b
UNIVERSITÄT
BERN

E-Voting Verifier

for the Swiss Post Voting System

Bachelor Thesis

Marc Günter
from
Ruswil LU, Switzerland

Faculty of Science, University of Bern

25. April 2023

Prof. Christian Cachin
Dr. Patrick Liniger
Mariarosaria Barbaraci
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

A verifier is used to validate an e-voting system. It conducts a series of tests on election data published by the e-voting system on a bulletin board. Collectively, the validation of the election data provides sufficient evidence to assess the correctness of the election results [15]. This thesis builds on the work of Dr. Patrick Liniger, who began developing a verifier for the Swiss Post Voting System in the Summer of 2022. The Swiss Post has made available a verifier specification [6], which serves as a template for developing a verifier for the Swiss Post Voting System. The main contribution of this work encompasses the implementation of five algorithms for Liniger's verifier. A thorough understanding of cryptographic protocols, such as the Elgamal encryption scheme and zero-knowledge proofs, was essential to implement these algorithms effectively.

Acknowledgements

I want to express my gratitude to Dr. Patrick Liniger and Mariarosaria Barbaraci for their supervision of my thesis. Their guidance and feedback were greatly appreciated. Furthermore, I would like to thank Prof. Christian Cachin for discussing possible thesis topics with me and setting me up with an exciting topic about e-voting.

Contents

Chapter 1

Introduction

1.1 E-voting

E-voting, or electronic voting, involves the use of electronic devices in the voting process. While the term is often associated with internet voting, where voters use an electronic device to cast their vote via the internet, it can also encompass the use of machines to scan and count physical ballots.

It is important to note that e-voting can be something other than a fully electronic solution. Voting instructions may still be sent via traditional mail, and e-voting may complement rather than completely replace traditional voting methods such as in-person voting or voting by mail [13].

For an e-voting system to be widely accepted, it must address various key considerations. According to Smyth [22], there are three fundamental properties that a state-of-the-art e-voting system should possess:

- **Individual verifiability**: A voter can check whether the e-voting system registered their vote.
- **Universal verifiability**: Anyone can check whether an election outcome corresponds to the registered votes.
- **Vote secrecy**: A voter's vote is not revealed to anyone.

1.2 (E-)Voting in Switzerland

Switzerland is known for its strong tradition of democracy and the direct participation of citizens in the political process. One of the hallmarks of the Swiss political system is the frequency of popular votes at the federal, cantonal, and municipal levels. The country holds more popular votes per capita than any other country.

Over one-third of all referendums ever held at a national level took place in Switzerland [16], a country with a population of just over 9 million. This level of direct citizen participation in the political process is unique worldwide, and it is one of the reasons that Switzerland is often cited as a model of democratic governance.

In an effort to modernize and digitalize governmental processes, the Swiss Government has embraced e-voting as part of its *E-Government* strategy. The *Vote électronique* project [3] began in the year 2000. Since 2004, fifteen cantons have conducted over three-hundred trials of e-voting, only allowing a small portion of the electorate to participate and following the principle of "security before speed" [4]. The trials often included Swiss citizens living abroad, who benefit greatly from e-voting as it ensures the timely arrival of their votes in Switzerland. Often votes from the Swiss abroad are not counted because their letters only arrive after the election has already happened [17].

Other advantages of e-voting include improved accessibility for people with disabilities and the elimination of invalid votes [6]. In some elections, a substantial part of the ballots is invalid. For example, in the Canton of Obwalden, there has been an election where as many as ten percent of the votes were invalid because some voters did not follow the voting instructions properly [24].

In Switzerland, e-voting is still seen as a complementary option rather than a replacement for traditional voting methods. Eligible voters will receive their voting instructions by mail and have the choice to cast their vote electronically, by mail, or as has been done for generations, in-person.

1.3 Swiss Post Voting System

The Swiss Post introduced an e-voting system that underwent testing in a few cantons for the first time in 2016. In 2019, the Swiss Post made the system's source code available to the public and invited hackers to participate in a public intrusion test. Unfortunately, the examination revealed severe flaws in the system, causing it to be canceled for the federal vote in May 2019 [2].

In response to these flaws, the Swiss Post decided to pause the testing of the system in elections from July 2019 [2]. Despite this, the development of the Swiss Post Voting System has continued.

In 2022, a second public intrusion test took place from August 8 to September 2 [18]. More than 3,400 hackers worldwide attempted over 600,000 attacks on the system and rated their findings on a low, medium, high, or critical severity scale.

The results of the second public intrusion test were encouraging, showing that the Voting System is now more secure than it was three years ago. No significant flaws were detected. The Swiss Post received only two low-severity findings from the hackers, one of which they confirmed to be a problem. Because of this success, the Swiss Post intended to make its improved Voting System available for use by interested cantons in 2023.

In March 2023, the Federal Council announced that three cantons have received regulatory approval to test the updated system until May 2025 [19]. For starters, around 1,2% of the Swiss electorate will thus have the chance to vote electronically, most of them being Swiss citizens living abroad.

1.4 Role of the Verifier in the voting process

E-voting systems need external and independent verifiers to be considered universally verifiable. Additional verifiers, apart from the one developed by the Swiss Post, published on the project's GitLab repository, need to be developed. If only the Swiss Post's verifier were to be used, the entity capable of manipulating election results would be in charge of verifying election results.

Verifiers conduct tests on the election data which the e-voting system has published on a private or public bulletin board. Collectively, these validations provide sufficient evidence to assess the correctness of the election results [15].

In Switzerland, the Federal Chancellery outlines the requirements for e-voting and holds the cantons responsible for appointing auditors who ensure that the election is conducted correctly. Auditors utilize verifiers as a technical tool to carry out their checks [9], and they guarantee that the election is conducted orderly and give voters peace of mind knowing that irregularities, if any, will be detected and addressed. While anyone can create a verifier, it is possible that only the designated auditors will be granted access to the election data by a canton.

Chapter 2

Zero-Knowledge Proofs

The chapter begins by introducing zero-knowledge proofs of statements and knowledge, both interactive and non-interactive. Then, we delve into the specifics of a generic non-interactive zero-knowledge proof that forms the basis for the proofs used in the Swiss Post Voting System and demonstrate how it meets the criteria for completeness, soundness, and zero-knowledge.

2.1 Introduction

2.1.1 Zero-Knowledge Proof

A *zero-nowledge proof* (ZKP) is a cryptographic concept first introduced in 1985 by Goldwasser, Micali, and Rackoff [14] and is defined as a proof ”that conveys no additional knowledge other than the correctness of the proposition in question”. It is common to differentiate between two kinds of ZKPs:

- **Zero-knowledge proof of statement**: A proof showing that a statement is true without revealing any additional information beyond the statement's truth.
  Example: Given a graph G, one can construct a proof, showing that the statement ”there exists a Hamiltonian circuit in G” is true, without revealing the circuit to others.

- **Zero-knowledge proof of knowledge**: A proof showing that someone possesses knowledge of a secret without revealing any information about it.
  Example: Given $n \in \mathbb{N}$, one can construct a proof, showing that one knows $p, q \in \mathbb{P}$ such that $n = p \cdot q$, without revealing the prime numbers to others.

The party constructing the proof is referred to as the prover P, while the one validating the proof is called the verifier V. In general, the following steps occur in an interaction between a prover and a verifier:

- P and V agree on a statement that P wants to prove or a secret that P wants to demonstrate knowledge of.

- P commits to some information that corresponds to the statement or secret but does not reveal the actual information.

- V challenges P to prove their claim by issuing a random challenge.

- P responds to the challenge by computing a response based on their initial commitment and the challenge.

- V checks whether the response is correct without gaining additional information beyond the statement's validity or knowledge of the secret. If the response is correct, V is convinced of the validity of the statement or that P knows the secret.

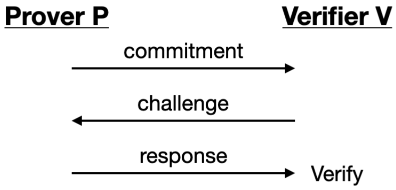

Figure 2.1. Interactions in a zero-knowledge proof of knowledge

A tuple consisting of a commitment, challenge, and response is referred to as a transcript of a zero-knowledge proof protocol. A zero-knowledge proof satisfies the following criteria:

- **Completeness**: An honest verifier always returns true if presented with a valid proof.
- **Soundness** (proofs of statement): A prover cannot convince a verifier of a false statement except with a small probability called the soundness error.
- **Special Soundness** (proofs of knowledge): An efficient algorithm K, referred to as knowledge extractor, can extract the knowledge if given two accepting transcripts.
- **Zero-knowledge**: A verifier does not learn additional information beyond the statement's validity or anything about the secret.

2.1.2 Non-Interactive Zero-Knowledge Proof

Non-interactive zero-knowledge proofs were introduced in 1987 by De Santis, Micali, and Persiano [20]. They came up with a concept where a prover can generate a proof that a verifier can validate without the need for any further interaction. This makes NIZKPs particularly useful in scenarios where the parties may not have a reliable or efficient channel for interactive communication, such as in some types of distributed systems, or in scenarios where the proof needs to be stored or transferred over a network.
ZKPs are turned non-interactive by using the Fiat-Shamir transform [11], where the existence of a random oracle is assumed. The random oracle chooses a random output for every input. If an input is given twice, the same output is produced again. In general, the following steps occur in a non-interactive zero-knowledge proof [12]:

- P and V agree on a statement that P wants to prove or a secret that P wants to demonstrate knowledge of.
- P commits to some information that corresponds to the statement or secret but does not reveal the actual information. P then passes the commitment as the input to a random oracle and receives an output in return, which replaces the challenge from the interactive ZKP.
- P generates a response to the output based on their initial commitment and the output (challenge) from the random oracle. P then makes the commitment and response available.
- V can, at some point, pass the commitment to the random oracle and use its output (challenge) to verify if P's response is valid, thus validating the proof.

Random oracles cannot be efficiently represented and, therefore, cannot exist in the real world. Hash functions are used in place of random oracles to achieve the same result [12].

2.2 NIZKP in Swiss Post Voting System

In this section, we will introduce a generic non-interactive zero-knowledge proof as outlined in the Swiss Post's Cryptographic Primitives specification [5]. This generic non-interactive zero-knowledge proof serves as the foundation for the zero-knowledge proofs used in the Swiss Post Voting System. The Swiss Post defines the hash function *RecursiveHash* in the same specification. Typically, a standard hash function is used. A statement, consisting of a homomorphism $\phi : \mathbb{G}_1 \mapsto \mathbb{G}_2$ between two algebraic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ and an image $y = \phi(w) \in \mathbb{G}_2$, is formed. The goal is to provide a proof of knowledge of the pre-image $w \in \mathbb{G}_1$ while keeping w a secret.

A prover performs the following steps:

- draw $b \in \mathbb{G}_1$ at random
- compute commitment $c = \phi(b)$
- compute challenge $e = RecursiveHash(\phi, y, c, auxiliaryData)$ by hashing the statement, commitment, and some other public data.
- compute response $z = b \star w^e$ (where $\star$ is the group operation for $\mathbb{G}_1$, and exponentiation is the repetition of that operation)
- output proof $\pi = (e, z)$

A verifier validates the proof as follows:

- compute $x = \phi(z)$
- compute challenge $c' = x \otimes y^{-e}$ (where $\otimes$ is the group operation for $\mathbb{G}_2$, and exponentiation is the repetition of that operation)
- if and only if $RecursiveHash(\phi, y, c', auxiliaryData) = e$, the proof is valid

The zero-knowledge proof must satisfy the following criteria:

Completeness

Proof: A verifier checks if $RecursiveHash(\phi, y, c', auxiliaryData) = e$ is true. Looking at the definition of e, it becomes clear that $c' = c$ must be satisfied for the condition to be true.

$$\begin{aligned} c' &= x \otimes y^{-e} \\ &= \phi(z) \otimes \phi(w)^{-e} \\ &= \phi(b \star w^e) \otimes \phi(w)^{-e} \\ &= \phi(b) \otimes \phi(w^e) \otimes \phi(w)^{-e} \\ &= c \otimes \phi(w)^e \otimes \phi(w)^{-e} \\ &= c \otimes 1 \\ &= c \quad \square \end{aligned}$$

Special Soundness

Proof: Given two accepting transcripts $\pi = (c, e, z)$ and $\pi' = (c, e', z')$ with $e \neq e'$, a knowledge extractor K extracts the witness w as follows.

$$\begin{aligned} c = \phi(b) = \phi(z \star \overline{w^e}) &= \phi(z' \star \overline{w^{e'}}) = \phi(b) = c \\ \iff \quad & z \star \overline{w^e} = z' \star \overline{w^{e'}} \\ \iff \quad & z \star \overline{w}^e = z' \star w^{\overline{e'}} \\ \iff \quad & z \star \overline{z'} = w^{e \star \overline{e'}} \\ \iff \quad & (z \star \overline{z'})^{\overline{e} \star e'} = w \quad \square \end{aligned}$$

Zero-knowledge

Proof: An accepting transcript is simulated.

- $c \leftarrow^R \mathbb{G}_2$
- $e \leftarrow^R \mathbb{G}_1$
- $z \leftarrow^R \mathbb{G}_1$

The simulated transcript (c, e, z) has the same distribution as an accepting transcript, meaning that someone else cannot tell the simulated one from the real one [21]. Thus the protocol is zero-knowledge.

Chapter 3

Elgamal encryption scheme and mixnets

In this chapter, we introduce the Elgamal encryption scheme and the concept of a mixnet, which are the fundamental components of a verifiable e-voting system. Additionally, we take a closer look at the mixnet used in the Swiss Post Voting System.

3.1 Multi-recipient Elgamal encryption scheme

The Swiss Post Voting System uses the Elgamal encryption scheme with its three well-known algorithms for key generation, encryption, and decryption. In addition, there is an algorithm for partially decrypting ciphertexts.

The encryption scheme is built on a group of quadratic residues $\mathbb{G}_q$ so that the discrete logarithm problem, as well as the Decisional Diffie-Hellman problem, are computationally difficult. All operations are performed modulo a prime number $p = 2q + 1$, and both p and q are large prime numbers to ensure security with $|p| = 3072$ bits and $|q| = 3071$ bits.

The Swiss Post Voting System uses a multi-recipient version of the Elgamal encryption scheme, which allows for efficient encryption and decryption of multiple messages simultaneously.

3.1.1 Key Generation

Algorithm 1 generates a multi-recipient Elgamal key pair $(sk, pk) \in (\mathbb{Z}_q^N \times \mathbb{G}_q^N), N \in \mathbb{N}^+$. The secret keys $sk_i \in \mathbb{Z}_q$ are chosen randomly using the algorithm *GenRandomInteger*. In contrast, the public keys $pk_i \in \mathbb{G}_q$ are calculated by raising the group generator g to the secret key sk_i.

Algorithm 1 GenKeyPair

```
for $i \in [0, N)$ do
    $sk_i \leftarrow$ GenRandomInteger$(q)$        ▷ see crypto primitives specification
    $pk_i \leftarrow g^{sk_i} \mod p$
end for
return $(sk, pk)$
```

3.1.2 Encryption

Algorithm 2 encrypts a multi-recipient message $m = (m_0, ..., m_{\ell-1}) \in \mathbb{G}_q^\ell$ consisting of ℓ messages with a public key $pk \in \mathbb{G}_q^N$ and produces a ciphertext $c = (\gamma, \phi_0, ..., \phi_{\ell-1}) \in \mathbb{H}_\ell$. Every message m_i is multiplied with a public key part pk_i raised to the power of a random number $r \in \mathbb{Z}_q$. Without the exponentiation with the random number r, the encryption would not be secure against chosen-plaintext attacks.

