



^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

1 Introduction	1
1.1 E-voting	1
1.2 (E-)Voting in Switzerland	1
1.3 Swiss Post Voting System	2
1.4 Role of the Verifier in the voting process	2
2 Zero-Knowledge Proofs	3
2.1 Introduction	3
2.1.1 Zero-Knowledge Proof	3
2.1.2 Non-Interactive Zero-Knowledge Proof	4
2.2 NIZKP in Swiss Post Voting System	4
3 Elgamal encryption scheme and mixnets	7
3.1 Multi-recipient Elgamal encryption scheme	7
3.1.1 Key Generation	7
3.1.2 Encryption	7
3.1.3 Decryption	8
3.1.4 Partial Decryption	8
3.2 Mixnet	8
3.2.1 Introduction	8
3.2.2 Re-encryption mixnet	9
3.2.3 Mixnet in the Swiss Post Voting System	9
3.2.4 Example of procedure in re-encryption mixnet	10
3.2.5 Bayer-Groth	11
4 Implemented Algorithms	12
4.1 swiss-post-voting-system package	12
4.2 VerifyProcessPlaintexts	12
4.3 VerifyMixDecOffline	13
4.4 VerifyOnlineControlComponentsBallotBox	13
4.5 VerifyTallyControlComponentBallotBox	15
4.6 VerifyOnlineControlComponents	15
4.7 Lessons learned	15
5 Conclusions	17
A Extra material	18
A.1 Algorithms	18
A.1.1 VerifyProcessPlaintexts	18
A.1.2 VerifyMixDecOffline	19
A.1.3 VerifyOnlineControlComponentsBallotBox	22
A.1.4 VerifyTallyControlComponentsBallotBox	24

A.1.5	VerifyOnlineControlComponents	26
A.2	Tests	27
A.2.1	Tests VerifyProcessPlaintexts	27
A.2.2	Tests VerifyMixDecOffline	32
A.2.3	Tests VerifyOnlineControlComponentsBallotBox	40
A.2.4	Tests VerifyTallyControlComponentsBallotBox	49
A.2.5	Tests VerifyOnlineControlComponents	56

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-knowledge 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.

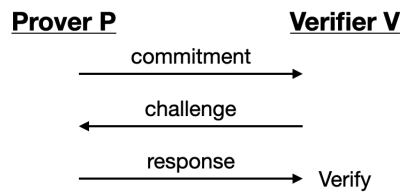


Figure 2.1. Interactions in a zero-knowledge proof of knowledge

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 $e = \text{RecursiveHash}(\phi, y, c, \text{auxiliaryData})$ 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 $\text{RecursiveHash}(\phi, y, c', \text{auxiliaryData}) = e$, the proof is valid

The zero-knowledge proof must satisfy the following criteria:

Completeness

Proof: A verifier checks if $\text{RecursiveHash}(\phi, y, c', \text{auxiliaryData}) = e$ 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.

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

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

```
1: for  $i \in [0, N)$  do
2:    $sk_i \leftarrow \text{GenRandomInteger}(q)$  ▷ see crypto primitives specification
3:    $pk_i \leftarrow g^{sk_i} \pmod p$ 
4: end for
5: return  $(sk, pk)$ 
```

3.1.2 Encryption

Algorithm 2 encrypts a multi-recipient message $m = (m_0, \dots, m_{\ell-1}) \in \mathbb{G}_q^\ell$ consisting of ℓ messages with a public key $pk \in \mathbb{G}_q^N$ and produces a ciphertext $c = (\gamma, \phi_0, \dots, \phi_{\ell-1}) \in \mathbb{H}^\ell$. 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

```
1:  $r \leftarrow \mathbb{Z}_q$ 
2:  $\gamma \leftarrow g^r \pmod p$ 
3: for  $i \in [0, \ell)$  do
4:    $\phi_i \leftarrow pk_i^r \cdot m_i \pmod p$ 
5: end for
6: return  $(\gamma, \phi_0, \dots, \phi_{\ell-1})$ 
```

3.1.3 Decryption

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

Algorithm 3 GetMessage

```
1: for  $i \in [0, \ell)$  do
2:    $m_i \leftarrow \phi_i \cdot \gamma^{-sk_i} \pmod p$ 
3: end for
4: return  $(m_0, \dots, m_{\ell-1})$ 
```

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 $c = (\gamma, \phi_0, \dots, \phi_{\ell-1}) \in \mathbb{H}_\ell$ is partially decrypted with the secret key $sk \in \mathbb{Z}_q^N$ and a ciphertext $c' = (\gamma, m_0, \dots, m_{\ell-1}) \in \mathbb{H}_\ell$ 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

```
1:  $(m_0, \dots, m_{\ell-1}) \leftarrow \mathbf{GetMessage}(c, sk)$ 
2: return  $(\gamma, m_0, \dots, m_{\ell-1})$ 
```

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 $\mathbf{pk} = \prod_{i=1}^n pk_i \pmod p$.

The first mixer in the mixnet performs the following operations:

1. *Mixer 1* receives a list of messages encrypted with the public key $\mathbf{pk} = \prod_{i=1}^n pk_i \pmod p$.
2. *Mixer 1* shuffles the messages and re-encrypts them with the public key \mathbf{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 $pk = \prod_{i=2}^n pk_i \pmod p$ as the public key \mathbf{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 $j = 2, \dots, n$, 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 $pk = \prod_{i=j}^n pk_i \pmod p$. 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 $pk = \prod_{i=j+1}^n pk_i \pmod p$ as the public key \mathbf{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

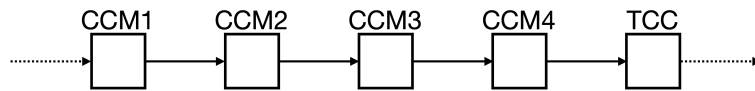


Figure 3.1. Overview of the mixnet components in the Swiss Post Voting System

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

public key $\mathbf{pk}_{EL} = \prod_{i=1}^5 pk_i \pmod p$.

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 $c \in \mathbb{H}_l$.

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

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 $\mathbf{pk} = \prod_{i=1}^5 pk_i \pmod p$ resulting in the ciphertext c_{mix} . In the case of the first mixer, this public key is the same as the election public key.

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

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} .

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

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 $\mathbf{pk} = \prod_{i=5}^5 pk_i \bmod p = pk_5$.

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

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

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

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 $m_i = \prod_{i=1}^{\psi} \hat{p}_i \pmod p$ 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

```

1:  $k \leftarrow 0$ 
2: for  $i \in [0, \hat{N}_C)$  do
3:   if  $m_i \neq 1$  then
4:      $\hat{\mathbf{p}}'_k \leftarrow \mathbf{Factorize}(m_i, \tilde{p}, \psi)$  ▷ see system specification
5:      $\hat{\mathbf{v}}'_k \leftarrow \mathbf{DecodeVotingOptions}(\hat{\mathbf{p}}'_k, pTable)$  ▷ see system specification
6:      $k \leftarrow k + 1$ 
7:   end if
8: end for
9: if  $(\hat{\mathbf{p}}'_0, \dots, \hat{\mathbf{p}}'_{N_C-1}) = L_{votes} \wedge (\hat{\mathbf{v}}'_0, \dots, \hat{\mathbf{v}}'_{N_C-1}) = L_{decodedVotes}$  then
10:  return  $\top$ 
11: else
12:  return  $\perp$ 
13: 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

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

Algorithm 7 VerifyOnlineControlComponentsBallotBox

```
1: if  $N_C \geq 1$  then
2:    $vcProofsVerif \leftarrow \mathbf{VerifyVotingClientProofs}(\mathbf{vc}_1, \mathbf{E1}_1, \widetilde{\mathbf{E1}}_1, \mathbf{E2}_1, \pi_{Exp,1}, \pi_{EqEnc,1},$ 
    $\mathbf{KMap}, \mathbf{pk}_{EL}, \mathbf{pk}_{CCR})$   $\triangleright$  see system specification
3: else
4:    $vcProofsVerif \leftarrow \top$ 
5: end if
6:  $\mathbf{c} \leftarrow \mathbf{GetMixnetInitialCiphertexts}_1(\hat{\delta}, vcMap_1, \mathbf{pk}_{EL})$   $\triangleright$  see system specification
7:  $shuffleProofsVerif \leftarrow \mathbf{VerifyMixDecOffline}(\mathbf{c}, \{\mathbf{c}_{mix,j}\}_{j=1}^4, \{\pi_{mix,j}\}_{j=1}^4, \{\mathbf{c}_{dec,j}\}_{j=1}^4,$ 
    $\{\pi_{dec,j}\}_{j=1}^4, \mathbf{pk}_{EL}, \{pk_j\}_{j=1}^5)$   $\triangleright$  see system specification
8: if  $vcProofsVerif \wedge shuffleProofsVerif$  then
9:   return  $\top$ 
10: else
11:  return  $\perp$ 
12: 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`

```
1: shuffleVerif  $\leftarrow$  VerifyShuffle( $\mathbf{c}_{dec,4}$ ,  $\mathbf{c}_{mix,5}$ ,  $\pi_{mix,5}$ ,  $pk_5$ )  $\triangleright$  see crypto primitives specification
2: decryptVerif  $\leftarrow$  VerifyDecryptions( $\mathbf{c}_{mix,5}$ ,  $pk_5$ ,  $\mathbf{m}$ ,  $\pi_{dec,5}$ ,  $i_{aux}$ )
    $\triangleright$  see crypto primitives specification
3: processVerif  $\leftarrow$  VerifyProcessPlaintexts(pTable,  $\mathbf{m}$ ,  $\psi$ ,  $\hat{\delta}$ ,  $L_{votes}$ ,  $L_{decodedVotes}$ )
    $\triangleright$  see crypto primitives specification
4: if shuffleVerif  $\wedge$  decryptVerif  $\wedge$  processVerif then
5:   return  $\top$ 
6: else
7:   return  $\perp$ 
8: 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`

```
1: for  $i \in [0, N_{bb})$  do
2:   Prepare input for verification of ballot box
3:   bbOnlineCCVerifi  $\leftarrow$  VerifyOnlineControlComponentsBallotBox(input)
    $\triangleright$  see Algorithm 7
4: end for
5: if bbOnlineCCVerifi  $\forall i$  then
6:   return  $\top$ 
7: else
8:   return  $\perp$ 
9: 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

```
1 # final_verification.py
2 def verify_process_plaintexts(
3     group: Group,
4     v_tilde: tuple[str, ...],
5     p_tilde: tuple[int, ...],
6     m: tuple[tuple[int, ...], ...],
7     psi: int,
8     delta_hat: int,
9     l_votes: tuple[tuple[int, ...], ...],
10    l_decoded_votes: tuple[tuple[str, ...], ...],
11 ) -> bool:
12
13     if not all(delta_hat == len(m_i) for m_i in m):
14         msg = f"Requirement 'all(delta_hat == len(m_i) for m_i in m)' not met: {
15             delta_hat=}, {m=}"
16         raise ValueError(msg)
17
18     n_c_hat = len(m)
19     if not n_c_hat >= 2:
20         msg = f"Requirement 'n_c_hat >= 2' not met: {n_c_hat=} elements in {m=}"
21         raise ValueError(msg)
22
23     n_c = len(l_votes)
24     if not n_c == len(l_decoded_votes):
25         msg = (
26             "Requirement 'len(l_votes) == len(l_decoded_votes)' not met: "
27             f"{n_c=} elements in l_votes, "
28             f"{len(l_decoded_votes)=} elements in l_decoded_votes"
29         )
30         raise ValueError(msg)
31
32     if n_c >= 2:
33         if not n_c_hat == n_c:
34             msg = f"Requirement 'n_c_hat == n_c' not met: {n_c_hat=}, {n_c=}"
35             raise ValueError(msg)
36         else:
37             if not n_c_hat == n_c + 2:
38                 msg = f"Requirement 'n_c_hat == n_c + 2' not met :{n_c_hat=}, {n_c=}"
39                 raise ValueError(msg)
40
41     p_hat_lst, v_hat_lst = [], []
42     one_tuple = (1,) * delta_hat
```

```

42     for m_i in m:
43         v_hat_k = []
44         if m_i != one_tuple:
45             p_hat_k = factorize(group=group, x=m_i[0], p_tilde=p_tilde, phi=psi)
46             p_hat_lst.append(p_hat_k)
47             for p in p_hat_k:
48                 v_hat_k.append(v_tilde[p_tilde.index(p)])
49             v_hat_lst.append(tuple(v_hat_k))
50     p_hat = tuple(p_hat_lst)
51     v_hat = tuple(v_hat_lst)
52
53     if p_hat != l_votes:
54         LOGGER.warning(
55             "verify_process_plaintexts failed because p_hat != l_votes",
56         )
57         return False
58
59     if v_hat != l_decoded_votes:
60         LOGGER.warning(
61             "verify_process_plaintexts failed because v_hat != l_decoded_votes",
62         )
63         return False
64
65     return True

```

Listing A.1. VerifyProcessPlaintexts

A.1.2 VerifyMixDecOffline

```

1 # mix_offline.py
2 def verify_mix_dec_offline(
3     group: Group,
4     delta_hat: int,
5     ee: str,
6     ballot_box_id: str,
7     c_init: tuple[MultiRecipientCiphertext, ...],
8     c_mix: tuple[tuple[MultiRecipientCiphertext, ...], ...],
9     pi_mix: tuple[ShuffleArgument, ...],
10    c_dec: tuple[tuple[MultiRecipientCiphertext, ...], ...],
11    pi_dec: tuple[tuple[Proofs, ...], ...],
12    el_pk: ElectionPublicKey,
13    ccm_el_pk: tuple[tuple[int, ...], ...],
14    eb_pk: ElectionPublicKey,
15 ) -> bool:
16
17     # pylint: disable=too-many-locals
18
19     n_c_hat = len(c_init)
20     if not n_c_hat >= 2:
21         msg = f"Requirement 'n_c_hat >= 2' not met: {n_c_hat=} elements in {c_init=}"
22         raise ValueError(msg)
23
24     if not all(len(c_mix_j) == n_c_hat for c_mix_j in c_mix):
25         msg = (
26             f"Requirement '# of ciphertexts in c_mix_j equal to # of initial"
27             f"ciphertexts' not met:"
28             f"{c_mix=}, {c_init=}"
29         )
30         raise ValueError(msg)
31
32     if not all((len(c_dec_j) == n_c_hat for c_dec_j in c_dec)):

