. Considering that $C_{\pi(i)} = M_{\pi(i)} * g^{sk_{\pi(i)}\rho'_{\pi(i)}}$ for a previously generated randomness ρ', one can finally write: $C'_i = M_{\pi(i)} * g^{sk_{\pi(i)}\rho'_{\pi(i)}} * g^{sk_{\pi(i)}*\rho_i} = M_{\pi(i)} * g^{sk_{\pi(i)}(\rho'_{\pi(i)}+\rho_i)}$.
Let's notice that this corresponds to the homomorphic encryption $E_{pk}(1 * M_i; \rho_i + \rho'_i) = E_{pk}(1;\rho'_i) * E_{pk}(M_i;\rho_i)$. In this case we use the sum of all the election secret keys and this corresponds to the election used for the first control component. However for each next control component i, the secret key corresponding to the control components $j < i$ are removed from the election key.

The function uses the randomness service and also a permutation generator, the $GenPermutation$ function[36]. This function generates a permutation of integers [0, N) with N corresponding in this case to the size of the list of ciphertexts. The Swiss post e-voting system implements the Fisher-Yates shuffle which allows to generate a permutation of N items uniformly at random without retries[37]. For each element, this algorithm randomly chooses a next element in the list to swap the current element with. The function is implemented correctly, and checks that N is positive. The system only implements one loop more than is necessary, as the element resulting in the last position can only be swapped with itself, but this has no consequence.

The multiplication of the ciphertexts is carefully done. They don't forget to also multiply the $\gamma = generator^{randomness}$ that is used for decryption. This corresponds to adding the two exponents equal to the different randomness generated for both encryptions.

[35] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/ShuffleService.java

[36] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/PermutationService.java

[37] https://en.wikipedia.org/wiki/Random$_{permutation}$

Overall, the implementation is correct. The necessary checks on the inputs are done and there are no security threat.

After this, the $GenVerifiableShuffle$ algorithm calls the $GetMatrixDimensions$ function.

GetMatrixDimensions

The function[38] takes the size of the list of received ciphertexts and computes the size-optimal number of rows and columns for creating a matrix. It computes size-optimal dimensions which are as close as possible to the dimensions of a square matrix so as to result in the smallest size of the shuffle argument and be the most efficient. The algorithm computes $\lfloor\sqrt{N}\rfloor$, and decrements its value until it finds an integer that divides N. This gives m the number of rows, and allows to compute $n = N/m$ the number of columns. This indeed gives the size-optimal dimensions and the implementation is correct.

After the matrix dimensions, the $GenVerifiableShuffle$ algorithm calls the $GetVerifiableCommitmentKey$ function and creates the $ShuffleStatement$ and $ShuffleWitness$.

GetVerifiableCommitmentKey

A commitment key represents a public key used for the calculation of a commitment. The $GetVerifiableCommitmentKey$ algorithm[39] creates a commitment key with n number of elements, where n is the number of columns in the matrix dimensions previously derived, to later be able to commit to each column. The key must be part of the quadratic residue group G_q. Due to the generation of the parameters p, q and g (see 5.1.1), every element of the quadratic residue group is a generator. The algorithm ensures that it is an element of the group by squaring the potential element of the commitment key. The implementation guarantees that no trivial element (0, 1, g) is chosen, and knowing q, the generation is deterministic to enable every party to derive the commitment key. The implementation also ensures that commitment keys are generated independently and verifiably at random. On conclusion the generation of the key meets the requirement explained in subsection 5.4.1.

[38] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/MatrixUtils.java

[39] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/CommitmentKeyService.java

ShuffleStatement

The shuffle statement contains the list of the ciphertext and the list of newly shuffled and re-encrypted ciphertexts which uses the permutation and randomness generated.

ShuffleWitness

The shuffle witness contains the permutations and the randomness used for the re-encryption of the ciphertexts. These are secret parameters, for which their knowledge must be proven without revealing them.

Finally, the $GenVerifiableShuffle$ algorithm computes a cryptographic argument for the validity of the shuffle calling $ShuffleArgument$ and using the shuffle statement and witness.

ShuffleArgument

The shuffle argument is computed calling the $GetShuffleArgument$ function[40] displayed on Figure 5.9. The algorithm follows the shuffle argument's generation (without verification) from the Bayer-Groth paper found on Figure 5.10. It computes a cryptographic argument of knowledge of the permutation and the randomness used to generate the new list of ciphertexts in $GenShuffle$. Precisely, the shuffle argument combines the multi-exponentiation argument and the product argument. They respectively prove that the product of a set of ciphertexts raised to a set of committed exponents gives a specific ciphertext, and that a set of committed values has a specific product.

