A Practical and Fully Distributed E-Voting Protocol for the Swiss Context

Véronique Cortier
Université de Lorraine, CNRS, Inria

Alexandre Debant
Université de Lorraine, CNRS, Inria

Henri Devillez
Université de Lorraine, CNRS, Inria

Olivier Esseiva
Swiss Post

Pierrick Gaudry
Université de Lorraine, CNRS, Inria

Audhild Høgåsen
Swiss Post

Chiara Spadafora
Swiss Post and Università di Trento

Abstract

Internet voting aims at guaranteeing both vote secrecy and verifiability. Advanced deployed systems, such as in Estonia or Switzerland, put a strong emphasis on *cast-as-intended*: even when a voter uses a corrupt device, they should be able to check that their intended vote is included in the tally. One approach relies on *return codes*, written on a voting card. This enables cast-as-intended even when the voter uses a single device that may be corrupted. While convenient for voters, this requires to generate the codes in a secure manner. This approach is deployed in Switzerland but the current system requires a single trusted setup authority: an offline component that is assumed to correctly generate the voting data.

We propose a novel protocol that allows to distribute the trust during the setup. At the heart of our system lies an asymmetric protocol where several parties create the voting material, while allowing one of these parties to remain offline during the voting phase. Usability constraints are taken into account in our design, both in terms of computational complexity (linear setup and tally) and in terms of user experience (we ask the voter to type a high-entropy string only once). Our protocol is therefore a candidate for the next generation e-voting protocol in Switzerland.

We provide an implementation of our protocol, demonstrating its practicability. Moreover, we formally prove vote privacy and verifiability. Our analysis takes into account the presence of short codes that an attacker can guess with non-negligible probability and reveals that making a significant change to the result remains exponentially hard.

1 Introduction

Electronic voting aims at guaranteeing the same security properties as traditional paper-based voting. Vote privacy and verifiability are two fundamental security properties of e-voting systems. *Vote privacy* guarantees that no one can learn how a voter voted. *Verifiability* ensures that the result corresponds to the votes of the voters. Real-world systems deployed, for instance, in Estonia [35], Australia [12], or Switzerland [47] have a strong emphasis on *cast-as-intended*. This implies that, even if the voting devices are corrupted (*e.g.,* controlled by a nation-scale attacker), the system must guarantee that the vote of a voter cannot be modified without detection. One common technique for cast-as-intended stems from the Benaloh challenge [6]. Whenever the voting device encrypts a vote, it can be challenged to reveal the randomness used for encryption, allowing a second device to audit the encryption and check that it encodes the right vote. The Benaloh challenge is used, for example, in StarVote [7], Helios [2] and, with some simplifications, in the Estonian protocol [35]. In all cases, this requires two devices: one to produce the ballot and one to audit it. Moreover, the audit step is optional and hence only a small fraction of voters actually audit their ballot [34].

Another less studied approach is the one in use in Switzerland. The Swiss regulation requires cast-as-intended to be achieved through *return codes*. Namely, before the election, voters receive a voting card like the one shown in Figure 1. Each candidate is associated with a short 4-digit return code. The codes are freshly generated for each voter. After selecting a candidate, the voter verifies that the code displayed by the voting device matches the code printed on their voting card.

The device, even if corrupted, may only obtain the correct return code if it did send the intended candidate in the encrypted ballot. This approach has been shown to have a better usability than approaches based on the Benaloh challenge [38]. A key challenge of this mechanism is that it should not jeopardize vote privacy. In particular, the system needs to be able to compute the return code without learning the corresponding vote.

In 2015, Scytl [28] proposed a voting protocol with such a mechanism for cast-as-intended. It was used in elections in several Swiss Cantons. However, due to a serious cryptographic flaw discovered in 2019 in zero-knowledge

This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006 and by the ERC Synergy Grant VePaSS (Grant agreement No. 101224640).

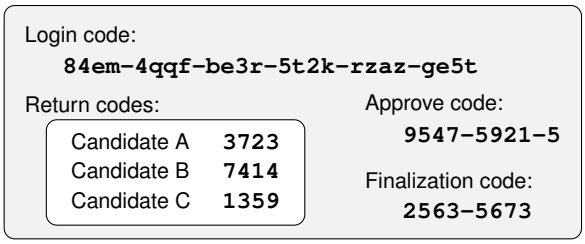


Figure 1: Voting card.

proofs [19], e-voting was temporarily discontinued throughout Switzerland. Swiss Post then fixed the protocol and re-implemented most of it [47]. E-voting was reintroduced with the fixed protocol in several Swiss Cantons in 2023.

Limitations. Even if the Swiss Post Voting System complies with the requirements from the Swiss Ordinance on electronic voting by the Federal Chancellery [25], it builds upon a single fully-trusted entity called **Setup Component** - SC. This entity is in charge of creating the voting material of each voter and sending it to a **Printer** who will print and send it to the expected voter. Moreover, SC shares with 4 online servers (called **Control Components** - CCs) the cryptographic material necessary to compute the return codes during the voting phase. Unlike SC, which is always fully trusted, the Swiss Post Voting System makes fewer assumptions about the CCs: none of them is trusted for vote privacy[1], and only one (out of the four) is trusted for cast-as-intended.

The SC therefore represents a critical single point of failure in the Swiss Post Voting System. If corrupted, it can break all desired security properties:

- it may add ballots on behalf of absentee voters since it knows the entire voting material;
- it may break vote privacy since the return codes are sent back in clear to voters during the voting phase and the SC knows the association between return codes and candidates;
- it may also modify the vote of the voters, in collusion with their voting devices, since it knows the return codes.

Even if the Setup Component is an offline component operated by the Cantons themselves to reduce the threats, the Swiss authorities have acknowledged this issue [24] and now aim to reduce the trust placed in this component [26]. Such a change to the Ordinance would make the Swiss Post Voting System, the only e-voting system currently in use in Switzerland, no longer compliant with the requirements.

Contributions. We propose a protocol that overcomes the identified limitation of the current system by balancing the trust between the SC and the CCs: either the SC or 1 out of 4 CCs is trusted for vote privacy and verifiability. Importantly, our protocol has been designed to respect the practical deployment constraints of such a protocol. In particular, it preserves the asymmetry between SC (an offline component) and the CCs (online components). This new protocol aligns with the Swiss authorities' aim of reducing the trust in SC.

The intuition of our protocol is simple: we consider the SC and the 4 CCs as 5 entities that generate the voting material in a Multi-Party-Computation (MPC) flavor. A key challenge is that the 5 entities play an asymmetric role. Indeed, SC needs to remain offline during the voting phase, where a return code is sent back to the voter. We therefore need to design an MPC protocol that generates the return codes during the setup, and later computes them during the voting phase among 4 components, while one of the 5 entities is offline. Moreover, this mechanism should preserve vote privacy. To achieve this, we combine mixnet and masking techniques during the setup: the SC masks and shuffles the list of possible voting options, so that the CCs only know the correspondence between the return codes and the *masked* voting options. Similarly, the CCs shuffle the list of possible votes to guarantee vote privacy against a dishonest SC (masking with an exponent is not done during the setup but on the fly during the voting phase). During the voting phase, similarly to the Swiss Post Voting System, the voting device (hereafter referred to as the Client) prepares both the encrypted ballot $\mathsf{enc}(v)$ and a masked version v^k of the vote in order to request the corresponding return code. It must also prove in zero-knowledge that the same v was encoded in both parts. The CCs then collectively compute the expected return code. We observe that the aforementioned proof of correspondence between the encrypted ballot and the masked vote is not an instance of the Maurer framework [39] and the zero-knowledge proof provided in the Swiss Post Voting System (Algorithm 5.4 of [47]) leaks additional data, without proving that it remains zero-knowledge. We therefore reworked this zero-knowledge proof and proved its security. Overall, our protocol remains linear in the number of voters and candidates and does not significantly impact the computational cost for the Client compared to the Swiss Post protocol. Moreover, our protocol relies on a generic group. This not only allows to opt for more efficient implementations (such as elliptic curves) but it also allows to get rid of non-standard cryptographic assumptions such as SGSP [29] in the current Swiss Post Voting System security proof [46].

We provide an implementation of our protocol to further evaluate its practicality. We considered a particularly demanding Swiss election scenario, namely the canton of Geneva, with 100,000 voters, 1,000 candidates and the possibility of selecting up to 100 candidates. Our implementation shows that it is possible to run the setup within a couple of hours while ballot casting can be performed within a few seconds. Our protocol complies with the E-voting Catalogue of Measures by the Confederation and Cantons [26], and is currently

[1] The Federal Chancellery allows to trust one out of 4 components for vote privacy, if they are operated by distinct companies, which has not been the case in practice.

under evaluation by the Chancellery. It is therefore a candidate for the next generation e-voting protocol in Switzerland.

We formally analyze our protocol and prove:

- vote privacy assuming the Printer and either SC or one of the CCs is honest;
- verifiability and in particular cast-as-intended assuming one of the CCs is honest. This trust assumption is needed for any protocol with return codes. Indeed, by design, the Client and the CCs must be able to return the appropriate code for any choice of the voter. Hence, if the voter wants to vote for A, the attacker may simulate a voting session with a ballot for A in order to compute the right return code and then restart a new session with a ballot for B. Therefore at least one CC must be honest to preserve cast-as-intended;
- eligibility assuming SC or one of the CCs is honest.

It is worth noting that we prove verifiability in a context where the attacker may win with non-negligible probability. Indeed, our protocol makes use of short codes (*e.g.,* return codes), that the adversary has a non-negligible probability to guess. Previous analyses of protocols with return codes [10, 46] considered a non quantitative verifiability notion [16] and prove that verifiability is satisfied up to the probability of guessing one code (plus a negligible function). However, the resulting guarantee is weak: the adversary may have a probability $\frac{1}{10000}$ to fully change the result of the election, since this is the probability of guessing one 4-digit return code. Moreover, existing verifiability definitions [16, 17, 36, 37] assume that voters did not detect any issue. This means that as soon as one voter complains, either the election should be stopped or there is no guarantee anymore. Therefore, we propose a verifiability definition that accounts both for the number k of complaints and the number d of vote modifications and we prove our protocol secure w.r.t. this definition, showing (roughly) that an adversary cannot modify more than $d+\varepsilon$ votes without triggering less than d complaints.

Related work. Historically, CHVote [32] has been designed to satisfy the Swiss requirements and therefore appears as the main competitor of the Swiss Post Voting System (previously Scytl protocol). It leverages oblivious transfer protocols to implement return codes while making no assumption about SC. Indeed, the voting material is entirely generated by the CCs themselves at the beginning of the protocol. However, unlike the Swiss Post Voting System and the Swiss Ordinance on electronic voting, this protocol assumes a high trust in the CCs: at least one of them must be trusted for both verifiability and vote privacy. If they are all dishonest, they may add ballots since they know the voting material. They may also break vote privacy since they know the return codes. In order for such protocol to be compliant with the Swiss Ordinance, the CCs must be operated by distinct entities. In practice, this separation has not been implemented in any elections in Switzerland, as it would require a more complex infrastructure and higher costs. Indeed, operating an online CC is a challenging task. It requires to implement a strict access control (logical and physical), strong DDoS protections, and to manage configurations and updates. In practice, in elections in Switzerland, the CCs are operated by a single company, with distinct administrative teams, while an offline component (e.g., SC) is operated by the election authorities (namely, the Cantons). The Cantons do not have their own infrastructure and cannot be online during the election. Adapting CHVote to support such an asymmetry is non-trivial. Moreover, several attacks against verifiability and privacy have been found against CHVote [14] and not yet fixed.

In the Swiss context, another idea is to use code voting [15, 31], as originally suggested by Chaum [13]: instead of entering their vote in the Client, the voter enters a code, where the correspondence between votes and codes is provided on the voting material. Intuitively, each voting code encodes or points to an encrypted ballot containing the vote. Return codes are then used to make sure that the Client does not change the vote (*e.g.,* encrypting another vote). The main advantage of this approach is that it guarantees vote privacy even against a dishonest Client. However, in addition to usability issues, the main drawback is that it places full trust in the SC that can now very easily swap votes, without even the help of the Client: it simply needs to write "B" instead of "A" in front of the code corresponding to A, and vice versa. Worse, this attack requires no cryptographic expertise and can be performed by a neighbor or roommate who intercepts the voting material and replaces it with a modified version.

Outside the Swiss context, a few deployed protocols aim at providing cast-as-intended guarantees. The Australian iVote protocol [12] provides cast-as-intended by telling their vote in clear by phone to the voter, severely diminishing vote privacy. The Estonian protocol [35] ensures cast-as-intended by displaying the randomness used for encryption. Similarly, CAISED [41] offers cast-as-intended by providing the randomness to a second device, but in a zero-knowledge way, in order to mitigate the risk of vote buying. However, both solutions assume a second device, and have lower usability than return codes [38].

2 High-level View of the Protocol

We first present here a high level view of the protocol, focusing on the view of the Voter.

2.1 Participants

Voters and their Clients. Voters vote from home, by connecting to a website. Beforehand, they have received voting material by postal mail. When studying the cast-as-intended

property, we need to consider a Voter and their Client (computer and software running on it) as separate entities.

Control Components. (CC_j, for $j = 1, \ldots, m$) These are m independent servers (typically, $m = 4$) that participate in the setup phase, and will stay online during the voting phase, in order to send return codes to the Voters.

Setup Component. (SC) This is a server that participates in the setup phase, and will remain offline during the voting phase. In some parts of the setup, SC behaves similarly to the CCs. In those cases, the corresponding data is indexed by $j = 0$, treating SC as CC_0.

Printer Component. (PC) At the end of the setup phase, this entity receives data to be printed and sent to the Voters by postal mail.

Talliers. (TC) These are servers in charge of generating the public key of the election and storing the decryption key in a distributed manner. They participate in the tally phase, after the voting phase.

Auditors. Throughout the protocol, audit data is collected, including public data, signatures, and zero-knowledge proofs, so that verification by independent auditors provides additional security properties.

We abstract away an entity that is in charge of preparing the list of legitimate Voters, together with their postal addresses (so that the Printer can send the voting material), the list of questions and voting options for each question, and a unique election identifier. We also assume that public keys of all the parties are known by everyone.

In the presentation, we view the Talliers as different entities but they are typically played by SC and the CCs.

2.2 Overview of the Phases

Setup phase. A key generation protocol is run by the Talliers to set the main public key of the election pk_{EL}. This is done in a standard way, with or without a threshold.

In parallel of the key generation, the setup protocol is run by the Setup and the Control Components in order to produce two pieces of data:

- **Data for the Printer.** At the end of the setup phase, the Printer receives (encrypted with its public key), for each Voter, the information to be printed on a voting card (as in Figure 1) and sent to them by postal mail.
- **Data for each Control Component.** During the setup phase, each CC will remember some information that will be required during the voting phase. In particular, for each Voter, they get a list of partial return codes, associated to a list of masked voting options.

At the end of the setup phase, a delay of several days (or even weeks, in the case where Voters are living abroad) must be set, so that Voters have time to receive the voting material from the Printer.

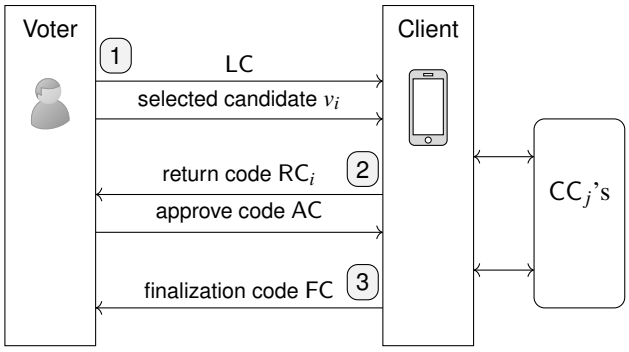


Figure 2: Voter's view of the voting phase. Step 1: authentication and candidate selection; Step 2: check validity and confirmation; Step 3: final check.

Voting phase. First, a Voter connects to their Client and enters the login code (LC), written on their voting card. The Client prompts the possible choices, and the Voter selects one of them, denoted by v_i. The Client prepares a ballot that contains $\{v_i\}_{\mathsf{pk}_{EL}}$, an encryption of v_i with the main election key pk_{EL}, and a masked version of v_i. This is sent to all the CCs.

From the masked version, each CC_j retrieves in its local database the corresponding partial return code $\mathsf{pRC}_{j,i}$ that is sent back. The Client recombines all of these into a return code RC_i that is shown to the Voter.

The Voter must check that this RC_i code corresponds to the one printed on their voting card aside the selected candidate v_i. If this is not the case, then the Voter must stop and complain. If the return code is correct, the Voter retrieves the approve code (AC) from the voting card and enters it to their Client.

The Client encrypts AC with a public key shared by the CCs, who engage in a small protocol based on plaintext equivalence test, to check that this is the correct code. If AC is correct, each CC_j records the ballot for tallying, and sends back a partial finalization code pFC_j. The Client combines all of these to retrieve the finalization code FC, which is shown to the Voter. The Voter must check that FC corresponds to the one written on their voting card, and complain otherwise.