Algorithm 2 GetCiphertext

```
$r \leftarrow \mathbb{Z}_q$
$\gamma \leftarrow g^r \mod p$
for $i \in [0, \ell)$ do
    $\phi_i \leftarrow pk_i^r \cdot m_i \mod p$
end for
return $(\gamma, \phi_0, ..., \phi_{\ell-1})$
```

3.1.3 Decryption

Algorithm 3 decrypts a ciphertext $c = (\gamma, \phi_0, ..., \phi_{\ell-1}) \in \mathbb{H}_\ell$ with the secret key $sk \in \mathbb{Z}_q^N$ and returns the message $m = (m_0, ..., m_{\ell-1}) \in \mathbb{G}_q^\ell$.

Algorithm 3 GetMessage

```
for $i \in [0, \ell)$ do
    $m_i \leftarrow \phi_i \cdot \gamma^{-sk_i} \mod p$
end for
return $(m_0, ..., m_{\ell-1})$
```

3.1.4 Partial Decryption

Algorithm 4 is used for the partial decryption of the encrypted votes in the mixnet, where each mixer removes its contribution to the encryption of the votes before sending them to the next mixer.
A ciphertext $c = (\gamma, \phi_0, ..., \phi_{\ell-1}) \in \mathbb{H}_\ell$ is partially decrypted with the secret key $sk \in \mathbb{Z}_q^N$ and a ciphertext $c' = (\gamma, m_0, ..., m_{\ell-1}) \in \mathbb{H}_\ell$ is returned. In contrast to the decryption algorithm, the γ is not removed as it is essential for further partial decryption by other components.

Algorithm 4 GetPartialDecryption

```
$(m_0, ..., m_{\ell-1}) \leftarrow$ GetMessage$(c, sk)$
return $(\gamma, m_0, ..., m_{\ell-1})$
```

3.2 Mixnet

3.2.1 Introduction

In 1981, Chaum introduced the concept of a mixnet [10] to tackle the traffic analysis problem, where an adversary can learn about the communication patterns between participants in a network by analyzing its traffic.
In a mixnet, messages pass through a series of mixers where they are permuted, and the connection between senders and receivers is concealed. In electronic voting, the input to the mixnet is a list of encrypted votes linked to voters. Every mixer removes a part of the encryption from the encrypted votes. After the last decryption step by the last mixer, the e-voting system can tally the plaintext votes, all while preserving vote secrecy.
Without the permutation of the votes, an adversary would know precisely how everyone voted, as each decrypted vote i would correspond to a specific encrypted vote i.
Since Chaum's mixnet is based on symmetric cryptography, one cannot prove that the shuffling and decryption were done correctly. The Swiss Post Voting System thus uses a mixnet based on asymmetric cryptography to allow for the construction of zero-knowledge proofs.

3.2.2 Re-encryption mixnet

A re-encryption mixnet consists of n mixers. For each mixer $j \in [1, n]$, there exists a key pair consisting of a secret key and public key $(sk_j, pk_j) \in (\mathbb{Z}_q^\delta \times \mathbb{G}_q^\delta), \delta \in \mathbb{N}^+$. The product of all these public keys is denoted as the public key $\mathbf{pk} = \prod_{i=1}^n pk_i \mod p$.

The first mixer in the mixnet performs the following operations:

1. *Mixer 1* receives a list of messages encrypted with the public key $\mathbf{pk} = \prod_{i=1}^n pk_i \mod p$.
2. *Mixer 1* shuffles the messages and re-encrypts them with the public key **pk**. The mixer generates a zero-knowledge proof of shuffle π_{mix} so that the other mixers can check that no messages were added, deleted, or modified.
3. *Mixer 1* partially decrypts the messages with its secret key $sk_1 \in \mathbb{Z}_q^\delta$. It generates a zero-knowledge proof of decryption π_{dec} that shows that the partially decrypted messages match the messages before the partial decryption step. The messages are now encrypted with the public key $pk = \prod_{i=2}^n pk_i \mod p$ as the public key **pk** contribution from the first mixer has been removed.
4. *Mixer 1* sends the partially decrypted messages and zero-knowledge proofs to the next mixer.

All the other mixers, for $j = 2, ..., n$, perform the following operations:

1. *Mixer j* receives a list of partially decrypted messages from the previous mixer $j - 1$ along with the proofs of shuffle and decryption of all the $j - 1$ mixers before it.
2. *Mixer j* verifies all the proofs of shuffle and decryption from the mixers before it. If something is wrong, the process is interrupted by an honest mixer j.
3. *Mixer j* shuffles the messages and re-encrypts them with the public key $pk = \prod_{i=j}^n pk_i \mod p$. The mixer generates a zero-knowledge proof of shuffle π_{mix} so that the other mixers can check that no messages were added, deleted, or modified.
4. *Mixer j* partially decrypts the messages with its secret key $sk_j \in \mathbb{Z}_q^\delta$. It generates a zero-knowledge proof of decryption π_{dec} that shows that the partially decrypted messages match the messages before the partial decryption step. The messages are now encrypted with the public key $pk = \prod_{i=j+1}^n pk_i \mod p$ as the public key **pk** contribution from *mixer j* has been removed.
5. *Mixer j* sends the partially decrypted messages and all the zero-knowledge proofs to the next *mixer j+1*.

After the n-th mixer performs the last decryption step, the result is a list of plaintext messages that can no longer be linked to the original senders, unless all mixers collude and combine their shuffles.

3.2.3 Mixnet in the Swiss Post Voting System

The mixnet in the Swiss Post Voting System consists of five mixers. The first four are called online control components, and the last is called the Tally control component. To each of the five mixers, there belongs a key pair consisting of a secret key $sk_j \in \mathbb{Z}_q^\delta$ and a public key $pk_j \in \mathbb{G}_q^\delta$ with $\delta \in \mathbb{N}^+$ denoting the number of elements of the election public key. The plaintext votes are encrypted with the election

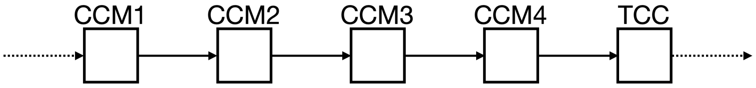

Figure 3.1. Overview of the mixnet components in the Swiss Post Voting System

public key $\mathbf{pk}_{EL} = \prod_{i=1}^{5} pk_i \mod p$.
The five mixers perform the operations described in section 3.2.2. The cantons in Switzerland are responsible for running elections meaning that each canton would have a separate instance of the Swiss Post Voting System. Not all the votes in the canton are mixed together, however. Each vote belongs to a ballot box corresponding to a municipality, and the mixing process is done separately for every ballot box.
The online control components are connected to the internet, communicating with the voting server. However, the Tally control component is run offline on the canton's premises. Its secret key sk_5 is constructed from the passwords belonging to electoral board members, while the other four components own their secret keys.
The electoral board's job is to observe the orderly processing of an election. This adds another security measure because all the members must agree that the election has proceeded orderly before the final decryption step can occur.

3.2.4 Example of procedure in re-encryption mixnet

The following showcases how a voter's vote is encrypted and how it is affected by the operations occurring in the first and last mixer.
A vote $m \in \mathbb{G}_q^l$ is encrypted with the election public key $\mathbf{pk}_{EL} \in \mathbb{G}_q^\delta$ together with a random number $r_0 \in \mathbb{Z}_q$ resulting in a ciphertext $c \in \mathbb{H}_l$.

$$\begin{aligned} c &= (\gamma, \phi) = \textbf{GetCiphertext}(m, r_0, \mathbf{pk}_{EL}) \\ &= (g^{r_0}, \mathbf{pk}_{EL}^{r_0} \cdot m) \end{aligned}$$

The voting server sends all the encrypted votes to the first mixer in the mixnet, where they are shuffled and re-encrypted. After the shuffling, every vote $c = (g^{r_0}, \mathbf{pk}_{EL}^{r_0} \cdot m)$ gets re-encrypted with the public key $\mathbf{pk} = \prod_{i=1}^{5} pk_i \mod p$ resulting in the ciphertext c_{mix}. In the case of the first mixer, this public key is the same as the election public key.

$$\begin{aligned} c_{mix} &= \textbf{GetCiphertext}(1, r_1, \mathbf{pk}) \cdot c \\ &= (g^{r_1}, \mathbf{pk}^{r_1} \cdot 1) \cdot (g^{r_0}, \mathbf{pk}_{EL}^{r_0} \cdot m) \\ &= (g^{r_1+r_0}, \mathbf{pk}_{EL}^{r_1+r_0} \cdot m) \end{aligned}$$

The first mixer removes its part of the encryption from the shuffled and re-encrypted votes. A shuffled and re-encrypted vote c_{mix} is partially decrypted with the mixer's secret key sk_1, resulting in the ciphertext c_{dec}.

$$\begin{aligned} c_{dec} &= \textbf{GetPartialDecryption}(c_{mix}, sk_1) \\ &= (g^{r_1+r_0}, \textbf{GetMessage}(c_{mix}, sk_1)) \\ &= (g^{r_1+r_0}, \underbrace{\mathbf{pk}_{EL}^{r_1+r_0} \cdot m}_{\phi} \cdot (\underbrace{g^{r_1+r_0}}_{\gamma})^{-sk_1}) \\ &= (g^{r_1+r_0}, \mathbf{pk}_{EL}^{r_1+r_0} \cdot m \cdot (g^{-sk_1})^{r_1+r_0}) \\ &= (g^{r_1+r_0}, \mathbf{pk}_{EL}^{r_1+r_0} \cdot m \cdot (pk_1^{-1})^{r_1+r_0}) \end{aligned}$$

Because pk_1 is a factor of the election public key $\mathbf{pk}_{EL}$, a shuffled, re-encrypted, and partially decrypted vote c_{dec} ends up looking like this:

$$c_{dec} = (g^{r_1+r_0}, \prod_{i=2}^{5} pk_i^{r_1+r_0} \cdot m)$$

The Tally control component, the last mixer in the mixnet, receives a list of encrypted votes c_{dec} from the previous mixer. Again every vote is shuffled and re-encrypted with the public key $\mathbf{pk} = \prod_{i=5}^{5} pk_i \mod p = pk_5$.

$$\begin{aligned} c_{mix} &= \textbf{GetCiphertext}(1, r_5, pk_5) \cdot c_{dec} \\ &= (g^{r_5}, {pk_5}^{r_5} \cdot 1) \cdot (g^{\hat{r}}, {pk_5}^{\hat{r}} \cdot m), \hat{r} := \sum_{k=0}^{4} r_k \mod p \\ &= (g^{r_5+\hat{r}}, {pk_5}^{r_5+\hat{r}} \cdot m) \end{aligned}$$

Then every shuffled and re-encrypted vote c_{mix} is partially decrypted with the Tally control components secret key sk_5.

$$\begin{aligned} c_{dec} &= \textbf{GetPartialDecryption}(c_{mix}, sk_5) \\ &= ... \\ c_{dec} &= (g^{r_5+\hat{r}}, m) \end{aligned}$$

The plaintext votes are no longer encrypted and can be tallied.

3.2.5 Bayer-Groth

The Swiss Post Voting System uses the Bayer-Groth [1] mixnet. Bayer and Groth developed an efficient zero-knowledge argument for the correctness of shuffle. The basic idea in an election with N votes is that the encrypted votes are arranged into a $n \times m$ matrix with $N = n \times m$. The prover then commits to the columns of the matrix.

Overall the proof of shuffle is quite complex and combines several zero-knowledge arguments into one. For a broad overview, one best watches Bayer's presentation of the paper given at *Eurocrypt 2012*.

Compared to another widely used verifiable mixnet of Terelius-Wikström [23], the mixnet of Bayer-Groth is more space-efficient by a factor of fifty.

Chapter 4

Implemented Algorithms

This chapter introduces the verifier for which additional algorithms were implemented. The algorithms are discussed in detail, but the pseudocode shown here is different from the one in the specifications for reasons of simplicity. For a deeper understanding of the algorithms, including the algorithms they rely on, one can refer to the system [7], crypto primitives [5], and verifier [6] specifications of the Swiss Post Voting System. All the produced Python program files are in the extra material section. The chapter finishes with a retrospective view of the working process and lessons learned.

4.1 swiss-post-voting-system package

The Swiss Post has published a verifier specification [6] on its GitLab repository for the Swiss Post Voting System. The document contains all the necessary information and pseudocode algorithms to develop a voting system verifier. Some of the required algorithms for the verifier are detailed in the system [7], and crypto primitives [5] specifications.

Dr. Patrick Liniger started developing a verifier for the Swiss Post Voting System in the Summer of 2022. It is implemented in Python, structured into a crypto-primitives, system, and verifier specification, and has test cases for many of the implemented pseudocode algorithms. Overall the code style is explicit and tries to follow the pseudocode version closely.

By the end of September 2022, Liniger had implemented most of the required algorithms from the system and crypto primitives specifications and a significant portion of the ones from the verifier specification.

The verification of the voting system is split into two phases. Weeks before the election date, the verifier validates the *SetupPhase* of the voting system to validate whether the election event has been configured correctly.

After the election, a second check is performed where the verifier validates the *TallyPhase* of the voting system to see whether the votes were mixed correctly and if the election outcome corresponds to the registered votes.

All the newly implemented algorithms belong to the validation of the *TallyPhase* because the other parts were already mostly finished.

4.2 VerifyProcessPlaintexts

Algorithm 5 verifies an operation that the Tally control component has performed in the Voting System. The Tally control component, the last mixer in the mixnet, performs the last decryption step, resulting in a list of plaintext votes that are not linkable to the original voters. It then processes the plaintext votes so that the election results can be generated.

A plaintext vote $m_i = \prod_{i=1}^{\psi} \hat{p}_i \mod p$ is the product of all selected encoded voting options a voter has selected, with each encoded voting option $\hat{p}_i$ being a different prime number. The number of voting options a voter can select is denoted by ψ. The Tally control component factorizes every plaintext vote

into its prime factors representing different voting options.
In the process, it generates a list L_{votes} containing all the selected voting options for all the voters and a list $L_{decodedVotes}$ containing all the selected plaintext voting options for all the voters.

Here is an example of the two lists in an election with three questions and three voters:

$$L_{votes} = ((5, 43, 61), (5, 37, 53), (5, 43, 73))$$
$$L_{decodedVotes} = ((Q1\_Yes, Q2\_No, Q3\_Yes), (Q1\_Yes, Q2\_Yes, Q3\_Abstain), (Q1\_Yes, Q2\_No, Q3\_No))$$

The verifier repeats this factorization procedure and checks whether the resulting lists equal those the Swiss Post Voting System provided as cryptographic evidence.