```



```

32     msg = (
33         f"Requirement '# of ciphertexts in c_dec_j equal to # of initial
ciphertexts' not met:"
34         f"{c_dec=}, {c_init=}"
35     )
36     raise ValueError(msg)
37
38     if not all((len(pi_dec_j) == n_c_hat for pi_dec_j in pi_dec)):
39         msg = (
40             f"Requirement '# of proofs in pi_dec_j equal to # of initial ciphertexts
' not met:"
41             f"{pi_dec=}, {c_init=}"
42         )
43         raise ValueError(msg)
44
45     if not all((len(proof.z) == delta_hat for proof in pi_dec_j) for pi_dec_j in
pi_dec):
46         msg = f"Requirement 'l == delta_hat' not met: {delta_hat=}, {pi_dec=}"
47         raise ValueError(msg)
48
49     ell = len(pi_dec[0][0].z)
50     delta = len(el_pk.public_key)
51     mu = len(ccm_el_pk[0])
52     if not 0 < ell <= delta <= mu:
53         msg = f"Requirement '0 < l <=      <=      ' not met: {ell=},      = {delta=},      =
{mu=}"
54         raise ValueError(msg)
55
56     if not delta == len(eb_pk.public_key):
57         msg = (
58             f"Requirement 'el_pk and eb_pk both consist of      elements' not met:"
59             f"len(el_pk) = {delta=}, len(eb_pk) = {len(eb_pk.public_key)=}"
60         )
61         raise ValueError(msg)
62
63     if not all(len(ccm_el_pk_j) == mu for ccm_el_pk_j in ccm_el_pk):
64         msg = f"Requirement 'all ccm_el_pk_j contain      elements' not met:      = {mu
=}, {ccm_el_pk=}"
65         raise ValueError(msg)
66
67     ccm_el_pk_prime = tuple(tuple(ccm_el_pl_j[:delta]) for ccm_el_pl_j in ccm_el_pk)
68
69     verify_shuffle_0_ok = verify_shuffle(
70         group=group,
71         ciphertexts=c_init,
72         shuffled_ciphertexts=c_mix[0],
73         shuffle_argument=pi_mix[0],
74         pk=el_pk.public_key,
75     )
76
77     if verify_shuffle_0_ok is False:
78         LOGGER.warning(
79             "verify_mix_dec_offline failed because 'verify_shuffle' failed for j=1",
80             # args:
81             group=group,
82             delta_hat=delta_hat,
83             ee=ee,
84             ballot_box_id=ballot_box_id,
85         )
86         return False
87
88     verify_decryptions_ok = verify_decryptions(
89         group=group,

```

```

90     c=c_mix[0],
91     pk=ccm_el_pk_prime[0],
92     c_prime=c_dec[0],
93     pi_dec=pi_dec[0],
94     i_aux=(ee, ballot_box_id, "MixDecOnline", "1"),
95 )
96
97 if verify_decryptions_ok is False:
98     LOGGER.warning(
99         "verify_mix_dec_offline failed because 'verify_decryptions' failed for j
100 =1",
101         # args:
102         group=group,
103         delta_hat=delta_hat,
104         ee=ee,
105         ballot_box_id=ballot_box_id,
106     )
107     return False
108
109 for j in range(2, 5):
110     el_pk_combined = combine_public_keys(
111         group=group, public_keys=ccm_el_pk_prime[j - 1 :] + (eb_pk.public_key,)
112     )
113     verify_shuffle_j_ok = verify_shuffle(
114         group=group,
115         ciphertexts=c_dec[j - 2],
116         shuffled_ciphertexts=c_mix[j - 1],
117         shuffle_argument=pi_mix[j - 1],
118         pk=el_pk_combined,
119     )
120
121     if verify_shuffle_j_ok is False:
122         LOGGER.warning(
123             f"verify_mix_dec_offline failed because 'verify_decryptions' failed
124 for {j=}",
125             # args:
126             group=group,
127             delta_hat=delta_hat,
128             ee=ee,
129             ballot_box_id=ballot_box_id,
130         )
131         return False
132
133     decrypt_verif_j_ok = verify_decryptions(
134         group=group,
135         c=c_mix[j - 1],
136         pk=ccm_el_pk_prime[j - 1],
137         c_prime=c_dec[j - 1],
138         pi_dec=pi_dec[j - 1],
139         i_aux=(ee, ballot_box_id, "MixDecOnline", str(j)),
140     )
141
142     if decrypt_verif_j_ok is False:
143         LOGGER.warning(
144             f"verify_mix_dec_offline failed because 'verify_decryptions' failed
145 for {j=}",
146             # args:
147             group=group,
148             delta_hat=delta_hat,
149             ee=ee,
150             ballot_box_id=ballot_box_id,

```

```

150     )
151     return False
152
153     return True

```

Listing A.2. VerifyMixDecOffline

A.1.3 VerifyOnlineControlComponentsBallotBox

```

1 # final_verification.py
2 def verify_online_control_components_ballot_box(
3     group: Group,
4     ee: str,
5     ballot_box_id: str,
6     psi: int,
7     el_pk: ElectionPublicKey,
8     ccm_el_pk: tuple[tuple[int, ...], ...],
9     eb_pk: ElectionPublicKey,
10    pk_bold_ccr: ChoiceReturnCodesEncryptionPublicKey,
11    delta_hat: int,
12    kmap: dict[str, int],
13    vc_bold_1: tuple[str, ...],
14    e1_bold_1: tuple[MultiRecipientCiphertext, ...],
15    e1_bold_tilde_1: tuple[MultiRecipientCiphertext, ...],
16    e2_bold_1: tuple[MultiRecipientCiphertext, ...],
17    pi_bold_exp_1: tuple[Proof, ...],
18    pi_bold_eqenc_1: tuple[Proof2, ...],
19    c_mix: tuple[tuple[MultiRecipientCiphertext, ...], ...],
20    pi_mix: tuple[ShuffleArgument, ...],
21    c_dec: tuple[tuple[MultiRecipientCiphertext, ...], ...],
22    pi_dec: tuple[tuple[Proofs, ...], ...],
23    p_tilde: tuple[int, ...],
24    v_tilde: tuple[str, ...],
25 ) -> bool:
26
27     if not all((len(proof.z) == delta_hat for proof in pi_dec_j) for pi_dec_j in
28 pi_dec):
29         msg = f"Requirement 'l == delta_hat' not met: {delta_hat=}, {pi_dec=}"
30         raise ValueError(msg)
31
32     n_c_hat = len(c_mix[0])
33     if not all(
34         len(c_mix_j) == len(c_dec_j) == len(pi_dec_j)
35         for (c_mix_j, c_dec_j, pi_dec_j) in zip(c_mix, c_dec, pi_dec)
36     ):
37         msg = (
38             "Requirement 'c_mix_j, c_dec_j and pi_dec_j are of size n_c_hat for all
39 j' not met: "
40             f"{n_c_hat=}, {c_mix=}, {c_dec=}, {pi_dec=}"
41         )
42         raise ValueError(msg)
43
44     if not n_c_hat >= 2:
45         msg = f"Requirement 'n_c_hat >= 2' not met: {n_c_hat=}"
46         raise ValueError(msg)
47
48     n_c = len(vc_bold_1)
49     if (
50         not len(vc_bold_1)
51         == len(e1_bold_1)
52         == len(e1_bold_tilde_1)
53         == len(e2_bold_1)

```

```

52     == len(pi_bold_exp_1)
53     == len(pi_bold_eqenc_1)
54 ):
55     msg = (
56         "Requirement 'vc_bold_1, e1_bold_1, e1_bold_tilde_1, e2_bold_1,
pi_bold_exp_1, "
57         f"pi_bold_eqenc_1 contain {n_c=} elements each' not met"
58     )
59     raise ValueError(msg)
60
61     if n_c >= 2:
62         if not n_c_hat == n_c:
63             msg = f"Requirement 'n_c_hat == n_c if n_c >= 2' not met: {n_c_hat=}, {
n_c=}"
64             raise ValueError(msg)
65         else:
66             if not n_c_hat == n_c + 2:
67                 msg = f"Requirement 'n_c_hat == n_c + 2 if n_c < 2' not met: {n_c_hat=},
{n_c=}"
68                 raise ValueError(msg)
69
70     if not len(set(vc_bold_1)) == len(vc_bold_1):
71         msg = "Requirement 'all elements in vc_bold_1 are distinct' not met"
72         raise ValueError(msg)
73
74     vc_map = {}
75     for (vc, e_1) in zip(vc_bold_1, e1_bold_1):
76         vc_map[vc] = e_1
77     n_c = len(vc_map)
78
79     if n_c >= 1:
80         verify_voting_client_proofs_ok = verify_voting_client_proofs(
81             group=group,
82             vc_bold_1=vc_bold_1,
83             e1_bold_1=e1_bold_1,
84             e1_bold_tilde_1=e1_bold_tilde_1,
85             e2_bold_1=e2_bold_1,
86             pi_bold_exp_1=pi_bold_exp_1,
87             pi_bold_eqenc_1=pi_bold_eqenc_1,
88             kmap=kmap,
89             el_pk=el_pk,
90             pk_bold_ccr=pk_bold_ccr,
91             delta_hat=delta_hat,
92             psi=psi,
93             ee=ee,
94             p_tilde=p_tilde,
95             v_tilde=v_tilde,
96         )
97     else:
98         verify_voting_client_proofs_ok = True
99
100     if verify_voting_client_proofs_ok is False:
101         LOGGER.warning(
102             "verify_online_control_components_ballot_box failed because "
103             "verify_voting_client_proofs failed",
104         )
105     return False
106
107     verify_mix_dec_offline_ok = verify_mix_dec_offline(
108         group=group,
109         delta_hat=delta_hat,
110         ee=ee,
111         ballot_box_id=ballot_box_id,

```

```

112     c_init=get_mixnet_initial_ciphertexts(
113         group=group, delta_hat=delta_hat, vc_map=vc_map, el_pk=el_pk
114     ),
115     c_mix=c_mix,
116     pi_mix=pi_mix,
117     c_dec=c_dec,
118     pi_dec=pi_dec,
119     el_pk=el_pk,
120     ccm_el_pk=ccm_el_pk,
121     eb_pk=eb_pk,
122 )
123
124 if verify_mix_dec_offline_ok is False:
125     LOGGER.warning(
126         "verify_online_control_components_ballot_box failed because "
127         "verify_mix_dec_offline failed",
128     )
129     return False
130
131 return True

```

Listing A.3. VerifyOnlineControlComponentsBallotBox

A.1.4 VerifyTallyControlComponentsBallotBox

```

1 # final_verification.py
2 def verify_tally_control_component_ballot_box(
3     group: Group,
4     ee: str,
5     ballot_box_id: str,
6     eb_pk: ElectionPublicKey,
7     v_tilde: tuple[str, ...],
8     p_tilde: tuple[int, ...],
9     psi: int,
10    delta_hat: int,
11    c_dec_4: tuple[MultiRecipientCiphertext, ...],
12    c_mix_5: tuple[MultiRecipientCiphertext, ...],
13    pi_mix_5: ShuffleArgument,
14    m: tuple[tuple[int, ...], ...],
15    pi_dec_5: tuple[Proofs, ...],
16    l_votes: tuple[tuple[int, ...], ...],
17    l_decoded_votes: tuple[tuple[str, ...], ...],
18 ) -> bool:
19
20     if not all(len(proof.z) == delta_hat for proof in pi_dec_5):
21         msg = f"Requirement 'l == delta_hat' not met: {delta_hat=}, {pi_dec_5=}"
22         raise ValueError(msg)
23
24     if not all(len(m_i) == delta_hat for m_i in m):
25         msg = f"Requirement 'all messages in m are of size delta_hat' not met: {
26         delta_hat=}, {m=}"
27         raise ValueError(msg)
28
29     n_c_hat = len(c_dec_4)
30     if not len(c_mix_5) == len(m) == len(pi_dec_5) == n_c_hat:
31         msg = (
32             "Requirement 'c_dec_4, c_mix_5, m and pi_dec_5 are of size n_c_hat for
33             all j' not met: "
34             f"{c_dec_4=}, {n_c_hat=}, {c_mix_5=}, {c_dec_4=}, {pi_dec_5=}"
35         )
36         raise ValueError(msg)

```

```

36     if not n_c_hat >= 2:
37         msg = f"Requirement 'n_c_hat >= 2' not met: {n_c_hat}"
38         raise ValueError(msg)
39
40     n_c = len(l_votes)
41     if not n_c == len(l_decoded_votes):
42         msg = (
43             "Requirement 'l_votes and l_decoded contain the same amount of elements'
44             not met: "
45             f"{l_votes=}, {l_decoded_votes=}"
46         )
47         raise ValueError(msg)
48
49     if n_c >= 2:
50         if not n_c_hat == n_c:
51             msg = f"Requirement 'n_c_hat == n_c if n_c >= 2' not met: {n_c_hat=}, {
52                 n_c=}"
53             raise ValueError(msg)
54         else:
55             if not n_c_hat == n_c + 2:
56                 msg = f"Requirement 'n_c_hat == n_c + 2 if n_c < 2' not met: {n_c_hat=},
57                     {n_c=}"
58                 raise ValueError(msg)
59
60     if not all(set(p_i).issubset(p_tilde) for p_i in l_votes):
61         msg = (
62             "Requirement 'selected voting options are a subset of voting options'
63             not met: "
64             f"{l_votes=}, {p_tilde=}"
65         )
66         raise ValueError(msg)
67
68     if not all(len(p_i) == len(set(p_i)) for p_i in l_votes):
69         msg = (
70             "Requirement 'a vote's selected encoded voting options must be distinct'
71             not met: "
72             f"{l_votes=}"
73         )
74         raise ValueError(msg)
75
76     i_aux = (ee, ballot_box_id, "MixDecOffline")
77     eb_pk_cut = eb_pk.public_key[:delta_hat]
78
79     verify_shuffle_ok = verify_shuffle(
80         group=group,
81         ciphertexts=c_dec_4,
82         shuffled_ciphertexts=c_mix_5,
83         shuffle_argument=pi_mix_5,
84         pk=eb_pk_cut,
85     )
86
87     if verify_shuffle_ok is False:
88         LOGGER.warning(
89             "verify_tally_control_component_ballot_box failed because verify_shuffle
90             failed",
91             group=group,
92             ee=ee,
93             ballot_box_id=ballot_box_id,
94             eb_pk=eb_pk,
95             # ...more data?
96         )
97     return False
98