The Voter's view is summarized in Figure 2.

Tally phase. At the end of the voting phase, the CCs agree on the list of ballots to be tallied. The transcript of their communications during the voting phase is also kept for auditing; this contains enough information to resolve a potential disagreement on the list of valid ballots. Then, the Talliers collectively mix and decrypt the ballots to obtain the result.

3 Complete protocol description

For the sake of readability, we present our protocol in a context of a single question, and where Voters choose one out of n candidates.

3.1 Key idea for return codes

The voting card of a Voter contains a list $(v_i, \mathsf{RC}_i)_{1\le i\le n}$, where RC_i is a short return code (typically 4 digits) corresponding to candidate v_i in a set $\mathcal{V}$. This return code RC_i will be obtained as the XOR of partial return codes $(\mathsf{pRC}_{j,i})_{1\le j\le m}$, sent by the m Control Components.

While the CCs must be able to send the appropriate partial code during the voting phase, they must not learn the choice v of the Voter. We take an ad-hoc approach, where most of the complexity is concentrated during the setup phase, so that the voting phase is light.

We mask the choice corresponding to a (partial) return code in two steps. On one hand, the set of encrypted $(v_i)_{1\le i\le n}$ is shuffled, where all parties contribute to the permutation, using a verifiable mixnet. The permutation must be transmitted to the Printer, so that it can display the return codes in front of the corresponding candidates. On the other hand, a masking based on exponentiation is also applied, so that the CCs cannot deduce the voting choice from the request of the Voter.

At the end of the setup phase, each CC_j gets a list of pairs $(v_{\sigma(i)}^{\mathsf{sv_{id}}},\ \mathsf{pRC}_{j,i})_{1\le i\le n}$, where σ is a permutation and $\mathsf{sv_{id}}$ is an exponent, both of them being unknown to SC and the CCs. At the same time, the Printer will get the permutation and the return codes, in order to print the list of pairs $(v_i,\ \mathsf{RC}_{\sigma^{-1}(i)})_{1\le i\le n}$ and transmit this to the Voter. The blinded vote $v^{\mathsf{sv_{id}}}$ will be jointly reconstructed at the voting phase by the CCs and the Client, which uses information generated by SC retrieved on the voting card.

The role of the setup phase is also to prepare the part where the Voter sends their approve code (if they are happy with the return code). Here, the information goes in the other direction and the constraints are different. This leads to a much simpler protocol. There is also a part for the preparation of the finalization code, which is even simpler.

Before explaining the details of how all of this is done, we present the cryptographic building blocks that are required.

3.2 Cryptographic building blocks

Let $\mathbb{G}$ be a cyclic group with generator g of prime order q, in which the Decisional Diffie-Hellman (DDH) problem is assumed to be difficult. The candidates are encoded by a set of n distinct and independent generators $v_1,\dots,v_n$ of the group $\mathbb{G}$. The correspondence between the candidates and the v_i's is public; it can be the result of a hash function that maps to $\mathbb{G}$. From now on, we often use only the v_i's, keeping this correspondence implicit.

3.2.1 ElGamal encryption

Our protocol uses ElGamal encryptions. We denote by $\{M\}_{\mathsf{pk}}$ a ciphertext $(g^r, M\,\mathsf{pk}^r)$, where r is a fresh random element of $\mathbb{Z}_q$, $M\in\mathbb{G}$ is the plaintext, and pk is the public key. The discrete logarithm sk of pk is the secret key that allows to decrypt. As a public-key encryption scheme, ElGamal is Ind-CPA secure under the DDH assumption. It supports a basic homomorphic property which allows in particular to easily rerandomize a ciphertext (without knowing sk), by point-wise multiplication with an encryption of the neutral in $\mathbb{G}$. More generally, the homomorphic property leads to simple and efficient zero-knowledge proofs (ZKP) for algebraic properties related to ciphertexts. We will list the ones we use below.

If several parties each have a public key pk_j, then, one can form the agglomerated public key $\mathsf{pk}=\prod\mathsf{pk}_j$. The corresponding secret key sk is the sum of the secret keys sk_j of all parties. A ciphertext (c_1,c_2) encrypted with pk can be decrypted in a distributed manner, where each party computes and sends $c_1^{\mathsf{sk}_j}$. Multiplying all these contributions gives c_1^{sk}, and c_2/c_1^{sk} is the plaintext. Adding a threshold feature (for robustness) to this simple distributed protocol is well known; see [8] for a modern treatment in the context of e-voting. For simplicity, we stick to the basic n-out-of-n version.

3.2.2 Classical zero-knowledge proofs

We list here some classical ZKPs that are used in our protocol, without recalling how they work, since they can be found in many places [2, 32, 47]. These are based on Sigma protocols turned into non-interactive versions via the Fiat-Shamir transform. All of them can be seen as instances of the general framework by Maurer [39], that we briefly recall in Appendix C.1. During the transform, it is important to hash as much context as possible: the election identifier, the complete public data required to verify the ZKP, the identity of the prover, and the precise step of the protocol where the proof is performed. This prevents any replay of the ZKP by a malicious party.

- $\mathsf{ZKP}_{\mathsf{DL}}$: Given $K=g^k$, this proves knowledge of k, the discrete logarithm of K (*e.g.,* a public key).
- $\mathsf{ZKP}_{\mathsf{enc}}$: Given a ciphertext $\{M\}_{\mathsf{pk}}=(g^r, M\,\mathsf{pk}^r)$ and the public key pk, this proves knowledge of r, the randomness used during the encryption. This implies the knowledge of the plaintext M.
- $\mathsf{ZKP}_{\mathsf{exp}}$: Given $K=g^k$, h and $H=h^k$, this proves that H has indeed been obtained by exponentiating the group element h by k, the discrete logarithm of the public element K (*e.g.,* during a proof of correct decryption).

In addition, we will also need the following, which is less standard. We give an algorithm for it in Appendix C.2. We remark that this is a zero-knowledge argument (and not a proof), because the zero-knowledge property relies on DDH.

- $\mathsf{ZKP}_{\mathsf{ptxte}}$: Given $c_1 = \{x\}_{\mathsf{pk}_1}$, $c_2 = \{x^k\}_{\mathsf{pk}_2}$ and $c_3 = g^k$, this proves that the plaintext of the ciphertext c_2 is equal to the plaintext v of c_1 raised to the power k, where k is the discrete logarithm of c_3.

3.2.3 Additive masking decryption algorithm (AMD)

Let pk be a public key, the secret key of which is shared as $\mathsf{sk} = \sum \mathsf{sk}_j$ between m parties, and let $c = \{M\}_{\mathsf{pk}}$ be a ciphertext. Moreover, each party holds a secret α_j, for which g^{α_j} is public. In our protocol, we want to compute $M^{\sum_j \alpha_j}$ at several instances, without revealing more information about any of the α_j or M.

This can be performed in a two-round protocol. First, each party homomorphically computes $c^{\alpha_j} = \{M^{\alpha_j}\}_{\mathsf{pk}}$ and broadcasts it. Then, the parties compute $\prod_{j=1}^{m} c^{\alpha_j} = \{M^{\sum_j \alpha_j}\}_{\mathsf{pk}}$ and decrypt in a distributed way the content of this ciphertext. To ensure that each party proceeds honestly, they compute as well in each round a ZKP that their output is correctly generated.

3.2.4 Plaintext equivalence test (PET)

Let pk be a public key, the secret key of which is shared as $\mathsf{sk} = \sum \mathsf{sk}_j$ between m parties, and let $c_1 = \{M_1\}_{\mathsf{pk}}$ and $c_2 = \{M_2\}_{\mathsf{pk}}$ be two ciphertexts. A PET is a protocol run between the parties to jointly decide (and prove to a third-party) whether $M_1 = M_2$, without anyone learning anything about M_1 and M_2 except this bit of information, including the holders of the secret key, assuming one of them is honest.

The general idea is to mask the plaintexts using the homomorphic property of ElGamal and to combine them into a ciphertext that, after decryption, gives either the neutral element, or a uniform element of the group, depending on the $M_1 = M_2$ bit. The details can be found for instance in [40].

3.2.5 Verifiable mixnets and variants

Let $[c_i]_{1 \le i \le n}$ be a list L of n ciphertexts for a public key pk. A shuffle operation consists in re-encrypting each c_i into another ciphertext $\tilde{c}_i$, and applying a random permutation σ to the list, leading to $\tilde{L} = \left[\tilde{c}_{\sigma(i)}\right]_{1 \le i \le n}$. When L, $\tilde{L}$ and pk are public, it is possible for the entity who performed the shuffle to publish a ZKP that proves that the multiset of plaintexts corresponding to L and $\tilde{L}$ are the same. We call this procedure Mix. It can be instantiated by the algorithms of Terelius and Wikström [48], Bayer and Groth [3] or Groth [30]. In what follows, we use Groth's, which is computationally the cheapest.

Verifiable mixnets are typically used during tally, so that the final output list contains the same votes as the initial ballots, but the link with the Voters is cut, assuming one of the servers is honest.

In our protocol, we use verifiable mixnets also during the setup phase, where the list of candidates is shuffled. In fact, we need primitives that do slightly more.

Let pk_1 and pk_2 be two public keys, and $K = g^k$ a public group element whose discrete logarithm k is known by the shuffler. Let $L = \left[\{x_i\}_{\mathsf{pk}_1}, \{y_i\}_{\mathsf{pk}_2}\right]_{1 \le i \le n}$ be a list of pairs of ciphertexts for both keys. The shuffler re-encrypts the two ciphertexts, and raises the second one to the power k (thus raising the associated plaintext to the same power). Then it applies a random permutation σ to the new pairs, and forms the output list $\tilde{L} = \left[\{\tilde{x}_{\sigma(i)}\}_{\mathsf{pk}_1}, \{\tilde{y}^k_{\sigma(i)}\}_{\mathsf{pk}_2}\right]_{1 \le i \le n}$. A ZKP is added, that proves that this has been performed as prescribed, without revealing the intermediate steps. This can be obtained with a cheap modification of Groth's approach (or of the other mixnets). We call this procedure DbleMixExp. When there is no need to exponentiate, things are much simpler and cheaper. The corresponding primitive is called DbleMix. The details are given in Appendix C.3.

3.3 Setup phase

The setup phase involves the Setup Component SC, the m Control Components $[\mathsf{CC}_j]_{1 \le j \le m}$, and the Printer. Before running the setup phase, everyone should know the public keys of all actors. During this phase, all messages are broadcast, and must be signed by the emitter. We don't make the signatures explicit. However, encryptions are explicit, and we denote by $\mathsf{pk}_{\mathsf{PS}}$ the public key of the Printer, and $\mathsf{pk}_{\mathsf{CC}}$ a public key whose associated secret key is shared between the CCs.

The set of all the communications (except the decryption part of the AMD) is seen as a transcript that is recorded and checked later on by auditors, before the voting phase starts.

As sketched in Subsection 3.1, the basis of the setup phase is a sub-protocol for masking the choices $v_1, \ldots, v_n$. This is described as Algorithm 1, whose main part is a round of DbleMixExp by SC and of DbleMix by all the CCs. A complication when turning this into a full return code sub-protocol is that SC is offline during the voting phase. The strategy to solve this is to use a so-called login code, printed on the voting card, that contains enough entropy to serve as a secret key, thus creating a secure channel to the Voter. This becomes Algorithm 2.

In addition to the generation of the return codes, Algorithm 2 also generates data for the approve code and the finalization code. All these codes are derived from a code seed CS_j that each party samples and sends to the Printer to simplify the handling of this data. The finalization code is handled similarly as the return codes, but it is much simpler because there is no associated vote that needs to be hidden, so that the masking part disappears.

As for the approve code, this goes in the other direction: this is a short code (typically 9 digits), denoted by AC, that the Voter reads on their voting card and types after having checked that the return code is correct. It is generated during the setup phase as the sum of partial approve codes pAC_j derived from CS_j that is picked at random by CC_j and sent

Parties : SC (party 0) and the CC_j (parties $1,2,\ldots,m$)
Context : $\mathsf{pk}_{\mathsf{CC}}$, $\mathsf{pk}_{\mathsf{PS}}$, $[v_i]_{1\le i\le n}$
Output: $\left[\{v_{\sigma(i)}\}_{\mathsf{pk}_{\mathsf{PS}}}, \{v_{\sigma(i)}^{\mathsf{sv}_{\mathsf{id}}}\}_{\mathsf{pk}_{\mathsf{CC}}}\right]_{1\le i\le n}$, $[\mathsf{pv}_j]_{1\le j\le m}$, π
(where σ and $\mathsf{sv}_{\mathsf{id}}$ are known by no party assuming one of them is honest. The data π is auxiliary data for verifiability.)

Initialization:
  Parties compute $L_{-1} = \left[\{v_i\}_{\mathsf{pk}_{\mathsf{PS}}}, \{v_i\}_{\mathsf{pk}_{\mathsf{CC}}}\right]_{1\le i\le n}$ where the encryptions are with randomness 0.
Round of mixnet:
  For j from 0 to m, party j does the following:
    Picks σ_j, a random permutation, and sv_j, a random exponent in $\mathbb{Z}_q$.
    Computes $\mathsf{pv}_j = g^{\mathsf{sv}_j}$.
    if j = 0: Applies DbleMixExp to L_{j-1}, with parameters σ_j and sv_j, to produce L_j, with associated ZKP π_j.
    if j ≠ 0: Applies DbleMix to L_{j-1}, with parameters σ_j, to produce L_j, with associated ZKP π_j.
    The data $(L_j, \pi_j, \mathsf{pv}_j)$ is broadcast.
Additive masking decryption:
  The CC_j for $1 \le j \le m$ compute the additive masking decryption of the content of $L_m[i][1] = \{v_{\sigma(i)}^{\mathsf{sv}_0}\}_{\mathsf{pk}_{\mathsf{CC}}}$ with their value sv_j. They obtain $v_{\sigma(i)}^{\mathsf{sv}_{\mathsf{id}}}$, where $\mathsf{sv}_{\mathsf{id}} = \mathsf{sv}_0 \sum_{j=1}^{4} \mathsf{sv}_j$.
Finalization:
  All the parties check all the ZKPs that they did not yet check, and they verify that $\mathsf{pv}_0 \neq 1$ and $\prod_{j=1}^{m} \mathsf{pv}_j \neq 1$.
  The final list is L_m, which has the expected format, with $\sigma = \sigma_m \circ \cdots \circ \sigma_1 \circ \sigma_0$.
  The output π is formed of all the data that has been broadcast.

Algorithm 1: Masking choices sub-protocol

(encrypted) to the Printer. Furthermore, a small protocol is run between the CCs so that they all get $\{g^{\mathsf{AC}}\}_{\mathsf{pk}_{\mathsf{CC}}}$.

We summarize all the parts of the setup phase in Figure 3, where we slightly interleave the sub-protocols, and we skip the zero-knowledge proofs for readability. This is run simultaneously for all the Voters. The sequence of computations / communications is as follows:

- SC initiates the protocol and sends data to the CCs and the Printer. Thereafter, it does nothing, except checking the validity of the transcript (which can be done also by an auditor).
- Then, the CCs make a round of mixnet and the AMD for the masking choices sub-protocol, and they do the rest with a few independent broadcasts between them. They finally send data to the Printer.
- At the end, the Printer decrypts the data that it received, with which it prepares the voting material: for each Voter, it computes $\mathsf{RC}_i = \oplus_{j=1}^{m} \mathsf{pRC}_{j,i}$, $\mathsf{AC} = \Sigma_{j=1}^{m} \mathsf{pAC}_j$, and $\mathsf{FC} = \oplus_{j=1}^{m} \mathsf{pFC}_j$. After associating RC_i to $v_{\sigma(i)}$, it organizes the layout as in Figure 1, prints the voting cards, and sends them by postal mail.

Parties : SC (party 0) and the CC_j (parties $1,2,\ldots,m$)
Context : $\mathsf{pk}_{\mathsf{CC}}$, $\mathsf{pk}_{\mathsf{PS}}$, $[v_i]_{1\le i\le n}$
Output : data for the Printer: $\{\mathsf{LC}\}_{\mathsf{pk}_{\mathsf{PS}}}$, $\left[\{v_{\sigma(i)}\}_{\mathsf{pk}_{\mathsf{PS}}}\right]_{1\le i\le n}$, $\left[\{\mathsf{CS}_j\}_{\mathsf{pk}_{\mathsf{PS}}}\right]_{1\le j\le m}$.
data for CC_j: $\mathsf{pk}_{\mathsf{id}}$, $[\mathsf{pv}_j]_{1\le j\le m}$, sv_j, $\left[v_{\sigma(i)}^{\mathsf{sv}_{\mathsf{id}}}\right]_{1\le i\le n}$, CS_j.
data for auditors.