Algorithm 5 VerifyProcessPlaintexts

```
$k \leftarrow 0$
for $i \in [0, \hat{N}_C)$ do
    if $m_i \neq 1$ then
        $\hat{\mathbf{p}}'_k \leftarrow$ Factorize$(m_i, \tilde{p}, \psi)$                    ▷ see system specification
        $\hat{\mathbf{v}}'_k \leftarrow$ DecodeVotingOptions$(\hat{\mathbf{p}}'_k, pTable)$    ▷ see system specification
        $k \leftarrow k + 1$
    end if
end for
if $(\hat{\mathbf{p}}'_0, ..., \hat{\mathbf{p}}'_{N_C-1}) = L_{votes} \wedge (\hat{\mathbf{v}}'_0, ..., \hat{\mathbf{v}}'_{N_C-1}) = L_{decodedVotes}$ then
    return $\top$
else
    return $\bot$
end if
```

4.3 VerifyMixDecOffline

In algorithm 6, the verifier repeats an operation the Tally control component has performed in the Voting System. Before the Tally control component shuffles, re-encrypts, and partially decrypts the input ciphertexts, it validates all the zero-knowledge proofs generated by the four online control components before it.
If all the shuffle proofs $\{\pi_{mix,j}\}_{j=1}^{4}$ and decryption proofs $\{\pi_{dec,j}\}_{j=1}^{4}$ are valid, the Tally control component is ensured that there was no manipulation in the mixnet operations of the four online control components.
The verifier also validates all these zero-knowledge proofs and returns true if the verification is successful.

4.4 VerifyOnlineControlComponentsBallotBox

Algorithm 7 repeats the Tally control component's verification of the four online control components for a specific ballot box. First the voting client's zero-knowledge proofs $\pi_{Exp,1}$ and $\pi_{EqEnc,1}$ are verified. The algorithm *GetMixnetInitialCiphertexts* is used to access the list of encrypted votes sorted lexicographically by the verification card ids of the voters.
Then it verifies the online control component's zero-knowledge proofs of shuffle $\{\pi_{mix,j}\}_{j=1}^{4}$ and decryption $\{\pi_{dec,j}\}_{j=1}^{4}$ using the algorithm *VerifyMixDecOffline*. The algorithm returns true if the verification is successful for all the zero-knowledge proofs.

Algorithm 6 VerifyMixDecOffline

```
$shuffleVerif_1 \leftarrow$ **VerifyShuffle**($\mathbf{c}, \mathbf{c}_{mix,1}, \pi_{mix,1}, \mathbf{pk}_{EL}$)    ▷ see crypto primitives specification
$decryptVerif_1 \leftarrow$ **VerifyDecryptions**($\mathbf{c}_{mix,1}, pk_1, \mathbf{c}_{dec,1}, \pi_{dec,1}, \mathbf{i}_{aux,1}$)
                                                            ▷ see crypto primitives specification
for $j \in [2, 4]$ do
    $shuffleVerif_j \leftarrow$ **VerifyShuffle**($\mathbf{c}_{dec,j-1}, \mathbf{c}_{mix,j}, \pi_{mix,j}, \prod_{i=j}^{5} pk_i \mod p$)
                                                            ▷ see crypto primitives specification
    $decryptVerif_j \leftarrow$ **VerifyDecryptions**($\mathbf{c}_{mix,j}, pk_j, \mathbf{c}_{dec,j}, \pi_{dec,j}, \mathbf{i}_{aux,j}$)
                                                            ▷ see crypto primitives specification
end for
if $(decryptVerif_j \wedge shuffleVerif_j)\ \forall\, j \in [1, 4]$ then
    return $\top$
else
    return $\perp$
end if
```

Algorithm 7 VerifyOnlineControlComponentsBallotBox

```
if $N_C \geq 1$ then
    $vcProofsVerif \leftarrow$ **VerifyVotingClientProofs**($\mathbf{vc}_1, \mathbf{E1}_1, \widetilde{\mathbf{E1}}_1, \mathbf{E2}_1, \pi_{Exp,1}, \pi_{EqEnc,1}$,
        **KMap**, $\mathbf{pk}_{EL}, \mathbf{pk}_{CCR}$)                    ▷ see system specification
else
    $vcProofsVerif \leftarrow \top$
end if
$\mathbf{c} \leftarrow$ **GetMixnetInitialCiphertexts**$_1$($\hat{\delta}, vcMap_1, \mathbf{pk}_{EL}$)          ▷ see system specification
$shuffleProofsVerif \leftarrow$ **VerifyMixDecOffline**($\mathbf{c}, \{\mathbf{c}_{mix,j}\}_{j=1}^{4}, \{\pi_{mix,j}\}_{j=1}^{4}, \{\mathbf{c}_{dec,j}\}_{j=1}^{4}$,
    $\{\pi_{dec,j}\}_{j=1}^{4}, \mathbf{pk}_{EL}, \{pk_j\}_{j=1}^{5}$)                          ▷ see system specification
if $vcProofsVerif \wedge shuffleProofsVerif$ then
    return $\top$
else
    return $\perp$
end if
```

4.5 VerifyTallyControlComponentBallotBox

Algorithm 8 verifies the operations of the Tally control component itself. First, it verifies the Tally control component's zero-knowledge proof of shuffle $\pi_{mix,5}$ and proof of decryption $\pi_{dec,5}$.
Then it checks the Tally control component's processing of the plaintext votes m using the algorithm *VerifyProcessPlaintexts*.

Algorithm 8 VerifyTallyControlComponentBallotBox

```
$shuffleVerif \leftarrow$ VerifyShuffle($\mathbf{c}_{dec,4}, \mathbf{c}_{mix,5}, \pi_{mix,5}, pk_5$)    ▷ see crypto primitives specification
$decryptVerif \leftarrow$ VerifyDecryptions($\mathbf{c}_{mix,5}, pk_5, \mathbf{m}, \pi_{dec,5}, i_{aux}$)
                                                                        ▷ see crypto primitives specification
$processVerif \leftarrow$ VerifyProcessPlaintexts($pTable, \mathbf{m}, \psi, \hat{\delta}, L_{votes}, L_{decodedVotes}$)
                                                                        ▷ see crypto primitives specification
if $shuffleVerif \wedge decryptVerif \wedge processVerif$ then
    return $\top$
else
    return $\perp$
end if
```

4.6 VerifyOnlineControlComponents

Algorithm 9 calls the algorithm *VerifyOnlineControlComponentsBallotBox* for all the ballot boxes and returns true if the verification is successful for all ballot boxes.

Algorithm 9 VerifyOnlineControlComponents

```
for $i \in [0, N_{bb})$ do
    Prepare input for verification of ballot box
    $bbOnlineCCVerif_i \leftarrow$ VerifyOnlineControlComponentsBallotBox($input$)
                                                                        ▷ see Algorithm 7
end for
if $bbOnlineCCVerif_i\ \forall\ i$ then
    return $\top$
else
    return $\perp$
end if
```

4.7 Lessons learned

In the beginning, I delved into the important cryptographic concepts used in the Swiss Post Voting System, including the Elgamal encryption scheme, homomorphic encryption, and zero-knowledge proofs. I watched portions of the Cryptographic Protocols course by Prof. Christian Cachin and supplemented my learning with the recommended literature and exercise sheets.
Despite having no prior experience with Python, I quickly began implementing the first algorithm, *VerifyProcessPlaintexts*, thanks to my previous exposure to programming languages like Ruby and Swift, which share similarities with Python.
I employed a test-driven development approach by writing tests before implementing each algorithm. This method made sense because it was important to find the right way of representing the data for the tests, guiding how I could write the algorithms. All the test data can be found on the GitLab repository of

the Swiss Post Voting System [8], stored in JSON files. Retrieving this data and making it accessible in the Python test files proved challenging, especially because the input names in the verifier specification often differed from those in the dataset. Once I had a better understanding of the steps involved in the algorithm, it became easier to identify the correct data for the tests.

After writing the tests, implementing the algorithms from the pseudocode was relatively straightforward, as the pseudocode provided a clear and comprehensive guide for the development process.

Overall, the implementation phase of this thesis was enjoyable. Occasional bugs slowed down the coding process, but with some assistance in refactoring the code, I produced functioning algorithms and well-documented test cases.

Chapter 5

Conclusions

This thesis aimed to further develop a verifier for the Swiss Post Voting System. To achieve this goal, a thorough understanding of zero-knowledge proofs, the Elgamal encryption scheme, and mixnets was crucial.

Five additional algorithms, as described in the verifier specification [6] and system specification [7] were implemented along with the necessary tests.

Currently, most of the required algorithms and verifications have been implemented. Still, as the Swiss Post Voting System continually evolves, the verifier is expected to remain a work in progress.

It remains uncertain whether the Swiss Post Voting System will eventually become accessible to the entirety of the Swiss electorate. Other trustworthy verifiers need to be developed in case of a more paramount appearance of e-voting in Switzerland. It would be interesting to explore the implementation of verifiers in different programming languages.

Appendix A

Extra material

A.1 Algorithms

A.1.1 VerifyProcessPlaintexts

```python
# final_verification.py
def verify_process_plaintexts(
    group: Group,
    v_tilde: tuple[str, ...],
    p_tilde: tuple[int, ...],
    m: tuple[tuple[int, ...], ...],
    psi: int,
    delta_hat: int,
    l_votes: tuple[tuple[int, ...], ...],
    l_decoded_votes: tuple[tuple[str, ...], ...],
) -> bool:

    if not all(delta_hat == len(m_i) for m_i in m):
        msg = f"Requirement 'all(delta_hat == len(m_i) for m_i in m)' not met: {delta_hat=}, {m=}"
        raise ValueError(msg)

    n_c_hat = len(m)
    if not n_c_hat >= 2:
        msg = f"Requirement 'n_c_hat >= 2' not met: {n_c_hat=} elements in {m=}"
        raise ValueError(msg)

    n_c = len(l_votes)
    if not n_c == len(l_decoded_votes):
        msg = (
            "Requirement 'len(l_votes) == len(l_decoded_votes)' not met: "
            f"{n_c=} elements in l_votes, "
            f"{len(l_decoded_votes)=} elements in l_decoded_votes"
        )
        raise ValueError(msg)

    if n_c >= 2:
        if not n_c_hat == n_c:
            msg = f"Requirement 'n_c_hat == n_c' not met: {n_c_hat=}, {n_c=}"
            raise ValueError(msg)
    else:
        if not n_c_hat == n_c + 2:
            msg = f"Requirement 'n_c_hat == n_c + 2' not met :{n_c_hat=}, {n_c=}"
            raise ValueError(msg)

    p_hat_lst, v_hat_lst = [], []
    one_tuple = (1,) * delta_hat
```

```python
    for m_i in m:
        v_hat_k = []
        if m_i != one_tuple:
            p_hat_k = factorize(group=group, x=m_i[0], p_tilde=p_tilde, phi=psi)
            p_hat_lst.append(p_hat_k)
            for p in p_hat_k:
                v_hat_k.append(v_tilde[p_tilde.index(p)])
            v_hat_lst.append(tuple(v_hat_k))
    p_hat = tuple(p_hat_lst)
    v_hat = tuple(v_hat_lst)

    if p_hat != l_votes:
        LOGGER.warning(
            "verify_process_plaintexts failed because p_hat != l_votes",
        )
        return False

    if v_hat != l_decoded_votes:
        LOGGER.warning(
            "verify_process_plaintexts failed because v_hat != l_decoded_votes",
        )
        return False

    return True
```

Listing A.1. VerifyProcessPlaintexts

A.1.2 VerifyMixDecOffline

```python
# mix_offline.py
def verify_mix_dec_offline(
    group: Group,
    delta_hat: int,
    ee: str,
    ballot_box_id: str,
    c_init: tuple[MultiRecipientCiphertext, ...],
    c_mix: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    pi_mix: tuple[ShuffleArgument, ...],
    c_dec: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    pi_dec: tuple[tuple[Proofs, ...], ...],
    el_pk: ElectionPublicKey,
    ccm_el_pk: tuple[tuple[int, ...], ...],
    eb_pk: ElectionPublicKey,
) -> bool:

    # pylint: disable=too-many-locals

    n_c_hat = len(c_init)
    if not n_c_hat >= 2:
        msg = f"Requirement ’n_c_hat >= 2’ not met: {n_c_hat=} elements in {c_init=}"
        raise ValueError(msg)

    if not all(len(c_mix_j) == n_c_hat for c_mix_j in c_mix):
        msg = (
            f"Requirement ’# of ciphertexts in c_mix_j equal to # of initial ciphertexts’ not met:"
            f"{c_mix=}, {c_init=}"
        )
        raise ValueError(msg)

    if not all((len(c_dec_j) == n_c_hat for c_dec_j in c_dec)):
```

```python
        msg = (
            f"Requirement '# of ciphertexts in c_dec_j equal to # of initial ciphertexts' not met:"
            f"{c_dec=}, {c_init=}"
        )
        raise ValueError(msg)

    if not all((len(pi_dec_j) == n_c_hat for pi_dec_j in pi_dec)):
        msg = (
            f"Requirement '# of proofs in pi_dec_j equal to # of initial ciphertexts' not met:"
            f"{pi_dec=}, {c_init=}"
        )
        raise ValueError(msg)

    if not all((len(proof.z) == delta_hat for proof in pi_dec_j) for pi_dec_j in pi_dec):
        msg = f"Requirement 'l == delta_hat' not met: {delta_hat=}, {pi_dec=}"
        raise ValueError(msg)

    ell = len(pi_dec[0][0].z)
    delta = len(el_pk.public_key)
    mu = len(ccm_el_pk[0])
    if not 0 < ell <= delta <= mu:
        msg = f"Requirement '0 < l <=    <=   ' not met: {ell=},    = {delta=},    = {mu=}"
        raise ValueError(msg)

    if not delta == len(eb_pk.public_key):
        msg = (
            f"Requirement 'el_pk and eb_pk both consist of    elements' not met:"
            f"len(el_pk) = {delta=}, len(eb_pk) = {len(eb_pk.public_key)=}"
        )
        raise ValueError(msg)

    if not all(len(ccm_el_pk_j) == mu for ccm_el_pk_j in ccm_el_pk):
        msg = f"Requirement 'all ccm_el_pk_j contain    elements' not met:    = {mu=}, {ccm_el_pk=}"
        raise ValueError(msg)

    ccm_el_pk_prime = tuple(tuple(ccm_el_pl_j[:delta]) for ccm_el_pl_j in ccm_el_pk)

    verify_shuffle_0_ok = verify_shuffle(
        group=group,
        ciphertexts=c_init,
        shuffled_ciphertexts=c_mix[0],
        shuffle_argument=pi_mix[0],
        pk=el_pk.public_key,
    )

    if verify_shuffle_0_ok is False:
        LOGGER.warning(
            "verify_mix_dec_offline failed because 'verify_shuffle' failed for j=1",
            # args:
            group=group,
            delta_hat=delta_hat,
            ee=ee,
            ballot_box_id=ballot_box_id,
        )
        return False

    verify_decryptions_ok = verify_decryptions(
        group=group,
```

```
        c=c_mix[0],
        pk=ccm_el_pk_prime[0],
        c_prime=c_dec[0],
        pi_dec=pi_dec[0],
        i_aux=(ee, ballot_box_id, "MixDecOnline", "1"),
    )

    if verify_decryptions_ok is False:
        LOGGER.warning(
            "verify_mix_dec_offline failed because 'verify_decryptions' failed for j=1",
            # args:
            group=group,
            delta_hat=delta_hat,
            ee=ee,
            ballot_box_id=ballot_box_id,
        )
        return False

    for j in range(2, 5):
        el_pk_combined = combine_public_keys(
            group=group, public_keys=ccm_el_pk_prime[j - 1 :] + (eb_pk.public_key,)
        )

        verify_shuffle_j_ok = verify_shuffle(
            group=group,
            ciphertexts=c_dec[j - 2],
            shuffled_ciphertexts=c_mix[j - 1],
            shuffle_argument=pi_mix[j - 1],
            pk=el_pk_combined,
        )

        if verify_shuffle_j_ok is False:

            LOGGER.warning(
                f"verify_mix_dec_offline failed because 'verify_decryptions' failed for {j=}",
                # args:
                group=group,
                delta_hat=delta_hat,
                ee=ee,
                ballot_box_id=ballot_box_id,
            )
            return False

        decrypt_verif_j_ok = verify_decryptions(
            group=group,
            c=c_mix[j - 1],
            pk=ccm_el_pk_prime[j - 1],
            c_prime=c_dec[j - 1],
            pi_dec=pi_dec[j - 1],
            i_aux=(ee, ballot_box_id, "MixDecOnline", str(j)),
        )

        if decrypt_verif_j_ok is False:
            LOGGER.warning(
                f"verify_mix_dec_offline failed because 'verify_decryptions' failed for {j=}",
                # args:
                group=group,
                delta_hat=delta_hat,
                ee=ee,
                ballot_box_id=ballot_box_id,
```

```
            )
            return False

    return True
```