The first step for the prover is to commit to the permutation. This is done by committing to $\pi(1), ..., \pi(N)$ using the Pedersen scheme in the Swiss post e-voting system's implementation. To follow, the prover receives a challenge x and commits to $x^{\pi(1)}, ..., x^{\pi(N)}$. In the implementation, every permutation $\pi(i)$ and $x^{\pi(i)}$ is placed in a matrix of the dimensions previously computed, and we compute a commitment for each column[41].
To check that the same permutation has been used in both commitments, two random challenges y and z are sent to the prover.
By using the homomorphic properties of the Pedersen scheme the prover, can commit to $d_1 - z = y\pi(1) + x^{\pi(1)}, ..., d_N - z = y\pi(N) + x^{\pi(N)} - z$ and using the argument from the section 5 of the Bayer-Groth paper, the prover shows $\sum_{i=1}^{N}(d_i - z) = \sum_{i=1}^{N}(yi + x^i - z)$. The implementation calls the $GetProductArgument$ function, which is the only one that was not checked during this project.

[40] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/ShuffleArgumentService.java

[41] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/CommitmentService.java

Algorithm 5.11 GetShuffleArgument: compute a cryptographic argument for the validity of the shuffle

Context:

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

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

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

A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

A commitment key $\mathbf{ck} = (h, g_1, ..., g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of

- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$ **s.t.** $0 < l \leq k$
- The shuffled and re-encrypted list of ciphertexts $\vec{C'} \in (\mathbb{H}_l)^N$

The witness composed of

- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q{}^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$

The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$ **s.t.** $2 \leq n \leq \nu$

Ensure: $\forall i \in [0, N) : \vec{C'_i} = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk}), \vec{C}_{\pi(i)})$

Ensure: $N = mn$

Operation:

1: $\mathbf{r} \leftarrow \mathsf{GenRandomVector}(q, m)$ ▷ See algorithm 3.2

2: $A \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{\pi(i)\}_{i=0}^{N-1}, m, n))$ ▷ Create a $n \times m$ matrix. See algorithm 5.14 and algorithm 5.13

3: $\mathbf{c}_A \leftarrow \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 5.8

4: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A))$

5: $\mathbf{s} \leftarrow \mathsf{GenRandomVector}(q, m)$

6: $\mathbf{b} \leftarrow \{x^{\pi(i)}\}_{i=0}^{N-1}$

7: $B \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\mathbf{b}, m, n))$

8: $\mathbf{c}_B \leftarrow \mathsf{GetCommitmentMatrix}(B, \mathbf{s}, \mathbf{ck})$

9: $y \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A))$

10: $z \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\text{"1"}, \mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A))$

▷ Both $\vec{C}$ and $\vec{C'}$ are passed in the vector forms here

11: $\mathsf{Zneg} \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{-z\}_{i=1}^{N}, m, n))$ ▷ Vector of length N, with all values being $q - z$

12: $\mathbf{c}_{-z} \leftarrow \mathsf{GetCommitmentMatrix}(\mathsf{Zneg}, \vec{0}, \mathbf{ck})$ ▷ A vector of length m, with all 0 values

13: $\mathbf{c}_D \leftarrow \mathbf{c}_A^y \mathbf{c}_B$ ▷ Entry-wise product

14: $D \leftarrow yA + B$

15: $\mathbf{t} \leftarrow y\mathbf{r} + \mathbf{s}$

16: $b \leftarrow \prod_{i=0}^{N-1}(yi + x^i - z)$

17: $\mathsf{pStatement} \leftarrow (\mathbf{c}_D \mathbf{c}_{-z}, b)$

18: $\mathsf{pWitness} \leftarrow (D + \mathsf{Zneg}, \mathbf{t})$

19: $\mathsf{productArgument} \leftarrow \mathsf{GetProductArgument}(\mathsf{pStatement}, \mathsf{pWitness})$ ▷ See algorithm 5.18

20: $\rho \leftarrow q - (\vec{\rho} \cdot \mathbf{b})$ ▷ Standard inner product $\sum_{i=0}^{N-1} \rho_i b_i$

21: $\vec{x} \leftarrow \{x^i\}_{i=0}^{N-1}$

22: $C \leftarrow \mathsf{GetCiphertextVectorExponentiation}(\vec{C}, \vec{x})$ ▷ See algorithm 4.9

23: $\mathsf{mStatement} \leftarrow (\mathsf{ToMatrix}(\vec{C'}, m, n), C, \mathbf{c}_B)$ ▷ See algorithm 5.13