Generation of login data:
  SC picks LC, a 128-bit long nonce and derives from it a secret key $\mathsf{sk}_{\mathsf{id}}$.
  SC broadcasts $\mathsf{pk}_{\mathsf{id}} = g^{\mathsf{sk}_{\mathsf{id}}}$, with a $\mathsf{ZKP}_{\mathsf{DL}}$.
  SC encrypts LC for the Printer, with a $\mathsf{ZKP}_{\mathsf{enc}}$.
Generation of return code data:
  Run the masking choices sub-protocol, where SC, instead of picking a random sv_0, derives it from LC.
  For $j = 1$ to m, CC_j does the following:
    Picks CS_j, a 128-bit long nonce, and encrypts it with $\mathsf{pk}_{\mathsf{PS}}$, with a $\mathsf{ZKP}_{\mathsf{enc}}$.
    Derives from CS_j the return codes $\left[\mathsf{pRC}_{j,i}\right]_{1\le i\le n}$, pAC_j and pFC_j.
    Broadcasts $\{g^{\mathsf{pAC}_j}\}_{\mathsf{pk}_{\mathsf{CC}}}$ with a $\mathsf{ZKP}_{\mathsf{enc}}$.
    Computes $\mathsf{enc}_{\mathsf{expAC}}^{\mathsf{pk}_{\mathsf{CC}}} = \prod_{j=1}^{m} \{g^{\mathsf{pAC}_j}\}_{\mathsf{pk}_{\mathsf{CC}}}$ and broadcasts a signature of it.
    Checks the ZKPs and signatures of others.

Algorithm 2: Code generation sub-protocol

3.4 Voting phase

3.4.1 Return code sub-protocol

The Voter starts by typing their login code LC on their Client. From it, the Client derives $\mathsf{sk}_{\mathsf{id}}$ and sv_0. The Voter also chooses a candidate v among the n possibilities.

Then, the Client encrypts the vote v with pk_{EL}, and computes a masked version of it $\{v^{\mathsf{sv}_0}\}_{\mathsf{pk}_{\mathsf{CC}}}$, encrypted with $\mathsf{pk}_{\mathsf{CC}}$. These ciphertexts are sent to the CCs, together with a ZKP that the same v is used in both ciphertexts, and a signature of all this data with $\mathsf{sk}_{\mathsf{id}}$.

The CCs check the ZKP and the signature, and compute the full mask of the vote. That is, they run the additive masking decryption algorithm to compute together $v^{\mathsf{sv}_{\mathsf{id}}} = v^{\mathsf{sv}_0 \sum_{j=1}^{4} \mathsf{sv}_j}$.

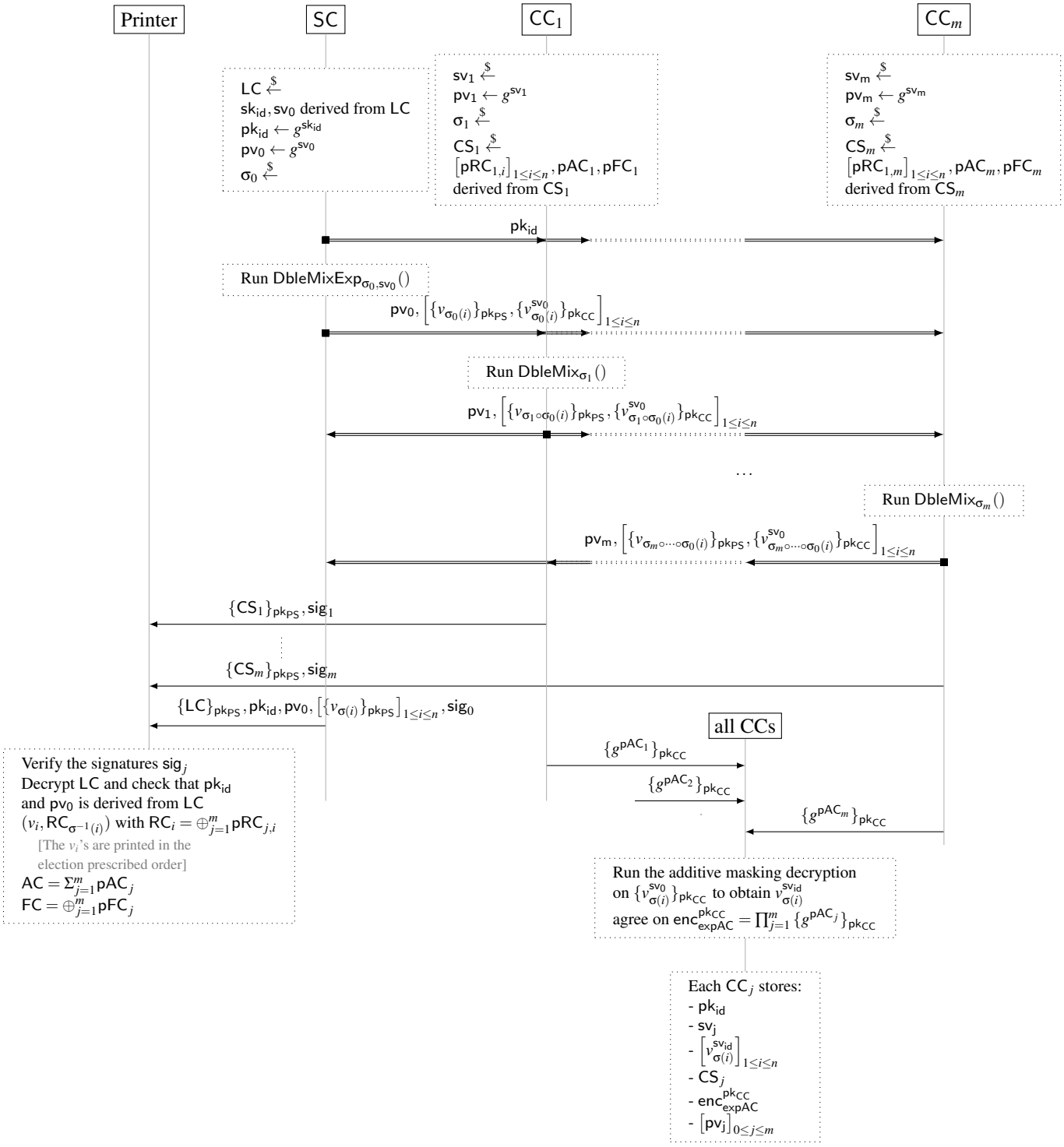


Figure 3: Setup phase. This shows the operations for one Voter. The real protocol runs this in parallel for all Voters. All zero-knowledge proofs and corresponding checks are omitted. σ is defined as $\sigma_m \circ \cdots \circ \sigma_0$ and $\mathsf{sv_{id}}$ as $\mathsf{sv}_0 \sum_{j=1}^m \mathsf{sv_j}$. The doubled arrows are broadcast communications. The Printer receives from this protocol a set of signatures $\left[\mathsf{sig}_j\right]_{1\le j\le m}$ from every party that sign all the data $\mathsf{pk_{id}}, \mathsf{pv_0}, \left[\{v_{\sigma(i)}\}_{\mathsf{pk_{PS}}}\right]_{1\le i\le n}, \left[\mathsf{CS}_j\right]_{1\le j\le m}$.

Afterward, each CC_j finds in its table $\left[v_{\sigma(i)}^{\mathsf{sv_{id}}}\right]_{1\le i\le n}$ which index k contains the $v^{\mathsf{sv_{id}}}$ sent by the Client. It sends back the corresponding partial return code $\mathsf{pRC}_{j,k}$, together with a signature of the ballot they received from the Client.

With this information, the Client verifies the signature and reconstructs the return code RC_k and shows it to the Voter, who can check that this is the one written on the voting card for the candidate v. If this is not the case, the protocol is aborted, and the Voter has to complain.

3.4.2 Approval sub-protocol

When happy with the return code, the Voter reads the approve code AC on the voting card, and types it on the Client. The Client computes g^{AC}, encrypts it with $\mathsf{pk}_{\mathsf{CC}}$, produces a proof of knowledge of the plaintext, and signs this with $\mathsf{sk}_{\mathsf{id}}$.

The CCs, as a whole, could decrypt this and also decrypt the version they had kept from the setup phase. However, we do not want them to learn the value of g^{AC} for the following reason: in the case where the Voter mistypes AC, they should be allowed a second try. On the other hand, a dishonest CC_j colluding with a dishonest device could impersonate the Voter and confirm a malicious vote in their place. For this reason, we resort to a PET protocol in order to compare the plaintexts without decrypting. If the PET confirms that the AC sent by the Voter is correct, then $\{v\}_{\mathsf{pk}_{EL}}$ is added to the list of ballots to be tallied, and audit information is kept, containing the transcript of all the communications.

Then the partial finalization codes are sent to the Client that reconstructs the finalization code, that the Voter can compare with the one on their voting card.

The whole voting phase protocol is summarized in Figure 4.

3.5 Tally

The tally phase can be performed as usual. For example, the Talliers proceed with a round of verifiable mixnets on the list of $\{v\}_{\mathsf{pk}_{EL}}$'s to be tallied, and then perform a verifiable decryption of them using their shared secret key.

A small complication arises, due to the absence of a public bulletin board. At the end of the voting phase, each CC_j has a list $\mathsf{L}_{\mathsf{confirmed},j}$ of ballots that must be tallied, together with audit data. They send this information to the Talliers and to the Auditors who check the validity of the audit data.

In the case of a disagreement between the lists, it is safe to take the union of them. Indeed, if a ballot is present in the list of one of the CCs, then the associated audit data contains the transcript of the PET protocol, that ensures that all the CCs participated in the voting phase for this ballot, and that they should all have included it in their final list.

We remark that such a pre-tally consensus procedure, which is required to avoid some verifiability attacks, has been added only recently in the Swiss Post Voting System and is still missing in CHVote.

3.6 Efficiency and Scalability

3.6.1 Proof-of-Concept Implementation

We have implemented our protocol in Rust [1], as a prototype whose purpose is to run benchmarks and validate that the computational costs are affordable. We used the Ed25519 elliptic curve with the Ristretto abstraction [33] as the underlying prime order group in which DDH is assumed to be difficult, and rely on the `curve25519-dalek` library [22] for an efficient implementation of it. When relevant, the fixed-base scalar multiplication and multi-scalar-multiplications (with appropriate pre-computations) are used.

The implementation does not include the communications and is mono-threaded. In the setup, which is the most critical part, the number of communication rounds is constant, and the computations are trivially parallelizable (*e.g.,* one per voter), so that this is not a problem. Another part that we did not implement is a proper distributed DKG for pk_{EL}, and the corresponding partial decryption during the tally. This is a standard situation, present in most voting protocols, and its cost is known to be affordable [8].

On the other hand, we implemented a simple extension of the protocol, namely the case where the voters choose n_{ans} out of n_{tot} candidates, instead of choosing only one. While this does not change the setup phase, the voting phase becomes more costly, and we wanted to assess the practicality of this important use case, especially for the Client which might be running on a not-so-powerful platform, like a smartphone. During its first step, the Client must compute n_{ans} ballots, each of them being 2 encryptions and 1 $\mathsf{ZKP}_{\mathsf{ptxte}}$. A natural optimization, that we did implement, is to use a multi-recipient ElGamal encryption [4]. This saves roughly a factor of 2 in the encryptions. The $\mathsf{ZKP}_{\mathsf{ptxte}}$ proof can be adapted to this setting in a straightforward way, and the cost is also mostly halved.

3.6.2 Benchmarks

For our test, we used a server running Linux, with two AMD Epyc 9275F processors and 768 GB of RAM. Our implementation uses a single core; in this situation the frequency can reach 4.8 GHz.

Experiments confirm that, as predicted by the theory, if we double the number of voters, the costs are doubled. Similarly, the costs grow linearly with the number of candidates for all the components during the setup, and linearly with the number of choices n_{ans}, during the voting phase and the tally. Due to this effectively measured multi-linear behavior, we ran an experiment with 100 voters and extrapolated the costs for an extreme case with $100,000$ voters who select 100 among $1,000$ candidates. The results are given in Figure 5 and confirm that the setup phase is the bottleneck. This phase can be done in advance of the election and can be parallelized. If each component is a standard 20-core server, this can be

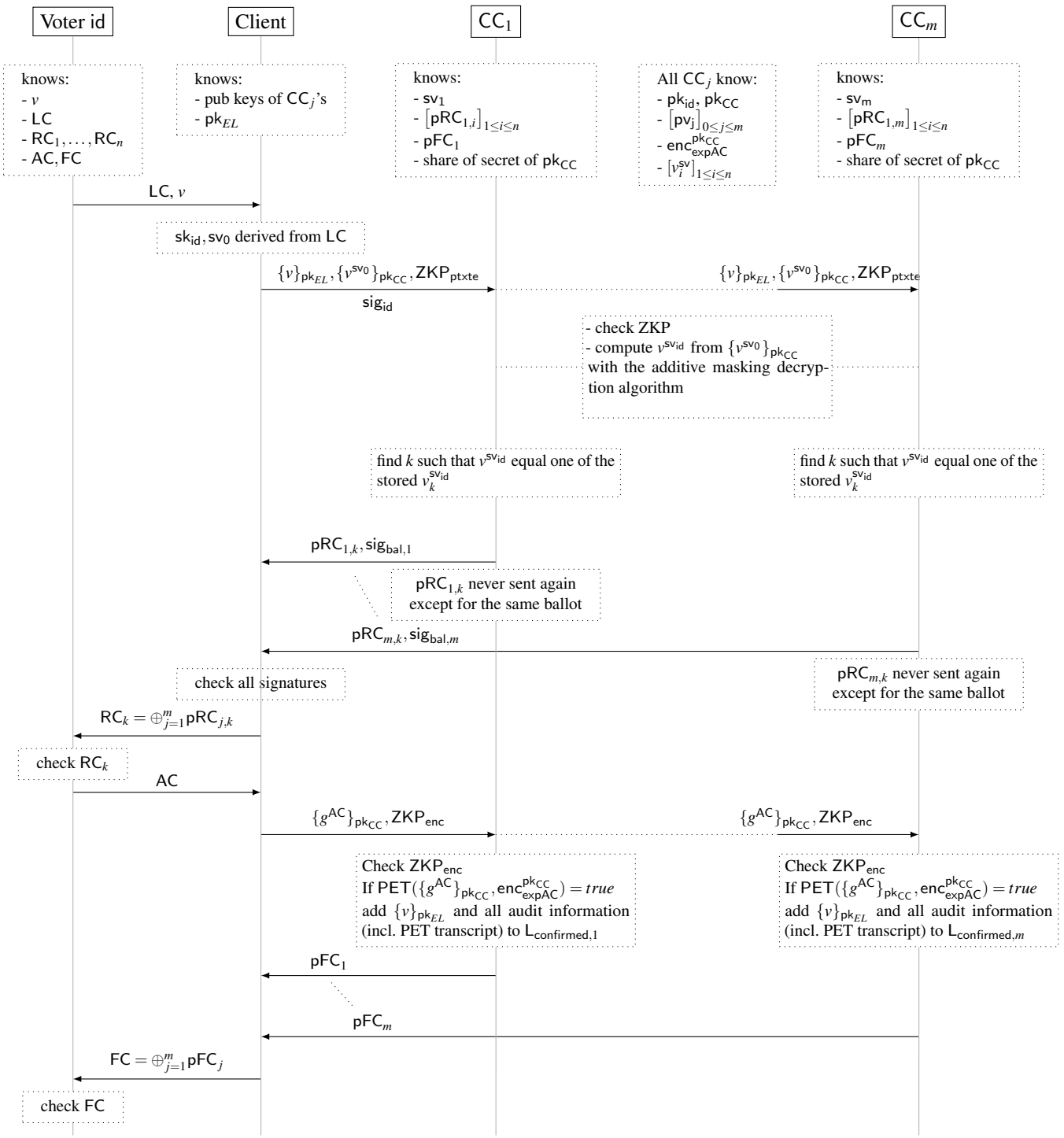


Figure 4: Voting phase.

	SC / Tallier	CC_j / Tallier	Client
Setup	25h36m	43h17m	—
Voting	—	1h46m (63ms per voter)	21ms
Tally	35m02s		—

Figure 5: Running times, on a single core, for each party for each phase, in an election with 100,000 voters who choose 100 among 1,000 candidates. The number of CCs is 4.

done in a few hours.

3.6.3 Comparison with the Swiss Post Voting System

A natural way to assess the practical acceptability of these costs is to compare them with those of the Swiss Post Voting System (SPVS), which has been deployed in multiple election events in Switzerland over the last few years.

The Swiss Post implementation is in Java, which does not provide the same level of optimization as Rust. Therefore, for a fair comparison, we compare the number of group exponentiations. SPVS requires a safe prime field and cannot use elliptic curves. As a consequence, for a similar security level, a full exponentiation in their group is up to 100 times more costly than in the Ristretto group.

We fix the number of CCs to 4, and consider only the dominant terms in the formulae. Let N be the number of voters, n_{tot} the number of candidates, and n_{ans} the number of selections. In SPVS, the cost of the setup phase for SC is $12n_{\text{tot}}N$ exponentiations, while in our protocol this is $66n_{\text{tot}}N$. This is in the same ballpark, and taking into account the faster exponentiations, the cost for the SC will actually be significantly reduced compared to SPVS.