Listing A.2. VerifyMixDecOffline

A.1.3 VerifyOnlineControlComponentsBallotBox

```
# final_verification.py
def verify_online_control_components_ballot_box(
    group: Group,
    ee: str,
    ballot_box_id: str,
    psi: int,
    el_pk: ElectionPublicKey,
    ccm_el_pk: tuple[tuple[int, ...], ...],
    eb_pk: ElectionPublicKey,
    pk_bold_ccr: ChoiceReturnCodesEncryptionPublicKey,
    delta_hat: int,
    kmap: dict[str, int],
    vc_bold_1: tuple[str, ...],
    e1_bold_1: tuple[MultiRecipientCiphertext, ...],
    e1_bold_tilde_1: tuple[MultiRecipientCiphertext, ...],
    e2_bold_1: tuple[MultiRecipientCiphertext, ...],
    pi_bold_exp_1: tuple[Proof, ...],
    pi_bold_eqenc_1: tuple[Proof2, ...],
    c_mix: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    pi_mix: tuple[ShuffleArgument, ...],
    c_dec: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    pi_dec: tuple[tuple[Proofs, ...], ...],
    p_tilde: tuple[int, ...],
    v_tilde: tuple[str, ...],
) -> bool:

    if not all((len(proof.z) == delta_hat for proof in pi_dec_j) for pi_dec_j in pi_dec):
        msg = f"Requirement ’l == delta_hat’ not met: {delta_hat=}, {pi_dec=}"
        raise ValueError(msg)

    n_c_hat = len(c_mix[0])
    if not all(
        len(c_mix_j) == len(c_dec_j) == len(pi_dec_j)
        for (c_mix_j, c_dec_j, pi_dec_j) in zip(c_mix, c_dec, pi_dec)
    ):
        msg = (
            "Requirement ’c_mix_j, c_dec_j and pi_dec_j are of size n_c_hat for all j’ not met: "
            f"{n_c_hat=}, {c_mix=}, {c_dec=}, {pi_dec=}"
        )
        raise ValueError(msg)

    if not n_c_hat >= 2:
        msg = f"Requirement ’n_c_hat >= 2’ not met: {n_c_hat=}"
        raise ValueError(msg)

    n_c = len(vc_bold_1)
    if (
        not len(vc_bold_1)
        == len(e1_bold_1)
        == len(e1_bold_tilde_1)
        == len(e2_bold_1)
```

```
        == len(pi_bold_exp_1)
        == len(pi_bold_eqenc_1)
    ):
        msg = (
            "Requirement ’vc_bold_1, e1_bold_1, e1_bold_tilde_1, e2_bold_1, pi_bold_exp_1, "
            f"pi_bold_eqenc_1 contain {n_c=} elements each’ not met"
        )
        raise ValueError(msg)

    if n_c >= 2:
        if not n_c_hat == n_c:
            msg = f"Requirement ’n_c_hat == n_c if n_c >= 2’ not met: {n_c_hat=}, {n_c=}"
            raise ValueError(msg)
    else:
        if not n_c_hat == n_c + 2:
            msg = f"Requirement ’n_c_hat == n_c + 2 if n_c < 2’ not met: {n_c_hat=}, {n_c=}"
            raise ValueError(msg)

    if not len(set(vc_bold_1)) == len(vc_bold_1):
        msg = "Requirement ’all elements in vc_bold 1 are distinct’ not met"
        raise ValueError(msg)

    vc_map = {}
    for (vc, e_1) in zip(vc_bold_1, e1_bold_1):
        vc_map[vc] = e_1
    n_c = len(vc_map)

    if n_c >= 1:
        verify_voting_client_proofs_ok = verify_voting_client_proofs(
            group=group,
            vc_bold_1=vc_bold_1,
            e1_bold_1=e1_bold_1,
            e1_bold_tilde_1=e1_bold_tilde_1,
            e2_bold_1=e2_bold_1,
            pi_bold_exp_1=pi_bold_exp_1,
            pi_bold_eqenc_1=pi_bold_eqenc_1,
            kmap=kmap,
            el_pk=el_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=delta_hat,
            psi=psi,
            ee=ee,
            p_tilde=p_tilde,
            v_tilde=v_tilde,
        )
    else:
        verify_voting_client_proofs_ok = True

    if verify_voting_client_proofs_ok is False:
        LOGGER.warning(
            "verify_online_control_components_ballot_box failed because "
            "verify_voting_client_proofs failed",
        )
        return False

    verify_mix_dec_offline_ok = verify_mix_dec_offline(
        group=group,
        delta_hat=delta_hat,
        ee=ee,
        ballot_box_id=ballot_box_id,
```

```
        c_init=get_mixnet_initial_ciphertexts(
            group=group, delta_hat=delta_hat, vc_map=vc_map, el_pk=el_pk
        ),
        c_mix=c_mix,
        pi_mix=pi_mix,
        c_dec=c_dec,
        pi_dec=pi_dec,
        el_pk=el_pk,
        ccm_el_pk=ccm_el_pk,
        eb_pk=eb_pk,
    )

    if verify_mix_dec_offline_ok is False:
        LOGGER.warning(
            "verify_online_control_components_ballot_box failed because "
            "verify_mix_dec_offline failed",
        )
        return False

    return True
```

Listing A.3. VerifyOnlineControlComponentsBallotBox

A.1.4 VerifyTallyControlComponentsBallotBox

```
# final_verification.py
def verify_tally_control_component_ballot_box(
    group: Group,
    ee: str,
    ballot_box_id: str,
    eb_pk: ElectionPublicKey,
    v_tilde: tuple[str, ...],
    p_tilde: tuple[int, ...],
    psi: int,
    delta_hat: int,
    c_dec_4: tuple[MultiRecipientCiphertext, ...],
    c_mix_5: tuple[MultiRecipientCiphertext, ...],
    pi_mix_5: ShuffleArgument,
    m: tuple[tuple[int, ...], ...],
    pi_dec_5: tuple[Proofs, ...],
    l_votes: tuple[tuple[int, ...], ...],
    l_decoded_votes: tuple[tuple[str, ...], ...],
) -> bool:

    if not all(len(proof.z) == delta_hat for proof in pi_dec_5):
        msg = f"Requirement ’l == delta_hat’ not met: {delta_hat=}, {pi_dec_5=}"
        raise ValueError(msg)

    if not all(len(m_i) == delta_hat for m_i in m):
        msg = f"Requirement ’all messages in m are of size delta_hat’ not met: {delta_hat=}, {m=}"
        raise ValueError(msg)

    n_c_hat = len(c_dec_4)
    if not len(c_mix_5) == len(m) == len(pi_dec_5) == n_c_hat:
        msg = (
            "Requirement ’c_dec_4, c_mix_5, m and pi_dec_5 are of size n_c_hat for all j’ not met: "
            f"{c_dec_4=}, {n_c_hat=}, {c_mix_5=}, {c_dec_4=}, {pi_dec_5=}"
        )
        raise ValueError(msg)

```

```
    if not n_c_hat >= 2:
        msg = f"Requirement ’n_c_hat >= 2’ not met: {n_c_hat=}"
        raise ValueError(msg)

    n_c = len(l_votes)
    if not n_c == len(l_decoded_votes):
        msg = (
            "Requirement ’l_votes and l_decoded contain the same amount of elements’ not met: "
            f"{l_votes=}, {l_decoded_votes=}"
        )
        raise ValueError(msg)

    if n_c >= 2:
        if not n_c_hat == n_c:
            msg = f"Requirement ’n_c_hat == n_c if n_c >= 2’ not met: {n_c_hat=}, {n_c=}"
            raise ValueError(msg)
    else:
        if not n_c_hat == n_c + 2:
            msg = f"Requirement ’n_c_hat == n_c + 2 if n_c < 2’ not met: {n_c_hat=}, {n_c=}"
            raise ValueError(msg)

    if not all(set(p_i).issubset(p_tilde) for p_i in l_votes):
        msg = (
            "Requirement ’selected voting options are a subset of voting options’ not met: "
            f"{l_votes=}, {p_tilde=}"
        )
        raise ValueError(msg)

    if not all(len(p_i) == len(set(p_i)) for p_i in l_votes):
        msg = (
            "Requirement ’a vote’s selected encoded voting options must be distinct’ not met: "
            f"{l_votes=}"
        )
        raise ValueError(msg)

    i_aux = (ee, ballot_box_id, "MixDecOffline")
    eb_pk_cut = eb_pk.public_key[:delta_hat]

    verify_shuffle_ok = verify_shuffle(
        group=group,
        ciphertexts=c_dec_4,
        shuffled_ciphertexts=c_mix_5,
        shuffle_argument=pi_mix_5,
        pk=eb_pk_cut,
    )

    if verify_shuffle_ok is False:
        LOGGER.warning(
            "verify_tally_control_component_ballot_box failed because verify_shuffle failed",
            group=group,
            ee=ee,
            ballot_box_id=ballot_box_id,
            eb_pk=eb_pk,
            # ...more data?
        )
        return False

```

```
    c_prime = tuple(MultiRecipientCiphertext(gamma=c.gamma, phis=m_i) for c, m_i in zip(c_mix_5, m))

    verify_decryptions_ok = verify_decryptions(
        group=group,
        c=c_mix_5,
        pk=eb_pk_cut,
        c_prime=c_prime,
        pi_dec=pi_dec_5,
        i_aux=i_aux,
    )

    if verify_decryptions_ok is False:
        LOGGER.warning(
            "verify_tally_control_component_ballot_box failed because verify_decryptions failed",
            group=group,
            ee=ee,
            ballot_box_id=ballot_box_id,
            eb_pk=eb_pk,
            # ...more data?
            c_prime=c_prime,
            i_aux=i_aux,
        )

        return False

    verify_process_plaintexts_ok = verify_process_plaintexts(
        group=group,
        v_tilde=v_tilde,
        p_tilde=p_tilde,
        m=m,
        psi=psi,
        delta_hat=delta_hat,
        l_votes=l_votes,
        l_decoded_votes=l_decoded_votes,
    )

    if verify_process_plaintexts_ok is False:
        LOGGER.warning(
            "verify_tally_control_component_ballot_box failed because verify_decryptions failed",
            group=group,
            ee=ee,
            ballot_box_id=ballot_box_id,
            eb_pk=eb_pk,
            # ...more data?
        )
        return False

    return True
```

Listing A.4. VerifyTallyControlComponentsBallotBox

A.1.5 VerifyOnlineControlComponents

```
# final_verification.py
def verify_online_control_components(
    group: Group,
    ee: str,
    ballot_box_ids: tuple[str, ...],
    psis: tuple[int, ...],
```

```python
    el_pk: ElectionPublicKey,
    ccm_el_pk: tuple[tuple[int, ...], ...],
    eb_pk: ElectionPublicKey,
    pk_bold_ccr: ChoiceReturnCodesEncryptionPublicKey,
    delta_hats: tuple[int, ...],
    kmaps: tuple[dict[str, int], ...],
    vc_bold_1s: tuple[tuple[str, ...], ...],
    e1_bold_1s: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    e1_bold_tilde_1s: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    e2_bold_1s: tuple[tuple[MultiRecipientCiphertext, ...], ...],
    pi_bold_exp_1s: tuple[tuple[Proof, ...], ...],
    pi_bold_eqenc_1s: tuple[tuple[Proof2, ...], ...],
    c_mixs: tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...],
    pi_mixs: tuple[tuple[ShuffleArgument, ...], ...],
    c_decs: tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...],
    pi_decs: tuple[tuple[tuple[Proofs, ...], ...], ...],
    p_tildes: tuple[tuple[int, ...], ...],
    v_tildes: tuple[tuple[str, ...], ...],
) -> bool:

    for j in range(len(ballot_box_ids)):
        if not verify_online_control_components_ballot_box(
            group=group,
            ee=ee,
            ballot_box_id=ballot_box_ids[j],
            psi=psis[j],
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=delta_hats[j],
            kmap=kmaps[j],
            vc_bold_1=vc_bold_1s[j],
            e1_bold_1=e1_bold_1s[j],
            e1_bold_tilde_1=e1_bold_tilde_1s[j],
            e2_bold_1=e2_bold_1s[j],
            pi_bold_exp_1=pi_bold_exp_1s[j],
            pi_bold_eqenc_1=pi_bold_eqenc_1s[j],
            c_mix=c_mixs[j],
            pi_mix=pi_mixs[j],
            c_dec=c_decs[j],
            pi_dec=pi_decs[j],
            p_tilde=p_tildes[j],
            v_tilde=v_tildes[j],
        ):
            LOGGER.warning(
                "verify_online_control_components failed because ",
            )
            return False
    return True
```