```

```

93     c_prime = tuple(MultiRecipientCiphertext(gamma=c.gamma, phis=m_i) for c, m_i in
94     zip(c_mix_5, m))
95
96     verify_decryptions_ok = verify_decryptions(
97         group=group,
98         c=c_mix_5,
99         pk=eb_pk_cut,
100        c_prime=c_prime,
101        pi_dec=pi_dec_5,
102        i_aux=i_aux,
103    )
104
105    if verify_decryptions_ok is False:
106        LOGGER.warning(
107            "verify_tally_control_component_ballot_box failed because
108            verify_decryptions failed",
109            group=group,
110            ee=ee,
111            ballot_box_id=ballot_box_id,
112            eb_pk=eb_pk,
113            # ...more data?
114            c_prime=c_prime,
115            i_aux=i_aux,
116        )
117
118        return False
119
120    verify_process_plaintexts_ok = verify_process_plaintexts(
121        group=group,
122        v_tilde=v_tilde,
123        p_tilde=p_tilde,
124        m=m,
125        psi=psi,
126        delta_hat=delta_hat,
127        l_votes=l_votes,
128        l_decoded_votes=l_decoded_votes,
129    )
130
131    if verify_process_plaintexts_ok is False:
132        LOGGER.warning(
133            "verify_tally_control_component_ballot_box failed because
134            verify_decryptions failed",
135            group=group,
136            ee=ee,
137            ballot_box_id=ballot_box_id,
138            eb_pk=eb_pk,
139            # ...more data?
140        )
141
142        return False
143
144    return True

```

Listing A.4. VerifyTallyControlComponentsBallotBox

A.1.5 VerifyOnlineControlComponents

```

1 # final_verification.py
2 def verify_online_control_components(
3     group: Group,
4     ee: str,
5     ballot_box_ids: tuple[str, ...],
6     psis: tuple[int, ...],

```

```

7   el_pk: ElectionPublicKey,
8   ccm_el_pk: tuple[tuple[int, ...], ...],
9   eb_pk: ElectionPublicKey,
10  pk_bold_ccr: ChoiceReturnCodesEncryptionPublicKey,
11  delta_hats: tuple[int, ...],
12  kmaps: tuple[dict[str, int], ...],
13  vc_bold_1s: tuple[tuple[str, ...], ...],
14  e1_bold_1s: tuple[tuple[MultiRecipientCiphertext, ...], ...],
15  e1_bold_tilde_1s: tuple[tuple[MultiRecipientCiphertext, ...], ...],
16  e2_bold_1s: tuple[tuple[MultiRecipientCiphertext, ...], ...],
17  pi_bold_exp_1s: tuple[tuple[Proof, ...], ...],
18  pi_bold_eqenc_1s: tuple[tuple[Proof2, ...], ...],
19  c_mixs: tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...],
20  pi_mixs: tuple[tuple[ShuffleArgument, ...], ...],
21  c_decs: tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...],
22  pi_decs: tuple[tuple[tuple[Proofs, ...], ...], ...],
23  p_tildes: tuple[tuple[int, ...], ...],
24  v_tildes: tuple[tuple[str, ...], ...],
25 ) -> bool:
26
27     for j in range(len(ballot_box_ids)):
28         if not verify_online_control_components_ballot_box(
29             group=group,
30             ee=ee,
31             ballot_box_id=ballot_box_ids[j],
32             psi=psis[j],
33             el_pk=el_pk,
34             ccm_el_pk=ccm_el_pk,
35             eb_pk=eb_pk,
36             pk_bold_ccr=pk_bold_ccr,
37             delta_hat=delta_hats[j],
38             kmap=kmaps[j],
39             vc_bold_1=vc_bold_1s[j],
40             e1_bold_1=e1_bold_1s[j],
41             e1_bold_tilde_1=e1_bold_tilde_1s[j],
42             e2_bold_1=e2_bold_1s[j],
43             pi_bold_exp_1=pi_bold_exp_1s[j],
44             pi_bold_eqenc_1=pi_bold_eqenc_1s[j],
45             c_mix=c_mixs[j],
46             pi_mix=pi_mixs[j],
47             c_dec=c_decs[j],
48             pi_dec=pi_decs[j],
49             p_tilde=p_tildes[j],
50             v_tilde=v_tildes[j],
51         ):
52             LOGGER.warning(
53                 "verify_online_control_components failed because ",
54             )
55             return False
56     return True

```

Listing A.5. VerifyOnlineControlComponents

A.2 Tests

A.2.1 Tests VerifyProcessPlaintexts

```

1 # test_verify_process_plaintexts.py
2
3 import json
4 from dataclasses import dataclass
5 from pathlib import Path

```



```

6 from typing import Final
7
8 from structlog.testing import capture_logs
9 from swiss_post_voting_system.crypto_primitives.elgamal import Group
10 from swiss_post_voting_system.verifier.final_verification import
    verify_process_plaintexts
11
12 from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
13
14 # electionEventContextPayload.json
15 data_electionEventContextPayload = json.loads(
16     (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
17 )
18
19 TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"
20
21 GROUP: Final[Group] = Group.from_dict(dct=data_electionEventContextPayload["
    encryptionGroup"])
22
23
24 @dataclass(frozen=True, slots=True)
25 class Data:
26     """
27     Data for the tests
28     """
29
30     ballot_box_id: str
31     get_delta_hat_context: int
32
33     def short_id_bb(self) -> str:
34         """
35         return the first 4 chars of the ballot_box_id
36         """
37         return self.ballot_box_id[:4]
38
39
40 DATA = (
41     Data(ballot_box_id="4120f03ccc8641389adf907c8c80f205", get_delta_hat_context=0),
42     Data(ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d", get_delta_hat_context=1),
43     Data(ballot_box_id="4600fb57269a426695193b57f694ed1c", get_delta_hat_context=2),
44     Data(ballot_box_id="1620dc54f5a147d492668dd34280261d", get_delta_hat_context=3),
45 )
46
47
48 def parse_payload() -> tuple[dict, dict, dict, dict]:
49     """
50     Parsing the payload
51     """
52
53     data_cc_ballot_box_payload: dict[str, dict] = {}
54     data_cc_shuffle_payload: dict[str, dict] = {}
55     data_tally_component_shuffle_payload: dict[str, dict] = {}
56     data_tally_component_votes_payload: dict[str, dict] = {}
57
58     for ballot_box_path in TALLY_BOXES_DIR.iterdir():
59         ballot_box_short = ballot_box_path.name[:4]
60
61         data_cc_ballot_box_payload[ballot_box_short] = {}
62         data = data_cc_ballot_box_payload[ballot_box_short]
63         for j in range(1, 5):
64             data[j] = json.loads(
65                 (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").
                    read_text()

```

```

66     )
67
68     data_cc_shuffle_payload[ballot_box_short] = {}
69     data = data_cc_shuffle_payload[ballot_box_short]
70     for j in range(1, 5):
71         data[j] = json.loads(
72             (ballot_box_path / f"controlComponentShufflePayload_{j}.json").
read_text()
73         )
74
75     data_tally_component_shuffle_payload[ballot_box_short] = json.loads(
76         (ballot_box_path / "tallyComponentShufflePayload.json").read_text()
77     )
78
79     data_tally_component_votes_payload[ballot_box_short] = json.loads(
80         (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
81     )
82     return (
83         data_cc_ballot_box_payload,
84         data_cc_shuffle_payload,
85         data_tally_component_shuffle_payload,
86         data_tally_component_votes_payload,
87     )
88
89
90 (
91     DATA_CC_BALLOT_BOX_PAYLOAD,
92     DATA_CC_SHUFFLE_PAYLOAD,
93     DATA_TC_SHUFFLE_PAYLOAD,
94     DATA_TC_VOTES_PAYLOAD,
95 ) = parse_payload()
96
97
98 def get_delta_hat(i: int) -> int:
99     """returns the number of allowed write-ins + 1 for this specific ballot box"""
100     return int(
101         data_electionEventContextPayload["electionEventContext"]["
verificationCardSetContexts"][i][
102             "numberOfWriteInFields"
103         ]
104         + 1
105     )
106
107
108 def get_v_tilde(i: int) -> tuple[str, ...]:
109     """returns list of actual voting options"""
110     v_tilde_lst = []
111     for k in data_electionEventContextPayload["electionEventContext"][
"verificationCardSetContexts"
112 ] [i] ["primesMappingTable"] ["pTable"]:
113         v_tilde_lst.append(k["actualVotingOption"])
114     return tuple(v_tilde_lst)
115
116
117
118 def get_p_tilde(i: int) -> tuple[int, ...]:
119     """returns list of actual encoded voting options"""
120     p_tilde_lst = []
121     for k in data_electionEventContextPayload["electionEventContext"][
"verificationCardSetContexts"
122 ] [i] ["primesMappingTable"] ["pTable"]:
123         p_tilde_lst.append(k["encodedVotingOption"])
124     return tuple(p_tilde_lst)
125
126

```

```

127
128 def get_m(json_data: dict) -> tuple[tuple[int, ...], ...]:
129     """returns the list of plaintext votes"""
130     return tuple(
131         tuple(int(i, 16) for i in k["message"])
132         for k in json_data["verifiablePlaintextDecryption"]["decryptedVotes"]
133     )
134
135
136 def get_write_in_voting_options(i: int) -> tuple[int, ...]:
137     """returns write-in voting options"""
138     p_tilde_write_ins_lst = []
139     for k in data_electionEventContextPayload["electionEventContext"][
140         "verificationCardSetContexts"
141     ][i]["primesMappingTable"]["pTable"]:
142         if str(k["actualVotingOption"]).startswith("WRITE_IN_"):
143             p_tilde_write_ins_lst.append(k["encodedVotingOption"])
144     return tuple(p_tilde_write_ins_lst)
145
146
147 def get_l_votes(json_data: dict) -> tuple[tuple[int, ...], ...]:
148     """returns list of all selected encoded voting options"""
149     return tuple(tuple(i) for i in json_data["votes"])
150
151
152 def get_l_decoded_votes(json_data: dict) -> tuple[tuple[str, ...], ...]:
153     """returns list of all selected decoded voting options"""
154     return tuple(tuple(i) for i in json_data["actualSelectedVotingOptions"])
155
156
157 def get_l_write_ins(json_data: dict) -> tuple[tuple[str, ...], ...]:
158     """returns list of all selected decoded write-in options"""
159     return tuple(tuple(i) for i in json_data["decodedWriteInVotes"])
160
161
162 def get_psi(json_data: dict) -> int:
163     """returns the number of selectable voting options"""
164     return len(json_data["votes"][0])
165
166
167 def test_ok() -> None:
168     """
169     All the tests that should not fail.
170     """
171
172     for data in DATA:
173         short_id_bb = data.short_id_bb()
174
175         is_ok = verify_process_plaintexts(
176             group=GROUP,
177             v_tilde=get_v_tilde(data.get_delta_hat_context),
178             p_tilde=get_p_tilde(data.get_delta_hat_context),
179             m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
180             psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
181             delta_hat=get_delta_hat(data.get_delta_hat_context),
182             l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
183             l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
184         )
185
186         assert is_ok
187
188

```

```

189 def test_fail() -> None:
190     """
191     Tests that must fail.
192     """
193
194     with capture_logs():
195         data = DATA[1]
196         short_id_bb = data.short_id_bb()
197         manipulated_m_1 = (
198             (10865, 1, 1),
199             (10865, 38369, 1),
200             (38369, 13365276363158976492414625067467080921481393000077236122849, 1),
201         )
202         is_ok = verify_process_plaintexts(
203             group=GROUP,
204             v_tilde=get_v_tilde(data.get_delta_hat_context),
205             p_tilde=get_p_tilde(data.get_delta_hat_context),
206             m=manipulated_m_1,
207             psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
208             delta_hat=get_delta_hat(data.get_delta_hat_context),
209             l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
210             l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb])),
211         )
212
213         assert is_ok is False, "verify_process_plaintexts should have failed due to
wrong m"
214
215         with capture_logs():
216             data = DATA[3]
217             short_id_bb = data.short_id_bb()
218             manipulated_l_votes = ((5, 59, 43), (5, 37, 53), (5, 73, 43))
219             manipulated_l_decoded_votes = (
220                 ("b9c57a40-a555-35e1-972c-a1a6b7e03381", "1-4", "1-5"),
221                 (
222                     "b9c57a40-a555-35e1-972c-a1a6b7e03381",
223                     "WRITE_IN_02c52035069d4d8ab78892b8882ec83b",
224                     "1-2",
225                 ),
226                 ("b9c57a40-a555-35e1-972c-a1a6b7e03381", "ae3a5d49-b15d-3e88-8bdd-
cbf965497a8c", "1-5"),
227             )
228             is_ok = verify_process_plaintexts(
229                 group=GROUP,
230                 v_tilde=get_v_tilde(data.get_delta_hat_context),
231                 p_tilde=get_p_tilde(data.get_delta_hat_context),
232                 m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
233                 psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
234                 delta_hat=get_delta_hat(data.get_delta_hat_context),
235                 l_votes=manipulated_l_votes,
236                 l_decoded_votes=manipulated_l_decoded_votes,
237             )
238
239             assert (
240                 is_ok is False
241             ), "verify_process_plaintexts should have failed due to wrong l_votes and
l_decoded_votes"
242
243             with capture_logs():
244                 data = DATA[0]
245                 short_id_bb = data.short_id_bb()
246                 manipulated_m_2 = ((17, 19), (17, 5), (5, 17))
247                 manipulated_delta_hat = 2