Still in the setup, for the CCs, the cost in our protocol is $95n_{\text{tot}}N$, while for SPVS this is only $2n_{\text{tot}}N$. This difference comes from the different trust model: in SPVS the CCs do not need to check any ZKPs since they trust SC for doing it. Even so, depending on the implementation details, the elliptic curve advantage might more than compensate, and, at worst, the costs will be similar.

During the voting phase, the cost for the Client is $2n_{\text{ans}} + O(1)$ exponentiations in SPVS, while in our case this is $8n_{\text{ans}} + O(1)$. For each CC, the cost per voter is $15n_{\text{ans}} + O(1)$ in SPVS versus $29n_{\text{ans}} + O(1)$ for our protocol. Due to the elliptic curve advantage, our protocol is therefore much cheaper.

For the tally, the SPVS takes advantage of the finite field group situation, and they encode many selected candidates in a single ElGamal ciphertext. Therefore, in SPVS, the cost of the tally does not depend on n_{ans}, while it does in our protocol. For elections where the voters must select a very large number of candidates, it might be the case that the elliptic curve advantage is not enough to compensate this. According to Figure 5, this remains much faster than the setup phase. Therefore, if the setup phase is practical, so is the tally.

4 Security proofs

We analyze our protocol in a computational model. Full details of the definitions and proofs are provided in Appendix F.

4.1 Verifiability

End-to-end verifiability [16] intuitively ensures that the result of the election corresponds to (1) the votes of all honest voters who voted; (2) plus some set of valid votes of size bounded by the number of corrupt voters.

We want to establish end-to-end verifiability against a dishonest Client, assuming that one of the CCs is honest. A dishonest Client that encrypts a candidate different from the voter's choice may always try to guess the return code and will succeed with non-negligible probability. In other words, the end-to-end verifiability definition from [16] used in previous analyzes [10, 46] of voting protocols with return codes, may always be broken with non-negligible probability. However, intuitively, guessing one return code only allows to modify one vote, which is not reflected in the definition of [16]. Therefore, we extend [16] to a quantitative definition in the spirit of [36, 37], where we bound the number of modifications that an attacker may perform. Formally, given two multisets of votes V_1 and V_2, we define the *distance* $dist(V_1, V_2)$ between V_1 and V_2, as the minimum number of atomic operations needed to go from V_1 to V_2, where an atomic operation is either the modification of one vote, the drop of one vote, or the addition of one vote.

Another limitation of standard verifiability definitions [16, 17, 36, 37] is that they provide guarantees as long as all verifications succeed. As soon as someone detects an issue, the election should be canceled. While this makes sense for some verifications such as checking the zero-knowledge proofs, this is no longer acceptable for protocols with cast-as-intended, where a voter may vote for A, receive codes corresponding to A and then pretend they actually voted for B. On the one hand, a single voter should not be given the opportunity to jeopardize the whole election. On the other hand, voters complaints may provide evidence of attacks and cannot be ignored.

Therefore, we propose a novel definition of end-to-end verifiability, that models the probability that an adversary modifies the result of the election up to d modifications, raising no more than k complaints from voters.

Definition 1 (Verifiability). *Let $\mathcal{ID}$ be a set of voters. A voting system V has (p,k,d)-verifiability if for all adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the game $\mathsf{Exp}^{\text{verif}}_{\mathcal{A},\mathsf{V},\mathcal{ID},k,d}(\lambda)$ defined in Figure 6 is bounded by $p + \varepsilon(\lambda)$, where ε is a negligible function.*

In $\mathsf{Exp}^{\text{verif}}_{\mathcal{A},\mathsf{V},\mathcal{ID},k,d}(\lambda)$, the adversary chooses the sets H and C of honest and dishonest voters from $\mathcal{ID}$. It selects which

$\mathsf{Exp}^{\mathrm{verif}}_{\mathcal{A},\mathsf{V},\mathcal{ID},k,d}(\lambda)$

$H, C, j_H \leftarrow \mathcal{A}(2^\lambda)$
if $H \cup C \neq \mathcal{ID}$ **or** $H \cap C \neq \emptyset$ **then return** 0
$\{(\mathsf{id}, \mathsf{usk}_{\mathsf{id}}, \mathsf{ccsk}_{j_H,\mathsf{id}}, \mathsf{upk}_{\mathsf{id}})\}_{\mathsf{id}\in\mathcal{ID}} \leftarrow \mathsf{Setup}_{\mathcal{A}}(2^\lambda)$
$H_{voted}, F_{check}, \mathsf{BB}_{j_H} \leftarrow \{\}$
$\mathcal{A}^{O\mathsf{vote}, O\mathsf{processVote}}()$
if $\exists(\mathsf{id},\star) \in H_{voted}$ such that $(\mathsf{id},\star) \notin F_{check}$ **then return** 0
$\mathsf{BB} \leftarrow \mathsf{FinalizeBB}_{\mathcal{A}}(\mathsf{BB}_{j_H})$
if $\mathsf{IsValid}(\mathsf{upk}_{\mathcal{ID}}, \mathsf{BB}) == 0$ **then return** 0
$(\mathsf{r}, \Pi) \leftarrow \mathsf{Tally}_{\mathcal{A}}(\mathsf{BB})$
if $\mathsf{VerifyTally}(\mathsf{BB}, \mathsf{r}, \Pi) == 0$ **then return** 0
$V_{complaint} = \{\mathsf{id} \,|\, (\mathsf{id},\mathsf{v}) \in H_{voted}, \exists(\mathsf{id}, \Gamma_{\mathsf{id}}) \in F_{check}$
and $\mathsf{VerifyVote}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}, \Gamma_{\mathsf{id}}) == 0\}$
if $|V_{complaint}| > k$ **then return** 0
Let $V_{check} = \{\mathsf{v}_{\mathsf{id}} | \mathsf{id} \in \mathcal{ID}, (\mathsf{id}, \mathsf{v}_{\mathsf{id}}) \in H_{voted}, (\mathsf{id},\star) \in F_{check}\}$
if $\exists V_{\mathcal{A}} \in \mathcal{V}^c$ such that $c \leq |C|$ **and** $dist(r, (V_{\mathcal{A}} \cup V_{check})) < d$
then return 0
else return 1

$O\mathsf{vote}(\mathsf{id}, \mathsf{v})$

if $\mathsf{v} \notin \mathcal{V}$ **then return** 0
if $\mathsf{id} \notin H$ **then return** 0
if $\exists(\mathsf{id},\star) \in H_{voted}$ **then return** 0
$H_{voted} \leftarrow H_{voted} \cup \{\mathsf{id}, \mathsf{v}\}$

$O\mathsf{processVote}(\mathsf{id})$

if $(\mathsf{id},\star) \in F_{check}$ **then return** 0
if $\exists(\mathsf{id},\mathsf{v}) \in H_{voted}$ **then**
$\Gamma_{\mathsf{id}}, \mathsf{BB}_{j_H} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}, BB_{j_H}, \mathsf{ccsk}_{j_H,\mathsf{id}})$
$F_{check} \leftarrow F_{check} \cup \{(\mathsf{id}, \Gamma_{\mathsf{id}})\}$
else $\mathsf{BB}_{j_H} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{BB}_{j_H}, \mathsf{ccsk}_{j_H,\mathsf{id}})$

Figure 6: Verifiability experiment.
The interactive process $\mathsf{Setup}_{\mathcal{A}}$ is defined by the process in Figure 3 where only the Printer and CC_{j_H} are honest. Similarly, $\mathsf{ProcessVote}_{\mathcal{A}}$ corresponds to the voting phase in Figure 4, where CC_{j_H} is honest, the Voter is honest or dishonest depending on its belonging to H or C, and the other parties are dishonest. The processes $\mathsf{FinalizeBB}_{\mathcal{A}}$ and $\mathsf{Tally}_{\mathcal{A}}$ correspond respectively to the consensus and tally phases sketched in Section 3.5 and explained in Appendix F. Concretely, $\mathsf{usk}_{\mathsf{id}}$ is the (private) voting material received by the voter (i.e., LC,RC_i,AC, FC...), $\mathsf{ccsk}_{j_H,\mathsf{id}}$ is the corresponding private data gathered by CC_j during the setup for this voter (i.e., $\mathsf{pRC}_{j_H}[i]$...), and $\mathsf{upk}_{\mathsf{id}}$ the public data associated to the voter (i.e., $\mathsf{pk}_{\mathsf{id}}$, pv_{j}...). Γ_{id} is the transcript of the voting phase from the honest CC view.

CC indexed by j_H is honest. It then runs the setup interacting with CC_{j_H} and starts the voting phase, interacting with honest voters and CC_{j_H}. Any honest voter that starts voting is assumed to perform all the expected verifications. They trigger one complaint if they detect an issue. In particular, a voter that started to vote and is stopped by the adversary will trigger a complaint. Then the adversary wins if he is able to produce a valid board, in interaction with CC_{j_H} such that less than k complaints have been triggered and such that the result r of the election cannot be explained with less than d modifications from the honest votes plus some set of dishonest votes, of size bounded by the number of corrupt voters.

In our protocol, this quantitative notion of verifiability is satisfied and the result depends on $p_{\mathsf{RC}}, p_{\mathsf{AC}}, p_{\mathsf{FC}}$, the probabilities of guessing, respectively, RC, AC or FC.

Theorem 1. *For any k and $d > k$, the protocol has (p,k,d)-verifiability if* PC *and one of the online CC are honest, for $p = \binom{d}{k}(p_{\mathsf{RC}}+1)\,q^{d-k}$, where $q = \max(p_{\mathsf{RC}}, p_{\mathsf{AC}}, p_{\mathsf{FC}}) + n\,p_{\mathsf{AC}}$ and n is the total number of voters.*

We first reduce the verifiability experiment to a pure probabilistic game, using several intermediate hybrid games that are indistinguishable up to a negligible probability. While our transitions for each game hop are standard, one difficulty arose in the modeling of the mixnet and the proofs of shuffle.

A single honest mixer participating in a mixnet composed of several secure shuffles does not trivially ensure the absence of a link between the input and the output. Indeed, a malicious party can "taint" a ciphertext, such that the taint is preserved even through honest mixes [21, 43], even though each individual mix is deemed secure.

To our knowledge, none of the formal security proofs in the game-based setting take the burden of explicitly proving the shuffling of a mixnet composed of dishonest parties. Some works assume the existence of a ZK-style SimProve algorithm outputting the transcript of all the rounds of the mixnet (but which does not model corruption of individual mixing servers) [11, 18], while other works assume a single honest tallier [42, 45]. In [9], a treatment of mixnets with more than one mixer is left as future work. Naturally, this question has already been

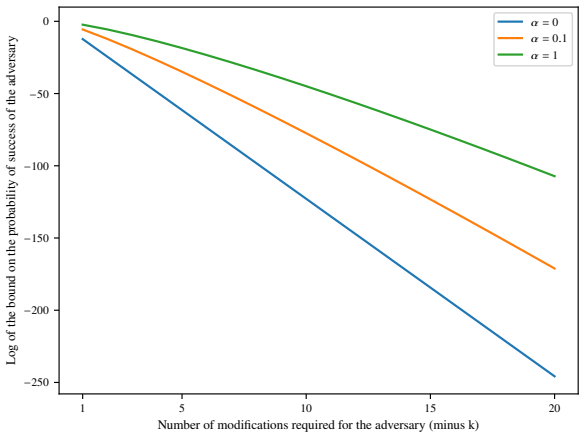


Figure 7: Probability of modifying $k+x$ votes without triggering more than k complaints, where k is set to α% of the total number of voters.

answered in the UC setting [49], but the heavy machinery of UC-style definition overcomplexifies the analysis of real-world voting schemes.

Instead of using a mixing algorithm as a black box, we model it as an ElGamal encryption scheme (which has perfect rerandomization) and a sound NIZK argument that the rerandomization and the shuffle are done correctly. This middle ground between abstracting the full mixnet construction entirely and fully specifying its algorithms enables us to prove our result in a setting with corrupted mixing servers, while being general enough to be compatible with well-known verifiable shuffles [3, 30, 48].

Then we study the winning probability of our pure game. To do this, we identify the optimal winning strategy and prove its optimality. We then provide an explicit formula to compute the resulting winning probability, yielding an exact bound, that is, a bound that is reached for the best adversary. We provide an over-approximation of the bound in Theorem 1 for the sake of readability. In particular, we note that this upper bound is greater than 1 as soon as $n > \frac{1}{p_{\mathsf{AC}}}$ but our actual bound is much tighter. To provide some intuition on the bound provided in Theorem 1, we display in Figure 7 the value of the bound when the number k of tolerated complaints is given as a percentage of the total number of voters. We consider the parameters $p_{\mathsf{RC}} = 10^{-4}, p_{\mathsf{AC}} = 10^{-9}, p_{\mathsf{FC}} = 10^{-8}, n = 10^5$, as typically set in the Swiss setting. Then our bound shows that the adversary may not modify more than $k+10$ votes with probability more than 10^{-50} without triggering at least 1% of complaints among the voters.

Eligibility. When all CCs are dishonest, they may collaborate with the dishonest Client to produce the expected return codes and then register another ballot, hence end-to-end verifiability does not hold. They may also drop any ballot since there is no public bulletin board. We show a weaker form of verifiability when all CCs are dishonest and SC is honest: no ballot can be added. In other words, the CCs are not able to add ballots in the name of absentee voters.

Formally, we define eligibility through a game where the adversary wins if it can produce valid ballots for voters for which it was not given the respective private credentials.

Theorem 2. *The protocol satisfies eligibility if* SC *and* PC *are honest.*

The formal definition of eligibility and the proof of the theorem are provided in Appendix F. The proof follows from the fact that ballots are signed.

4.2 Privacy

To prove vote privacy, we consider the well known definition proposed by Benaloh [5]. While this definition is too restrictive for complex tally functions [9], it perfectly suits our setting where the result of the election is simply the multiset of votes. On the other hand, we had to extend the definition to a more complex setting, where both the setup and the tally are interactive and the voting phase has two main phases (the ballot is sent and then confirmed).

Theorem 3. *The protocol has vote privacy if* PC, TC, *and either* SC *or one of the online* CC*s are honest.*

Intuitively, the TC need to be trusted as a whole (according to the chosen threshold) since they detain the secret key of the election. Similarly, at least one CC or SC needs to be honest since they each detain one share of the masking factor $\mathsf{sv}_{\mathsf{id}}$. The formal definition of vote privacy and the proof of the theorem are provided in Appendix F. While the definition of vote privacy is as expected, the proof requires several intermediate reductions, as in the verifiability proof, to account for the various cryptographic ingredients of our protocol.

5 Discussion

Our protocol satisfies the properties required by the Swiss regulation. Vote privacy is ensured assuming either SC or one of the CCs is honest, thanks to the fact that the honest authority permutes the voting options and detains a private share of $\mathsf{sv}_{\mathsf{id}}$. Importantly, SC remains offline during the voting phase, providing an additional layer of practical security. End-to-end verifiability holds assuming one of the CCs is honest. If they are all dishonest, they may always drop ballots, hence verifiability cannot be guaranteed. To overcome this limitation, it would be necessary to introduce a public bulletin board, which is not currently foreseen in the Swiss context, nor in most politically binding elections. In any case, eligibility is preserved assuming SC or one of the CCs is honest.

In our protocol, the only component that is always trusted is the Printer and its channel to the Voters. This comes from

the fact that, for usability reasons, Voters may only receive their voting material at once, by post. We may imagine in the future that the Voters have access to more sophisticated online services (at least for authentication), which could open new possibilities to weaken the trust on the Printer.

References

[1] Artifact: Proof-of-concept Rust implementation of the protocol. Available at https://inria.hal.science/hal-05422264, 2026.

[2] Ben Adida. Helios: Web-based Open-Audit Voting. In *USENIX'08*, pages 335–348, 2008.

[3] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 2012.

[4] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. Randomness re-use in multi-recipient encryption schemeas. In Yvo Desmedt, editor, *PKC 2003*, pages 85–99. Springer, 2003.

[5] Josh Benaloh. Verifiable secret-ballot elections. Technical report, Yale University Department of Computer Science, 1987.

[6] Josh Benaloh. Simple verifiable elections. *EVT*, 2006.

[7] Josh Benaloh, Michael D. Byrne, Bryce Eakin, Philip T. Kortum, Neal McBurnett, Olivier Pereira, Philip B. Stark, Dan S. Wallach, Gail Fisher, Julian Montoya, Michelle Parker, and Michael Winn. Star-vote: A secure, transparent, auditable, and reliable voting system. In *EVT/WOTE*. USENIX Association, 2013.

[8] Josh Benaloh, Michael Naehrig, and Olivier Pereira. REACTIVE: Rethinking effective approaches concerning trustees in verifiable elections. In *E-Vote-ID 2025*, pages 17–37. Springer, 2005.

[9] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. Sok: A comprehensive analysis of game-based ballot privacy definitions. In *IEEE Symposium on Security and Privacy*, pages 499–516. IEEE Computer Society, 2015.