Listing A.5. VerifyOnlineControlComponents

A.2 Tests

A.2.1 Tests VerifyProcessPlaintexts

```python
# test_verify_process_plaintexts.py

import json
from dataclasses import dataclass
from pathlib import Path
```

```python
from typing import Final

from structlog.testing import capture_logs
from swiss_post_voting_system.crypto_primitives.elgamal import Group
from swiss_post_voting_system.verifier.final_verification import verify_process_plaintexts

from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR

# electionEventContextPayload.json
data_electionEventContextPayload = json.loads(
    (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
)

TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"

GROUP: Final[Group] = Group.from_dict(dct=data_electionEventContextPayload["encryptionGroup"])


@dataclass(frozen=True, slots=True)
class Data:
    """
    Data for the tests
    """

    ballot_box_id: str
    get_delta_hat_context: int

    def short_id_bb(self) -> str:
        """
        return the first 4 chars of the ballot_box_id
        """
        return self.ballot_box_id[:4]


DATA = (
    Data(ballot_box_id="4120f03ccc8641389adf907c8c80f205", get_delta_hat_context=0),
    Data(ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d", get_delta_hat_context=1),
    Data(ballot_box_id="4600fb57269a426695193b57f694ed1c", get_delta_hat_context=2),
    Data(ballot_box_id="1620dc54f5a147d492668dd34280261d", get_delta_hat_context=3),
)


def parse_payload() -> tuple[dict, dict, dict, dict]:
    """
    Parsing the payload
    """

    data_cc_ballot_box_payload: dict[str, dict] = {}
    data_cc_shuffle_payload: dict[str, dict] = {}
    data_tally_component_shuffle_payload: dict[str, dict] = {}
    data_tally_component_votes_payload: dict[str, dict] = {}

    for ballot_box_path in TALLY_BOXES_DIR.iterdir():
        ballot_box_short = ballot_box_path.name[:4]

        data_cc_ballot_box_payload[ballot_box_short] = {}
        data = data_cc_ballot_box_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").read_text()
```

```
            )

        data_cc_shuffle_payload[ballot_box_short] = {}
        data = data_cc_shuffle_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentShufflePayload_{j}.json").read_text()
            )

        data_tally_component_shuffle_payload[ballot_box_short] = json.loads(
            (ballot_box_path / "tallyComponentShufflePayload.json").read_text()
        )

        data_tally_component_votes_payload[ballot_box_short] = json.loads(
            (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
        )
    return (
        data_cc_ballot_box_payload,
        data_cc_shuffle_payload,
        data_tally_component_shuffle_payload,
        data_tally_component_votes_payload,
    )


(
    DATA_CC_BALLOT_BOX_PAYLOAD,
    DATA_CC_SHUFFLE_PAYLOAD,
    DATA_TC_SHUFFLE_PAYLOAD,
    DATA_TC_VOTES_PAYLOAD,
) = parse_payload()


def get_delta_hat(i: int) -> int:
    """returns the number of allowed write-ins + 1 for this specific ballot box"""
    return int(
        data_electionEventContextPayload["electionEventContext"]["verificationCardSetContexts"][i][
            "numberOfWriteInFields"
        ]
        + 1
    )


def get_v_tilde(i: int) -> tuple[str, ...]:
    """returns list of actual voting options"""
    v_tilde_lst = []
    for k in data_electionEventContextPayload["electionEventContext"][
        "verificationCardSetContexts"
    ][i]["primesMappingTable"]["pTable"]:
        v_tilde_lst.append(k["actualVotingOption"])
    return tuple(v_tilde_lst)


def get_p_tilde(i: int) -> tuple[int, ...]:
    """returns list of actual encoded voting options"""
    p_tilde_lst = []
    for k in data_electionEventContextPayload["electionEventContext"][
        "verificationCardSetContexts"
    ][i]["primesMappingTable"]["pTable"]:
        p_tilde_lst.append(k["encodedVotingOption"])
    return tuple(p_tilde_lst)

```

```
def get_m(json_data: dict) -> tuple[tuple[int, ...], ...]:
    """returns the list of plaintext votes"""
    return tuple(
        tuple(int(i, 16) for i in k["message"])
        for k in json_data["verifiablePlaintextDecryption"]["decryptedVotes"]
    )


def get_write_in_voting_options(i: int) -> tuple[int, ...]:
    """returns write-in voting options"""
    p_tilde_write_ins_lst = []
    for k in data_electionEventContextPayload["electionEventContext"][
        "verificationCardSetContexts"
    ][i]["primesMappingTable"]["pTable"]:
        if str(k["actualVotingOption"]).startswith("WRITE_IN_"):
            p_tilde_write_ins_lst.append(k["encodedVotingOption"])
    return tuple(p_tilde_write_ins_lst)


def get_l_votes(json_data: dict) -> tuple[tuple[int, ...], ...]:
    """returns list of all selected encoded voting options"""
    return tuple(tuple(i) for i in json_data["votes"])


def get_l_decoded_votes(json_data: dict) -> tuple[tuple[str, ...], ...]:
    """returns list of all selected decoded voting options"""
    return tuple(tuple(i) for i in json_data["actualSelectedVotingOptions"])


def get_l_write_ins(json_data: dict) -> tuple[tuple[str, ...], ...]:
    """returns list of all selected decoded write-in options"""
    return tuple(tuple(i) for i in json_data["decodedWriteInVotes"])


def get_psi(json_data: dict) -> int:
    """returns the number of selectable voting options"""
    return len(json_data["votes"][0])


def test_ok() -> None:
    """
    All the tests that should not fail.
    """

    for data in DATA:
        short_id_bb = data.short_id_bb()

        is_ok = verify_process_plaintexts(
            group=GROUP,
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
    short_id_bb]),
        )

        assert is_ok
```

```python
def test_fail() -> None:
    """
    Tests that must fail.
    """

    with capture_logs():
        data = DATA[1]
        short_id_bb = data.short_id_bb()
        manipulated_m_1 = (
            (10865, 1, 1),
            (10865, 38369, 1),
            (38369, 13365276363158976492414625067467080921481393000077236122849, 1),
        )
        is_ok = verify_process_plaintexts(
            group=GROUP,
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            m=manipulated_m_1,
            psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )

    assert is_ok is False, "verify_process_plaintexts should have failed due to wrong m"

    with capture_logs():
        data = DATA[3]
        short_id_bb = data.short_id_bb()
        manipulated_l_votes = ((5, 59, 43), (5, 37, 53), (5, 73, 43))
        manipulated_l_decoded_votes = (
            ("b9c57a40-a555-35e1-972c-a1a6b7e03381", "1-4", "1-5"),
            (
                "b9c57a40-a555-35e1-972c-a1a6b7e03381",
                "WRITE_IN_02c52035069d4d8ab78892b8882ec83b",
                "1-2",
            ),
            ("b9c57a40-a555-35e1-972c-a1a6b7e03381", "ae3a5d49-b15d-3e88-8bdd-cbf965497a8c", "1-5"),
        )
        is_ok = verify_process_plaintexts(
            group=GROUP,
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            l_votes=manipulated_l_votes,
            l_decoded_votes=manipulated_l_decoded_votes,
        )

    assert (
        is_ok is False
    ), "verify_process_plaintexts should have failed due to wrong l_votes and l_decoded_votes"

    with capture_logs():
        data = DATA[0]
        short_id_bb = data.short_id_bb()
        manipulated_m_2 = ((17, 19), (17, 5), (5, 17))
        manipulated_delta_hat = 2
```

```
        is_ok = verify_process_plaintexts(
            group=GROUP,
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            m=manipulated_m_2,
            psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            delta_hat=manipulated_delta_hat,
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )

    assert (
        is_ok is False
    ), "verify_process_plaintexts should have failed because of wrong m and delta_hat"

    with capture_logs():
        data = DATA[2]
        short_id_bb = data.short_id_bb()
        manipulated_v_tilde = (
            "8814c2b6-8c73-38e8-99e6-830bffdf32c6",
            "57a30570-1722-3a7e-a8f9-7dd643d7f339",
            "b9c57a40-a555-35e1-972c-a1a6b7e03381",
        )
        manipulated_p_tilde = (17, 5, 19)
        is_ok = verify_process_plaintexts(
            group=GROUP,
            v_tilde=manipulated_v_tilde,
            p_tilde=manipulated_p_tilde,
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )
    assert is_ok is False, "verify_process_plaintexts should have failed because of wrong pTable"


if __name__ == "__main__":
    test_ok()
    test_fail()

```

Listing A.6. Tests VerifyProcessPlaintexts

A.2.2 Tests VerifyMixDecOffline

```
# test_verify_mix_dec_offline.py
import json
from dataclasses import dataclass
from pathlib import Path
from typing import Any, Final

from structlog.testing import capture_logs
from swiss_post_voting_system.crypto_primitives.elgamal import (
    ElectionPublicKey,
    Group,
    MultiRecipientCiphertext,
)
```

```
from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
    MultiExponentiationArgument,
    ProductArgument,
    ShuffleArgument,
    SingleValueProductArgument,
)
from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proofs
from swiss_post_voting_system.system.mix_offline import verify_mix_dec_offline
from swiss_post_voting_system.system.mix_online import
    get_mixnet_initial_ciphertexts

from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR

TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"

ELECTION_EVENT_CONTEXT_PAYLOAD_DICT = json.loads(
    (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
)

GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["
    encryptionGroup"])


@dataclass(frozen=True, slots=True)
class Data:
    """
    Data for the tests
    """

    ballot_box_id: str
    get_delta_hat_context: int

    def short_id(self) -> str:
        """
        return the first 4 chars of the ballot_box_id
        """
        return self.ballot_box_id[:4]


DATA = (
    Data(ballot_box_id="4120f03ccc8641389adf907c8c80f205", get_delta_hat_context=0),
    Data(ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d", get_delta_hat_context=1),
    Data(ballot_box_id="4600fb57269a426695193b57f694ed1c", get_delta_hat_context=2),
    Data(ballot_box_id="1620dc54f5a147d492668dd34280261d", get_delta_hat_context=3),
)


def parse_payload() -> tuple[dict, dict]:
    """
    Parsing the payload
    """

    data_cc_ballot_box_payload: dict[str, dict] = {}
    data_cc_shuffle_payload: dict[str, dict] = {}
    for ballot_box_path in TALLY_BOXES_DIR.iterdir():
        ballot_box_short = ballot_box_path.name[:4]

        data_cc_ballot_box_payload[ballot_box_short] = {}
        data = data_cc_ballot_box_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").
    read_text()
```

```
            )

        data_cc_shuffle_payload[ballot_box_short] = {}
        data = data_cc_shuffle_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentShufflePayload_{j}.json").read_text()
            )
    return data_cc_ballot_box_payload, data_cc_shuffle_payload


DATA_CC_BALLOT_BOX_PAYLOAD, DATA_CC_SHUFFLE_PAYLOAD = parse_payload()


def get_election_event_id() -> str:
    """returns election event ID ee"""

    # the str(...) is only here to make mypy happy...
    return str(ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["electionEventId"])


def get_c_init(ballot_box: str) -> tuple[MultiRecipientCiphertext, ...]:
    """returns mix net initial ciphertexts"""

    return get_mixnet_initial_ciphertexts(
        group=GROUP,
        delta_hat=get_delta_hat(3),
        vc_map=get_vc_map(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[ballot_box][1]),
        el_pk=get_el_pk(),
    )


def get_delta_hat(i: int) -> int:
    """returns the number of allowed write-ins + 1 for this specific ballot box"""
    return int(
        ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["verificationCardSetContexts"][
            i
        ]["numberOfWriteInFields"]
        + 1
    )


def get_vc_map(json_data: dict) -> dict[str, MultiRecipientCiphertext]:
    """returns vcMap used for the calculation of c_init_1"""
    vc_map = {}
    for i in json_data["confirmedEncryptedVotes"]:
        vc = i["contextIds"]["verificationCardId"]
        e_1 = MultiRecipientCiphertext(
            gamma=int(i["encryptedVote"]["gamma"], 16),
            phis=tuple(int(x, 16) for x in i["encryptedVote"]["phis"]),
        )
        vc_map[vc] = e_1
    return vc_map


# pylint: disable=too-many-branches
def get_c_mix(ballot_box: str) -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
    """returns preceding shuffled votes"""
    return tuple(get_c_mix_j(DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j in range(1, 5))
```

```python


def get_c_mix_j(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns preceding shuffled votes"""
    c_mix_j_lst = []
    for i in json_data["verifiableShuffle"]["shuffledCiphertexts"]:
        c_mix_j_lst.append(
            MultiRecipientCiphertext(
                gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
            )
        )
    return tuple(c_mix_j_lst)


def get_pi_mix(ballot_box: str) -> tuple[ShuffleArgument, ...]:
    """returns preceding shuffled votes"""

    return tuple(
        get_shuffle_argument(json_data=DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j in range(1, 5)
    )


def get_shuffle_argument(
    json_data: dict[str, dict[str, dict[str, dict[str, Any]]]]
) -> ShuffleArgument:
    """returns a preceding shuffle proof"""
    e_lst = []
    for i in json_data["verifiableShuffle"]["shuffleArgument"]["multiExponentiationArgument"]["E"]:
        e_lst.append(
            MultiRecipientCiphertext(
                gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
            )
        )
    e = tuple(e_lst)

    return ShuffleArgument(
        c_a=tuple(int(i, 16) for i in json_data["verifiableShuffle"]["shuffleArgument"]["c_A"]),
        c_b=tuple(int(i, 16) for i in json_data["verifiableShuffle"]["shuffleArgument"]["c_B"]),
        product_argument=ProductArgument(
            c_b=None,
            hadamard_arg=None,
            single_value_product_arg=SingleValueProductArgument(
                c_d=int(
                    json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
                        "singleValueProductArgument"
                    ]["c_d"],
                    16,
                ),
                c_lower_delta=int(
                    json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
                        "singleValueProductArgument"
                    ]["c_delta"],
                    16,
                ),
                c_upper_delta=int(
                    json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
```

```
                    "singleValueProductArgument"
                ]["c_Delta"],
                16,
            ),
            a_tilde=tuple(
                int(i, 16)
                for i in json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
                    "singleValueProductArgument"
                ]["a_tilde"]
            ),
            b_tilde=tuple(
                int(i, 16)
                for i in json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
                    "singleValueProductArgument"
                ]["b_tilde"]
            ),
            r_tilde=int(
                json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
                    "singleValueProductArgument"
                ]["r_tilde"],
                16,
            ),
            s_tilde=int(
                json_data["verifiableShuffle"]["shuffleArgument"]["productArgument"][
                    "singleValueProductArgument"
                ]["s_tilde"],
                16,
            ),
        ),
    ),
    multi_exponentiation_argument=MultiExponentiationArgument(
        c_a_0=int(
            json_data["verifiableShuffle"]["shuffleArgument"]["multiExponentiationArgument"][
                "c_A_0"
            ],
            16,
        ),
        c_b=tuple(
            int(i, 16)
            for i in json_data["verifiableShuffle"]["shuffleArgument"][
                "multiExponentiationArgument"
            ]["c_B"]
        ),
        e=e,
        a=tuple(
            int(i, 16)
            for i in json_data["verifiableShuffle"]["shuffleArgument"][
                "multiExponentiationArgument"
            ]["a"]
        ),
        r=int(
            json_data["verifiableShuffle"]["shuffleArgument"]["multiExponentiationArgument"][
                "r"
            ],
            16,
        ),
        b=int(
```

```
                json_data["verifiableShuffle"]["shuffleArgument"]["multiExponentiationArgument"][
                    "b"
                ],
                16,
            ),
            s=int(
                json_data["verifiableShuffle"]["shuffleArgument"]["multiExponentiationArgument"][
                    "s"
                ],
                16,
            ),
            tau=int(
                json_data["verifiableShuffle"]["shuffleArgument"]["multiExponentiationArgument"][
                    "tau"
                ],
                16,
            ),
        ),
    )


def get_c_dec(ballot_box: str) -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
    """returns preceding partially decrypted votes"""

    return tuple(get_c_dec_j(json_data=DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j in range(1, 5))


def get_c_dec_j(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns preceding partially decrypted votes"""
    c_dec_j_lst = []
    for i in json_data["verifiableDecryptions"]["ciphertexts"]:
        c_dec_j_lst.append(
            MultiRecipientCiphertext(
                gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
            )
        )
    return tuple(c_dec_j_lst)


def get_pi_dec(ballot_box: str) -> tuple[tuple[Proofs, ...], ...]:
    """returns preceding decryption proofs"""

    return tuple(
        get_pi_dec_j(json_data=DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j in range(1, 5)
    )


def get_pi_dec_j(json_data: dict) -> tuple[Proofs, ...]:
    """returns preceding decryption proofs"""
    pi_dec_j_lst = []
    for i in json_data["verifiableDecryptions"]["decryptionProofs"]:
        pi_dec_j_lst.append(Proofs(e=int(i["e"], 16), z=tuple(int(x, 16) for x in i["z"])))
    return tuple(pi_dec_j_lst)


def get_el_pk() -> ElectionPublicKey:
    """returns election public key"""
```

```
    return ElectionPublicKey(
        tuple(
            int(x, 16)
            for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
                "electionPublicKey"
            ]
        )
    )


def get_ccm_el_pk() -> tuple[tuple[int, ...], ...]:
    """returns CCM election public keys"""

    return tuple(
        tuple(int(key, 16) for key in j["ccmjElectionPublicKey"])
        for j in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
            "combinedControlComponentPublicKeys"
        ]
    )


def get_eb_pk() -> ElectionPublicKey:
    """returns electoral board public key"""
    return ElectionPublicKey(
        tuple(
            int(x, 16)
            for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
                "electoralBoardPublicKey"
            ]
        )
    )


def test_ok() -> None:
    """
    All the tests that should not fail.
    """

    for data in DATA:
        short_id = data.short_id()
        is_ok = verify_mix_dec_offline(
            group=GROUP,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            c_init=get_c_init(short_id),
            c_mix=get_c_mix(short_id),
            pi_mix=get_pi_mix(short_id),
            c_dec=get_c_dec(short_id),
            pi_dec=get_pi_dec(short_id),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
        )

        assert is_ok


def test_fail() -> None:
    """
    Tests that must fail.
    """