```

```

248     is_ok = verify_process_plaintexts(
249         group=GROUP,
250         v_tilde=get_v_tilde(data.get_delta_hat_context),
251         p_tilde=get_p_tilde(data.get_delta_hat_context),
252         m=manipulated_m_2,
253         psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
254         delta_hat=manipulated_delta_hat,
255         l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
256         l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
257     )
258
259     assert (
260         is_ok is False
261     ), "verify_process_plaintexts should have failed because of wrong m and
delta_hat"
262
263     with capture_logs():
264         data = DATA[2]
265         short_id_bb = data.short_id_bb()
266         manipulated_v_tilde = (
267             "8814c2b6-8c73-38e8-99e6-830bffd32c6",
268             "57a30570-1722-3a7e-a8f9-7dd643d7f339",
269             "b9c57a40-a555-35e1-972c-a1a6b7e03381",
270         )
271         manipulated_p_tilde = (17, 5, 19)
272         is_ok = verify_process_plaintexts(
273             group=GROUP,
274             v_tilde=manipulated_v_tilde,
275             p_tilde=manipulated_p_tilde,
276             m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
277             psi=get_psi(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
278             delta_hat=get_delta_hat(data.get_delta_hat_context),
279             l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
280             l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
281         )
282         assert is_ok is False, "verify_process_plaintexts should have failed because of
wrong pTable"
283
284
285 if __name__ == "__main__":
286     test_ok()
287     test_fail()
288

```

Listing A.6. Tests VerifyProcessPlaintexts

A.2.2 Tests VerifyMixDecOffline

```

1 # test_verify_mix_dec_offline.py
2 import json
3 from dataclasses import dataclass
4 from pathlib import Path
5 from typing import Any, Final
6
7 from structlog.testing import capture_logs
8 from swiss_post_voting_system.crypto_primitives.elgamal import (
9     ElectionPublicKey,
10     Group,
11     MultiRecipientCiphertext,
12 )

```

```

13 from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
14     MultiExponentiationArgument,
15     ProductArgument,
16     ShuffleArgument,
17     SingleValueProductArgument,
18 )
19 from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proofs
20 from swiss_post_voting_system.system.mix_offline import verify_mix_dec_offline
21 from swiss_post_voting_system.system.mix_online import
22     get_mixnet_initial_ciphertexts
23
24 from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
25
26 TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"
27
28 ELECTION_EVENT_CONTEXT_PAYLOAD_DICT = json.loads(
29     (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
30 )
31
32 GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["
33     encryptionGroup"])
34
35 @dataclass(frozen=True, slots=True)
36 class Data:
37     """
38     Data for the tests
39     """
40     ballot_box_id: str
41     get_delta_hat_context: int
42
43     def short_id(self) -> str:
44         """
45         return the first 4 chars of the ballot_box_id
46         """
47         return self.ballot_box_id[:4]
48
49
50 DATA = (
51     Data(ballot_box_id="4120f03ccc8641389adf907c8c80f205", get_delta_hat_context=0),
52     Data(ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d", get_delta_hat_context=1),
53     Data(ballot_box_id="4600fb57269a426695193b57f694ed1c", get_delta_hat_context=2),
54     Data(ballot_box_id="1620dc54f5a147d492668dd34280261d", get_delta_hat_context=3),
55 )
56
57
58 def parse_payload() -> tuple[dict, dict]:
59     """
60     Parsing the payload
61     """
62
63     data_cc_ballot_box_payload: dict[str, dict] = {}
64     data_cc_shuffle_payload: dict[str, dict] = {}
65     for ballot_box_path in TALLY_BOXES_DIR.iterdir():
66         ballot_box_short = ballot_box_path.name[:4]
67
68         data_cc_ballot_box_payload[ballot_box_short] = {}
69         data = data_cc_ballot_box_payload[ballot_box_short]
70         for j in range(1, 5):
71             data[j] = json.loads(
72                 (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").
73                 read_text())

```

```

73         )
74
75         data_cc_shuffle_payload[ballot_box_short] = {}
76         data = data_cc_shuffle_payload[ballot_box_short]
77         for j in range(1, 5):
78             data[j] = json.loads(
79                 (ballot_box_path / f"controlComponentShufflePayload_{j}.json").
read_text()
80             )
81         return data_cc_ballot_box_payload, data_cc_shuffle_payload
82
83
84 DATA_CC_BALLOT_BOX_PAYLOAD, DATA_CC_SHUFFLE_PAYLOAD = parse_payload()
85
86
87 def get_election_event_id() -> str:
88     """returns election event ID ee"""
89
90     # the str(...) is only here to make mypy happy...
91     return str(ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["
electionEventId"])
92
93
94 def get_c_init(ballot_box: str) -> tuple[MultiRecipientCiphertext, ...]:
95     """returns mix net initial ciphertexts"""
96
97     return get_mixnet_initial_ciphertexts(
98         group=GROUP,
99         delta_hat=get_delta_hat(3),
100         vc_map=get_vc_map(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[ballot_box][1]),
101         el_pk=get_el_pk(),
102     )
103
104
105 def get_delta_hat(i: int) -> int:
106     """returns the number of allowed write-ins + 1 for this specific ballot box"""
107     return int(
108         ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["
verificationCardSetContexts"][
109             i
110         ]["numberOfWriteInFields"]
111         + 1
112     )
113
114
115 def get_vc_map(json_data: dict) -> dict[str, MultiRecipientCiphertext]:
116     """returns vcMap used for the calculation of c_init_1"""
117     vc_map = {}
118     for i in json_data["confirmedEncryptedVotes"]:
119         vc = i["contextIds"]["verificationCardId"]
120         e_1 = MultiRecipientCiphertext(
121             gamma=int(i["encryptedVote"]["gamma"], 16),
122             phis=tuple(int(x, 16) for x in i["encryptedVote"]["phis"]),
123         )
124         vc_map[vc] = e_1
125     return vc_map
126
127
128 # pylint: disable=too-many-branches
129 def get_c_mix(ballot_box: str) -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
130     """returns preceding shuffled votes"""
131     return tuple(get_c_mix_j(DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j in range
(1, 5))

```

```

132
133
134 def get_c_mix_j(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
135     """returns preceding shuffled votes"""
136     c_mix_j_lst = []
137     for i in json_data["verifiableShuffle"]["shuffledCiphertexts"]:
138         c_mix_j_lst.append(
139             MultiRecipientCiphertext(
140                 gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
141             )
142         )
143     return tuple(c_mix_j_lst)
144
145
146 def get_pi_mix(ballot_box: str) -> tuple[ShuffleArgument, ...]:
147     """returns preceding shuffled votes"""
148
149     return tuple(
150         get_shuffle_argument(json_data=DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j
151         in range(1, 5)
152     )
153
154 def get_shuffle_argument(
155     json_data: dict[str, dict[str, dict[str, dict[str, Any]]]]
156 ) -> ShuffleArgument:
157     """returns a preceding shuffle proof"""
158     e_lst = []
159     for i in json_data["verifiableShuffle"]["shuffleArgument"]["
160 multiExponentiationArgument"]["E"]:
161         e_lst.append(
162             MultiRecipientCiphertext(
163                 gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
164             )
165         )
166     e = tuple(e_lst)
167
168     return ShuffleArgument(
169         c_a=tuple(int(i, 16) for i in json_data["verifiableShuffle"]["
170 shuffleArgument"]["c_A"]),
171         c_b=tuple(int(i, 16) for i in json_data["verifiableShuffle"]["
172 shuffleArgument"]["c_B"]),
173         product_argument=ProductArgument(
174             c_b=None,
175             hadamard_arg=None,
176             single_value_product_arg=SingleValueProductArgument(
177                 c_d=int(
178                     json_data["verifiableShuffle"]["shuffleArgument"]["
179 productArgument"][
180 "singleValueProductArgument"
181 ]["c_d"],
182                 16,
183             ),
184             c_lower_delta=int(
185                 json_data["verifiableShuffle"]["shuffleArgument"]["
186 productArgument"][
187 "singleValueProductArgument"
188 ]["c_delta"],
189                 16,
190             ),
191             c_upper_delta=int(
192                 json_data["verifiableShuffle"]["shuffleArgument"]["
193 productArgument"][

```



```

188         "singleValueProductArgument"
189         ][ "c_Delta" ],
190         16,
191     ),
192     a_tilde=tuple(
193         int(i, 16)
194         for i in json_data["verifiableShuffle"]["shuffleArgument"]["
productArgument" ] [
195             "singleValueProductArgument"
196             ][ "a_tilde" ]
197     ),
198     b_tilde=tuple(
199         int(i, 16)
200         for i in json_data["verifiableShuffle"]["shuffleArgument"]["
productArgument" ] [
201             "singleValueProductArgument"
202             ][ "b_tilde" ]
203     ),
204     r_tilde=int(
205         json_data["verifiableShuffle"]["shuffleArgument"]["
productArgument" ] [
206             "singleValueProductArgument"
207             ][ "r_tilde" ],
208         16,
209     ),
210     s_tilde=int(
211         json_data["verifiableShuffle"]["shuffleArgument"]["
productArgument" ] [
212             "singleValueProductArgument"
213             ][ "s_tilde" ],
214         16,
215     ),
216 ),
217 ),
218 multi_exponentiation_argument=MultiExponentiationArgument(
219     c_a_0=int(
220         json_data["verifiableShuffle"]["shuffleArgument"]["
multiExponentiationArgument" ] [
221             "c_A_0"
222         ],
223         16,
224     ),
225     c_b=tuple(
226         int(i, 16)
227         for i in json_data["verifiableShuffle"]["shuffleArgument" ] [
228             "multiExponentiationArgument"
229         ][ "c_B" ]
230     ),
231     e=e,
232     a=tuple(
233         int(i, 16)
234         for i in json_data["verifiableShuffle"]["shuffleArgument" ] [
235             "multiExponentiationArgument"
236         ][ "a" ]
237     ),
238     r=int(
239         json_data["verifiableShuffle"]["shuffleArgument" ] [
multiExponentiationArgument" ] [
240             "r"
241         ],
242         16,
243     ),
244     b=int(

```

```

245         json_data["verifiableShuffle"]["shuffleArgument"]["
multiExponentiationArgument"] [
246             "b"
247         ],
248         16,
249     ),
250     s=int(
251         json_data["verifiableShuffle"]["shuffleArgument"]["
multiExponentiationArgument"] [
252             "s"
253         ],
254         16,
255     ),
256     tau=int(
257         json_data["verifiableShuffle"]["shuffleArgument"]["
multiExponentiationArgument"] [
258             "tau"
259         ],
260         16,
261     ),
262 ),
263 )
264
265
266 def get_c_dec(ballot_box: str) -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
267     """returns preceding partially decrypted votes"""
268
269     return tuple(get_c_dec_j(json_data=DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j
in range(1, 5))
270
271
272 def get_c_dec_j(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
273     """returns preceding partially decrypted votes"""
274     c_dec_j_lst = []
275     for i in json_data["verifiableDecryptions"]["ciphertexts"]:
276         c_dec_j_lst.append(
277             MultiRecipientCiphertext(
278                 gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
279             )
280         )
281     return tuple(c_dec_j_lst)
282
283
284 def get_pi_dec(ballot_box: str) -> tuple[tuple[Proofs, ...], ...]:
285     """returns preceding decryption proofs"""
286
287     return tuple(
288         get_pi_dec_j(json_data=DATA_CC_SHUFFLE_PAYLOAD[ballot_box][j]) for j in
range(1, 5)
289     )
290
291
292 def get_pi_dec_j(json_data: dict) -> tuple[Proofs, ...]:
293     """returns preceding decryption proofs"""
294     pi_dec_j_lst = []
295     for i in json_data["verifiableDecryptions"]["decryptionProofs"]:
296         pi_dec_j_lst.append(Proofs(e=int(i["e"], 16), z=tuple(int(x, 16) for x in i[
"z"])))
297     return tuple(pi_dec_j_lst)
298
299
300 def get_el_pk() -> ElectionPublicKey:
301     """returns election public key"""

```

```

302     return ElectionPublicKey(
303         tuple(
304             int(x, 16)
305             for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
306                 "electionPublicKey"
307             ]
308         )
309     )
310
311
312 def get_ccm_el_pk() -> tuple[tuple[int, ...], ...]:
313     """returns CCM election public keys"""
314
315     return tuple(
316         tuple(int(key, 16) for key in j["ccmjElectionPublicKey"])
317         for j in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
318             "combinedControlComponentPublicKeys"
319         ]
320     )
321
322
323 def get_eb_pk() -> ElectionPublicKey:
324     """returns electoral board public key"""
325     return ElectionPublicKey(
326         tuple(
327             int(x, 16)
328             for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
329                 "electoralBoardPublicKey"
330             ]
331         )
332     )
333
334
335 def test_ok() -> None:
336     """
337     All the tests that should not fail.
338     """
339
340     for data in DATA:
341         short_id = data.short_id()
342         is_ok = verify_mix_dec_offline(
343             group=GROUP,
344             delta_hat=get_delta_hat(data.get_delta_hat_context),
345             ee=get_election_event_id(),
346             ballot_box_id=data.ballot_box_id,
347             c_init=get_c_init(short_id),
348             c_mix=get_c_mix(short_id),
349             pi_mix=get_pi_mix(short_id),
350             c_dec=get_c_dec(short_id),
351             pi_dec=get_pi_dec(short_id),
352             el_pk=get_el_pk(),
353             ccm_el_pk=get_ccm_el_pk(),
354             eb_pk=get_eb_pk(),
355         )
356
357         assert is_ok
358
359
360 def test_fail() -> None:
361     """
362     Tests that must fail.
363     """
364