[10] David Bernhard, Véronique Cortier, Pierrick Gaudry, Mathieu Turuani, and Bogdan Warinschi. Verifiability analysis of CHVote. Cryptology ePrint Archive, Report 2018/1052, 2018. https://eprint.iacr.org/2018/1052.

[11] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643. Springer, 2012.

[12] Ian Brightwell, Jordi Cucurull, David Galindo, and Sandra Guasch. An Overview of the iVote 2015 Voting System. New South Wales Electoral Commission, Australia, 2015.

[13] David Chaum. Sure Vote: Technical Overview. In *Workshop on trustworthy elections (WOTE)*, 2001.

[14] Véronique Cortier, Alexandre Debant, and Pierrick Gaudry. Breaking verifiability and vote privacy in CHVote. In *Esorics 2025 - 30th European Symposium on Research in Computer Security*, 2025.

[15] Véronique Cortier, Alexandre Debant, and Florian Moser. Code voting: when simplicity meets security. In *ESORICS 2024 - 29th European Symposium on Research in Computer Security*, Bydgoszcz, Poland, 2024.

[16] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachene. Election verifiability for Helios under weaker trust assumptions. In *19th European Symposium on Research in Computer Security (ESORICS'14)*, volume 8713 of *LNCS*, pages 327–344, Wroclaw, Poland, September 2014. Springer.

[17] Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. Sok: Verifiability notions for e-voting protocols. In *IEEE Symposium on Security and Privacy*, pages 779–798. IEEE Computer Society, 2016.

[18] Véronique Cortier, Joseph Lallemand, and Bogdan Warinschi. Fifty shades of ballot privacy: Privacy against a malicious board. In *CSF*, pages 17–32. IEEE, 2020.

[19] Chris Culnane, Aleksander Essex, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. Knights and knaves run elections: Internet voting and undetectable electoral fraud. *IEEE Secur. Priv.*, 17(4):62–70, 2019.

[20] Ivan Damgård. On σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 84:84–105, 2002.

[21] George Danezis. Breaking four mix-related schemes based on universal re-encryption. *Int. J. Inf. Sec.*, 6(6):393–402, 2007.

[22] Dalek Cryptography Developers. Dalek elliptic curve cryptography. https://dalek.rs/. Rust package `curve25519-dalek`.

[23] Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge L. Villar. An algebraic framework for diffie-hellman assumptions. In *CRYPTO (2)*, volume 8043

of *Lecture Notes in Computer Science*, pages 129–147. Springer, 2013.

[24] Federal Chancellery. Redesign and relaunch of trials – Final report of the Steering Committee Vote électronique (SC VE). https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting/berichte-und-studien.html, November 2020.

[25] Federal Chancellery. Federal Chancellery Ordinance on Electronic Voting. https://www.fedlex.admin.ch/eli/cc/2022/336/en, July 2022.

[26] Federal Chancellery. E-voting Catalogue of Measures by the Confederation and Cantons, Approved by the Vote électronique Steering Committee (SC VE): 4 August 2023. https://www.bk.admin.ch/en/catalogue-of-measures, 2023.

[27] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[28] David Galindo, Sandra Guasch, and Jordi Puiggali. Neuchâtel's Cast-as-Intended Verification Mechanism. In *International Conference on E-Voting and Identity (VOTE-ID)*, pages 3–18. Springer, 2015.

[29] Kristian Gjøsteen. The Norwegian internet voting protocol. In *Vote-ID 2011*. Springer, 2012.

[30] Jens Groth. A verifiable secret shuffle of homomorphic encryptions. In *PKC 2003*. Springer, 2005. Extended version at https://eprint.iacr.org/2005/246.

[31] Rolf Haenni, Reto E Koenig, and Eric Dubuis. Private Internet Voting on Untrusted Voting Devices. In *International Conference on Financial Cryptography and Data Security*, 2023.

[32] Rolf Haenni, Reto E Koenig, Philipp Locher, and Eric Dubuis. CHVote Protocol Specification. *Cryptology ePrint Archive*, 2017.

[33] Mike Hamburg and Henry de Valence. The Ristretto group. Available at https://ristretto.group/.

[34] Sven Heiberg, Arnis Parsovs, and Jan Willemson. Log analysis of Estonian internet voting 2013–2015. In *Vote-ID 2015*. Springer, 2015.

[35] Sven Heiberg and Jan Willemson. Verifiable internet voting in Estonia. In *International Conference on Electronic Voting (EVOTE)*, 2014.

[36] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. End-to-end verifiable elections in the standard model. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 468–498. Springer, 2015.

[37] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, privacy, and coercion-resistance: New insights from a case study. In *IEEE Symposium on Security and Privacy*, pages 538–553. IEEE Computer Society, 2011.

[38] Karola Marky, Marie-Laure Zollinger, Peter B. Roenne, Peter Y. A. Ryan, Tim Grube, and Kai Kunze. Investigating usability and user experience of individually verifiable internet voting schemes. *ACM Trans. Comput. Hum. Interact.*, 28(5):30:1–30:36, 2021.

[39] Ueli Maurer. Unifying zero-knowledge proofs of knowledge. In Bart Preneel, editor, *AFRICACRYPT 2009*, pages 272–286. Springer, 2009.

[40] Eleanor McMurtry, Olivier Pereira, and Vanessa Teague. When is a test not a proof? In *Computer Security – ESORICS 2020*, pages 23–41. Springer, 2020.

[41] Johannes Mueller and Tomasz Truderung. CAISED: a protocol for cast-as-intended verifiability with a second device. In *International Joint Conference on Electronic Voting*, pages 123–139. Springer, 2023.

[42] Johannes Müller, Balázs Pejó, and Ivan Pryvalov. Devos: Deniable yet verifiable vote updating. *Proc. Priv. Enhancing Technol.*, 2024(1):357–378, 2024.

[43] Birgit Pfitzmann and Andreas Pfitzmann. How to break the direct RSA-implementation of mixes. In *EUROCRYPT*, volume 434 of *Lecture Notes in Computer Science*, pages 373–381. Springer, 1989.

[44] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *FOCS*, pages 543–553. IEEE Computer Society, 1999.

[45] Ben Smyth. Verifiability of Helios mixnet. In *Financial Cryptography Workshops*, volume 10958 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2018.

[46] Swiss Post. Protocol of the Swiss Post Voting System: Computational Proof of Complete Verifiability and Privacy. Version 1.5.0. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/master/Protocol/Swiss_Post_Voting_Protocol_Computational_proof.pdf?ref_type=heads, 2026.

[47] Swiss Post. Swiss Post Voting System: System Specification. Version 1.6.1. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/master/

System/System_Specification.pdf?ref_type=heads, 2026.

[48] Björn Terelius and Douglas Wikström. Proofs of restricted shuffles. In *Progress in Cryptology – AFRICACRYPT 2010*, pages 100–113. Springer, 2010.

[49] Douglas Wikström. A universally composable mix-net. In *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 317–335. Springer, 2004.

A Open Science

Our implementation is provided as an artifact [1] that is made available under an open source licence. This is a prototype that is not meant to be used in production, but allows to reproduce the benchmarks of Figure 5.

The prototype is entirely written in Rust, with the Cargo build system to handle dependencies. It should be easy to compile and run in any hardware/software environment.

We refer to the `Readme.md` file for instructions.

B Ethical Considerations

We did not identify any ethical issues related to this work:

- No attack on any existing product was revealed, so there is no need of responsible disclosure.
- The experiments that were performed did not involve any human being.
- The computational costs required for the benchmarks were small enough to have negligible environmental impact compared to *e.g.,* traveling to the conference.
- The use of Internet voting could be controversial; since we anchor this work in the realm of the Swiss Federal regulations, we estimate that the associated risks are taken into account.

C Non-classical zero-knowledge arguments

C.1 Maurer's framework

In [39], Maurer describes a framework that covers many zero-knowledge proofs. The (slightly simplified) setting is as follows. Let G_1 and G_2 be groups that are finite products of cyclic groups of order q where q is a prime. Let $\phi : G_1 \to G_2$ be a group homomorphism that is efficiently computable by all parties. Let z be a public element of G_2. The purpose of the protocol is to let the prover prove in zero-knowledge that they know an element x in G_1 such that $\phi(x) = z$. We describe the interactive version, which can be turned into a non-interactive proof via the Fiat-Shamir transform.

The prover picks k, a uniform random element of G_1, computes $t = \phi(k)$, and sends t to the verifier. Then, the verifier sends a uniform random element c in $\mathbb{Z}_q$. The prover computes a response $r = kx^c$, and sends it to the verifier. The verifier computes $\phi(r)$ and tz^c, and accepts the proof if and only if they are equal.

We refer to the original paper for the security proof.

In the non-interactive version, where c is obtained by hashing t and the context, the prover can publish only c and r, because t can be recomputed by the verifier as $t = \phi(r)/z^c$,

and then it has to check that indeed c is obtained by hashing this value.

In terms of cost of the protocol, this amounts to one ϕ-computation and one exponentiation in G_1 for the prover, and one ϕ-computation and one exponentiation in G_2 for the verifier (the rest is considered to be negligible).

C.2 $\mathsf{ZKP}_{\mathsf{ptxte}}$: Same plaintext, exponentiated

The $\mathsf{ZKP}_{\mathsf{ptxte}}$ zero-knowledge argument is used during the voting phase by the Client to prove that it encrypted the same vote v in the part that goes to the tally and in the part that is used by the CC_j to select the appropriate return code.

The public data is

- the group $\mathbb{G}$, with generator g; and a second independent generator h;
- pk_1, pk_2, two public keys;
- $C_1 = (c_{1,1}, c_{1,2}) = (g^{r_1}, x\,\mathsf{pk}_1^{r_1})$, an encryption of x with randomness r_1;
- $C_2 = (c_{2,1}, c_{2,2}) = (g^{r_2}, x^k\,\mathsf{pk}_2^{r_2})$, an encryption of the plaintext x, raised to the power k, with randomness r_2;
- $K = g^k$, a commitment to an non-zero exponent k.

The prover proves that they know x, k, r_1 and r_2 such that C_1, C_2, K, pk_1 and pk_2 correspond to these values.

Intuitively, we want to link the plaintext of C_1 and C_2 by raising $c_{1,2}$ to the power k and comparing it to $c_{2,2}$. However, doing so introduces a quadratic term $r_1 k$. To circumvent this issue, the prover publishes an extra ElGamal ciphertext $D = (d_1, d_2) = (g^{r_1 k + r^\star}, x^k \mathsf{pk}_1^{r_1 k + r^\star})$. D can be linked to C_2 with a Sigma protocol, and proving that $D = C_1^k\,(g, \mathsf{pk}_1)^{r^\star}$ ties the relation to C_1. Since D is an ElGamal ciphertext for which no one knows the secret key, we can prove the zero-knowledge property under the DDH assumption. In our protocol, this key corresponds to pk_{EL}, for which the adversary never knows the associated secret key.

Additional data published by the prover:

- $d_1 = g^{r_1 k + r^\star}$, where $r^\star$ is a fresh, uniform random in $\mathbb{Z}_q$;
- $d_2 = x^k \mathsf{pk}_1^{r_1 k + r^\star}$.

Maurer proof: The prover proves that they know $\alpha, k, r_2, r^\star$ such that $d_1 = g^\alpha, K = g^k, c_{2,1} = g^{r_2}, d_1 = c_{1,1}^k\, g^{r^\star}, d_2 = c_{1,2}^k\, \mathsf{pk}_1^{r^\star}, c_{2,2}/d_2 = \mathsf{pk}_1^{-\alpha} \mathsf{pk}_2^{r_2}$.

In Maurer's framework, the map $\phi : \mathbb{Z}_q^4 \to G^6$ is defined by

$$(\rho_1, \rho_2, \rho_3, \rho_4) \mapsto (g^{\rho_1}, g^{\rho_2}, g^{\rho_3}, c_{1,1}^{\rho_2}\, g^{\rho_4}, c_{1,2}^{\rho_2}\, \mathsf{pk}_1^{\rho_4}, \mathsf{pk}_1^{-\rho_1}\, \mathsf{pk}_2^{\rho_3}),$$

which is publicly computable and a group homomorphism. The prover can prove in zero-knowledge that they know a pre-image of this (public) image.

Knowledge-Soundness: The extractor of this Maurer proof gives a tuple $(\alpha, k, r_2, r^\star)$. Given $\alpha, r^\star$ and k, we can reconstruct the discrete log of $c_{1,1}$, $r_1 = (\alpha - r^\star)\, k^{-1}$ since $k \neq 0$.

Complexity: The number of exponentiations for the prover is 11. Indeed, 2 exponentiations are required for computing the additional published data. Then 9 exponentiations are needed for computing ϕ.

For the verifier, the number of exponentiations amounts to 15. They come from 9 exponentiations for computing ϕ, and from 6 exponentiations for exponentiations in the co-domains of ϕ.

C.3 Variants of Groth's Verifiable Shuffle

In the literature, the most used verifiable shuffles are the one of Terelius-Wikström [48], the one of Bayer-Groth [3], and, to a lesser extent, the one of Groth [30]. Groth's algorithm is the cheapest in terms of computational cost and is versatile enough to cover all our needs. We describe here the variants required by our protocol.

C.3.1 Overview of Groth's Algorithms

In [30], the algorithms can be instantiated with any homomorphic commitment scheme, and any homomorphic cryptosystem. We use Pedersen's commitment, with the same underlying group as in the ElGamal encryption.

An important building block is a sub-protocol called "Shuffle of known content argument", which does not involve any ciphertext. It is used in subsequent algorithms to commit a permutation and to show (in zero-knowledge) that this permutation is applied to other data. It is present in all the variants, and the sub-protocol is called only once if the same permutation is applied to several lists.

The main protocol is given in Figure 3 of [30]. This is a zero-knowledge argument of shuffle of a list of ciphertexts. It is presented as a 7-round interactive protocol, that can be turned in a non-interactive protocol with the Fiat-Shamir heuristic. This provides an instance of the Mix algorithm that is used in our protocol.

It can be noted that the operations on the ciphertexts are limited and well isolated in this protocol. This opens the opportunity to derive variants. Most of them are already present in Groth's article:

- Decryption shuffle: in Figure 5 of [30], there is a protocol that combines the proof of correct shuffle together with a partial decryption and the corresponding proof of correct exponentiation. We did not use it in our implementation, because we want to keep the decryption separate from the shuffles. This variant is a very good basis for the DbleMixExp algorithm that is needed in our setup, as shown below.

- Parallel shuffle: the 7th paragraph of Section 6 of [30] explains briefly how to shuffle two lists with the same permutation. We took this version without any modification for implementing DbleMix used by the CCs in the setup. (While not explicitly said in this paragraph, a different R_d must be used for each list.)
- Multi-recipient ElGamal: the description in Figure 3 of [30] works for any homomorphic encryption scheme. Therefore it can be directly used with the multi-recipient ElGamal scheme that we implemented to optimize the case where the voters must select many candidates. In Figure 3, the only modification occurs in the computation of E_d.

C.3.2 Simultaneous Shuffling and Exponentiating

The only variant that is missing is DbleMixExp, the prover shows that two lists have been shuffled with the same permutation, the second list being raised to a secret exponent x, for which $X = g^x$ is public. We explain here how to simultaneously shuffle and exponentiate, i.e. a MixExp protocol. This can then be put in parallel with the plain shuffle of another list with exactly the same technique as for DbleMix.

We take the same notation as in [30]. Our starting point is Figure 5. The modifications to obtain MixExp are as follows. The output list of ciphertexts verifies $(U_i, V_i) = \left(g^{R_i} u_{\pi(i)}^{x_s}, Y^{R_i} v_{\pi(i)}^{x_s}\right)$, where R_i are the rerandomization factors, π is the permutation, Y is the public key, and x_s is the secret exponent for which $y_s = g^{x_s}$ is public.

The values U_d and V_d that replace the commitment E_d of Figure 3 must be adapted: instead of having a single hiding randomizer r_V with its mask d_V, we now need two randomizers r_U and r_V, and two masks d_U and d_V. These 4 values are committed with 4 Pedersen commitments instead of the 2 commitments C_1 and C_2 in Figure 3.

After the first interaction with the verifier, the value U also has to be duplicated in a pair (U, V), with slightly modified formulae.

In the final round, the prover provides check values f_U and Z_U in addition to the f_V and Z_V values, so that the verifier can check the 4 commitments. And finally, the two big formulae of the verifier that involve the ciphertexts are modified to take into account the different relation between (U_i, V_i) and (u_i, v_i).

The security proof follows mutatis mutandis the one given by Groth in Section 5 of [30] for the simultaneous shuffle and partial decryption.

For completeness, we provide pseudocode of this MixExp protocol.

Common input: lists $(u_i, v_i)_{1\le i\le n}$ and $(U_i, V_i)_{1\le i\le n}$ of ciphertexts, public key Y, commit to exponent y_s.

Prover's secret input: permutation π, exponent x_s such that $y_s = g^{x_s}$, scalars $R_1, \dots, R_n$ such that $(U_i, V_i) = \left(g^{R_i} u_{\pi(i)}^{x_s}, Y^{R_i} v_{\pi(i)}^{x_s}\right)$.