```

```
    data = DATA[0]
    short_id = data.short_id()

    wrong_data = DATA[2]
    wrong_short_id = wrong_data.short_id()

    el_pk = get_el_pk()
    eb_pk = get_eb_pk()
    ccm_el_pk = get_ccm_el_pk()

    with capture_logs():
        is_ok = verify_mix_dec_offline(
            group=GROUP,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            c_init=get_c_init(wrong_short_id),
            c_mix=get_c_mix(short_id),
            pi_mix=get_pi_mix(short_id),
            c_dec=get_c_dec(short_id),
            pi_dec=get_pi_dec(short_id),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
        )

    assert is_ok is False, "verify_mix_dec_offline should have failed because of wrong c_init"

    with capture_logs():
        is_ok = verify_mix_dec_offline(
            group=GROUP,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            c_init=get_c_init(short_id),
            c_mix=get_c_mix(wrong_short_id),
            pi_mix=get_pi_mix(short_id),
            c_dec=get_c_dec(short_id),
            pi_dec=get_pi_dec(short_id),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
        )

    assert is_ok is False, "verify_mix_dec_offline should have failed because of wrong c_mix"

    with capture_logs():
        is_ok = verify_mix_dec_offline(
            group=GROUP,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            c_init=get_c_init(short_id),
            c_mix=get_c_mix(short_id),
            pi_mix=get_pi_mix(wrong_short_id),
            c_dec=get_c_dec(short_id),
            pi_dec=get_pi_dec(short_id),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
        )
```

```

    assert is_ok is False, "verify_mix_dec_offline should have failed because of wrong pi_mix"

    with capture_logs():
        is_ok = verify_mix_dec_offline(
            group=GROUP,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            c_init=get_c_init(short_id),
            c_mix=get_c_mix(short_id),
            pi_mix=get_pi_mix(short_id),
            c_dec=get_c_dec(wrong_short_id),
            pi_dec=get_pi_dec(short_id),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
        )

    assert is_ok is False, "verify_mix_dec_offline should have failed because of wrong c_dec"

    with capture_logs():
        is_ok = verify_mix_dec_offline(
            group=GROUP,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            c_init=get_c_init(short_id),
            c_mix=get_c_mix(short_id),
            pi_mix=get_pi_mix(short_id),
            c_dec=get_c_dec(short_id),
            pi_dec=get_pi_dec(wrong_short_id),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
        )

    assert is_ok is False, "verify_mix_dec_offline should have failed because of wrong pi_dec"


if __name__ == "__main__":
    test_ok()
    test_fail()

```

Listing A.7. Tests VerifyMixDecOffline

A.2.3 Tests VerifyOnlineControlComponentsBallotBox

```
# test_verify_online_control_components_ballot_box.py

import json
from dataclasses import dataclass
from pathlib import Path
from typing import Final

from structlog.testing import capture_logs
from swiss_post_voting_system.crypto_primitives.elgamal import (
    ChoiceReturnCodesEncryptionPublicKey,
```

```python
    Group,
)
from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
    MultiRecipientCiphertext,
)
from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proof, Proof2
from swiss_post_voting_system.verifier.final_verification import (
    verify_online_control_components_ballot_box,
)

from swiss_post_voting_system_tests.system_tests.test_verify_mix_dec_offline import (
    get_c_dec,
    get_c_mix,
    get_ccm_el_pk,
    get_eb_pk,
    get_el_pk,
    get_pi_dec,
    get_pi_mix,
)
from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
from swiss_post_voting_system_tests.verifier_tests.test_verify_process_plaintexts import (
    get_p_tilde,
    get_v_tilde,
)

ELECTION_EVENT_CONTEXT_PAYLOAD_DICT = json.loads(
    (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
)

SETUP_VCS_DIR: Final[Path] = DATASETS_DIR / "dataset1/setup/verification_card_sets"
TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"

GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["encryptionGroup"])


@dataclass(frozen=True, slots=True)
class Data:
    """
    Data for the tests
    """

    verification_card_set_id: str
    ballot_box_id: str
    get_delta_hat_context: int

    def short_id_vcs(self) -> str:
        """
        return the first 4 chars of the verification_card_set_id
        """
        return self.verification_card_set_id[:4]

    def short_id_bb(self) -> str:
        """
        return the first 4 chars of the ballot_box_id
        """
        return self.ballot_box_id[:4]


DATA = (
```

```python
    Data(
        verification_card_set_id="73e2eed19de9494ea9eaf93968e9b428",
        ballot_box_id="4120f03ccc8641389adf907c8c80f205",
        get_delta_hat_context=0,
    ),
    Data(
        verification_card_set_id="3880a1b0f49341d68f3c9fec15782063",
        ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d",
        get_delta_hat_context=1,
    ),
    Data(
        verification_card_set_id="ae82cc64b620433da892983df6363d8c",
        ballot_box_id="4600fb57269a426695193b57f694ed1c",
        get_delta_hat_context=2,
    ),
    Data(
        verification_card_set_id="fe9bb7092993440eb51235f0efa5d19b",
        ballot_box_id="1620dc54f5a147d492668dd34280261d",
        get_delta_hat_context=3,
    ),
)


def parse_payload() -> tuple[dict, dict, dict, dict]:
    """
    Parsing the payload
    """

    data_cc_ballot_box_payload: dict[str, dict] = {}
    data_cc_shuffle_payload: dict[str, dict] = {}
    data_tally_component_votes_payload: dict[str, dict] = {}
    for ballot_box_path in TALLY_BOXES_DIR.iterdir():
        ballot_box_short = ballot_box_path.name[:4]

        data_cc_ballot_box_payload[ballot_box_short] = {}
        data = data_cc_ballot_box_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").read_text()
            )

        data_cc_shuffle_payload[ballot_box_short] = {}
        data = data_cc_shuffle_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentShufflePayload_{j}.json").read_text()
            )

        data_tally_component_votes_payload[ballot_box_short] = json.loads(
            (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
        )

    data_setup_component_tally_data_payload: dict[str, dict] = {}
    for vcs_path in SETUP_VCS_DIR.iterdir():
        vcs_short = vcs_path.name[:4]

        data_setup_component_tally_data_payload[vcs_short] = json.loads(
            (vcs_path / "setupComponentTallyDataPayload.json").read_text()
        )

    return (
```

```python
        data_cc_ballot_box_payload,
        data_cc_shuffle_payload,
        data_setup_component_tally_data_payload,
        data_tally_component_votes_payload,
    )


(
    DATA_CC_BALLOT_BOX_PAYLOAD,
    DATA_CC_SHUFFLE_PAYLOAD,
    DATA_SC_TALLY_DATA_PAYLOAD,
    DATA_TC_VOTES_PAYLOAD,
) = parse_payload()


def get_election_event_id() -> str:
    """returns election event ID ee"""
    # the str(...) is only here to make mypy happy...
    return str(ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["
    electionEventId"])


def get_psi(json_data: dict) -> int:
    """returns number of selectable voting options"""
    return len(json_data["confirmedEncryptedVotes"][0]["
    encryptedPartialChoiceReturnCodes"]["phis"])


def get_pk_bold_ccr() -> ChoiceReturnCodesEncryptionPublicKey:
    """returns choice return codes encryption public key"""
    return ChoiceReturnCodesEncryptionPublicKey(
        tuple(
            int(x, 16)
            for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
                "choiceReturnCodesEncryptionPublicKey"
            ]
        )
    )


def get_delta_hat(i: int) -> int:
    """returns the number of allowed write-ins + 1 for this specific ballot box"""
    return int(
        ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["
    verificationCardSetContexts"][
            i
        ]["numberOfWriteInFields"]
        + 1
    )


def get_kmap(json_data: dict) -> dict[str, int]:
    """returns key-value map of the verification card public keys"""
    kmap = {}
    for (vc_id, public_key) in zip(
        json_data["verificationCardIds"], json_data["verificationCardPublicKeys"]
    ):
        kmap[str(vc_id)] = int(public_key[0], 16)
    return kmap


def get_vc_bold_1(json_data: dict) -> tuple[str, ...]:
    """returns control component’s list of confirmed verification card IDs"""
```

```python
    return tuple(
        i["contextIds"]["verificationCardId"] for i in json_data["confirmedEncryptedVotes"]
    )


def get_e1_bold_1(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns control component’s list of encrypted, confirmed votes"""
    return tuple(
        MultiRecipientCiphertext(
            gamma=int(i["encryptedVote"]["gamma"], 16),
            phis=tuple(int(x, 16) for x in i["encryptedVote"]["phis"]),
        )
        for i in json_data["confirmedEncryptedVotes"]
    )


def get_e1_bold_tilde_1(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns control component’s list of exponentiated, encrypted, confirmed votes"""
    return tuple(
        MultiRecipientCiphertext(
            gamma=int(i["exponentiatedEncryptedVote"]["gamma"], 16),
            phis=tuple(int(x, 16) for x in i["exponentiatedEncryptedVote"]["phis"]),
        )
        for i in json_data["confirmedEncryptedVotes"]
    )


def get_e2_bold_1(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns control component’s list of encrypted, partial Choice Return Codes"""
    return tuple(
        MultiRecipientCiphertext(
            gamma=int(i["encryptedPartialChoiceReturnCodes"]["gamma"], 16),
            phis=tuple(int(x, 16) for x in i["encryptedPartialChoiceReturnCodes"]["phis"]),
        )
        for i in json_data["confirmedEncryptedVotes"]
    )


def get_pi_bold_exp_1(json_data: dict) -> tuple[Proof, ...]:
    """returns control component’s list of exponentiation proofs"""
    return tuple(
        Proof(
            e=int(i["exponentiationProof"]["e"], 16),
            z=int(i["exponentiationProof"]["z"], 16),
        )
        for i in json_data["confirmedEncryptedVotes"]
    )


def get_pi_bold_eqenc_1(json_data: dict) -> tuple[Proof2, ...]:
    """returns control component’s list of plaintext equality proofs"""
    return tuple(
        Proof2(
            e=int(i["plaintextEqualityProof"]["e"], 16),
            z=(
                int(i["plaintextEqualityProof"]["z"][0], 16),
                int(i["plaintextEqualityProof"]["z"][1], 16),
            ),
        )
        for i in json_data["confirmedEncryptedVotes"]
```

```
    )


def test_ok() -> None:
    """
    All the tests that should not fail.
    """

    for data in DATA:
        short_id_bb = data.short_id_bb()
        short_id_vcs = data.short_id_vcs()

        is_ok = verify_online_control_components_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
            pk_bold_ccr=get_pk_bold_ccr(),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
            vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_tilde_1=get_e1_bold_tilde_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_eqenc_1=get_pi_bold_eqenc_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            c_mix=get_c_mix(short_id_bb),
            pi_mix=get_pi_mix(short_id_bb),
            c_dec=get_c_dec(short_id_bb),
            pi_dec=get_pi_dec(short_id_bb),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
        )

        assert is_ok


def test_fail() -> None:
    """
    Tests that must fail.
    """

    data = DATA[0]
    short_id_bb = data.short_id_bb()
    short_id_vcs = data.short_id_vcs()

    wrong_data = DATA[2]
    wrong_short_id_bb = wrong_data.short_id_bb()

    el_pk = get_el_pk()
    eb_pk = get_eb_pk()
    ccm_el_pk = get_ccm_el_pk()
```

```
    pk_bold_ccr = get_pk_bold_ccr()

    with capture_logs():
        is_ok = verify_online_control_components_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
            vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_tilde_1=get_e1_bold_tilde_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb][1]
            ),
            e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_eqenc_1=get_pi_bold_eqenc_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            c_mix=get_c_mix(short_id_bb),
            pi_mix=get_pi_mix(short_id_bb),
            c_dec=get_c_dec(short_id_bb),
            pi_dec=get_pi_dec(short_id_bb),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components_ballot_box should have failed due to wrong e1_bold_tilde_1"

    with capture_logs():
        is_ok = verify_online_control_components_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
            vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_tilde_1=get_e1_bold_tilde_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_exp_1=get_pi_bold_exp_1(
```

```
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb][1]
            ),
            pi_bold_eqenc_1=get_pi_bold_eqenc_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            c_mix=get_c_mix(short_id_bb),
            pi_mix=get_pi_mix(short_id_bb),
            c_dec=get_c_dec(short_id_bb),
            pi_dec=get_pi_dec(short_id_bb),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components_ballot_box should have failed due to wrong pi_bold_exp_1"

    with capture_logs():
        is_ok = verify_online_control_components_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
            vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_tilde_1=get_e1_bold_tilde_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_eqenc_1=get_pi_bold_eqenc_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            c_mix=get_c_mix(short_id_bb),
            pi_mix=get_pi_mix(wrong_short_id_bb),
            c_dec=get_c_dec(short_id_bb),
            pi_dec=get_pi_dec(short_id_bb),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components_ballot_box should have failed because of wrong pi_mix"

    with capture_logs():
        is_ok = verify_online_control_components_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
```

```
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
            vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_tilde_1=get_e1_bold_tilde_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb][1]),
            pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_eqenc_1=get_pi_bold_eqenc_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            c_mix=get_c_mix(short_id_bb),
            pi_mix=get_pi_mix(short_id_bb),
            c_dec=get_c_dec(short_id_bb),
            pi_dec=get_pi_dec(short_id_bb),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components_ballot_box should have failed because of wrong e2_bold_1"

    with capture_logs():
        is_ok = verify_online_control_components_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            el_pk=el_pk,
            ccm_el_pk=ccm_el_pk,
            eb_pk=eb_pk,
            pk_bold_ccr=pk_bold_ccr,
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
            vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            e1_bold_tilde_1=get_e1_bold_tilde_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
            ),
            e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            pi_bold_eqenc_1=get_pi_bold_eqenc_1(
                json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb][1]
            ),
            c_mix=get_c_mix(short_id_bb),
            pi_mix=get_pi_mix(short_id_bb),
            c_dec=get_c_dec(short_id_bb),
            pi_dec=get_pi_dec(short_id_bb),
```

```
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components_ballot_box should have failed due to wrong pi_bold_eqenc_1"


if __name__ == "__main__":
    test_ok()
    test_fail()