```

```

365 data = DATA[0]
366 short_id = data.short_id()
367
368 wrong_data = DATA[2]
369 wrong_short_id = wrong_data.short_id()
370
371 el_pk = get_el_pk()
372 eb_pk = get_eb_pk()
373 ccm_el_pk = get_ccm_el_pk()
374
375 with capture_logs():
376     is_ok = verify_mix_dec_offline(
377         group=GROUP,
378         delta_hat=get_delta_hat(data.get_delta_hat_context),
379         ee=get_election_event_id(),
380         ballot_box_id=data.ballot_box_id,
381         c_init=get_c_init(wrong_short_id),
382         c_mix=get_c_mix(short_id),
383         pi_mix=get_pi_mix(short_id),
384         c_dec=get_c_dec(short_id),
385         pi_dec=get_pi_dec(short_id),
386         el_pk=el_pk,
387         ccm_el_pk=ccm_el_pk,
388         eb_pk=eb_pk,
389     )
390
391 assert is_ok is False, "verify_mix_dec_offline should have failed because of
wrong c_init"
392
393 with capture_logs():
394     is_ok = verify_mix_dec_offline(
395         group=GROUP,
396         delta_hat=get_delta_hat(data.get_delta_hat_context),
397         ee=get_election_event_id(),
398         ballot_box_id=data.ballot_box_id,
399         c_init=get_c_init(short_id),
400         c_mix=get_c_mix(wrong_short_id),
401         pi_mix=get_pi_mix(short_id),
402         c_dec=get_c_dec(short_id),
403         pi_dec=get_pi_dec(short_id),
404         el_pk=el_pk,
405         ccm_el_pk=ccm_el_pk,
406         eb_pk=eb_pk,
407     )
408
409 assert is_ok is False, "verify_mix_dec_offline should have failed because of
wrong c_mix"
410
411 with capture_logs():
412     is_ok = verify_mix_dec_offline(
413         group=GROUP,
414         delta_hat=get_delta_hat(data.get_delta_hat_context),
415         ee=get_election_event_id(),
416         ballot_box_id=data.ballot_box_id,
417         c_init=get_c_init(short_id),
418         c_mix=get_c_mix(short_id),
419         pi_mix=get_pi_mix(wrong_short_id),
420         c_dec=get_c_dec(short_id),
421         pi_dec=get_pi_dec(short_id),
422         el_pk=el_pk,
423         ccm_el_pk=ccm_el_pk,
424         eb_pk=eb_pk,
425     )

```

```

426
427     assert is_ok is False, "verify_mix_dec_offline should have failed because of
wrong pi_mix"
428
429     with capture_logs():
430         is_ok = verify_mix_dec_offline(
431             group=GROUP,
432             delta_hat=get_delta_hat(data.get_delta_hat_context),
433             ee=get_election_event_id(),
434             ballot_box_id=data.ballot_box_id,
435             c_init=get_c_init(short_id),
436             c_mix=get_c_mix(short_id),
437             pi_mix=get_pi_mix(short_id),
438             c_dec=get_c_dec(wrong_short_id),
439             pi_dec=get_pi_dec(short_id),
440             el_pk=el_pk,
441             ccm_el_pk=ccm_el_pk,
442             eb_pk=eb_pk,
443         )
444
445     assert is_ok is False, "verify_mix_dec_offline should have failed because of
wrong c_dec"
446
447     with capture_logs():
448         is_ok = verify_mix_dec_offline(
449             group=GROUP,
450             delta_hat=get_delta_hat(data.get_delta_hat_context),
451             ee=get_election_event_id(),
452             ballot_box_id=data.ballot_box_id,
453             c_init=get_c_init(short_id),
454             c_mix=get_c_mix(short_id),
455             pi_mix=get_pi_mix(short_id),
456             c_dec=get_c_dec(short_id),
457             pi_dec=get_pi_dec(wrong_short_id),
458             el_pk=el_pk,
459             ccm_el_pk=ccm_el_pk,
460             eb_pk=eb_pk,
461         )
462
463     assert is_ok is False, "verify_mix_dec_offline should have failed because of
wrong pi_dec"
464
465
466 if __name__ == "__main__":
467     test_ok()
468     test_fail()
469

```

Listing A.7. Tests VerifyMixDecOffline

A.2.3 Tests VerifyOnlineControlComponentsBallotBox

```

1 # test_verify_online_control_components_ballot_box.py
2
3 import json
4 from dataclasses import dataclass
5 from pathlib import Path
6 from typing import Final
7
8 from structlog.testing import capture_logs
9 from swiss_post_voting_system.crypto_primitives.elgamal import (
10     ChoiceReturnCodesEncryptionPublicKey,

```

```

11     Group,
12 )
13 from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
14     MultiRecipientCiphertext,
15 )
16 from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proof,
17     Proof2
18 from swiss_post_voting_system.verifier.final_verification import (
19     verify_online_control_components_ballot_box,
20 )
21 from swiss_post_voting_system_tests.system_tests.test_verify_mix_dec_offline import
22     (
23     get_c_dec,
24     get_c_mix,
25     get_ccm_el_pk,
26     get_eb_pk,
27     get_el_pk,
28     get_pi_dec,
29     get_pi_mix,
30 )
31 from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
32 from swiss_post_voting_system_tests.verifier_tests.test_verify_process_plaintexts
33     import (
34     get_p_tilde,
35     get_v_tilde,
36 )
37 ELECTION_EVENT_CONTEXT_PAYLOAD_DICT = json.loads(
38     (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
39 )
40 SETUP_VCS_DIR: Final[Path] = DATASETS_DIR / "dataset1/setup/verification_card_sets"
41 TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"
42
43 GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["
44     encryptionGroup"])
45
46 @dataclass(frozen=True, slots=True)
47 class Data:
48     """
49     Data for the tests
50     """
51
52     verification_card_set_id: str
53     ballot_box_id: str
54     get_delta_hat_context: int
55
56     def short_id_vcs(self) -> str:
57         """
58         return the first 4 chars of the verification_card_set_id
59         """
60         return self.verification_card_set_id[:4]
61
62     def short_id_bb(self) -> str:
63         """
64         return the first 4 chars of the ballot_box_id
65         """
66         return self.ballot_box_id[:4]
67
68
69 DATA = (

```

```

70 Data(
71     verification_card_set_id="73e2eed19de9494ea9eaf93968e9b428",
72     ballot_box_id="4120f03ccc8641389adf907c8c80f205",
73     get_delta_hat_context=0,
74 ),
75 Data(
76     verification_card_set_id="3880a1b0f49341d68f3c9fec15782063",
77     ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d",
78     get_delta_hat_context=1,
79 ),
80 Data(
81     verification_card_set_id="ae82cc64b620433da892983df6363d8c",
82     ballot_box_id="4600fb57269a426695193b57f694ed1c",
83     get_delta_hat_context=2,
84 ),
85 Data(
86     verification_card_set_id="fe9bb7092993440eb51235f0efa5d19b",
87     ballot_box_id="1620dc54f5a147d492668dd34280261d",
88     get_delta_hat_context=3,
89 ),
90 )
91
92
93 def parse_payload() -> tuple[dict, dict, dict, dict]:
94     """
95     Parsing the payload
96     """
97
98     data_cc_ballot_box_payload: dict[str, dict] = {}
99     data_cc_shuffle_payload: dict[str, dict] = {}
100     data_tally_component_votes_payload: dict[str, dict] = {}
101     for ballot_box_path in TALLY_BOXES_DIR.iterdir():
102         ballot_box_short = ballot_box_path.name[:4]
103
104         data_cc_ballot_box_payload[ballot_box_short] = {}
105         data = data_cc_ballot_box_payload[ballot_box_short]
106         for j in range(1, 5):
107             data[j] = json.loads(
108                 (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").
109                 read_text()
110             )
111
112         data_cc_shuffle_payload[ballot_box_short] = {}
113         data = data_cc_shuffle_payload[ballot_box_short]
114         for j in range(1, 5):
115             data[j] = json.loads(
116                 (ballot_box_path / f"controlComponentShufflePayload_{j}.json").
117                 read_text()
118             )
119
120         data_tally_component_votes_payload[ballot_box_short] = json.loads(
121             (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
122         )
123
124     data_setup_component_tally_data_payload: dict[str, dict] = {}
125     for vcs_path in SETUP_VCS_DIR.iterdir():
126         vcs_short = vcs_path.name[:4]
127
128         data_setup_component_tally_data_payload[vcs_short] = json.loads(
129             (vcs_path / "setupComponentTallyDataPayload.json").read_text()
130         )
131
132     return (

```

```

131     data_cc_ballot_box_payload,
132     data_cc_shuffle_payload,
133     data_setup_component_tally_data_payload,
134     data_tally_component_votes_payload,
135 )
136
137
138 (
139     DATA_CC_BALLOT_BOX_PAYLOAD,
140     DATA_CC_SHUFFLE_PAYLOAD,
141     DATA_SC_TALLY_DATA_PAYLOAD,
142     DATA_TC_VOTES_PAYLOAD,
143 ) = parse_payload()
144
145
146 def get_election_event_id() -> str:
147     """returns election event ID ee"""
148     # the str(...) is only here to make mypy happy...
149     return str(ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["electionEventId"])
150
151
152 def get_psi(json_data: dict) -> int:
153     """returns number of selectable voting options"""
154     return len(json_data["confirmedEncryptedVotes"][0][0]["encryptedPartialChoiceReturnCodes"]["phis"])
155
156
157 def get_pk_bold_ccr() -> ChoiceReturnCodesEncryptionPublicKey:
158     """returns choice return codes encryption public key"""
159     return ChoiceReturnCodesEncryptionPublicKey(
160         tuple(
161             int(x, 16)
162             for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
163                 "choiceReturnCodesEncryptionPublicKey"
164             ]
165         )
166     )
167
168
169 def get_delta_hat(i: int) -> int:
170     """returns the number of allowed write-ins + 1 for this specific ballot box"""
171     return int(
172         ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["verificationCardSetContexts"][
173             i
174         ]["numberOfWriteInFields"]
175         + 1
176     )
177
178
179 def get_kmap(json_data: dict) -> dict[str, int]:
180     """returns key-value map of the verification card public keys"""
181     kmap = {}
182     for (vc_id, public_key) in zip(
183         json_data["verificationCardIds"], json_data["verificationCardPublicKeys"]
184     ):
185         kmap[str(vc_id)] = int(public_key[0], 16)
186     return kmap
187
188
189 def get_vc_bold_l(json_data: dict) -> tuple[str, ...]:
190     """returns control component's list of confirmed verification card IDs"""

```



```

191     return tuple(
192         i["contextIds"]["verificationCardId"] for i in json_data["
confirmedEncryptedVotes"]
193     )
194
195
196 def get_e1_bold_1(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
197     """returns control component's list of encrypted, confirmed votes"""
198     return tuple(
199         MultiRecipientCiphertext(
200             gamma=int(i["encryptedVote"]["gamma"], 16),
201             phis=tuple(int(x, 16) for x in i["encryptedVote"]["phis"]),
202         )
203         for i in json_data["confirmedEncryptedVotes"]
204     )
205
206
207 def get_e1_bold_tilde_1(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
208     """returns control component's list of exponentiated, encrypted, confirmed votes
"""
209     return tuple(
210         MultiRecipientCiphertext(
211             gamma=int(i["exponentiatedEncryptedVote"]["gamma"], 16),
212             phis=tuple(int(x, 16) for x in i["exponentiatedEncryptedVote"]["phis"]),
213         )
214         for i in json_data["confirmedEncryptedVotes"]
215     )
216
217
218 def get_e2_bold_1(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
219     """returns control component's list of encrypted, partial Choice Return Codes"""
220     return tuple(
221         MultiRecipientCiphertext(
222             gamma=int(i["encryptedPartialChoiceReturnCodes"]["gamma"], 16),
223             phis=tuple(int(x, 16) for x in i["encryptedPartialChoiceReturnCodes"]["
phis"]),
224         )
225         for i in json_data["confirmedEncryptedVotes"]
226     )
227
228
229 def get_pi_bold_exp_1(json_data: dict) -> tuple[Proof, ...]:
230     """returns control component's list of exponentiation proofs"""
231     return tuple(
232         Proof(
233             e=int(i["exponentiationProof"]["e"], 16),
234             z=int(i["exponentiationProof"]["z"], 16),
235         )
236         for i in json_data["confirmedEncryptedVotes"]
237     )
238
239
240 def get_pi_bold_eqenc_1(json_data: dict) -> tuple[Proof2, ...]:
241     """returns control component's list of plaintext equality proofs"""
242     return tuple(
243         Proof2(
244             e=int(i["plaintextEqualityProof"]["e"], 16),
245             z=(
246                 int(i["plaintextEqualityProof"]["z"][0], 16),
247                 int(i["plaintextEqualityProof"]["z"][1], 16),
248             ),
249         )
250         for i in json_data["confirmedEncryptedVotes"]