24: $\mathsf{mWitness} \leftarrow (B, \mathbf{s}, \rho)$

25: $\mathsf{multiExponentiationArgument} \leftarrow \mathsf{GetMultiExponentiationArgument}(\mathsf{mStatement}, \mathsf{mWitness})$ ▷ See algorithm 5.15

Output:

$\mathsf{shuffleArgument}\ (\mathbf{c}_A, \mathbf{c}_B, \mathsf{productArgument}, \mathsf{multiExponentiationArgument}) \in \mathbb{G}_q{}^m \times \mathbb{G}_q{}^m \times ... \times ...$

▷ See algorithm 5.18 and algorithm 5.15 for their respective domains

Figure 5.9: GetShuffleArgument algorithm [16]

Common reference string: pk, ck.
Statement: $\boldsymbol{C}, \boldsymbol{C'} \in \mathbb{H}^N$ with $N = mn$.
Prover's witness: $\pi \in \Sigma_N$ and $\boldsymbol{\rho} \in \mathbb{Z}_q^N$ such that $\boldsymbol{C'} = \mathcal{E}_{pk}(\mathbf{1}; \boldsymbol{\rho})\boldsymbol{C}_\pi$.
Initial message: Pick $\boldsymbol{r} \leftarrow \mathbb{Z}_q^m$, set $\boldsymbol{a} = \{\pi(i)\}_{i=1}^N$ and compute $\boldsymbol{c}_A = \mathrm{com}_{ck}(\boldsymbol{a}; \boldsymbol{r})$.
Send: $\boldsymbol{c}_A$
Challenge: $x \leftarrow \mathbb{Z}_q^*$.
Answer Pick $\boldsymbol{s} \in \mathbb{Z}_q^m$, set $\boldsymbol{b} = \{x^{\pi(i)}\}_{i=1}^N$ and compute $\boldsymbol{c}_B = \mathrm{com}_{ck}(\boldsymbol{b}; \boldsymbol{s})$.
Send: $\boldsymbol{c}_B$
Challenge: $y, z \leftarrow \mathbb{Z}_q^*$.
Answer: Define $\boldsymbol{c}_{-z} = \mathrm{com}_{ck}(-z, \ldots, -z; \mathbf{0})$ and $\boldsymbol{c}_D = \boldsymbol{c}_A^y \boldsymbol{c}_B$. Compute $\boldsymbol{d} = y\boldsymbol{a} + \boldsymbol{b}$, and $\boldsymbol{t} = y\boldsymbol{r} + \boldsymbol{s}$. Engage in a product argument as described in Sect. 5 of openings $d_1 - z, \ldots, d_N - z$ and $\boldsymbol{t}$ such that

$$\boldsymbol{c}_D \boldsymbol{c}_{-z} = \mathrm{com}_{ck}(\boldsymbol{d} - \boldsymbol{z}; \boldsymbol{t}) \qquad \text{and} \qquad \prod_{i=1}^{N}(d_i - z) = \prod_{i=1}^{N}(yi + x^i - z)\ .$$

Compute $\rho = -\boldsymbol{\rho} \cdot \boldsymbol{b}$ and set $\boldsymbol{x} = (x, x^2, \ldots, x^N)^T$. Engage in a multi-exponentiation argument as described in Sect. 4 of $\boldsymbol{b}$, $\boldsymbol{s}$ and ρ such that

$$\boldsymbol{C}^{\boldsymbol{x}} = \mathcal{E}_{pk}(1; \rho)\boldsymbol{C'}^{\boldsymbol{b}} \qquad \text{and} \qquad \boldsymbol{c}_B = \mathrm{com}_{ck}(\boldsymbol{b}; \boldsymbol{s})$$

The two arguments can be run in parallel. Furthermore, the multi-exponentiation argument can be started in round 3 after the computation of the commitments $\boldsymbol{c}_B$.
Verification: The verifier checks $\boldsymbol{c}_A, \boldsymbol{c}_B \in \mathbb{G}^m$ and computes $\boldsymbol{c}_{-z}, \boldsymbol{c}_D$ as described above and computes $\prod_{i=1}^{N}(yi + x^i - z)$ and $\boldsymbol{C}^{\boldsymbol{x}}$. The verifier accepts if the product and multi-exponentiation arguments both are valid.

Figure 5.10: Bayer-groth shuffle argument computation [1]

It uses a derived pStatement representing the statement used for the calculation of a product argument, computable with the knowledge of the verifier. It also uses a derived pWitness representing a witness for a product argument, containing computations using the secret values we committed to and thus non-reproductible by an outside party.
This demonstrates that the same permutation has been used in both cases and means that the prover has a commitment to $x_1, ..., x_N$ permuted in an order that was fixed before the prover saw x.