Prover's operations:

Pick r, r_d, R_d, $d_1, \dots, d_n$ in $\mathbb{Z}_q$.
Let $c = \mathsf{com}(\pi(1), \dots, \pi(n); r)$.
Let $c_d = \mathsf{com}(-d_1, \dots, -d_n; r_d)$.
Pick r_U, d_U, r_V, d_V, d_x in $\mathbb{Z}_q$.
Let $D = g^{d_x}$.
Let $U_d = \prod_{1\le i\le n} U_i^{-d_i} g^{R_d + r_U}$.
Let $V_d = \prod_{1\le i\le n} V_i^{-d_i} Y^{R_d} g^{r_V}$.
Pick r_1, r_2, r_3, r_4.
Let $C_1 = \mathsf{com}(r_U; r_1)$ and $C_2 = \mathsf{com}(d_U; r_2)$.
Let $C_3 = \mathsf{com}(r_V; r_3)$ and $C_4 = \mathsf{com}(d_V; r_4)$.
Send c, c_d, U_d, V_d, D, C_1, C_2, C_3, C_4 to verifier.
Receive challenges $t_1, \dots, t_n$ from verifier.
Let $f_i = t_{\pi(i)} + d_i$ and $Z = \sum_{1\le i\le n} t_{\pi(i)} R_i + R_d$.
Let $U = g^{d_U} \left(\prod_{1\le i\le n} u_i^{t_i}\right)^{d_x}$.
Let $V = g^{d_V} \left(\prod_{1\le i\le n} v_i^{t_i}\right)^{d_x}$.
Send $f_1, \dots, f_n$, Z, U, V to verifier.
Receive challenges λ, e from verifier.
Set $\rho = \lambda r + r_d$ and run the shuffle of known content argument protocol $\mathsf{Arg}(\pi, \rho | c^\lambda c_d \mathsf{com}(f_1, \dots, f_n; 0)) = \mathsf{com}(\lambda\pi(1) + t_{\pi(1)}, \dots, \lambda\pi(n) + t_{\pi(n)}; \rho)$.
Let $f = ex_s + d_x$.
Let $f_U = er_U + d_U$ and $z_U = er_1 + r_2$.
Let $f_V = er_V + d_V$ and $z_V = er_3 + r_4$.
Send f, f_U, z_U, f_V, z_V to verifier.

Verifier's operations:

Verify the shuffle of known content argument.
Check that $y_s^e D = g^f$.
Check that $C_1^e C_2 = \mathsf{com}(f_U, z_U)$ and $C_3^e C_4 = \mathsf{com}(f_V, z_V)$.
Check that $(\prod_{1\le i\le n} u_i^{t_i})^{-f} (\prod_{1\le i\le n} U_i^{f_i}\, U_d)^e U = g^{eZ + f_U}$.
Check that $(\prod_{1\le i\le n} v_i^{t_i})^{-f} (\prod_{1\le i\le n} V_i^{f_i}\, V_d)^e V = Y^{eZ} g^{f_V}$.

C.3.3 Complexities of Variants of Shuffles

In all the variants, the first step for the prover is to compute a Pedersen commitment of $\pi(1), \dots, \pi(n)$, seen as elements of $\mathbb{Z}_q$. These are short exponents, so that, depending on the implementation, it can be made cheaper than a generic Pedersen commitment but we count it as a generic operation that costs $n + O(1)$ exponentiations.

In the following table, we provide the number of exponentiations for the prover and the verifier, omitting the $O(1)$ terms, for the different variants required in our protocol. In the formulae, n is the length of the lists and k is the number of recipients in the case of multi-recipient ElGamal.

Variant	Prover's cost	Verifier's cost
Mix	$7n$	$7n$
Mix (multi-recipient)	$n(k+7)$	$n(2k+6)$
DbleMix	$9n$	$11n$
DbleMixExp	$16n$	$11n$

D Complexity of each phase of the protocol

We recall the notation: m is the number of CCs; n_{tot} is the number of candidates; n_{ans} is the number of selections; N is the number of Voters; n_{bal} is the number of ballots that have been cast. We count the number of group exponentiations (Exp) and focus on the dominant terms, so that all the counts are to be understood with an additional $+O(1)$ term.

D.1 Setup phase

We count the operations for each voter, and the results have to be multiplied by N.

Using the mixnet described above, the mixing procedure costs $22n_{\text{tot}}$ exponentiations for SC ($6n_{\text{tot}}$ to compute the data, and $16n_{\text{tot}}$ for the proof), and $13n_{\text{tot}}$ exponentiations for the online CCs (the exponentiation by sv_0 is omitted). Verifying SC's and each CC's proof costs $11n_{\text{tot}}$. Each CC also requires $11n_{\text{tot}}$ exponentiations to verify each of the other CC's mixing procedures.

The additive masking decryption procedure costs $8n_{\text{tot}}$ exponentiations plus $10n_{\text{tot}}$ per other party, to check their ZKPs.

The cost of the encryption of the partial return codes is negligible, since they can all be batched together.

Therefore, the overall cost of the setup for SC is $(22+11m)n_{\text{tot}}$ exponentiations per voter, and for the CCs the cost is $(11+21m)n_{\text{tot}}$ exponentiations per voter.

D.2 Voting phase

We assume that the multi-recipient ElGamal is used to handle the case where the number n_{ans} of selections is large.

During the first step, the Client prepares the ballot and the associated proof and signature. This costs $8n_{\text{ans}}+9$ exponentiations. In its second and final steps, the costs no longer depends on n_{ans}. We count 9 additional exponentiations. In total, this gives $8n_{\text{ans}}+18$, where the first term dominates when the number of selections to be done is large.

For each Voter, the cost for the online CCs is small: checking the signature and the ZKP of the voter costs $6n_{\text{ans}}+O(1)$ and running the masking decryption algorithm costs $(5+(m-1)6)n_{\text{ans}}$ exponentiations. Again, the step after sending the partial return codes does not depend on n_{ans} and is therefore negligible. The total cost per voter for the CC's is therefore $(5+6m)n_{\text{ans}}$ exponentiations.

D.3 Tally phase

The costs of producing and proving the shuffle is $2n_{\text{ans}}n_{\text{bal}}$ exponentiations. The cost of verification is the same. Assuming that there are $m+1$ talliers (the SC and the m CCs), we get a cost of $2mn_{\text{ans}}n_{\text{bal}}$ exponentiations for each tallier.

The partial decryption, with their proofs, adds an additional $2mn_{\text{ans}}n_{\text{bal}}$ contribution to the cost.

E Purely probabilistic guessing game

In our security proof of verifiability, we reduce the final hybrid to the guessing experiment $\mathsf{Exp}^{\text{guess}}_{\mathcal{A},k,d,n}()$, which is presented in Figure 8. The goal of the adversary is to reach $D=d$ while keeping $K\le k$, by making at most n OguessAC, OguessRC or OguessFC requests. Each request can increase D, but can also increase K. Especially, an OguessAC call never increases K, but increases D only with a probability p_{AC} while an OguessRC (resp. OguessFC) call always increases D, but increases K with probability p_{RC} (resp. p_{FC}). p_{AC}, p_{RC} and p_{FC} respectively represent the probability of guessing the codes AC, FC and RC, which is $2^{-l_{\mathsf{AC}}}$, $2^{-l_{\mathsf{RC}}}$ and $2^{-l_{\mathsf{FC}}}$ for codes of lengths l_{AC}, l_{RC} and l_{FC}.

This game is purely probabilistic, or in other words, it is not based on any hardness assumption and is hard to win even for an unbounded adversary (given well selected parameters). It is not trivial to find an optimal strategy, because the adversary receives the information whether the call succeeded or not, and can react adaptively to reevaluate its best future move. Actually, it is sometimes suboptimal to set the number of each call request before the game: if we take a negligible p_{RC} and $k=1$, it is always worth it to make an OguessRC call to increase D. This call almost surely increases K, and making another OguessRC call in that case is almost certainly leading to a losing state, but there is a tiny probability that the call succeeds, and in that call it is optimal to call it a second time. That means that we cannot decide in advance how many OguessRC calls we will make before making the first one. Still, the game has enough structure for us to compute an optimal strategy $\mathcal{S}^\star(k,d,n)$, and evaluate the best winning probability $f(k,d,n)$.

There are a few observations that can help find an optimal strategy. First, it is never optimal for the adversary to call an $O\mathsf{guessRC}(i)$ request with $i\neq 1$, since its probability of increasing K can only be greater otherwise. Also, both $O\mathsf{guessRC}(1)$ and $O\mathsf{guessFC}()$ are identical, up to the probability of success. So if we write $p_{max}=max(p_{FC},p_{RC})$, it is always optimal to call the oracle with the high probability of success p_{max}. Without any loss of generality, we consider next that $p_{\mathsf{RC}}\ge p_{\mathsf{FC}}$, and never call $O\mathsf{guessFC}()$ in the guessing experiment.

First, we can design an optimal strategy for the easier case when $k=0$. Then, we will use this optimal strategy as a building block for the general case.

Lemma 1. *For any $d>0$ and $n\ge d$, the best probability of winning the guessing experiment $\mathsf{Exp}^{\text{guess}}_{\mathcal{S}^\star_0,0,d,n}()$ is equal to*

$$f(0,d,n)=\sum_{s=d}^{n-1}\binom{n-1}{s}p_{\mathsf{AC}}^{s}(1-p_{\mathsf{AC}})^{n-1-s}+\sum_{t=1}^{d}p_{\max}^{t}\binom{n-t-1}{d-t}p_{\mathsf{AC}}^{d-t}(1-p_{\mathsf{AC}})^{n-d}$$

$\mathsf{Exp}^{\text{guess}}_{\mathcal{A},k,d,n}()$

$D \leftarrow 0, K \leftarrow 0, N \leftarrow 0$

$\mathcal{A}^{O\text{guessRC},O\text{guessAC},O\text{guessFC}}()$

if $D >= d$ **and** $K \leq k$ **and** $N \leq n$ **then return** 1

else return 0

$O\text{guessRC}(i)$

if $i < 1$ **then return** 0

N += 1

Sample $b = \begin{cases} 1 & \text{with prob. } p^i_{\mathsf{RC}} \\ 0 & \text{with prob. } 1 - p^i_{\mathsf{RC}} \end{cases}$

D += 1

K += (1-b)

return b

$O\text{guessAC}()$

N += 1

Sample $b = \begin{cases} 1 & \text{with prob. } p_{\mathsf{AC}} \\ 0 & \text{with prob. } 1 - p_{\mathsf{AC}} \end{cases}$

D += b

return b

$O\text{guessFC}()$

N += 1

Sample $b = \begin{cases} 1 & \text{with prob. } p_{\mathsf{FC}} \\ 0 & \text{with prob. } 1 - p_{\mathsf{FC}} \end{cases}$

D += 1

K += 1-b

return b

Figure 8: Guessing experiment.

where $p_{\max} = \max(p_{\mathsf{AC}}, p_{\mathsf{FC}}, p_{\mathsf{RC}})$.

Proof: First, there is an optimal strategy $\mathcal{S}^\star_0$ that consists in only making $O\text{guessAC}$ calls, and then only make $O\text{guessRC}$ calls with the leftover budget. We name them respectively A-calls and R-calls.

Indeed, if an optimal strategy makes an A-call after an R-call, it never worsens our probability of success to move the R-call after the A-call. We must succeed that R-call (otherwise, we have one complaint making us lose), and thus cannot adapt to the result of that call. If the optimal strategy is winning, it always makes that A-call, and reordering these calls does not change the probability of winning. By applying this argument recursively, we can move all of the A-calls before the R-calls.

There are two values that can help any strategy to adaptively choose when to switch to a A-calls regime to the R-calls regime : the number of times we still have to increase D (given by $d-D$), and the number of remaining calls $T = n - N$ we can still make. There are three cases:

- $d - D < T$: We have to win $d - D$ more calls in the next T calls. If we decide to switch to the R-calls regime, we will use only $d - D$ of these calls, hence we will not use all of our budget. It is thus never worth it to switch to the $O\text{guessRC}$ regime here.
- $d - D > T$: We have to win more calls than the number of turns we have. In that case, we always lose the game.
- $d - D = T$: We need to win all of our remaining calls. It is optimal to choose the call that has the better probability of success $p_{max} = \max p_{\mathsf{RC}}, p_{\mathsf{AC}}$.

An optimal strategy is thus to only do A-calls until we only have $d - D$ calls left, and then switch to the call that has the better individual probability of success. We denote by $T^\star$ the random variable accounting the number of steps left when we do the switch.

We can compute the probability of success of this strategy as follows:

$$
\begin{aligned}
f(0,d,n) &= Pr[d = D \text{ before } T = D - d] \\
&\quad + Pr[d = D \text{ not before } T = D - d] \\
&= Pr[\text{At least } d \text{ wins in the first } n-1 \text{ calls}] \\
&\quad + \sum_{t=1}^{d} Pr[d = D | T^\star = t] \, Pr[T^\star = t] \\
&= Pr[\text{At least } d \text{ wins in the first } n-1 \text{ calls}] \\
&\quad + \sum_{t=1}^{d} Pr[t \text{ successes in the last } t \text{ calls}] \\
&\quad \times Pr\begin{bmatrix} d-t \text{ successes in the first } n-t-1 \text{ calls} \\ \text{and the } n-t\text{-th call is not a success} \end{bmatrix} \\
&= \sum_{s=d}^{n-1} \binom{n-1}{s} p^s_{\mathsf{AC}} (1 - p_{\mathsf{AC}})^{n-1-s} \\
&\quad + \sum_{t=1}^{d} p^t_{\max} \binom{n-t-1}{d-t} p^{d-t}_{\mathsf{AC}} (1 - p_{\mathsf{AC}})^{n-d} \qquad \square
\end{aligned}
$$

Now, we can use this strategy for the case $k = 0$ to build an optimal strategy for the general case.

Lemma 2. *For any $k \geq 1$, $d > k$ and $n \geq d$, the best probability of winning the guessing experiment $\mathsf{Exp}^{\text{guess}}_{\mathcal{S},k,d,n}()$ is equal*

to

$$f(k,d,n) = \sum_{i=0}^{k-1} \binom{d}{i} (1-p_{\mathsf{RC}})^i p_{\mathsf{RC}}^{d-i} + \sum_{j=k}^{d} \binom{j-1}{j-k} p_{\mathsf{RC}}^{j-k} (1-p_{\mathsf{RC}})^k f(0,d-j,n-j).$$

Proof: If $k \neq 0$, there is an optimal strategy that makes one R-call as its first move. Indeed, given an optimal strategy $\mathcal{S}^\star$, we can design another one $\mathcal{S}^R$ that starts with an R-call. Then, it simulates the optimal strategy, and mimics its behavior until $\mathcal{S}^\star$ makes its first R-call. That is, each time $\mathcal{S}^\star$ makes an A-call, $\mathcal{S}^R$ makes the same call, and forwards the bit response b. When $\mathcal{S}^\star$ makes its first R-call, we give it the bit b we received for our first call. At this point, we have the same number N, K and D. Then, we continue mimicking all of $\mathcal{S}^\star$'s calls.

There are two possibilities:

- $\mathcal{S}^\star$ makes at least one R-call, and in that case, $\mathcal{S}^R$ made the exact same calls (but in a different order), so our probability of winning is the same.
- $\mathcal{S}^\star$ never makes an R-call, but in that case $\mathcal{S}^R$ made a call increasing D for free, since one such call never busts our complaints budget and always increases D.

We can apply this argument recursively and make R-calls as long as we still have one free complaint left (that is, $K < k$). However, if $K = k$, one extra R-call can make us lose the game (by busting our complaint budget), and this R-call might not be optimal anymore. Then, we can treat the end of the game as a guessing experiment with $k = 0$, and use the strategy of lemma 1.

Let $T^\star$ be a random variable counting the number of R-calls we make until we use all of our complaints budget. Given the best probability $f(0,d,n)$ at winning $\mathsf{Exp}^{\text{guess}}_{\mathcal{S},0,d,n}()$, we can compute the best probability of winning $\mathsf{Exp}^{\text{guess}}_{\mathcal{S},k,d,n}()$ as

$$\begin{aligned}
&f(k,d,n)\\
&= Pr[\text{at most } k-1 \text{ complaints among the first } d \text{ calls}]\\
&\quad + Pr[\text{the } k\text{-th complaint happens before } D = d]\\
&= \sum_{i=0}^{k-1} \binom{d}{i} (1-p_{\mathsf{RC}})^i p_{\mathsf{RC}}^{d-i}\\
&\quad + \sum_{j=k}^{d} Pr[T^\star = j] Pr[\mathsf{Exp}^{\text{guess}}_{\mathcal{S}^0,0,d-j,n-j}() = 1]\\
&= \sum_{i=0}^{k-1} \binom{d}{i} (1-p_{\mathsf{RC}})^i p_{\mathsf{RC}}^{d-i}\\
&\quad + \sum_{j=k}^{d} Pr\left[\begin{array}{l}k-1 \text{ fails in the first } j-1\\ \text{calls and the } j\text{-th call fails}\end{array}\right] f(0,d-j,n-j)\\
&= \sum_{i=0}^{k-1} \binom{d}{i} (1-p_{\mathsf{RC}})^i p_{\mathsf{RC}}^{d-i}\\
&\quad + \sum_{j=k}^{d} \binom{j-1}{j-k} p_{\mathsf{RC}}^{j-k} (1-p_{\mathsf{RC}})^k f(0,d-j,n-j) \quad \square
\end{aligned}$$

From lemma 1 and 2, we can highlight one optimal strategy. This strategy makes OguessRC calls until we reach $K = k$, and then makes OguessAC calls until $n - N = d - D$. At that point, we cannot fail any call anymore, and the strategy makes the call with the best individual success probability, either OguessRC or OguessAC. While the probability of lemma 2 is exact, because our optimal strategy reaches it, it is hard to have an intuition to interpret it. For this reason, we propose an upper bound of this probability, which albeit not tight, is easier to interpret.