```

```

251 )
252
253
254 def test_ok() -> None:
255     """
256     All the tests that should not fail.
257     """
258
259     for data in DATA:
260         short_id_bb = data.short_id_bb()
261         short_id_vcs = data.short_id_vcs()
262
263         is_ok = verify_online_control_components_ballot_box(
264             group=GROUP,
265             ee=get_election_event_id(),
266             ballot_box_id=data.ballot_box_id,
267             psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
268             el_pk=get_el_pk(),
269             ccm_el_pk=get_ccm_el_pk(),
270             eb_pk=get_eb_pk(),
271             pk_bold_ccr=get_pk_bold_ccr(),
272             delta_hat=get_delta_hat(data.get_delta_hat_context),
273             kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
274             vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
275 ] [1]),
276             el_bold_1=get_el_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
277 ] [1]),
278             el_bold_tilde_1=get_el_bold_tilde_1(
279                 json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
280             ),
281             e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
282 ] [1]),
283             pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[
284 short_id_bb][1]),
285             pi_bold_eqenc_1=get_pi_bold_eqenc_1(
286                 json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
287             ),
288             c_mix=get_c_mix(short_id_bb),
289             pi_mix=get_pi_mix(short_id_bb),
290             c_dec=get_c_dec(short_id_bb),
291             pi_dec=get_pi_dec(short_id_bb),
292             p_tilde=get_p_tilde(data.get_delta_hat_context),
293             v_tilde=get_v_tilde(data.get_delta_hat_context),
294         )
295
296         assert is_ok
297
298
299 def test_fail() -> None:
300     """
301     Tests that must fail.
302     """
303
304     data = DATA[0]
305     short_id_bb = data.short_id_bb()
306     short_id_vcs = data.short_id_vcs()
307
308     wrong_data = DATA[2]
309     wrong_short_id_bb = wrong_data.short_id_bb()
310
311     el_pk = get_el_pk()
312     eb_pk = get_eb_pk()
313     ccm_el_pk = get_ccm_el_pk()

```

```

310 pk_bold_ccr = get_pk_bold_ccr()
311
312 with capture_logs():
313     is_ok = verify_online_control_components_ballot_box(
314         group=GROUP,
315         ee=get_election_event_id(),
316         ballot_box_id=data.ballot_box_id,
317         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
318         el_pk=el_pk,
319         ccm_el_pk=ccm_el_pk,
320         eb_pk=eb_pk,
321         pk_bold_ccr=pk_bold_ccr,
322         delta_hat=get_delta_hat(data.get_delta_hat_context),
323         kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
324         vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
325         e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
326         e1_bold_tilde_1=get_e1_bold_tilde_1(
327             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb][1]
328         ),
329         e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
330         pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[
short_id_bb][1]),
331         pi_bold_eqenc_1=get_pi_bold_eqenc_1(
332             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
333         ),
334         c_mix=get_c_mix(short_id_bb),
335         pi_mix=get_pi_mix(short_id_bb),
336         c_dec=get_c_dec(short_id_bb),
337         pi_dec=get_pi_dec(short_id_bb),
338         p_tilde=get_p_tilde(data.get_delta_hat_context),
339         v_tilde=get_v_tilde(data.get_delta_hat_context),
340     )
341
342 assert (
343     is_ok is False
344 ), "verify_online_control_components_ballot_box should have failed due to wrong
e1_bold_tilde_1"
345
346 with capture_logs():
347     is_ok = verify_online_control_components_ballot_box(
348         group=GROUP,
349         ee=get_election_event_id(),
350         ballot_box_id=data.ballot_box_id,
351         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
352         el_pk=el_pk,
353         ccm_el_pk=ccm_el_pk,
354         eb_pk=eb_pk,
355         pk_bold_ccr=pk_bold_ccr,
356         delta_hat=get_delta_hat(data.get_delta_hat_context),
357         kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
358         vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
359         e1_bold_1=get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
360         e1_bold_tilde_1=get_e1_bold_tilde_1(
361             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
362         ),
363         e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
364         pi_bold_exp_1=get_pi_bold_exp_1(

```

```

365         json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb][1]
366     ),
367     pi_bold_eqenc_1=get_pi_bold_eqenc_1(
368         json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
369     ),
370     c_mix=get_c_mix(short_id_bb),
371     pi_mix=get_pi_mix(short_id_bb),
372     c_dec=get_c_dec(short_id_bb),
373     pi_dec=get_pi_dec(short_id_bb),
374     p_tilde=get_p_tilde(data.get_delta_hat_context),
375     v_tilde=get_v_tilde(data.get_delta_hat_context),
376 )
377
378 assert (
379     is_ok is False
380 ), "verify_online_control_components_ballot_box should have failed due to wrong
pi_bold_exp_1"
381
382 with capture_logs():
383     is_ok = verify_online_control_components_ballot_box(
384         group=GROUP,
385         ee=get_election_event_id(),
386         ballot_box_id=data.ballot_box_id,
387         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
388         el_pk=el_pk,
389         ccm_el_pk=ccm_el_pk,
390         eb_pk=eb_pk,
391         pk_bold_ccr=pk_bold_ccr,
392         delta_hat=get_delta_hat(data.get_delta_hat_context),
393         kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
394         vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
]
] [1]),
395         el_bold_1=get_el_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
]
] [1]),
396         el_bold_tilde_1=get_el_bold_tilde_1(
397             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
398         ),
399         e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
]
] [1]),
400         pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[
short_id_bb][1]),
401         pi_bold_eqenc_1=get_pi_bold_eqenc_1(
402             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]
403         ),
404         c_mix=get_c_mix(short_id_bb),
405         pi_mix=get_pi_mix(wrong_short_id_bb),
406         c_dec=get_c_dec(short_id_bb),
407         pi_dec=get_pi_dec(short_id_bb),
408         p_tilde=get_p_tilde(data.get_delta_hat_context),
409         v_tilde=get_v_tilde(data.get_delta_hat_context),
410     )
411
412 assert (
413     is_ok is False
414 ), "verify_online_control_components_ballot_box should have failed because of
wrong pi_mix"
415
416 with capture_logs():
417     is_ok = verify_online_control_components_ballot_box(
418         group=GROUP,
419         ee=get_election_event_id(),
420         ballot_box_id=data.ballot_box_id,
421         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),

```

```

422     el_pk=el_pk,
423     ccm_el_pk=ccm_el_pk,
424     eb_pk=eb_pk,
425     pk_bold_ccr=pk_bold_ccr,
426     delta_hat=get_delta_hat(data.get_delta_hat_context),
427     kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
428     vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
429     el_bold_1=get_el_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
430     el_bold_tilde_1=get_el_bold_tilde_1(
431         json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb] [1]
432     ),
433     e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[
wrong_short_id_bb] [1]),
434     pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[
short_id_bb] [1]),
435     pi_bold_eqenc_1=get_pi_bold_eqenc_1(
436         json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb] [1]
437     ),
438     c_mix=get_c_mix(short_id_bb),
439     pi_mix=get_pi_mix(short_id_bb),
440     c_dec=get_c_dec(short_id_bb),
441     pi_dec=get_pi_dec(short_id_bb),
442     p_tilde=get_p_tilde(data.get_delta_hat_context),
443     v_tilde=get_v_tilde(data.get_delta_hat_context),
444 )
445
446 assert (
447     is_ok is False
448 ), "verify_online_control_components_ballot_box should have failed because of
wrong e2_bold_1"
449
450 with capture_logs():
451     is_ok = verify_online_control_components_ballot_box(
452         group=GROUP,
453         ee=get_election_event_id(),
454         ballot_box_id=data.ballot_box_id,
455         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb] [1]),
456         el_pk=el_pk,
457         ccm_el_pk=ccm_el_pk,
458         eb_pk=eb_pk,
459         pk_bold_ccr=pk_bold_ccr,
460         delta_hat=get_delta_hat(data.get_delta_hat_context),
461         kmap=get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[short_id_vcs]),
462         vc_bold_1=get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
463         el_bold_1=get_el_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
464         el_bold_tilde_1=get_el_bold_tilde_1(
465             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb] [1]
466         ),
467         e2_bold_1=get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb
] [1]),
468         pi_bold_exp_1=get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[
short_id_bb] [1]),
469         pi_bold_eqenc_1=get_pi_bold_eqenc_1(
470             json_data=DATA_CC_BALLOT_BOX_PAYLOAD[wrong_short_id_bb] [1]
471         ),
472         c_mix=get_c_mix(short_id_bb),
473         pi_mix=get_pi_mix(short_id_bb),
474         c_dec=get_c_dec(short_id_bb),
475         pi_dec=get_pi_dec(short_id_bb),

```

```

476         p_tilde=get_p_tilde(data.get_delta_hat_context),
477         v_tilde=get_v_tilde(data.get_delta_hat_context),
478     )
479
480     assert (
481         is_ok is False
482     ), "verify_online_control_components_ballot_box should have failed due to wrong
pi_bold_eqenc_1"
483
484
485 if __name__ == "__main__":
486     test_ok()
487     test_fail()
488

```

Listing A.8. Tests VerifyOnlineControlComponentsBallotBox

A.2.4 Tests VerifyTallyControlComponentsBallotBox

```

1 # test_verify_tally_control_components_ballot_box.py
2
3 import json
4 from dataclasses import dataclass
5 from pathlib import Path
6 from typing import Any, Final
7
8 from structlog.testing import capture_logs
9 from swiss_post_voting_system.crypto_primitives.elgamal import Group
10 from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
11     MultiExponentiationArgument,
12     MultiRecipientCiphertext,
13     ProductArgument,
14     ShuffleArgument,
15     SingleValueProductArgument,
16 )
17 from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proofs
18 from swiss_post_voting_system.verifier.final_verification import (
19     verify_tally_control_component_ballot_box,
20 )
21
22 from swiss_post_voting_system_tests.system_tests.test_verify_mix_dec_offline import
23     (
24         get_delta_hat,
25         get_eb_pk,
26 )
27 from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
28 from swiss_post_voting_system_tests.verifier_tests.test_verify_process_plaintexts
29     import (
30         get_l_decoded_votes,
31         get_l_votes,
32         get_m,
33         get_p_tilde,
34         get_v_tilde,
35 )
36
37 ELECTION_EVENT_CONTEXT_PAYLOAD = json.loads(
38     (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
39 )
40
41 SETUP_VCS_DIR: Final[Path] = DATASETS_DIR / "dataset1/setup/verification_card_sets"
42 TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"

```

```

42 GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD["
    encryptionGroup"])
43
44
45 @dataclass(frozen=True, slots=True)
46 class Data:
47     """
48     Data for the tests
49     """
50
51     verification_card_set_id: str
52     ballot_box_id: str
53     get_delta_hat_context: int
54
55     def short_id_vcs(self) -> str:
56         """
57         return the first 4 chars of the verification_card_set_id
58         """
59         return self.verification_card_set_id[:4]
60
61     def short_id_bb(self) -> str:
62         """
63         return the first 4 chars of the ballot_box_id
64         """
65         return self.ballot_box_id[:4]
66
67
68 DATA = (
69     Data(
70         verification_card_set_id="73e2eed19de9494ea9eaf93968e9b428",
71         ballot_box_id="4120f03ccc8641389adf907c8c80f205",
72         get_delta_hat_context=0,
73     ),
74     Data(
75         verification_card_set_id="3880a1b0f49341d68f3c9fec15782063",
76         ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d",
77         get_delta_hat_context=1,
78     ),
79     Data(
80         verification_card_set_id="ae82cc64b620433da892983df6363d8c",
81         ballot_box_id="4600fb57269a426695193b57f694ed1c",
82         get_delta_hat_context=2,
83     ),
84     Data(
85         verification_card_set_id="fe9bb7092993440eb51235f0efa5d19b",
86         ballot_box_id="1620dc54f5a147d492668dd34280261d",
87         get_delta_hat_context=3,
88     ),
89 )
90
91
92 def parse_payload() -> tuple[dict, dict, dict, dict, dict]:
93     """
94     Parsing the payload
95     """
96
97     data_cc_ballot_box_payload: dict[str, dict] = {}
98     data_cc_shuffle_payload: dict[str, dict] = {}
99     data_tally_component_shuffle_payload: dict[str, dict] = {}
100     data_tally_component_votes_payload: dict[str, dict] = {}
101     for ballot_box_path in TALLY_BOXES_DIR.iterdir():
102         ballot_box_short = ballot_box_path.name[:4]
103

```

```

104     data_cc_ballot_box_payload[ballot_box_short] = {}
105     data = data_cc_ballot_box_payload[ballot_box_short]
106     for j in range(1, 5):
107         data[j] = json.loads(
108             (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").
read_text()
109         )
110
111     data_cc_shuffle_payload[ballot_box_short] = {}
112     data = data_cc_shuffle_payload[ballot_box_short]
113     for j in range(1, 5):
114         data[j] = json.loads(
115             (ballot_box_path / f"controlComponentShufflePayload_{j}.json").
read_text()
116         )
117
118     data_tally_component_shuffle_payload[ballot_box_short] = json.loads(
119         (ballot_box_path / "tallyComponentShufflePayload.json").read_text()
120     )
121
122     data_tally_component_votes_payload[ballot_box_short] = json.loads(
123         (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
124     )
125
126     data_setup_component_tally_data_payload: dict[str, dict] = {}
127     for vcs_path in SETUP_VCS_DIR.iterdir():
128         vcs_short = vcs_path.name[:4]
129
130         data_setup_component_tally_data_payload[vcs_short] = json.loads(
131             (vcs_path / "setupComponentTallyDataPayload.json").read_text()
132         )
133
134     return (
135         data_cc_ballot_box_payload,
136         data_cc_shuffle_payload,
137         data_setup_component_tally_data_payload,
138         data_tally_component_shuffle_payload,
139         data_tally_component_votes_payload,
140     )
141
142
143 (
144     DATA_CC_BALLOT_BOX_PAYLOAD,
145     DATA_CC_SHUFFLE_PAYLOAD,
146     DATA_SC_TALLY_DATA_PAYLOAD,
147     DATA_TC_SHUFFLE_PAYLOAD,
148     DATA_TC_VOTES_PAYLOAD,
149 ) = parse_payload()
150
151
152 def get_election_event_id() -> str:
153     """returns election event ID ee"""
154     # the str(...) is only here to make mypy happy...
155     return str(ELECTION_EVENT_CONTEXT_PAYLOAD["electionEventContext"]["
electionEventId"])
156
157
158 def get_psi(json_data: dict) -> int:
159     """returns number of selectable voting options"""
160     return len(json_data["confirmedEncryptedVotes"][0][
"encryptedPartialChoiceReturnCodes"]["phis"])
161
162