Finally, the algorithm computes an mStatement, the statement for the multi exponent argument, and an mWitness, the witness for the multi exponentiation argument. Using the multi-exponentiation argument from section 4 of Bayer-Groth paper, the prover demonstrates that there exist randomizers $\rho_1, ..., \rho_N$ such that $\prod_{i=1}^{N} C_i^{x^i} = E_{pk}(1; \rho) \prod_{i=1}^{N} C_i'^{x^{\pi(i)}}$. In the implementation this part is computed by a function called $GetMultiExponentiationArgument$[42] which implements the protocol on Figure 5.11 from the Bayer-Grother paper.

However, as we are in a non-interactive setting, every time a challenge is supposedly sent by the verifier, the challenge is actually derived from all the environment variables, inputs of the function, and values supposedly previously sent to the challenger. When two different challenges are needed in a row with no computation added in between, an integer string is prepended to differentiate them and avoid replay.

At the end of the $GenVerifiableShuffle$ function, the control component outputs the Shuf-

[42] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/MultiExponentiationArgumentService.java

Common reference string: pk, ck.
Statement: $\boldsymbol{C}_1, \ldots, \boldsymbol{C}_m \in \mathbb{H}^n$, $C \in \mathbb{H}$, and $\boldsymbol{c}_A \in \mathbb{G}^m$
Prover's witness: $A = \{\boldsymbol{a}_j\}_{j=1}^m \in \mathbb{Z}_q^{n \times m}$, $\boldsymbol{r} \in \mathbb{Z}_q^m$, and $\rho \in \mathbb{Z}_q$ such that

$$C = \mathcal{E}_{pk}(1; \rho) \prod_{i=1}^{m} \boldsymbol{C}_i^{\boldsymbol{a}_i} \qquad \text{and} \qquad \boldsymbol{c}_A = \text{com}_{ck}(A; \boldsymbol{r})$$

Initial message: Pick $\boldsymbol{a}_0 \leftarrow \mathbb{Z}_q^n, r_0 \leftarrow \mathbb{Z}_q$, and $b_0, s_0, \tau_0 \ldots, b_{2m-1}, s_{2m-1}, \tau_{2m-1} \leftarrow \mathbb{Z}_q$ and set $b_m = 0, s_m = 0, \tau_m = \rho$. Compute for $k = 0, \ldots, 2m-1$

$$c_{A_0} = \text{com}_{ck}(\boldsymbol{a}_0; r_0) \; , \; c_{B_k} = \text{com}_{ck}(b_k; s_k) \; , \; E_k = \mathcal{E}_{pk}(G^{b_k}; \tau_k) \prod_{\substack{i=1, j=0 \\ j=(k-m)+i}}^{m,m} \boldsymbol{C}_i^{\boldsymbol{a}_j}$$

Send: $c_{A_0}, \{c_{B_k}\}_{k=0}^{2m-1}, \{E_k\}_{k=0}^{2m-1}$.
Challenge: $x \leftarrow \mathbb{Z}_q^*$.
Answer: Set $\boldsymbol{x} = (x, x^2, \ldots, x^m)^T$ and compute

$$\boldsymbol{a} = \boldsymbol{a}_0 + A\boldsymbol{x} \qquad r = r_0 + \boldsymbol{r} \cdot \boldsymbol{x} \qquad b = b_0 + \sum_{k=1}^{2m-1} b_k x^k$$
$$s = s_0 + \sum_{k=1}^{2m-1} s_k x^k \qquad \tau = \tau_0 + \sum_{k=1}^{2m-1} \tau_k x^k \; .$$

Send: $\boldsymbol{a}, r, b, s, \tau$.
Verification: Check $c_{A_0}, c_{B_0}, \ldots, c_{B_{2m-1}} \in \mathbb{G}$, and $E_0, \ldots, E_{2m-1} \in \mathbb{H}$, and $\boldsymbol{a} \in \mathbb{Z}_q^n$, and $r, b, s, \tau \in \mathbb{Z}_q$, and accept if $c_{B_m} = \text{com}_{ck}(0; 0)$ and $E_m = C$, and

$$c_{A_0} \boldsymbol{c}_A^{\boldsymbol{x}} = \text{com}_{ck}(\boldsymbol{a}; r) \qquad c_{B_0} \prod_{k=1}^{2m-1} c_{B_k}^{x^k} = \text{com}_{ck}(b; \boldsymbol{s})$$
$$E_0 \prod_{k=1}^{2m-1} E_k^{x^k} = \mathcal{E}_{pk}(G^b; \tau) \prod_{i=1}^{m} \boldsymbol{C}_i^{x^{m-i}\boldsymbol{a}} \; .$$

Figure 5.11: Bayer-Groth multi exponentiation argument computation [1]

fleArgument and the new list of shuffled and re-encrypted ciphertexts. The new list of ciphertexts is given to the next control component to run the same algorithm and in turn partially decrypt the list. However, if this was the last control component running the algorithm offline, the final output is the list of plaintexts representing the votes.