First, we can bound the probability $f(0,d,n)$ of lemma 1:

$$\begin{aligned}
f(0,d,n) &= \sum_{s=d}^{n-1} \binom{n-1}{s} p_{\mathsf{AC}}^s (1-p_{\mathsf{AC}})^{n-1-s}\\
&\quad + \sum_{t=1}^{d} p_{\max}^t \binom{n-t-1}{d-t} p_{\mathsf{AC}}^{d-t} (1-p_{\mathsf{AC}})^{n-d}\\
&= Pr[Bin(n-1, p_{\mathsf{AC}}) \geq d]\\
&\quad + \sum_{t=1}^{d} \binom{n-t-1}{d-t} p_{\mathsf{AC}}^{d-t} (1-p_{\mathsf{AC}})^{n-d} p_{\max}^t\\
&= \sum_{s=d}^{n-1} \binom{n-1}{s} p_{\mathsf{AC}}^s (1-p_{\mathsf{AC}})^{n-1-s}\\
&\quad + \sum_{s=0}^{d-1} p_{\max}^{d-s} \binom{n-1}{s} p_{\mathsf{AC}}^s (1-p_{\mathsf{AC}})^{n-1-s}
\end{aligned}$$

Dropping the $(1-p_{\mathsf{AC}})$ terms, and bounding $\binom{n-1}{s} p_{\mathsf{AC}}^s \leq \frac{((n-1)p_{\mathsf{AC}})^s}{s!} \leq \binom{d}{s}((n-1)p_{\mathsf{AC}})^s \leq \binom{d}{s}(np_{\mathsf{AC}})^s$ yields

$$\begin{aligned}
f(0,d,n) &\leq \sum_{s=d}^{n-1} \binom{n-1}{s} p_{\mathsf{AC}}^s (1-p_{\mathsf{AC}})^{n-1-s}\\
&\quad + \sum_{s=0}^{d-1} p_{\max}^{d-s} \binom{n-1}{s} p_{\mathsf{AC}}^s (1-p_{\mathsf{AC}})^{n-1-s}\\
&\leq (np_{\mathsf{AC}})^d + \sum_{s=0}^{d-1} \binom{d}{s} p_{\max}^{d-s} (np_{\mathsf{AC}})^s,
\end{aligned}$$

which is the binomial expansion of $(p_{\max} + np_{\mathsf{AC}})^d = q^d$, for $q = p_{\max} + np_{\mathsf{AC}}$. We can proceed similarly for the part relative to lemma 2.

$$\begin{aligned}
f(0,d,n) &\leq \sum_{i=0}^{k-1} \binom{d}{i} (1-p_{\mathsf{RC}})^i p_{\mathsf{RC}}^{d-i}\\
&\quad + \sum_{j=k}^{d} \binom{j-1}{j-k} p_{\mathsf{RC}}^{j-k} (1-p_{\mathsf{RC}})^k F(d-j,n-j)\\
&\leq \sum_{i=d-k+1}^{d} \binom{d}{i} p_{\mathsf{RC}}^i (1-p_{\mathsf{RC}})^{d-i}
\end{aligned}$$

$$
\begin{aligned}
&+\sum_{j=k}^{d}\binom{j-1}{j-k}p_{\mathsf{RC}}^{j-k}q^{d-j}\\
&\leq\binom{d}{d-k+1}p_{\mathsf{RC}}^{d-k+1}+q^{d-k}\sum_{j=k}^{d}\binom{j-1}{j-k}\\
&\leq\binom{d}{k-1}p_{\mathsf{RC}}^{d-k+1}+q^{d-k}\binom{d}{k}\\
&\leq\binom{d}{k}(p_{\mathsf{RC}}+1)\,q^{d-k}
\end{aligned}
$$

where we use the hockey stick identity to bound the sum of binomials. Finally, we find the bound on the best probability of winning the guessing game, $f(k,d,n)\leq\binom{d}{k}(p_{\mathsf{RC}}+1)\,q^{d-k}$, where $q=p_{\max}+n\,p_{\mathsf{AC}}$.

F Details of the computational proofs

F.1 Model

In order to prove Theorem 1-3 in the computational model, we need to formally define the settings of the voting protocol. As usual, we describe the participants of the protocol, the various algorithms run in an execution of the protocol, and the game-based definitions of the security properties.

F.1.1 Participants

We use the same participants as defined in Section 2.1. We denote by $\mathcal{ID}$ the set of voters participating in the election. For the Talliers, we consider a particular case where the talliers are composed of the online CCs, as well as an extra component played by the electoral board TC_{EB}. This case is aligned with the Swiss setting. Merging CC with all the talliers but one renders the proof slightly more complex since they use their public key $\mathsf{pk}_{\mathsf{CC}}$ as a part of the election key $\mathsf{pk}_{\mathsf{EL}}$. We show that the protocol preserves privacy nevertheless.It is easy to extract from our security proofs the more abstract theorem where the Talliers are left unspecified.

F.1.2 Algorithms

In order to define the security properties of a voting protocol, we need to specify the syntax of the algorithms composing the protocols.

Definition 2 (Voting System)**.** *A Voting System is a tuple of algorithms* (SetupParams, SetupKey, Setup, CreateBallot, ProcessVote, VerifyVote, FinalizeBB, IsValid, Tally, VerifyTally)*:*

- SetupParams *is an algorithm run by* PC *that generates common public parameters, including the cryptographic parameters of the group used by all the cryptographic primitives and the encoding of the several candidates the voter can vote for. The public parameters* pp *is considered an implicit argument for all the following algorithms.*
- SetupKey *is a protocol run by* $\mathsf{PC},\mathsf{SC},\mathsf{TC}_{EB}$ *and each* CC_j*. It takes as input the security parameters and outputs the party's private and public key. We denote by* keys *the set of public keys, which is shared to every participant of the election. The set of public keys* keys *is considered an implicit argument for all the following algorithms, and the secret key* sk *of a party is considered an implicit argument for all the following algorithms run by this party.*
- Setup *is a protocol run by the* PC, SC *and each* CC_j*. It takes as input all the parties public keys and outputs:*
  - $(\mathsf{id},\mathsf{ccsk}_{j,\mathsf{id}},\mathsf{upk}_{\mathsf{id}})$ *to* CC_j*.*
  - $(\mathsf{id},\mathsf{usk}_{\mathsf{id}},\mathsf{upk}_{\mathsf{id}})$ *to* PC*.*
  - $(\mathsf{id},\mathsf{upk}_{\mathsf{id}})$ *to* TC_{EB}*.*

  We write the set $\{(\mathsf{id},\mathsf{upk}_{\mathsf{id}})\}_{\mathsf{id}\in\mathcal{ID}}$ *as* $\mathsf{upk}_{\mathcal{ID}}$*.* Setup *also outputs for each party a boolean* success*, which indicates if the party notices any failure event during the protocol (*e.g.*, through the verification of the signatures and the proofs).*
- CreateBallot *is a protocol run by the voter and the Client. The voter takes as input their credential* $\mathsf{usk}_{\mathsf{id}}$ *and a valid voting intention* v*, and interacts with the Client which outputs a ballot* b *(later to be sent to all the CCs).*
- ProcessVote *is a protocol run by the voter, the Client and the online CCs.*

  The voter takes as input the credential $\mathsf{usk}_{\mathsf{id}}$ *and their voting intention* v*, the Client takes as input a ballot* b *and the* CC*s take as input the CC's part of the voter's credential* $\mathsf{ccsk}_{\mathsf{id}}$*, and the CC's view of the bulletin board* BB_j*.*

  At the end of the protocol, the voter receives as output a transcript of this interaction Γ_{id} *and each* CC_j *updates their view with a finalized ballot* b^{final}*.*
- VerifyVote *is an algorithm run by a voter that takes as input the credential* $\mathsf{usk}_{\mathsf{id}}$*, the voting intention* v *and the transcript* Γ_{id} *of the* FinalizeVote *protocol and outputs either* 1 *if the voter is convinced or* 0 *otherwise.*
- FinalizeBB *is a protocol run by all the online CCs. Each* CC_j *takes as input their view of the bulletin board* BB_i *and they all output a bulletin board* BB *to be tallied.*
- IsValid *is an algorithm that takes as input the public credential of a voter* $\mathsf{upk}_{\mathsf{id}}$ *and a ballot* b*, and outputs* 1 *if this ballot is valid for this credential, and* 0 *otherwise.*

We extend this algorithm to take as well as input the public part of the setup $\{\mathsf{upk}_{\mathsf{id}}\}_{\mathsf{id}\in I_{\mathcal{D}}}$ *and the finalized bulletin board* BB *and outputs* 1 *if the bulletin board is valid and should be tallied or* 0 *otherwise.*

- Tally *is a protocol run by* TC_{EB} *and the online* CCs*. Each party takes as input the bulletin board* BB *and outputs the result of the election* r *and a proof that the tally is correctly computed* Π*.*

- VerifyTally *is an algorithm which takes the bulletin board* BB*, the result of the election* r *and the proof of tally* Π *and returns* 0 *or* 1*.*

F.1.3 Security definitions

We can now formally define the security properties. First, verifiability is defined as in Def. 1, but we make a few changes of experiment $\mathsf{Exp}^{\text{verif}}_{\mathcal{A},\mathsf{V},I_{\mathcal{D}},k,d}(\lambda)$ in Figure 9 to fit the syntax of the previous section. The changes are highlighted in blue.

Next, we define eligibility. For this property, we consider that all the online CCs are dishonest, but the setup component SC is honest. In the experiment, the adversary chooses a set of absentee voters, and a set of corrupted voters. For the latter, the adversary can freely create ballots using their credentials, and publish a bulletin board BB. At the end of the experiment, the adversary wins if it manages to create a ballot on BB which is valid for the public credentials of one of the absentee voters.

Definition 3 (Eligibility)**.** *A voting system* V *has eligibility if for all adversary* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins the game* $\mathsf{Exp}^{\text{elig}}_{\mathcal{A},\mathsf{V},I_{\mathcal{D}}}(\lambda)$ *defined in Figure 10 is bounded by* $\varepsilon(\lambda)$*, where* ε *is a negligible function.*

Finally, we define privacy. For this security property, we consider two scenarios: one where all online CCs are corrupted but SC is honest, and one where SC and all but one CC are corrupted. We capture both in a single game, using a color code in Figure 11 to distinguish them (green for the former, blue for the latter).

Definition 4 (Privacy)**.** *A voting system* V *has privacy if for all adversary* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins the game* $\mathsf{Exp}^{\text{priv}}_{\mathcal{A},\mathsf{V},I_{\mathcal{D}}}(\lambda)$ *defined in Figure 11 is bounded by* $\varepsilon(\lambda)$*, where* ε *is a negligible function.*

F.2 Building blocks

This section presents the formal definition of the algorithms that we use in our protocol. Compared to Section 3.2, we give here the exact syntax of the algorithms, as well as the security properties we require from them.

F.2.1 Computational setting

We rely on an efficient GrpSetup algorithm to generate $\mathbb{G}$. This algorithm takes as input the security parameters λ and generates the group parameters of a suitable cryptographic group. More precisely, $\mathsf{GrpSetup}(\lambda)$ outputs the parameters $(\mathbb{G},p,g)$, where $\mathbb{G}$ is a group of prime order p and g is a random generator of the group.

We only assume that the *DDH* assumption holds for this group, which means that for any PPT adversary $\mathcal{A}$, and random values $x,y,z \leftarrow \mathbb{Z}_p$, the probability that $\mathcal{A}$ distinguishes (g,g^x,g^y,g^{xy}) from (g,g^x,g^y,g^z) is negligible in 2^λ. $(\mathbb{G},p,g)$ is considered to be an implicit argument for all of the primitives we are using the protocol.

F.2.2 NIZK

We use in our protocol a generic Non-Interactive Zero-Knowledge (NIZK) proof (or argument) system as a pair of algorithms $(\mathsf{Prove},\mathsf{Verify})$ for a relation R:

- $\mathsf{Prove}(x,w)$ takes as input a statement x, a witness w, and outputs a proof π that $(x,w)\in R$. As a shorthand, we use the notation: $\mathsf{Prove}(\{(x,w):\phi(x,w)\})$ to compute a proof for the relation $R=\{(x,w):\phi(x,w)\}$.

  We also use a couple of shortcuts for common relations:

  - $\mathsf{Prove}^{DL}(x=(g,y),w) :=$ $\mathsf{quadProve}(\{(x,w):y=g^w\})$.
  - $\mathsf{Prove}^{Enc}(x=(g,\mathsf{pk},c),w=(m,r)) :=$ $\mathsf{Prove}(\{(x,w):c=\mathsf{Enc}(\mathsf{pk},m;r)\})$.
  - $\mathsf{Prove}^{Exp}(x=(g_1,g_2,y_1,y_2),w) :=$ $\mathsf{Prove}(\{(x,w):y_1=g_1^x\wedge y_2=g_2^x\})$

- $\mathsf{Verify}(x,\pi)$ takes as input a statement x and a proof π and outputs 1 if the proof is convincing that there exists w such that $(x,w)\in R$ and 0 otherwise. As a shorthand, we use the notation $\mathsf{Verify}(\pi)$ and leave the statement and the relation implicit.

All the proofs that we use are so-called Σ-protocols [20] with the strong Fiat-Shamir heuristic [27]. We seek several properties from this proof system:

- **Completeness**: For all $(x,w)\in R$, $\mathsf{Verify}(\mathsf{Prove}(x,w))$ outputs 1 with an overwhelming probability.

- **Soundness**: For all x such that for all w, $(x,w)\notin R$, and for any PPT adversary $\mathcal{A}$, $\mathcal{A}(x)$ outputs a proof π such that $\mathsf{Verify}(x,\pi)=1$ with negligible probability.

- **Zero-Knowledge**: There exists a PPT simulator $\mathcal{S}$ such that for any $(x,w)\in R$, $\pi\leftarrow\mathsf{Prove}(x,w)$ and $\pi\leftarrow\mathcal{S}(x)$ are computationally indistinguishable. We commonly say that we simulate a proof as a shorthand of using $\mathcal{S}$ on x.