```

Listing A.8. Tests VerifyOnlineControlComponentsBallotBox

A.2.4 Tests VerifyTallyControlComponentsBallotBox

```
# test_verify_tally_control_components_ballot_box.py

import json
from dataclasses import dataclass
from pathlib import Path
from typing import Any, Final

from structlog.testing import capture_logs
from swiss_post_voting_system.crypto_primitives.elgamal import Group
from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
    MultiExponentiationArgument,
    MultiRecipientCiphertext,
    ProductArgument,
    ShuffleArgument,
    SingleValueProductArgument,
)
from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proofs
from swiss_post_voting_system.verifier.final_verification import (
    verify_tally_control_component_ballot_box,
)

from swiss_post_voting_system_tests.system_tests.test_verify_mix_dec_offline import (
    get_delta_hat,
    get_eb_pk,
)
from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
from swiss_post_voting_system_tests.verifier_tests.test_verify_process_plaintexts import (
    get_l_decoded_votes,
    get_l_votes,
    get_m,
    get_p_tilde,
    get_v_tilde,
)

ELECTION_EVENT_CONTEXT_PAYLOAD = json.loads(
    (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
)

SETUP_VCS_DIR: Final[Path] = DATASETS_DIR / "dataset1/setup/verification_card_sets"
TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"

```

```python
GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD["encryptionGroup"])


@dataclass(frozen=True, slots=True)
class Data:
    """
    Data for the tests
    """

    verification_card_set_id: str
    ballot_box_id: str
    get_delta_hat_context: int

    def short_id_vcs(self) -> str:
        """
        return the first 4 chars of the verification_card_set_id
        """
        return self.verification_card_set_id[:4]

    def short_id_bb(self) -> str:
        """
        return the first 4 chars of the ballot_box_id
        """
        return self.ballot_box_id[:4]


DATA = (
    Data(
        verification_card_set_id="73e2eed19de9494ea9eaf93968e9b428",
        ballot_box_id="4120f03ccc8641389adf907c8c80f205",
        get_delta_hat_context=0,
    ),
    Data(
        verification_card_set_id="3880a1b0f49341d68f3c9fec15782063",
        ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d",
        get_delta_hat_context=1,
    ),
    Data(
        verification_card_set_id="ae82cc64b620433da892983df6363d8c",
        ballot_box_id="4600fb57269a426695193b57f694ed1c",
        get_delta_hat_context=2,
    ),
    Data(
        verification_card_set_id="fe9bb7092993440eb51235f0efa5d19b",
        ballot_box_id="1620dc54f5a147d492668dd34280261d",
        get_delta_hat_context=3,
    ),
)


def parse_payload() -> tuple[dict, dict, dict, dict, dict]:
    """
    Parsing the payload
    """

    data_cc_ballot_box_payload: dict[str, dict] = {}
    data_cc_shuffle_payload: dict[str, dict] = {}
    data_tally_component_shuffle_payload: dict[str, dict] = {}
    data_tally_component_votes_payload: dict[str, dict] = {}
    for ballot_box_path in TALLY_BOXES_DIR.iterdir():
        ballot_box_short = ballot_box_path.name[:4]

```

```
        data_cc_ballot_box_payload[ballot_box_short] = {}
        data = data_cc_ballot_box_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").read_text()
            )

        data_cc_shuffle_payload[ballot_box_short] = {}
        data = data_cc_shuffle_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentShufflePayload_{j}.json").read_text()
            )

        data_tally_component_shuffle_payload[ballot_box_short] = json.loads(
            (ballot_box_path / "tallyComponentShufflePayload.json").read_text()
        )

        data_tally_component_votes_payload[ballot_box_short] = json.loads(
            (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
        )

    data_setup_component_tally_data_payload: dict[str, dict] = {}
    for vcs_path in SETUP_VCS_DIR.iterdir():
        vcs_short = vcs_path.name[:4]

        data_setup_component_tally_data_payload[vcs_short] = json.loads(
            (vcs_path / "setupComponentTallyDataPayload.json").read_text()
        )

    return (
        data_cc_ballot_box_payload,
        data_cc_shuffle_payload,
        data_setup_component_tally_data_payload,
        data_tally_component_shuffle_payload,
        data_tally_component_votes_payload,
    )


(
    DATA_CC_BALLOT_BOX_PAYLOAD,
    DATA_CC_SHUFFLE_PAYLOAD,
    DATA_SC_TALLY_DATA_PAYLOAD,
    DATA_TC_SHUFFLE_PAYLOAD,
    DATA_TC_VOTES_PAYLOAD,
) = parse_payload()


def get_election_event_id() -> str:
    """returns election event ID ee"""
    # the str(...) is only here to make mypy happy...
    return str(ELECTION_EVENT_CONTEXT_PAYLOAD["electionEventContext"]["electionEventId"])


def get_psi(json_data: dict) -> int:
    """returns number of selectable voting options"""
    return len(json_data["confirmedEncryptedVotes"][0]["encryptedPartialChoiceReturnCodes"]["phis"])


```

```python
def get_c_dec_4(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns last online control component’s partially decrypted votes"""
    c_dec_4_lst = []
    for i in json_data["verifiableDecryptions"]["ciphertexts"]:
        c_dec_4_lst.append(
            MultiRecipientCiphertext(
                gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
            )
        )
    return tuple(c_dec_4_lst)


def get_c_mix_5(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
    """returns preceding shuffled votes"""
    c_mix_5_lst = []
    for i in json_data["verifiableShuffle"]["shuffledCiphertexts"]:
        c_mix_5_lst.append(
            MultiRecipientCiphertext(
                gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
            )
        )
    return tuple(c_mix_5_lst)


def get_pi_mix_5(json_data: dict[str, dict[str, dict[str, dict[str, Any]]]]) -> ShuffleArgument:
    """returns a preceding shuffle proof"""
    data = json_data["verifiableShuffle"]["shuffleArgument"]
    e_lst = []
    for i in data["multiExponentiationArgument"]["E"]:
        e_lst.append(
            MultiRecipientCiphertext(
                gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
            )
        )
    e = tuple(e_lst)

    return ShuffleArgument(
        c_a=tuple(int(i, 16) for i in data["c_A"]),
        c_b=tuple(int(i, 16) for i in data["c_B"]),
        product_argument=ProductArgument(
            c_b=None,
            hadamard_arg=None,
            single_value_product_arg=SingleValueProductArgument(
                c_d=int(
                    data["productArgument"]["singleValueProductArgument"]["c_d"],
                    16,
                ),
                c_lower_delta=int(
                    data["productArgument"]["singleValueProductArgument"]["c_delta"],
                    16,
                ),
                c_upper_delta=int(
                    data["productArgument"]["singleValueProductArgument"]["c_Delta"],
                    16,
                ),
                a_tilde=tuple(
                    int(i, 16)
                    for i in data["productArgument"]["singleValueProductArgument"]["a_tilde"]
                ),
```

```python
                b_tilde=tuple(
                    int(i, 16)
                    for i in data["productArgument"]["singleValueProductArgument"]["b_tilde"]
                ),
                r_tilde=int(
                    data["productArgument"]["singleValueProductArgument"]["r_tilde"],
                    16,
                ),
                s_tilde=int(
                    data["productArgument"]["singleValueProductArgument"]["s_tilde"],
                    16,
                ),
            ),
        ),
        multi_exponentiation_argument=MultiExponentiationArgument(
            c_a_0=int(
                data["multiExponentiationArgument"]["c_A_0"],
                16,
            ),
            c_b=tuple(int(i, 16) for i in data["multiExponentiationArgument"]["c_B"]),
            e=e,
            a=tuple(int(i, 16) for i in data["multiExponentiationArgument"]["a"]),
            r=int(
                data["multiExponentiationArgument"]["r"],
                16,
            ),
            b=int(
                data["multiExponentiationArgument"]["b"],
                16,
            ),
            s=int(
                data["multiExponentiationArgument"]["s"],
                16,
            ),
            tau=int(
                data["multiExponentiationArgument"]["tau"],
                16,
            ),
        ),
    )


def get_pi_dec_5(json_data: dict) -> tuple[Proofs, ...]:
    """returns preceding decryption proofs"""
    pi_dec_5_lst = []
    for i in json_data["verifiablePlaintextDecryption"]["decryptionProofs"]:
        pi_dec_5_lst.append(Proofs(e=int(i["e"], 16), z=tuple(int(x, 16) for x in i["z"])))
    return tuple(pi_dec_5_lst)


def test_ok() -> None:
    """
    All the tests that should not fail.
    """

    for data in DATA:
        short_id_bb = data.short_id_bb()

```

```python
        is_ok = verify_tally_control_component_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            eb_pk=get_eb_pk(),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
            c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )

        assert is_ok


def test_fail() -> None:
    """
    Tests that must fail.
    """
    data = DATA[0]
    short_id_bb = data.short_id_bb()

    wrong_data = DATA[2]
    wrong_short_id_bb = wrong_data.short_id_bb()

    with capture_logs():
        is_ok = verify_tally_control_component_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            eb_pk=get_eb_pk(),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[wrong_short_id_bb][4]),
            c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[wrong_short_id_bb]),
            pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[wrong_short_id_bb]),
            pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )

    assert is_ok is False, (
        "verify_tally_control_component_ballot_box should have failed"
        "due to wrong c_dec_4, c_mix_5 and m"
    )

    with capture_logs():
        is_ok = verify_tally_control_component_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
```

```
            ballot_box_id=data.ballot_box_id,
            eb_pk=get_eb_pk(),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
            c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[wrong_short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[wrong_short_id_bb]),
        )

    assert is_ok is False, (
        "verify_tally_control_component_ballot_box should have failed"
        "due to wrong l_votes and l_decoded_votes"
    )

    with capture_logs():
        is_ok = verify_tally_control_component_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            eb_pk=get_eb_pk(),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
            c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[wrong_short_id_bb]),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )

    assert (
        is_ok is False
    ), "verify_tally_control_component_ballot_box should have failed because of wrong pi_mix_5"

    with capture_logs():
        is_ok = verify_tally_control_component_ballot_box(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_id=data.ballot_box_id,
            eb_pk=get_eb_pk(),
            v_tilde=get_v_tilde(data.get_delta_hat_context),
            p_tilde=get_p_tilde(data.get_delta_hat_context),
            psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
            delta_hat=get_delta_hat(data.get_delta_hat_context),
            c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
            c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
            pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[wrong_short_id_bb]),
```

```
            l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
            l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
        )
    assert is_ok is False, "verify_process_plaintexts should have failed because of wrong pi_dec_5"


if __name__ == "__main__":
    test_ok()
    test_fail()