As is shown on Figure 5.8, the implementation of the $GenVerifiableShuffle$ algorithms calls a few different functions. They were all verified and these functions compute the intended output with a correct implementation regarding the methdology chapter.

5.4.5 GenVerifiableDecryptions

The $GenVerifiableDecryptions$[43] function provides a verifiable partial decryption of a list of ciphertexts.

GetPartialDecryption

This function takes as input the list of ciphertexts previously shuffled and re-encrypted. This list was encrypted using the election key computed using the product of all CCM election public keys of the control component which haven not run the $MixDecOnline$ algorithm yet. The $GetPartialDecryption$ decrypts the list of ciphertexts according to the ElGamal scheme (same scheme used for encryption), using as input its election secret key.
Informally, using its secret key, the control component removes from the encryption of the vote its contribution. The vote is not encrypted anymore using its secret key and can be decrypted by the next control components.
To understand in details what happens, we continue the example from 5.4.4. $GenShuffle$ output a new list of ciphertexts such that $C'_i = M_{\pi(i)} * g^{sk_{\pi(i)}(\rho'_{\pi(i)}+\rho_i)} = M_{\pi(i)} * g^{(sk1_{\pi(i)}+sk2_{\pi(i)}+sk3_{\pi(i)}+sk4_{\pi(i)})*(\rho'_{\pi(i)}+\rho_i)}$ (in the case of the first control component). The partial decryption uses the election secret key $sk1$ and computes $C''_i = C'_i * \gamma^{-sk1} = C'_i * g^{(\rho'_{\pi(i)}+\rho_i)*(-sk1)} = M_{\pi(i)} * g^{(sk1_{\pi(i)}+sk2_{\pi(i)}+sk3_{\pi(i)}+sk4_{\pi(i)})*(\rho'_{\pi(i)}+\rho_i)} * g^{(\rho'_{\pi(i)}+\rho_i)*(-sk1)} = M_{\pi(i)} * g^{(sk2_{\pi(i)}+sk3_{\pi(i)}+sk4_{\pi(i)})*(\rho'_{\pi(i)}+\rho_i)}$ for $i = 1, ..., N$. This results in the vote not being encrypted anymore relative to the first control component.

The implementation uses the ElGamal decryption scheme from section 5.1.1, and there is nothing to notice, everything is implemented correctly.

[43] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/zeroknowledgeproofs/ZeroKnowledgeProofService.java

GenDecryptionProof

The $GenDecryptionProof$ algorithm computes a proof of validity for the fresh decryption using a scheme derived from the Schnorr protocol in a non-interactive setting[44].

On Figure 5.12 and 5.13, one can find the $GenDecryptionProof$ algorithm and its main function $ComputePhiDecryption$.

Algorithm 6.2 GenDecryptionProof: Generate a proof of validity for the provided decryption

Context:

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

Input:

A multi-recipient ciphertext $\mathbf{C} = (\gamma, \phi_0, \dots, \phi_{l-1}) \in \mathbb{H}_l$
A multi-recipient key pair $(\mathbf{pk}, \mathbf{sk}) \in \mathbb{G}_q^k \times \mathbb{Z}_q^k$
A multi-recipient message $\mathbf{m} = (m_0, \dots, m_{l-1}) \in {\mathbb{G}_q}^l$ s.t. $\mathbf{m} = \mathsf{GetMessage}(\mathbf{C}, \mathbf{sk})$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in ({\mathbb{A}_{UCS}}^*)^*$

Require: $0 < l \leq k$

Operation:

1: $\mathbf{b} \leftarrow \mathsf{GenRandomVector}(q, l)$ ▷ See algorithm 3.2
2: $\mathbf{c} \leftarrow \mathsf{ComputePhiDecryption}(\mathbf{b}, \gamma)$ ▷ See algorithm 6.1
3: $\mathbf{f} \leftarrow (p, q, g, \gamma)$
4: $(\mathsf{pk}'_0, \dots, \mathsf{pk}'_{l-1}) \leftarrow \mathsf{CompressPublicKey}(\mathbf{pk}, l)$ ▷ See algorithm 4.5
5: **for** $i \in [0, l)$ **do**
6: $\quad y_i \leftarrow \mathsf{pk}'_i$
7: $\quad y_{l+i} \leftarrow \frac{\phi_i}{m_i}$
8: **end for**
9: $\mathbf{h}_{\mathsf{aux}} \leftarrow (\texttt{"DecryptionProof"}, (\phi_0, \dots, \phi_{l-1}), \mathbf{m}, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
10: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, (y_0, \dots, y_{2l-1}), \mathbf{c}, \mathbf{h}_{\mathsf{aux}}))$ ▷ See algorithms 2.8 and 3.6
11: $\mathbf{sk}' \leftarrow \mathsf{CompressSecretKey}(\mathbf{sk}, l)$ ▷ See algorithm 4.6
12: $\mathbf{z} \leftarrow \mathbf{b} + e \cdot \mathbf{sk}'$

Output:

Proof $(e, \mathbf{z}) \in \mathbb{Z}_q \times {\mathbb{Z}_q}^l$

Figure 5.12: GenDecryptionProof algorithm [16]

The $ComputePhiDecryption$ function[45] computes a list of g^{b_i} and γ^{b_i}. γ represents the generator g exponentiated to the randomness used for encryption, and b_i are elements of a random vector of the same size as the list of ciphertexts.The phi function maps the election secret key, for which we need to prove the knowledge, to its public key and the decryption of the ciphertext. Indeed, if the vector b corresponds to the election secret key, this list is equal to the list of election public keys and the list of election public keys exponentiated to the randomness used for encryption.
These are actually computed next in the $GenDecryptionProof$.
As we are in a non-interactive setting, these results are not sent to the challenger but hashed to

[44] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/zeroknowledgeproofs/DecryptionProofService.java
[45] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/zeroknowledgeproofs/DecryptionProofService.java

Algorithm 6.1 ComputePhiDecryption: Compute the ϕ-function for decryption

Context:

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

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

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

Input:

Preimage $(x_0, \ldots, x_{l-1}) \in {\mathbb{Z}_q}^l$

Base $\gamma \in \mathbb{G}_q$

Operation:

1: **for** $i \in [0, l)$ **do**

2: $y_i \leftarrow g^{x_i}$ ▷ $y_i = \mathsf{pk}_i$ when $x_i = \mathsf{sk}_i$

3: $y_{l+i} \leftarrow \gamma^{x_i}$ ▷ $y_{l+i} = g^{\mathsf{sk}_i \cdot r} = \frac{\phi_i}{m_i}$ when $\gamma = g^r$ and $x_i = \mathsf{sk}_i$

4: **end for**

▷ All symbols used in the comments above are aligned with algorithms 4.4 and 4.7

Output:

The image $(y_0, \ldots, y_{2l-1}) \in {\mathbb{G}_q}^{2l}$

This algorithm implies that for the multi-recipient ElGamal key pair $(\mathbf{pk}, \mathbf{sk})$ and the valid decryption $m = (m_0, \ldots, m_{l-1})$ of the ciphertext $(\gamma, \phi_0, \ldots, \phi_{l-1})$, the computation of the ComputePhiDecryption$(\mathbf{sk}, \gamma)$ would yield $(\mathsf{pk}_0, \ldots, \mathsf{pk}_{l-1}, \frac{\phi_0}{m_0}, \ldots, \frac{\phi_{l-1}}{m_{l-1}})$.

Figure 5.13: ComputePhiDecryption algorithm
[16]

the variable e.
Finally, the function computes $z = b + e * sk$ where sk is the election secret key used for encryption, and in this case the witness.
This computation is explained in more detail during the verification. Briefly, the elements b and sk are unknown to an outside party, but the public key g^{sk} is known as well as e which is part of the output with z, and it is possible to prove knowledge of b by computing $g^z = g^{b+e*sk} = g^b * (g^{sk})^e$.

The implementation of $GenDecryptionproof$ is correct, as well as that of all the different functions it calls. Again, all the necessary checks are performed (cross-checks, inputs) and the java best practises are followed.

Regarding the security of the system, as the implementation is correct, it relies theoretically on the scheme used.

5.4.6 VerifyOnlineTally and VerifyOfflineTally

In the $VerifyOnlineTall$ and $VerifyOfflineTally$ algorithms, the proof of the decryption and the shuffling are verified and run in the verifier[46] [47].