```



```

163 def get_c_dec_4(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
164     """returns last online control component's partially decrypted votes"""
165     c_dec_4_lst = []
166     for i in json_data["verifiableDecryptions"]["ciphertexts"]:
167         c_dec_4_lst.append(
168             MultiRecipientCiphertext(
169                 gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
170             )
171         )
172     return tuple(c_dec_4_lst)
173
174
175 def get_c_mix_5(json_data: dict) -> tuple[MultiRecipientCiphertext, ...]:
176     """returns preceding shuffled votes"""
177     c_mix_5_lst = []
178     for i in json_data["verifiableShuffle"]["shuffledCiphertexts"]:
179         c_mix_5_lst.append(
180             MultiRecipientCiphertext(
181                 gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
182             )
183         )
184     return tuple(c_mix_5_lst)
185
186
187 def get_pi_mix_5(json_data: dict[str, dict[str, dict[str, dict[str, Any]]]]) ->
188     ShuffleArgument:
189     """returns a preceding shuffle proof"""
190     data = json_data["verifiableShuffle"]["shuffleArgument"]
191     e_lst = []
192     for i in data["multiExponentiationArgument"]["E"]:
193         e_lst.append(
194             MultiRecipientCiphertext(
195                 gamma=int(i["gamma"], 16), phis=tuple(int(x, 16) for x in i["phis"])
196             )
197         )
198     e = tuple(e_lst)
199
200     return ShuffleArgument(
201         c_a=tuple(int(i, 16) for i in data["c_A"]),
202         c_b=tuple(int(i, 16) for i in data["c_B"]),
203         product_argument=ProductArgument(
204             c_b=None,
205             hadamard_arg=None,
206             single_value_product_arg=SingleValueProductArgument(
207                 c_d=int(
208                     data["productArgument"]["singleValueProductArgument"]["c_d"],
209                     16,
210                 ),
211                 c_lower_delta=int(
212                     data["productArgument"]["singleValueProductArgument"]["c_delta"]
213                 ),
214                 c_upper_delta=int(
215                     data["productArgument"]["singleValueProductArgument"]["c_Delta"]
216                 ),
217                 a_tilde=tuple(
218                     int(i, 16)
219                     for i in data["productArgument"]["singleValueProductArgument"]["
220 a_tilde"]
221                 ),

```

```

222         b_tilde=tuple(
223             int(i, 16)
224             for i in data["productArgument"]["singleValueProductArgument"]["
b_tilde"]
225         ),
226         r_tilde=int(
227             data["productArgument"]["singleValueProductArgument"]["r_tilde"
],
228             16,
229         ),
230         s_tilde=int(
231             data["productArgument"]["singleValueProductArgument"]["s_tilde"
],
232             16,
233         ),
234     ),
235 ),
236 multi_exponentiation_argument=MultiExponentiationArgument(
237     c_a_0=int(
238         data["multiExponentiationArgument"]["c_A_0"],
239         16,
240     ),
241     c_b=tuple(int(i, 16) for i in data["multiExponentiationArgument"]["c_B"
]),
242     e=e,
243     a=tuple(int(i, 16) for i in data["multiExponentiationArgument"]["a"]),
244     r=int(
245         data["multiExponentiationArgument"]["r"],
246         16,
247     ),
248     b=int(
249         data["multiExponentiationArgument"]["b"],
250         16,
251     ),
252     s=int(
253         data["multiExponentiationArgument"]["s"],
254         16,
255     ),
256     tau=int(
257         data["multiExponentiationArgument"]["tau"],
258         16,
259     ),
260 ),
261 )
262
263
264 def get_pi_dec_5(json_data: dict) -> tuple[Proofs, ...]:
265     """returns preceding decryption proofs"""
266     pi_dec_5_lst = []
267     for i in json_data["verifiablePlaintextDecryption"]["decryptionProofs"]:
268         pi_dec_5_lst.append(Proofs(e=int(i["e"], 16), z=tuple(int(x, 16) for x in i[
"z"])))
269     return tuple(pi_dec_5_lst)
270
271
272 def test_ok() -> None:
273     """
274     All the tests that should not fail.
275     """
276
277     for data in DATA:
278         short_id_bb = data.short_id_bb()
279

```

```

280     is_ok = verify_tally_control_component_ballot_box(
281         group=GROUP,
282         ee=get_election_event_id(),
283         ballot_box_id=data.ballot_box_id,
284         eb_pk=get_eb_pk(),
285         v_tilde=get_v_tilde(data.get_delta_hat_context),
286         p_tilde=get_p_tilde(data.get_delta_hat_context),
287         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
288         delta_hat=get_delta_hat(data.get_delta_hat_context),
289         c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
290         c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
291         pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
292         m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
293         pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
294         l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
295         l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
296     )
297
298     assert is_ok
299
300
301 def test_fail() -> None:
302     """
303     Tests that must fail.
304     """
305     data = DATA[0]
306     short_id_bb = data.short_id_bb()
307
308     wrong_data = DATA[2]
309     wrong_short_id_bb = wrong_data.short_id_bb()
310
311     with capture_logs():
312         is_ok = verify_tally_control_component_ballot_box(
313             group=GROUP,
314             ee=get_election_event_id(),
315             ballot_box_id=data.ballot_box_id,
316             eb_pk=get_eb_pk(),
317             v_tilde=get_v_tilde(data.get_delta_hat_context),
318             p_tilde=get_p_tilde(data.get_delta_hat_context),
319             psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
320             delta_hat=get_delta_hat(data.get_delta_hat_context),
321             c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[wrong_short_id_bb
] [4]),
322             c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[wrong_short_id_bb
]),
323             pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
324             m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[wrong_short_id_bb]),
325             pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
326             l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
327             l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
328         )
329
330     assert is_ok is False, (
331         "verify_tally_control_component_ballot_box should have failed"
332         "due to wrong c_dec_4, c_mix_5 and m"
333     )
334
335     with capture_logs():
336         is_ok = verify_tally_control_component_ballot_box(
337             group=GROUP,
338             ee=get_election_event_id(),

```

```

339     ballot_box_id=data.ballot_box_id,
340     eb_pk=get_eb_pk(),
341     v_tilde=get_v_tilde(data.get_delta_hat_context),
342     p_tilde=get_p_tilde(data.get_delta_hat_context),
343     psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
344     delta_hat=get_delta_hat(data.get_delta_hat_context),
345     c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
346     c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
347     pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
348     m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
349     pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
350     l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[wrong_short_id_bb]),
351     l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
wrong_short_id_bb]),
352 )
353
354 assert is_ok is False, (
355     "verify_tally_control_component_ballot_box should have failed"
356     "due to wrong l_votes and l_decoded_votes"
357 )
358
359 with capture_logs():
360     is_ok = verify_tally_control_component_ballot_box(
361         group=GROUP,
362         ee=get_election_event_id(),
363         ballot_box_id=data.ballot_box_id,
364         eb_pk=get_eb_pk(),
365         v_tilde=get_v_tilde(data.get_delta_hat_context),
366         p_tilde=get_p_tilde(data.get_delta_hat_context),
367         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
368         delta_hat=get_delta_hat(data.get_delta_hat_context),
369         c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
370         c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
371         pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[
wrong_short_id_bb]),
372         m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
373         pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
374         l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
375         l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
376     )
377
378 assert (
379     is_ok is False
380 ), "verify_tally_control_component_ballot_box should have failed because of
wrong pi_mix_5"
381
382 with capture_logs():
383     is_ok = verify_tally_control_component_ballot_box(
384         group=GROUP,
385         ee=get_election_event_id(),
386         ballot_box_id=data.ballot_box_id,
387         eb_pk=get_eb_pk(),
388         v_tilde=get_v_tilde(data.get_delta_hat_context),
389         p_tilde=get_p_tilde(data.get_delta_hat_context),
390         psi=get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[short_id_bb][1]),
391         delta_hat=get_delta_hat(data.get_delta_hat_context),
392         c_dec_4=get_c_dec_4(json_data=DATA_CC_SHUFFLE_PAYLOAD[short_id_bb][4]),
393         c_mix_5=get_c_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
394         pi_mix_5=get_pi_mix_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
395         m=get_m(json_data=DATA_TC_SHUFFLE_PAYLOAD[short_id_bb]),
396         pi_dec_5=get_pi_dec_5(json_data=DATA_TC_SHUFFLE_PAYLOAD[
wrong_short_id_bb]),

```

```

397         l_votes=get_l_votes(json_data=DATA_TC_VOTES_PAYLOAD[short_id_bb]),
398         l_decoded_votes=get_l_decoded_votes(json_data=DATA_TC_VOTES_PAYLOAD[
short_id_bb]),
399     )
400     assert is_ok is False, "verify_process_plaintexts should have failed because of
wrong pi_dec_5"
401
402
403 if __name__ == "__main__":
404     test_ok()
405     test_fail()
406

```

Listing A.9. Tests VerifyTallyControlComponentsBallotBox

A.2.5 Tests VerifyOnlineControlComponents

```

1 # test_verify_online_control_components.py
2
3 import json
4 from dataclasses import dataclass
5 from pathlib import Path
6 from typing import Final
7
8 from structlog.testing import capture_logs
9 from swiss_post_voting_system.crypto_primitives.elgamal import (
10     ChoiceReturnCodesEncryptionPublicKey,
11     Group,
12 )
13 from swiss_post_voting_system.crypto_primitives.mixnet_arguments_containers import (
14     MultiRecipientCiphertext,
15     ShuffleArgument,
16 )
17 from swiss_post_voting_system.crypto_primitives.zeroknowledgeproofs import Proof,
Proof2, Proofs
18 from swiss_post_voting_system.verifier.final_verification import
verify_online_control_components
19
20 from swiss_post_voting_system_tests.system_tests.test_verify_mix_dec_offline import
(
21     get_c_dec,
22     get_c_mix,
23     get_ccm_el_pk,
24     get_eb_pk,
25     get_el_pk,
26     get_pi_dec,
27     get_pi_mix,
28 )
29 from swiss_post_voting_system_tests.verifier_tests.config import DATASETS_DIR
30 from swiss_post_voting_system_tests.verifier_tests.
test_verify_online_control_components_ballot_box import (
31     get_delta_hat,
32     get_e1_bold_1,
33     get_e1_bold_tilde_1,
34     get_e2_bold_1,
35     get_kmap,
36     get_pi_bold_eqenc_1,
37     get_pi_bold_exp_1,
38     get_psi,
39     get_vc_bold_1,
40 )
41 from swiss_post_voting_system_tests.verifier_tests.test_verify_process_plaintexts
import (

```

```

42     get_p_tilde,
43     get_v_tilde,
44 )
45
46 ELECTION_EVENT_CONTEXT_PAYLOAD_DICT = json.loads(
47     (DATASETS_DIR / "dataset1/setup/electionEventContextPayload.json").read_text()
48 )
49
50 SETUP_VCS_DIR: Final[Path] = DATASETS_DIR / "dataset1/setup/verification_card_sets"
51 TALLY_BOXES_DIR: Final[Path] = DATASETS_DIR / "dataset1/tally/ballot_boxes/"
52
53 GROUP: Final[Group] = Group.from_dict(dct=ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["
54     encryptionGroup"])
55
56 @dataclass(frozen=True, slots=True)
57 class Data:
58     """
59     Data for the tests
60     """
61
62     verification_card_set_id: str
63     ballot_box_id: str
64     get_delta_hat_context: int
65
66     def short_id_vcs(self) -> str:
67         """
68         return the first 4 chars of the verification_card_set_id
69         """
70         return self.verification_card_set_id[:4]
71
72     def short_id_bb(self) -> str:
73         """
74         return the first 4 chars of the ballot_box_id
75         """
76         return self.ballot_box_id[:4]
77
78
79 DATA = (
80     Data(
81         verification_card_set_id="73e2eed19de9494ea9eaf93968e9b428",
82         ballot_box_id="4120f03ccc8641389adf907c8c80f205",
83         get_delta_hat_context=0,
84     ),
85     Data(
86         verification_card_set_id="3880a1b0f49341d68f3c9fec15782063",
87         ballot_box_id="0a7b0d1d302e451c97a2a1bc667ca89d",
88         get_delta_hat_context=1,
89     ),
90     Data(
91         verification_card_set_id="ae82cc64b620433da892983df6363d8c",
92         ballot_box_id="4600fb57269a426695193b57f694ed1c",
93         get_delta_hat_context=2,
94     ),
95     Data(
96         verification_card_set_id="fe9bb7092993440eb51235f0efa5d19b",
97         ballot_box_id="1620dc54f5a147d492668dd34280261d",
98         get_delta_hat_context=3,
99     ),
100 )
101
102
103 def parse_payload() -> tuple[dict, dict, dict, dict]:

```

```

104     """
105     Parsing the payload
106     """
107
108     data_cc_ballot_box_payload: dict[str, dict] = {}
109     data_cc_shuffle_payload: dict[str, dict] = {}
110     data_tally_component_votes_payload: dict[str, dict] = {}
111     for ballot_box_path in TALLY_BOXES_DIR.iterdir():
112         ballot_box_short = ballot_box_path.name[:4]
113
114         data_cc_ballot_box_payload[ballot_box_short] = {}
115         data = data_cc_ballot_box_payload[ballot_box_short]
116         for j in range(1, 5):
117             data[j] = json.loads(
118                 (ballot_box_path / f"controlComponentBallotBoxPayload_{j}.json").
119                 read_text()
120             )
121
122         data_cc_shuffle_payload[ballot_box_short] = {}
123         data = data_cc_shuffle_payload[ballot_box_short]
124         for j in range(1, 5):
125             data[j] = json.loads(
126                 (ballot_box_path / f"controlComponentShufflePayload_{j}.json").
127                 read_text()
128             )
129
130         data_tally_component_votes_payload[ballot_box_short] = json.loads(
131             (ballot_box_path / "tallyComponentVotesPayload.json").read_text()
132         )
133
134     data_setup_component_tally_data_payload: dict[str, dict] = {}
135     for vcs_path in SETUP_VCS_DIR.iterdir():
136         vcs_short = vcs_path.name[:4]
137
138         data_setup_component_tally_data_payload[vcs_short] = json.loads(
139             (vcs_path / "setupComponentTallyDataPayload.json").read_text()
140         )
141
142     return (
143         data_cc_ballot_box_payload,
144         data_cc_shuffle_payload,
145         data_setup_component_tally_data_payload,
146         data_tally_component_votes_payload,
147     )
148
149     (
150     DATA_CC_BALLOT_BOX_PAYLOAD,
151     DATA_CC_SHUFFLE_PAYLOAD,
152     DATA_SC_TALLY_DATA_PAYLOAD,
153     DATA_TC_VOTES_PAYLOAD,
154 ) = parse_payload()
155
156 def get_election_event_id() -> str:
157     """returns election event ID ee"""
158     # the str(...) is only here to make mypy happy...
159     return str(ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"]["
160     electionEventId"])
161
162 def get_ballot_box_ids() -> tuple[str, ...]:
163     """returns ballot box ids"""

```

```

164     return tuple(data.ballot_box_id for data in DATA)
165
166
167 def get_psis() -> tuple[int, ...]:
168     """returns number of selectable voting options"""
169     return tuple(
170         get_psi(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1]) for
171         data in DATA
172     )
173
174 def get_pk_bold_ccr() -> ChoiceReturnCodesEncryptionPublicKey:
175     """returns choice return codes encryption public key"""
176     return ChoiceReturnCodesEncryptionPublicKey(
177         tuple(
178             int(x, 16)
179             for x in ELECTION_EVENT_CONTEXT_PAYLOAD_DICT["electionEventContext"][
180                 "choiceReturnCodesEncryptionPublicKey"
181             ]
182         )
183     )
184
185
186 def get_delta_hats() -> tuple[int, ...]:
187     """returns the number of allowed write-ins + 1 for this specific ballot box"""
188     return tuple(get_delta_hat(i=i) for i in range(len(DATA)))
189
190
191 def get_kmaps() -> tuple[dict[str, int], ...]:
192     """returns key-value map of the verification card public keys"""
193     return tuple(
194         get_kmap(json_data=DATA_SC_TALLY_DATA_PAYLOAD[data.short_id_vcs()]) for data
195         in DATA
196     )
197
198 def get_vc_bold_ls() -> tuple[tuple[str, ...], ...]:
199     """returns control component's list of confirmed verification card IDs"""
200     return tuple(
201         get_vc_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1])
202         for data in DATA
203     )
204
205 def get_e1_bold_ls() -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
206     """returns control component's list of encrypted, confirmed votes"""
207     return tuple(
208         get_e1_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1])
209         for data in DATA
210     )
211
212 def get_e1_bold_tilde_ls() -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:
213     """returns control component's list of exponentiated, encrypted, confirmed votes
214     """
215     return tuple(
216         get_e1_bold_tilde_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()
217         ][1])
218         for data in DATA
219     )
220
221 def get_e2_bold_ls() -> tuple[tuple[MultiRecipientCiphertext, ...], ...]:

```



```

221     """returns control component's list of encrypted, partial Choice Return Codes"""
222     return tuple(
223         get_e2_bold_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()][1])
224     )
225
226
227 def get_pi_bold_exp_1s() -> tuple[tuple[Proof, ...], ...]:
228     """returns control component's list of exponentiation proofs"""
229     return tuple(
230         get_pi_bold_exp_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()]
231 ] [1])
232         for data in DATA
233     )
234
235 def get_pi_bold_eqenc_1s() -> tuple[tuple[Proof2, ...], ...]:
236     """returns control component's list of plaintext equality proofs"""
237     return tuple(
238         get_pi_bold_eqenc_1(json_data=DATA_CC_BALLOT_BOX_PAYLOAD[data.short_id_bb()]
239 ] [1])
240         for data in DATA
241     )
242
243 def get_c_mixs() -> tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...]:
244     """returns preceding shuffled votes"""
245     return tuple(get_c_mix(data.short_id_bb()) for data in DATA)
246
247
248 def get_pi_mixs() -> tuple[tuple[ShuffleArgument, ...], ...]:
249     """returns preceding shuffled votes"""
250     return tuple(get_pi_mix(data.short_id_bb()) for data in DATA)
251
252
253 def get_c_dec_s() -> tuple[tuple[tuple[MultiRecipientCiphertext, ...], ...], ...]:
254     """returns preceding partially decrypted votes"""
255     return tuple(get_c_dec(data.short_id_bb()) for data in DATA)
256
257
258 def get_pi_dec_s() -> tuple[tuple[tuple[Proofs, ...], ...], ...]:
259     """returns preceding decryption proofs"""
260     return tuple(get_pi_dec(data.short_id_bb()) for data in DATA)
261
262
263 def get_p_tildes() -> tuple[tuple[int, ...], ...]:
264     """returns list of actual encoded voting options"""
265     return tuple(get_p_tilde(data.get_delta_hat_context) for data in DATA)
266
267
268 def get_v_tildes() -> tuple[tuple[str, ...], ...]:
269     """returns list of actual voting options"""
270     return tuple(get_v_tilde(data.get_delta_hat_context) for data in DATA)
271
272
273 def test_ok() -> None:
274     """
275     All the tests that should not fail.
276     """
277     is_ok = verify_online_control_components(
278         group=GROUP,
279         ee=get_election_event_id(),
280         ballot_box_ids=get_ballot_box_ids(),

```

```

281     psis=get_psis(),
282     el_pk=get_el_pk(),
283     ccm_el_pk=get_ccm_el_pk(),
284     eb_pk=get_eb_pk(),
285     pk_bold_ccr=get_pk_bold_ccr(),
286     delta_hats=get_delta_hats(),
287     kmaps=get_kmaps(),
288     vc_bold_ls=get_vc_bold_ls(),
289     e1_bold_ls=get_e1_bold_ls(),
290     e1_bold_tilde_ls=get_e1_bold_tilde_ls(),
291     e2_bold_ls=get_e2_bold_ls(),
292     pi_bold_exp_ls=get_pi_bold_exp_ls(),
293     pi_bold_eqenc_ls=get_pi_bold_eqenc_ls(),
294     c_mixs=get_c_mixs(),
295     pi_mixs=get_pi_mixs(),
296     c_decs=get_c_decs(),
297     pi_decs=get_pi_decs(),
298     p_tildes=get_p_tildes(),
299     v_tildes=get_v_tildes(),
300 )
301
302 assert is_ok
303
304
305 def test_fail() -> None:
306     """
307     Tests that must fail.
308     """
309
310     with capture_logs():
311         is_ok = verify_online_control_components(
312             group=GROUP,
313             ee=get_election_event_id(),
314             ballot_box_ids=get_ballot_box_ids(),
315             psis=get_psis(),
316             el_pk=get_el_pk(),
317             ccm_el_pk=get_ccm_el_pk(),
318             eb_pk=get_eb_pk(),
319             pk_bold_ccr=get_pk_bold_ccr(),
320             delta_hats=get_delta_hats(),
321             kmaps=get_kmaps(),
322             vc_bold_ls=get_vc_bold_ls(),
323             e1_bold_ls=get_e1_bold_ls(),
324             e1_bold_tilde_ls=get_e1_bold_tilde_ls(),
325             e2_bold_ls=tuple(get_e2_bold_ls()[i] for i in [2, 0, 3, 1]),
326             pi_bold_exp_ls=get_pi_bold_exp_ls(),
327             pi_bold_eqenc_ls=get_pi_bold_eqenc_ls(),
328             c_mixs=get_c_mixs(),
329             pi_mixs=get_pi_mixs(),
330             c_decs=get_c_decs(),
331             pi_decs=get_pi_decs(),
332             p_tildes=get_p_tildes(),
333             v_tildes=get_v_tildes(),
334         )
335
336     assert (
337         is_ok is False
338     ), "verify_online_control_components should have failed because of mixed
339     e2_bold_ls"
340
341     with capture_logs():
342         is_ok = verify_online_control_components(

```

```

343     ee=get_election_event_id(),
344     ballot_box_ids=get_ballot_box_ids(),
345     psis=get_psis(),
346     el_pk=get_el_pk(),
347     ccm_el_pk=get_ccm_el_pk(),
348     eb_pk=get_eb_pk(),
349     pk_bold_ccr=get_pk_bold_ccr(),
350     delta_hats=get_delta_hats(),
351     kmaps=get_kmaps(),
352     vc_bold_ls=get_vc_bold_ls(),
353     el_bold_ls=get_el_bold_ls(),
354     el_bold_tilde_ls=get_el_bold_tilde_ls(),
355     e2_bold_ls=get_e2_bold_ls(),
356     pi_bold_exp_ls=get_pi_bold_exp_ls(),
357     pi_bold_eqenc_ls=get_pi_bold_eqenc_ls(),
358     c_mixs=tuple(get_c_mixs()[i] for i in [2, 3, 0, 1]),
359     pi_mixs=get_pi_mixs(),
360     c_decs=tuple(get_c_decs()[i] for i in [2, 3, 0, 1]),
361     pi_decs=get_pi_decs(),
362     p_tildes=get_p_tildes(),
363     v_tildes=get_v_tildes(),
364 )
365
366 assert (
367     is_ok is False
368 ), "verify_online_control_components should have because of mixed c_mixs and
c_decs"
369
370 with capture_logs():
371     is_ok = verify_online_control_components(
372         group=GROUP,
373         ee=get_election_event_id(),
374         ballot_box_ids=get_ballot_box_ids(),
375         psis=get_psis(),
376         el_pk=get_el_pk(),
377         ccm_el_pk=get_ccm_el_pk(),
378         eb_pk=get_eb_pk(),
379         pk_bold_ccr=get_pk_bold_ccr(),
380         delta_hats=tuple(get_delta_hats()[i] for i in [3, 2, 1, 0]),
381         kmaps=get_kmaps(),
382         vc_bold_ls=get_vc_bold_ls(),
383         el_bold_ls=tuple(get_el_bold_ls()[i] for i in [3, 2, 1, 0]),
384         el_bold_tilde_ls=get_el_bold_tilde_ls(),
385         e2_bold_ls=get_e2_bold_ls(),
386         pi_bold_exp_ls=get_pi_bold_exp_ls(),
387         pi_bold_eqenc_ls=get_pi_bold_eqenc_ls(),
388         c_mixs=get_c_mixs(),
389         pi_mixs=get_pi_mixs(),
390         c_decs=get_c_decs(),
391         pi_decs=get_pi_decs(),
392         p_tildes=get_p_tildes(),
393         v_tildes=get_v_tildes(),
394     )
395
396 assert (
397     is_ok is False
398 ), "verify_online_control_components should have failed because of mixed
delta_hats"
399
400 with capture_logs():
401     is_ok = verify_online_control_components(
402         group=GROUP,
403         ee=get_election_event_id(),

```

```

404     ballot_box_ids=get_ballot_box_ids(),
405     psis=get_psis(),
406     el_pk=get_el_pk(),
407     ccm_el_pk=get_ccm_el_pk(),
408     eb_pk=get_eb_pk(),
409     pk_bold_ccr=get_pk_bold_ccr(),
410     delta_hats=get_delta_hats(),
411     kmaps=get_kmaps(),
412     vc_bold_ls=get_vc_bold_ls(),
413     el_bold_ls=get_el_bold_ls(),
414     el_bold_tilde_ls=get_el_bold_tilde_ls(),
415     e2_bold_ls=get_e2_bold_ls(),
416     pi_bold_exp_ls=get_pi_bold_exp_ls(),
417     pi_bold_eqenc_ls=tuple(get_pi_bold_eqenc_ls()[i] for i in [2, 1, 3, 0]),
418     c_mixs=get_c_mixs(),
419     pi_mixs=get_pi_mixs(),
420     c_decs=get_c_decs(),
421     pi_decs=get_pi_decs(),
422     p_tildes=get_p_tildes(),
423     v_tildes=get_v_tildes(),
424 )
425
426 assert (
427     is_ok is False
428 ), "verify_online_control_components should have failed because of mixed
pi_bold_eqenc_ls"
429
430 with capture_logs():
431     is_ok = verify_online_control_components(
432         group=GROUP,
433         ee=get_election_event_id(),
434         ballot_box_ids=get_ballot_box_ids(),
435         psis=get_psis(),
436         el_pk=get_el_pk(),
437         ccm_el_pk=get_ccm_el_pk(),
438         eb_pk=get_eb_pk(),
439         pk_bold_ccr=get_pk_bold_ccr(),
440         delta_hats=get_delta_hats(),
441         kmaps=get_kmaps(),
442         vc_bold_ls=get_vc_bold_ls(),
443         el_bold_ls=get_el_bold_ls(),
444         el_bold_tilde_ls=get_el_bold_tilde_ls(),
445         e2_bold_ls=get_e2_bold_ls(),
446         pi_bold_exp_ls=get_pi_bold_exp_ls(),
447         pi_bold_eqenc_ls=get_pi_bold_eqenc_ls(),
448         c_mixs=get_c_mixs(),
449         pi_mixs=get_pi_mixs(),
450         c_decs=get_c_decs(),
451         pi_decs=get_pi_decs(),
452         p_tildes=tuple(get_p_tildes()[i] for i in [2, 1, 3, 0]),
453         v_tildes=tuple(get_v_tildes()[i] for i in [2, 1, 3, 0]),
454     )
455
456 assert (
457     is_ok is False
458 ), "verify_online_control_components should have failed because of mixed pTable"
459
460
461 if __name__ == "__main__":
462     test_ok()
463     test_fail()
464

```

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. Kalina, “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