```

Listing A.9. Tests VerifyTallyControlComponentsBallotBox

A.2.5 Tests VerifyOnlineControlComponents

```
# test_verify_online_control_components.py

import json
from dataclasses import dataclass
from pathlib import Path
from typing import Final

from structlog.testing import capture_logs
from swiss_post_voting_system.crypto_primitives.elgamal import (
    ChoiceReturnCodesEncryptionPublicKey,
    Group,
)
from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
    MultiRecipientCiphertext,
    ShuffleArgument,
)
from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proof, Proof2, Proofs
from swiss_post_voting_system.verifier.final_verification import verify_online_control_components

from swiss_post_voting_system_tests.system_tests.test_verify_mix_dec_offline import (
    get_c_dec,
    get_c_mix,
    get_ccm_el_pk,
    get_eb_pk,
    get_el_pk,
    get_pi_dec,
    get_pi_mix,
)
from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
from swiss_post_voting_system_tests.verifier_tests.test_verify_online_control_components_ballot_box import (
    get_delta_hat,
    get_e1_bold_1,
    get_e1_bold_tilde_1,
    get_e2_bold_1,
    get_kmap,
    get_pi_bold_eqenc_1,
    get_pi_bold_exp_1,
    get_psi,
    get_vc_bold_1,
)
from swiss_post_voting_system_tests.verifier_tests.test_verify_process_plaintexts import (
```

```
    get_p_tilde,
    get_v_tilde,
)

ELECTION_EVENT_CONTEXT_PAYLOAD_DICT = json.loads(
    (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
)

SETUP_VCS_DIR: Final[Path] = DATASETS_DIR / "dataset1/setup/verification_card_sets"
TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"

GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["encryptionGroup"])


@dataclass(frozen=True, slots=True)
class Data:
    """
    Data for the tests
    """

    verification_card_set_id: str
    ballot_box_id: str
    get_delta_hat_context: int

    def short_id_vcs(self) -> str:
        """
        return the first 4 chars of the verification_card_set_id
        """
        return self.verification_card_set_id[:4]

    def short_id_bb(self) -> str:
        """
        return the first 4 chars of the ballot_box_id
        """
        return self.ballot_box_id[:4]


DATA = (
    Data(
        verification_card_set_id="73e2eed19de9494ea9eaf93968e9b428",
        ballot_box_id="4120f03ccc8641389adf907c8c80f205",
        get_delta_hat_context=0,
    ),
    Data(
        verification_card_set_id="3880a1b0f49341d68f3c9fec15782063",
        ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d",
        get_delta_hat_context=1,
    ),
    Data(
        verification_card_set_id="ae82cc64b620433da892983df6363d8c",
        ballot_box_id="4600fb57269a426695193b57f694ed1c",
        get_delta_hat_context=2,
    ),
    Data(
        verification_card_set_id="fe9bb7092993440eb51235f0efa5d19b",
        ballot_box_id="1620dc54f5a147d492668dd34280261d",
        get_delta_hat_context=3,
    ),
)


def parse_payload() -> tuple[dict, dict, dict, dict]:
```

```
    """
    Parsing the payload
    """

    data_cc_ballot_box_payload: dict[str, dict] = {}
    data_cc_shuffle_payload: dict[str, dict] = {}
    data_tally_component_votes_payload: dict[str, dict] = {}
    for ballot_box_path in TALLY_BOXES_DIR.iterdir():
        ballot_box_short = ballot_box_path.name[:4]

        data_cc_ballot_box_payload[ballot_box_short] = {}
        data = data_cc_ballot_box_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").read_text()
            )

        data_cc_shuffle_payload[ballot_box_short] = {}
        data = data_cc_shuffle_payload[ballot_box_short]
        for j in range(1, 5):
            data[j] = json.loads(
                (ballot_box_path / f"controlComponentShufflePayload_{j}.json").read_text()
            )

        data_tally_component_votes_payload[ballot_box_short] = json.loads(
            (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
        )

    data_setup_component_tally_data_payload: dict[str, dict] = {}
    for vcs_path in SETUP_VCS_DIR.iterdir():
        vcs_short = vcs_path.name[:4]

        data_setup_component_tally_data_payload[vcs_short] = json.loads(
            (vcs_path / "setupComponentTallyDataPayload.json").read_text()
        )

    return (
        data_cc_ballot_box_payload,
        data_cc_shuffle_payload,
        data_setup_component_tally_data_payload,
        data_tally_component_votes_payload,
    )


(
    DATA_CC_BALLOT_BOX_PAYLOAD,
    DATA_CC_SHUFFLE_PAYLOAD,
    DATA_SC_TALLY_DATA_PAYLOAD,
    DATA_TC_VOTES_PAYLOAD,
) = parse_payload()


def get_election_event_id() -> str:
    """returns election event ID ee"""
    # the str(...) is only here to make mypy happy...
    return str(ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["electionEventId"])


def get_ballot_box_ids() -> tuple[str, ...]:
    """returns ballot box ids"""
```

```
    return tuple(data.ballot_box_id for data in DATA)


def get_psis() -> tuple[int, ...]:
    """returns number of selectable voting options"""
    return tuple(
        get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1]) for data in DATA
    )


def get_pk_bold_ccr() -> ChoiceReturnCodesEncryptionPublicKey:
    """returns choice return codes encryption public key"""
    return ChoiceReturnCodesEncryptionPublicKey(
        tuple(
            int(x, 16)
            for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
                "choiceReturnCodesEncryptionPublicKey"
            ]
        )
    )


def get_delta_hats() -> tuple[int, ...]:
    """returns the number of allowed write-ins + 1 for this specific ballot box"""
    return tuple(get_delta_hat(i=i) for i in range(len(DATA)))


def get_kmaps() -> tuple[dict[str, int], ...]:
    """returns key-value map of the verification card public keys"""
    return tuple(
        get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[data.short_id_vcs()]) for data in DATA
    )


def get_vc_bold_1s() -> tuple[tuple[str, ...], ...]:
    """returns control component’s list of confirmed verification card IDs"""
    return tuple(
        get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1]) for data in DATA
    )


def get_e1_bold_1s() -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
    """returns control component’s list of encrypted, confirmed votes"""
    return tuple(
        get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1]) for data in DATA
    )


def get_e1_bold_tilde_1s() -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
    """returns control component’s list of exponentiated, encrypted, confirmed votes"""
    return tuple(
        get_e1_bold_tilde_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1])
        for data in DATA
    )


def get_e2_bold_1s() -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
```

```
    """returns control component's list of encrypted, partial Choice Return Codes"""
    return tuple(
        get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1])
    for data in DATA
    )


def get_pi_bold_exp_1s() -> tuple[tuple[Proof, ...], ...]:
    """returns control component's list of exponentiation proofs"""
    return tuple(
        get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()
    ][1])
        for data in DATA
    )


def get_pi_bold_eqenc_1s() -> tuple[tuple[Proof2, ...], ...]:
    """returns control component's list of plaintext equality proofs"""
    return tuple(
        get_pi_bold_eqenc_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()
    ][1])
        for data in DATA
    )


def get_c_mixs() -> tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...]:
    """returns preceding shuffled votes"""
    return tuple(get_c_mix(data.short_id_bb()) for data in DATA)


def get_pi_mixs() -> tuple[tuple[ShuffleArgument, ...], ...]:
    """returns preceding shuffled votes"""
    return tuple(get_pi_mix(data.short_id_bb()) for data in DATA)


def get_c_decs() -> tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...]:
    """returns preceding partially decrypted votes"""
    return tuple(get_c_dec(data.short_id_bb()) for data in DATA)


def get_pi_decs() -> tuple[tuple[tuple[Proofs, ...], ...], ...]:
    """returns preceding decryption proofs"""
    return tuple(get_pi_dec(data.short_id_bb()) for data in DATA)


def get_p_tildes() -> tuple[tuple[int, ...], ...]:
    """returns list of actual encoded voting options"""
    return tuple(get_p_tilde(data.get_delta_hat_context) for data in DATA)


def get_v_tildes() -> tuple[tuple[str, ...], ...]:
    """returns list of actual voting options"""
    return tuple(get_v_tilde(data.get_delta_hat_context) for data in DATA)


def test_ok() -> None:
    """
    All the tests that should not fail.
    """
    is_ok = verify_online_control_components(
        group=GROUP,
        ee=get_election_event_id(),
        ballot_box_ids=get_ballot_box_ids(),
```

```python
        psis=get_psis(),
        el_pk=get_el_pk(),
        ccm_el_pk=get_ccm_el_pk(),
        eb_pk=get_eb_pk(),
        pk_bold_ccr=get_pk_bold_ccr(),
        delta_hats=get_delta_hats(),
        kmaps=get_kmaps(),
        vc_bold_1s=get_vc_bold_1s(),
        e1_bold_1s=get_e1_bold_1s(),
        e1_bold_tilde_1s=get_e1_bold_tilde_1s(),
        e2_bold_1s=get_e2_bold_1s(),
        pi_bold_exp_1s=get_pi_bold_exp_1s(),
        pi_bold_eqenc_1s=get_pi_bold_eqenc_1s(),
        c_mixs=get_c_mixs(),
        pi_mixs=get_pi_mixs(),
        c_decs=get_c_decs(),
        pi_decs=get_pi_decs(),
        p_tildes=get_p_tildes(),
        v_tildes=get_v_tildes(),
    )

    assert is_ok


def test_fail() -> None:
    """
    Tests that must fail.
    """

    with capture_logs():
        is_ok = verify_online_control_components(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_ids=get_ballot_box_ids(),
            psis=get_psis(),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
            pk_bold_ccr=get_pk_bold_ccr(),
            delta_hats=get_delta_hats(),
            kmaps=get_kmaps(),
            vc_bold_1s=get_vc_bold_1s(),
            e1_bold_1s=get_e1_bold_1s(),
            e1_bold_tilde_1s=get_e1_bold_tilde_1s(),
            e2_bold_1s=tuple(get_e2_bold_1s()[i] for i in [2, 0, 3, 1]),
            pi_bold_exp_1s=get_pi_bold_exp_1s(),
            pi_bold_eqenc_1s=get_pi_bold_eqenc_1s(),
            c_mixs=get_c_mixs(),
            pi_mixs=get_pi_mixs(),
            c_decs=get_c_decs(),
            pi_decs=get_pi_decs(),
            p_tildes=get_p_tildes(),
            v_tildes=get_v_tildes(),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components should have failed because of mixed e2_bold_1s"

    with capture_logs():
        is_ok = verify_online_control_components(
            group=GROUP,
```

```
            ee=get_election_event_id(),
            ballot_box_ids=get_ballot_box_ids(),
            psis=get_psis(),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
            pk_bold_ccr=get_pk_bold_ccr(),
            delta_hats=get_delta_hats(),
            kmaps=get_kmaps(),
            vc_bold_1s=get_vc_bold_1s(),
            e1_bold_1s=get_e1_bold_1s(),
            e1_bold_tilde_1s=get_e1_bold_tilde_1s(),
            e2_bold_1s=get_e2_bold_1s(),
            pi_bold_exp_1s=get_pi_bold_exp_1s(),
            pi_bold_eqenc_1s=get_pi_bold_eqenc_1s(),
            c_mixs=tuple(get_c_mixs()[i] for i in [2, 3, 0, 1]),
            pi_mixs=get_pi_mixs(),
            c_decs=tuple(get_c_decs()[i] for i in [2, 3, 0, 1]),
            pi_decs=get_pi_decs(),
            p_tildes=get_p_tildes(),
            v_tildes=get_v_tildes(),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components should have because of mixed c_mixs and c_decs"

    with capture_logs():
        is_ok = verify_online_control_components(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_ids=get_ballot_box_ids(),
            psis=get_psis(),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
            pk_bold_ccr=get_pk_bold_ccr(),
            delta_hats=tuple(get_delta_hats()[i] for i in [3, 2, 1, 0]),
            kmaps=get_kmaps(),
            vc_bold_1s=get_vc_bold_1s(),
            e1_bold_1s=tuple(get_e1_bold_1s()[i] for i in [3, 2, 1, 0]),
            e1_bold_tilde_1s=get_e1_bold_tilde_1s(),
            e2_bold_1s=get_e2_bold_1s(),
            pi_bold_exp_1s=get_pi_bold_exp_1s(),
            pi_bold_eqenc_1s=get_pi_bold_eqenc_1s(),
            c_mixs=get_c_mixs(),
            pi_mixs=get_pi_mixs(),
            c_decs=get_c_decs(),
            pi_decs=get_pi_decs(),
            p_tildes=get_p_tildes(),
            v_tildes=get_v_tildes(),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components should have failed because of mixed delta_hats"

    with capture_logs():
        is_ok = verify_online_control_components(
            group=GROUP,
            ee=get_election_event_id(),
```

```
            ballot_box_ids=get_ballot_box_ids(),
            psis=get_psis(),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
            pk_bold_ccr=get_pk_bold_ccr(),
            delta_hats=get_delta_hats(),
            kmaps=get_kmaps(),
            vc_bold_1s=get_vc_bold_1s(),
            e1_bold_1s=get_e1_bold_1s(),
            e1_bold_tilde_1s=get_e1_bold_tilde_1s(),
            e2_bold_1s=get_e2_bold_1s(),
            pi_bold_exp_1s=get_pi_bold_exp_1s(),
            pi_bold_eqenc_1s=tuple(get_pi_bold_eqenc_1s()[i] for i in [2, 1, 3, 0]),
            c_mixs=get_c_mixs(),
            pi_mixs=get_pi_mixs(),
            c_decs=get_c_decs(),
            pi_decs=get_pi_decs(),
            p_tildes=get_p_tildes(),
            v_tildes=get_v_tildes(),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components should have failed because of mixed pi_bold_eqenc_1s"

    with capture_logs():
        is_ok = verify_online_control_components(
            group=GROUP,
            ee=get_election_event_id(),
            ballot_box_ids=get_ballot_box_ids(),
            psis=get_psis(),
            el_pk=get_el_pk(),
            ccm_el_pk=get_ccm_el_pk(),
            eb_pk=get_eb_pk(),
            pk_bold_ccr=get_pk_bold_ccr(),
            delta_hats=get_delta_hats(),
            kmaps=get_kmaps(),
            vc_bold_1s=get_vc_bold_1s(),
            e1_bold_1s=get_e1_bold_1s(),
            e1_bold_tilde_1s=get_e1_bold_tilde_1s(),
            e2_bold_1s=get_e2_bold_1s(),
            pi_bold_exp_1s=get_pi_bold_exp_1s(),
            pi_bold_eqenc_1s=get_pi_bold_eqenc_1s(),
            c_mixs=get_c_mixs(),
            pi_mixs=get_pi_mixs(),
            c_decs=get_c_decs(),
            pi_decs=get_pi_decs(),
            p_tildes=tuple(get_p_tildes()[i] for i in [2, 1, 3, 0]),
            v_tildes=tuple(get_v_tildes()[i] for i in [2, 1, 3, 0]),
        )

    assert (
        is_ok is False
    ), "verify_online_control_components should have failed because of mixed pTable"


if __name__ == "__main__":
    test_ok()
    test_fail()

```

Listing A.10. Tests VerifyOnlineControlComponents

Bibliography

[1] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings* (D. Pointcheval and T. Johansson, eds.), vol. 7237 of *Lecture Notes in Computer Science*, pp. 263–280, Springer, 2012.

[2] B. BK, "Milestones," n.d. `https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting/chronik.html`, Accessed on February 5, 2023.

[3] B. BK, "Vote électronique," n.d. `https://www.bk.admin.ch/bk/de/home/politische-rechte/e-voting.html`, Accessed on February 5, 2023.

[4] B. BK, "Überblick," n.d. `https://www.bk.admin.ch/bk/de/home/politische-rechte/e-voting/ueberblick.html`, Accessed on February 5, 2023.

[5] S. Bot, "Cryptographic primitives of the swiss post voting system," 2022. `https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/Crypto-Primitives-Specification.pdf`, Accessed on February 5, 2023.

[6] S. Bot, "Swiss post voting system verifier specification," 2022. `https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/master/System/Verifier_Specification.pdf`, Accessed on February 5, 2023.

[7] S. Bot, "System specification of the swiss post voting system," 2022. `https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/master/System/System_Specification.pdf`, Accessed on February 8, 2023.

[8] S. Bot, "Datasets gitlab," 2023. `https://gitlab.com/swisspost-evoting/verifier/verifier/-/tree/master/datasets`, Accessed on February 8, 2023.

[9] T. S. F. Chancellery, "Federal chancellery ordinance on electronic voting (oev)," 2022. `https://www.fedlex.admin.ch/eli/cc/2022/336/en`, Accessed on February 10, 2023.

[10] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, pp. 84–88, 1981.

[11] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings* (A. M. Odlyzko, ed.), vol. 263 of *Lecture Notes in Computer Science*, pp. 186–194, Springer, 1986.

[12] S. Gandlur and E. Kalinina, "Zero knowledge and fiat-shamir for nizk," 2019. `https://courses.grainger.illinois.edu/cs598dk/fa2019/Files/lecture08.pdf`, Accessed on February 17, 2023.

[13] J. Gerlach and U. Gasser, “Three case studies from switzerland: E-voting.” Berkman Center Research Publication No. 2009-03.1, 2009. https://cyber.harvard.edu/sites/cyber.law.harvard.edu/files/Gerlach-Gasser_SwissCases_Evoting.pdf, Accessed on February 16, 2023.

[14] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof-systems (extended abstract),” in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA* (R. Sedgewick, ed.), pp. 291–304, ACM, 1985.

[15] R. Haenni, E. Dubuis, R. E. Koenig, and P. Locher, “Process models for universally verifiable elections,” in *Electronic Voting - Third International Joint Conference, E-Vote-ID 2018, Bregenz, Austria, October 2-5, 2018, Proceedings* (R. Krimmer, M. Volkamer, V. Cortier, R. Goré, M. Hapsara, U. Serdült, and D. Duenas-Cid, eds.), vol. 11143 of *Lecture Notes in Computer Science*, pp. 84–99, Springer, 2018.

[16] B. Kaufmann, “The way to modern direct democracy in switzerland,” 2019. https://www.houseofswitzerland.org/swissstories/history/way-modern-direct-democracy-switzerland, Accessed on February 5, 2023.

[17] O. of the Swiss Abroad, “E-voting,” n.d. https://www.swisscommunity.org/en/voting-co-determination/political-topics/e-voting, Accessed on February 16, 2023.

[18] S. Post, “Hackers put e-voting system to the test,” 2022. https://www.post.ch/en/about-us/news/2022/hackers-put-e-voting-system-to-the-test, Accessed on February 5, 2023.

[19] B. Rigendinger and S. Jaberg, “Die schweiz hat wieder ein e-voting-system,” 2023. https://www.swissinfo.ch/ger/wirtschaft/schweiz-e-voting-system/48330162, Accessed on April 12, 2023.

[20] A. D. Santis, S. Micali, and G. Persiano, “Non-interactive zero-knowledge proof systems,” in *Advances in Cryptology - CRYPTO ’87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings* (C. Pomerance, ed.), vol. 293 of *Lecture Notes in Computer Science*, pp. 52–72, Springer, 1987.

[21] N. P. Smart, *Cryptography Made Simple*. Information Security and Cryptography, Springer, 2016.

[22] B. Smyth, “A foundation for secret, verifiable elections.” Cryptology ePrint Archive, Paper 2018/225, 2018. https://eprint.iacr.org/2018/225, Accessed on February 10, 2023.

[23] B. Terelius and D. Wikström, “Proofs of restricted shuffles,” in *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings* (D. J. Bernstein and T. Lange, eds.), vol. 6055 of *Lecture Notes in Computer Science*, pp. 100–113, Springer, 2010.

[24] P. Unterschütz, “Wer mitwählen will, muss die regeln einhalten,” 2022. https://www.luzernerzeitung.ch/zentralschweiz/obwalden/gesamterneuerungswahlen-wer-mitwaehlen-will-muss-die-regeln-einhalten-ld.2251779, Accessed on February 5, 2023.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ruswil, 25.04.2023
Ort/Datum

M. Guenter
Unterschrift