[46] https://gitlab.com/swisspost-evoting/verifier/verifier/-/blob/master/verifier-block3/src/main/java/ch/post/it/evoting/verifier/block/block3/verifications/VerifyOnlineDecryptionProofs.java

[47] https://gitlab.com/swisspost-evoting/verifier/verifier/-/blob/master/verifier-block4/src/main/java/ch/post/it/evoting/verifier/block/block4/verifications/VerifyOfflineDecryptionProofs.java

5.4.7 VerifyShuffle

The $VerifyShuffle$ algorithm[48] shown on Figure 5.14 verifies the correctness of the shuffle argument computed in the $GetShuffleArgument$ algorithm for the list of ciphertexts and their shuffled and re-encrypted list. It computes the matrix dimensions the same way as in the

Algorithm 5.2 `VerifyShuffle`: Verify the output of a previously generated verifiable shuffle

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

Input:
A vector of unshuffled ciphertexts $\mathbf{C} \in (\mathbb{H}_l)^N$
A vector of shuffled, re-encrypted ciphertexts $\mathbf{C'} \in (\mathbb{H}_l)^N$
A Bayer-Groth shuffleArgument ▷ See algorithm 5.11 for the domain
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$ ▷ This public key is passed as context to all sub-arguments

Require: $0 < l \leq k$
Require: $2 \leq N \leq q - 3$

Operation:
1: $(m, n) \leftarrow$ GetMatrixDimensions(N) ▷ See algorithm 5.5
2: $\mathbf{ck} \leftarrow$ GetVerifiableCommitmentKey(n) ▷ See algorithm 5.6 ▷ This commitment key is passed as context to all sub-arguments
3: shuffleStatement $\leftarrow (\mathbf{C}, \mathbf{C'})$
4: **return** VerifyShuffleArgument(shuffleStatement, shuffleArgument, m, n) ▷ See algorithm 5.12

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

Figure 5.14: VerifyShuffle algorithm
[16]

generation of the proof and derives the deterministic commitment key. Finally, it calls the $VerifyShuffleArgument$ function using as inputs the list ciphertexts, the list of shuffled and re-encrypted ciphertexts, and the shuffleArgument computed during the proof.

The $VerifyShuffleArgument$ function first re-computes the challenges x, y, z. Then, it derives the part of the proof computable without the knowledge of witness: the $pStatement$ and $mStatement$, as during the generation of the proof. Now, instead of generating the proof it verifies them using two functions: $VerifyProductArgument$ and $VerifyMultiExponentiationArgument$.
$VerifyProductArgument$ takes as input the $productArgument$ generated during the proof and verifies the equality stated during the generation using the $pStatement$. Again, as its sibling in the proof this function was not checked.
$VerifyMultiExponentiationArgument$ verifies the $multiExponentiationProof$ using the $mStatement$. It computes the verification part of the protocol on Figure 5.11.

The whole function corresponds to the Verification part explained on Figure 5.10 and is

[48] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/mixnet/MixnetService.java

correctly implemented.

5.4.8 VerifyDecryptions

The function $VerifyDecryptions$[49] receives for each ciphertext its proof composed of the pair (e, z) computed in 5.4.5. For each cipherext, it verifies the proof.

First, it applies the $ComputePhiDecryption$ function to z to construct a list x. This computes a list of $g^z = g^{b+e*sk} = g^b * (g^{sk})^e$, and $g^{z*\rho}$.

Then, as during the generation of the proof, the algorithm constructs a list y containing the list of election public keys g^{sk} prepended to the list of election public keys exponentiated to the randomness used for encryption $g^{sk*\rho}$.

Next, it multiplies both list x and y, with y raised to the exponent e. For the first l elements this is equal to : $[g^z] * [y^{-e}] = [g^{b+e*sk}] * [y^{-e}] = [g^b * g^{e*sk}] * [(g^{sk})^{-e}] = [g^b * g^{e*sk'}] * [(g^{-e*sk}] = g^b$. For the next l elements this results in: $[g^{\rho*b+\rho*e*sk}] * [g^{sk} * \rho]^{-e} = [g^{\rho*b} * g^{\rho*e*sk}] * [g^{sk} * \rho]^{-e} = g^{\rho*b}$. To sum up, if the prover was honest, this computation retrieves the output (g^b, $g^{\rho*b}$) of the function $ComputePhiDecryption$ with input b, without knowing the value of b.

At this point, the verifier has all elements to compute the hash e' with the same inputs as during $GenDecryptionProof$.

The trustworthy verifier compares the newly generated e' to the received e. If everything was done correctly, they are equal.

No error were found in the implementation. As the previous algorithm, the security relies on the scheme used for the commitment and re-encryption. The Pedersen commitment scheme and the Bayer-Groth mixnet are currently considered as secure, and consequently is this part of Swiss post e-voting system theoretically.

[49] https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/src/main/java/ch/post/it/evoting/cryptoprimitives/zeroknowledgeproofs/DecryptionProofService.java

Chapter 6

Conclusion

In this project, we covered some specific part of the Swiss post e-voting system and analyzed their implementation and the theory behind them.

After reviewing all parts covered in the analysis section, one doubt remains about the potential oracle implemented in the $MixDecOnline$ and $MixDecOffline$ functions and mentioned in section 5.4.3. This would deserve more attention. Either it needs to be modified, or they need to make clear that they allow this case which removes some anonymity in a particular situation. Otherwise, overall no security issue was found.

There are a few inconsistencies between the documentation and the theory, but with no security impact. In some functions, Java best practises are not always respected, however the majority of the code does follow them and it is known that the implementation will be improved in this sense.

Overall, the system is still under development and some future work is planned[1]. Through the work done in this project, we have a high level of confidence on the security of the Swiss post e-voting system.

[1] https://gitlab.com/swisspost-evoting/e-voting/e-voting

Bibliography

[1] Stephanie Bayer and Jens Groth. “Efficient Zero-Knowledge Argument for Correctness of a Shuffle”. In: *EUROCRYPT 2012, LNCS 7237.*. 2012, pp. 263–280.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. “Keying Hash Functions for Message Authentication”. In: *Advances in Cryptology - CRYPTO ’96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Vol. 1109. Springer, 1996.

[3] Amos Fiat and Ad Shamir. “How To Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *Advances in Cryptology — CRYPTO’ 86. Lecture Notes in Computer Science*. 1987.

[4] Taher El Gamal. “A public key cryptosystem and a signature scheme based on discrete logarithms.” In: *G. R. Blakley and David Chaum, editors, CRYPTO, volume 196 of Lecture Notes in Computer Science*. Springer, 1984, pp. 10–18.

[5] NHenri Gilbert and Helena Handschuh. “Security Analysis of SHA-256 and Sisters.” In: *Cryptography 2003*. 2003, pp. 175–193.

[6] Jens Groth. “Homomorphic Trapdoor Commitments to Group Elements”. In: *IACR Cryptol. ePrint Arch.* 2009 (2009), p. 7.

[7] Thomas Haines, Rajeev Gore, and Bhavesh Sharma. *Did you mix me? Formally Verifying Verifiable Mix Nets in Electronic Voting*. Cryptology ePrint Archive, Report 2020/1114. 2020.

[8] RSA Laboratories. “PKCS 1 v2.1: RSA Cryptography Standard”. In: 2020.

[9] Derrick Lehmer. “Tests for primality by the converse of Fermat’s theorem”. In: *Bulletin of the American Mathematical Society* 33.3 (1927), pp. 327–340.

[10] Philipp Locher, Rolf Haenni, and Reto E. Koenig. “Analysis of the Cryptographic Implementation of the Swiss Post Voting Protocol”. In: July 19, 2019.

[11] Ueli Maurer. “Unifying Zero-Knowledge Proofs of Knowledge”. In: *AFRICACRYPT 2009, LNCS 5580*. 2009, pp. 272–286.

[12] Torben Pryds Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Advances in Cryptology - CRYPT0 ’91, LNCS 576*. 1992, pp. 129–140.

[13] Swiss Post. “Cryptographic Primitives of the Swiss Post Voting System - Version 0.9.11”. In: *E-voting documentation*. 2021.

[14] Swiss Post. “Infrastructure whitepaper of the Swiss Post e-voting system”. In: *E-voting documentation*. 2021.

[15] Swiss Post. “Protocol of the Swiss Post Voting System - Version 0.9.11”. In: *E-voting documentation*. 2021.

[16] Swiss Post. “Swiss Post Voting System - System specification Version 0.9.7”. In: *E-voting documentation*. 2021.

[17] Swiss Post. “SwissPost Voting System architecture document - Version 0.9.1”. In: *E-voting documentation*. 2021.

[18] Michael O Rabin. “Probabilistic algorithm for testing primality”. In: *Journal of Number Theory* 12.1 (1980), pp. 128–138.

[19] National Institute of Standards and Technology. “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions”. In: *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION*. 2015.

[20] Yiannis Tsiounis and Moti Yung. “ On the security of elgamal based encryption.” In: *Hideki Imai and Yuliang Zheng, editors, Public Key Cryptography, volume 1431 of Lecture Notes in Computer Science*. Springer, 1998, pp. 117–134.