$\mathsf{Exp}^{\text{verif}}_{\mathcal{A},\mathsf{V},\mathcal{ID},k,d}(\lambda)$

$H, C, j_H \leftarrow \mathcal{A}(2^\lambda)$
if $H \cup C \neq \mathcal{ID}$ **or** $H \cap C \neq \emptyset$ **then return** 0
$\mathsf{pp} \leftarrow \mathsf{SetupParams}(2^\lambda)$
$\mathsf{keys} \leftarrow \mathsf{SetupKey}_{\mathcal{A}}(2^\lambda)$
$\{(\mathsf{id}, \mathsf{usk}_{\mathsf{id}}, \mathsf{ccsk}_{j_H,\mathsf{id}}, \mathsf{upk}_{\mathsf{id}})\}_{\mathsf{id}\in\mathcal{ID}}, \mathsf{success} \leftarrow \mathsf{Setup}_{\mathcal{A}}(2^\lambda)$
if $\mathsf{success} == 0$ **then return** 0
$H_{voted}, F_{check}, \mathsf{BB}_{j_H} \leftarrow \{\}$
$\mathcal{A}^{\mathcal{O}\mathsf{vote},\mathcal{O}\mathsf{processVote}}()$
if $\exists(\mathsf{id}, \star) \in H_{voted}$ such that $(\mathsf{id}, \star) \notin F_{check}$ **then return** 0
$\mathsf{BB} \leftarrow \mathsf{FinalizeBB}_{\mathcal{A}}(\mathsf{BB}_{j_H})$
if $\mathsf{IsValid}(\mathsf{upk}_{\mathcal{ID}}, \mathsf{BB}) == 0$ **then return** 0
$(\mathsf{r}, \Pi) \leftarrow \mathsf{Tally}_{\mathcal{A}}(\mathsf{BB})$
if $\mathsf{VerifyTally}(\mathsf{BB}, \mathsf{r}, \Pi) == 0$ **then return** 0
$V_{complaint} = \{\mathsf{id} \,|\, (\mathsf{id}, \mathsf{v}) \in H_{voted}, \exists(\mathsf{id}, \Gamma_{\mathsf{id}}) \in F_{check}$
and $\mathsf{VerifyVote}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}, \Gamma_{\mathsf{id}}) == 0\}$
if $|V_{complaint}| > k$ **then return** 0
Let $V_{check} = \{\mathsf{v}_{\mathsf{id}} | \mathsf{id} \in \mathcal{ID}, (\mathsf{id}, \mathsf{v}_{\mathsf{id}}) \in H_{voted}, (\mathsf{id}, \star) \in F_{check}\}$
if $\exists V_{\mathcal{A}} \in \mathcal{V}^c$ such that $c \leq |C|$ **and** $dist(r, (V_{\mathcal{A}} \cup V_{check})) < d$
then return 0
else return 1

$\mathcal{O}\mathsf{vote}(\mathsf{id}, \mathsf{v})$

if $\mathsf{v} \notin \mathcal{V}$ **then return** 0
if $\mathsf{id} \notin H$ **then return** 0
if $\exists(\mathsf{id}, \star) \in H_{voted}$ **then return** 0
$H_{voted} \leftarrow H_{voted} \cup \{\mathsf{id}, \mathsf{v}\}$

$\mathcal{O}\mathsf{processVote}(\mathsf{id}, \mathsf{b})$

if $(\mathsf{id}, \star) \in F_{check}$ **then return** 0
if $\exists(\mathsf{id}, \mathsf{v}) \in H_{voted}$ **then**
$\Gamma_{\mathsf{id}}, \mathsf{BB}_{j_H} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}, \mathsf{b}, \mathsf{BB}_{j_H}, \mathsf{ccsk}_{j_H,\mathsf{id}})$
$F_{check} \leftarrow F_{check} \cup \{(\mathsf{id}, \Gamma_{\mathsf{id}})\}$
else $\mathsf{BB}_{j_H} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{b}, \mathsf{BB}_{j_H}, \mathsf{ccsk}_{j_H,\mathsf{id}})$

Figure 9: Verifiability experiment, as used in the proof of security. The changes of this experiment compared to the experiment in Fig 6 are highlighted in blue. In particular, we run explicitly the generation of the setup parameters and of all the keys, and we add the boolean flag success indicating if the setup was executed correctly by all the parties. Since we split CreateBallot and ProcessVote in our modeling, the ballot must be given as input to the ProcessVote algorithm.

- **Knowledge-Soundness**: There exists a PPT extractor $\mathcal{E}$ such that for any $(x, w) \in R$ and any PPT adversary $\mathcal{A}$, if $\mathcal{A}()$ outputs a proof x, π such that $\mathsf{Verify}(x, \pi) = 1$, then $\mathcal{E}$ outputs w' such that $(x, w') \in R$ with overwhelming probability. We commonly say that we extract the witness as a shorthand of using the extractor $\mathcal{E}$ on $\mathcal{A}$.

In the security proof, we rely on the soundness of the adversary's proofs, while simulating ourselves the proofs the honest parties have to produce. The adversary could use these simulated proofs to craft unsound proofs which escape the soundness definition [44]. At a first glance, we would need extra properties, the simulation-soundness and the simulation-soundness extractability, which respectively guarantee that despite seeing these simulated proofs, the adversary cannot break the soundness of the proof system, and we can still extract a witness from the adversary. However in our protocol, the identity (or the $\mathsf{upk}_{\mathsf{id}}$ for the case of the voters) of the prover is always public information. By adding it in the context of the Fiat-Shamir hash function, we obtain a separation of domains, and none of these domains are both used in simulated proofs and adversary proofs, which solves this issue.

In particular, the Σ-protocols for the relations DL, enc and exp, presented in Section 3.2.2, as well as the Σ-protocol given in Appendix C.2, satisfy these properties [20].

F.2.3 ElGamal

We define the ElGamal cryptosystem as a tuple of algorithms $(\mathsf{GenKey}, \mathsf{Enc}, \mathsf{Dec})$ augmented with an extra algorithm Rand:

- $\mathsf{GenKey}(2^\lambda)$: 1) Generate a random secret key $x \xleftarrow{\$} \mathbb{Z}_q$. 2) Set $y = g^x$ and compute a proof of knowledge of x: $\pi = \mathsf{Prove}^{DL}((g, y), x)$. Output $\mathsf{sk} \leftarrow x, \mathsf{pk} \leftarrow (y, \pi)$.

  In any of the following algorithms, if pk is parsed as $(\star, \pi)$ such that $\mathsf{Verify}(\pi) = 0$, abort and output 0.

- $\mathsf{Enc}(\mathsf{pk}, m)$: 1) Parse pk as (y, π). 2) Generate a random $r \xleftarrow{\$} \mathbb{Z}_q$. 3) Compute $c_1 = g^r, c_2 = my^r$. Output $c \leftarrow (c_1, c_2)$.

$\mathsf{Exp}^{\text{elig}}_{\mathcal{A},\mathsf{V},I\mathcal{D}}(\lambda)$

$\mathsf{pp} \leftarrow \mathsf{SetupParams}(2^\lambda)$
$A, C \leftarrow \mathcal{A}(2^\lambda)$
if $A \cup C \neq I\mathcal{D}$ **or** $A \cap C \neq \emptyset$ **then return** 0
$\mathsf{keys} \leftarrow \mathsf{SetupKey}_{\mathcal{A}}(2^\lambda)$
$\{(\mathsf{id}, \mathsf{usk}_{\mathsf{id}}, \mathsf{upk}_{\mathsf{id}})\}_{\mathsf{id}\in I\mathcal{D}}, \mathsf{success} \leftarrow \mathsf{Setup}_{\mathcal{A}}(2^\lambda)$
if $\mathsf{success} == 0$ **then return** 0
$\mathsf{BB} \leftarrow \mathcal{A}(\{\mathsf{usk}_{\mathsf{id}}\}_{\mathsf{id}\in C})$
if $\mathsf{IsValid}(\mathsf{upk}_{I\mathcal{D}}, \mathsf{BB}) == 0$ **then return** 0
if $\exists \mathsf{id} \in A$ **and** $\exists \mathsf{b} \in \mathsf{BB}$ **and** $\mathsf{IsValid}(\mathsf{upk}_{\mathsf{id}}, \mathsf{b}) == 1$
  then return 1
else return 0

Figure 10: The eligibility experiment.

- $\mathsf{Dec}(\mathsf{sk}, c)$: 1) Parse c as (c_1, c_2). 2) Compute $m \leftarrow c_2 c_1^{-\mathsf{sk}}$. Output m.
- $\mathsf{Rand}(\mathsf{pk}, c)$: 1) Parse pk as (y, π) and c as (c_1, c_2). 2) Generate a random $r \xleftarrow{\$} \mathbb{Z}_q$. 3) Compute $c'_1 = c_1 g^r, c'_2 = y^r c_2$. Output $c' \leftarrow (c'_1, c'_2)$.

We also extend ElGamal to a distributed key setup when the public key is a list of keys $\mathsf{PK} = [\mathsf{pk}_i]_{1\leq i\leq n}$. In that case, we define $Y = \prod_{i=1}^n y_i$ for each $(y_i, \pi_i) \in \mathsf{PK}$ and use the previous algorithms with PK as the public key. If any of the pk_i contains a proof that does not verify, each algorithm outputs 0. We also define additional algorithms to decrypt a ciphertext:

- $\mathsf{Enc}(PK, m)$: 1) Parse each pk_i as (y_i, π_i). 2) Compute $Y = \prod_{i=1}^n y_i$. 3) Compute $c = \mathsf{Enc}(Y, m)$. Output c.
- $\mathsf{Rand}(PK, c)$: 1) Parse each pk_i as (y_i, π_i). 2) Compute $Y = \prod_{i=1}^n y_i$. 3) Compute $c' = \mathsf{Rand}(Y, c)$. Output c'.
- $\mathsf{PartDec}(\mathsf{sk}, c)$: 1) Parse c as (c_1, c_2). 2) Compute $\mathsf{df} \leftarrow c_1^{\mathsf{sk}}$. Output df.
- $\mathsf{Combine}(DF, c)$: 1) Parse DF as $[\mathsf{df}_i]_{1\leq i\leq n}$. 2) Compute $m \leftarrow c_2 \prod_{i=1}^n \mathsf{df}_i^{-1}$. Output m.

When $\mathsf{Enc}/\mathsf{Rand}/\mathsf{Dec}$ is applied to a vector of components, we consider that the algorithm is applied to each component independently, and each output grouped together in a vector.

F.2.4 Signature scheme

We also use a signature scheme in various parts of the protocol. We define a signature scheme as a tuple of three algorithms $(\mathsf{GenKey}, \mathsf{Sign}, \mathsf{Verify})$:

- $\mathsf{GenKey}(2^\lambda)$ takes as input the security parameters, and output a pair of keys : sk, the signing key, and pk, the verification key.
- $\mathsf{Sign}(\mathsf{sk}, m)$ takes as input a signing key sk and a message m, and outputs a signature sig.
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ takes as input a verifying key, a message, and signature, and outputs 1 if the signature is valid, or 0 otherwise.

We need our signature scheme to be existentially unforgeable, and we assume the verifying key to be of the form $\mathsf{pk} = g^{\mathsf{sk}}$ (such as the Schnorr signature scheme, which we used in our implementation for example).

F.2.5 Plaintext equivalence test

We use in our protocol a plaintext equivalence test (PET) to check if two ciphertexts have the same content or not, without revealing their content. We model the PET as a protocol PET jointly run by n parties each having an ElGamal pair of keys $(\mathsf{sk}_i, \mathsf{pk}_i)$ and an algorithm Verify.

We write an execution of the algorithm as $\mathsf{PET}(c_1, c_2)$, where each party takes as input two ciphertexts c_1 and c_2 and their pair of keys. At the end of the protocol, each party learns a bit e indicating if $\mathsf{Dec}(\mathsf{sk}, c_1) = \mathsf{Dec}(\mathsf{sk}, c_2)$, where sk is the aggregated key, as well as a transcript trans. Verify takes as input c_1, c_2, all the public keys and the transcript, and outputs 1 if the transcript is convincing that c_1 and c_2 have the same content and 0 otherwise.

For the security properties we aim to prove, we need the PET to have these properties:

- **Completeness**: For any c_1 and c_2 such that $\mathsf{Dec}(\mathsf{sk}, c_1) = \mathsf{Dec}(\mathsf{sk}, c_2)$, for any execution $f, \mathsf{trans} \leftarrow \mathsf{PET}(c_1, c_2)$ $f = 1$ and $\mathsf{Verify}(c_1, c_2, \mathsf{trans}) = 1$ with overwhelming probability.
- **Soundness**: For any c_1 and c_2 such that $\mathsf{Dec}(\mathsf{sk}, c_1) \neq \mathsf{Dec}(\mathsf{sk}, c_2)$, for any execution $f, \mathsf{trans} \leftarrow \mathsf{PET}(c_1, c_2)$ $f = 0$ and $\mathsf{Verify}(c_1, c_2, \mathsf{trans}) = 0$ with overwhelming probability.
- **Zero-Knowledge**: There exists a PPT simulator $\mathcal{S}$ such that for any c_1 and c_2, the distributions $f, \mathsf{trans} \leftarrow \mathsf{PET}(c_1, c_2)$ and $f, \mathsf{trans} \leftarrow \mathcal{S}(c_1, c_2, \mathsf{Dec}(\mathsf{sk}, c_1) = \mathsf{Dec}(\mathsf{sk}, c_2))$ are computationally indistinguishable, where $\mathcal{S}$ does not have access to one of the secret key.

F.2.6 Mixnet

Finally, we define a mixnet as a pair of algorithms $(\mathsf{Mix}, \mathsf{VerifyMix})$. Mix is composed from the ElGamal cryptosystem and a NIZK proof system $(\mathsf{Prove}, \mathsf{Verify})$, and it mixes a list of pairs of ciphertexts (under different keys) and exponentiates one of them with a private scalar:

$\mathsf{Exp}^{\mathrm{priv}}_{\mathcal{A},\mathsf{V},I\mathcal{D}}(\lambda)$

$\beta \leftarrow \{0,1\}$
$\mathsf{pp} \leftarrow \mathsf{SetupParams}(2^\lambda)$
$R \leftarrow \{\}$
$\mathsf{BB}_{j_H} \leftarrow \{\}$
$H, C, j_H \leftarrow \mathcal{A}(2^\lambda)$
if $H \cup C \neq I\mathcal{D}$ **or** $H \cap C \neq \emptyset$ **then return** 0
$\mathsf{keys} \leftarrow \mathsf{SetupKey}_{\mathcal{A}}(2^\lambda)$
$\{(\mathsf{id}, \mathsf{usk}_{\mathsf{id}}, \mathsf{ccsk}_{j_H,\mathsf{id}}, \mathsf{upk}_{\mathsf{id}})\}_{\mathsf{id}\in I\mathcal{D}}, \mathsf{success} \leftarrow \mathsf{Setup}_{\mathcal{A}}(2^\lambda)$
$\{(\mathsf{id}, \mathsf{usk}_{\mathsf{id}}, \mathsf{upk}_{\mathsf{id}})\}_{\mathsf{id}\in I\mathcal{D}}, \mathsf{success} \leftarrow \mathsf{Setup}_{\mathcal{A}}(2^\lambda)$
if $\mathsf{success} == 0$ **then return** 0
$\mathcal{A}^{O\mathsf{voteLR}, O\mathsf{processVote}}(\{\mathsf{usk}_{\mathsf{id}}\}_{\mathsf{id}\in C})$
$\mathsf{BB} \leftarrow \mathsf{FinalizeBB}_{\mathcal{A}}(\mathsf{BB}_{j_H})$
$\mathsf{BB} \leftarrow \mathcal{A}(2^\lambda)$
if $\mathsf{IsValid}(\mathsf{upk}_{I\mathcal{D}}, \mathsf{BB}) == 0$ **then return** 0
Let $V_0 = \{\mathsf{v}_0 | \exists(\mathsf{id}, \mathsf{b}, \mathsf{v}_0, \mathsf{v}_1) \in R \wedge \exists(\mathsf{b}, \star, \mathsf{b}^{\mathsf{final}}) \in \mathsf{BB}\}$
Let $V_1 = \{\mathsf{v}_1 | \exists(\mathsf{id}, \mathsf{b}, \mathsf{v}_0, \mathsf{v}_1) \in R \wedge \exists(\mathsf{b}, \star, \mathsf{b}^{\mathsf{final}}) \in \mathsf{BB})\}$
if $V_0 \neq V_1$ **then return** 0
$(\mathsf{r}, \Pi) \leftarrow \mathsf{Tally}_{\mathcal{A}}(\mathsf{BB})$
$\mathsf{guess} \leftarrow \mathcal{A}(\mathsf{r}, \Pi)$
return $\mathsf{guess} == \beta$

$O\mathsf{voteLR}(\mathsf{id}, \mathsf{v}_0, \mathsf{v}_1)$

if $\mathsf{v}_0 \notin \mathcal{V}$ **or** $\mathsf{v}_1 \notin \mathcal{V}$ **then return** 0
if $\mathsf{id} \notin H$ **then return** 0
if $\exists(\mathsf{id}, \star, \star, \star) \in R$ **then return** 0
$\mathsf{b} \leftarrow \mathsf{CreateBallot}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}_\beta)$
$R \leftarrow R \cup \{\mathsf{id}, \mathsf{b}, \mathsf{v}_0, \mathsf{v}_1\}$
return b

$O\mathsf{processVote}(\mathsf{id})$

if $\mathsf{id} \in H$ **then**
  if $\nexists(\mathsf{id}, \mathsf{b}, \mathsf{v}_0, \mathsf{v}_1) \in R$ **then return** 0
  We parse the unique b such that
  $(\mathsf{id}, \mathsf{b}, \mathsf{v}_0, \mathsf{v}_1) \in R$
  $\Gamma_{\mathsf{id}}, \mathsf{BB}_{j_H} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}_\beta, \mathsf{b}, \mathsf{BB}_{j_H}, \mathsf{ccsk}_{j_H,\mathsf{id}})$
  $\Gamma_{\mathsf{id}} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}_\beta, \mathsf{b})$
  return $\mathsf{VerifyVote}(\mathsf{usk}_{\mathsf{id}}, \mathsf{v}_\beta, \Gamma_{\mathsf{id}})$
else $\mathsf{BB}_{j_H} \leftarrow \mathsf{ProcessVote}_{\mathcal{A}}(\mathsf{BB}_{j_H}, \mathsf{ccsk}_{j_H,\mathsf{id}})$

Figure 11: The privacy experiment and its oracles. The lines highlighted in green are the lines that are executed when SC is honest, while the lines highlighted in blue are the ones that are run when one of the online CCs is honest. All the other lines are run in both cases.

- MixSetup takes as input a list of pairs of ciphertexts $L_{in} = [(c_{i,1}, c_{i,2})]_{1\le i\le n}$, a pair of public keys $(\mathsf{pk}_1, \mathsf{pk}_2)$, a permutation σ, a scalar sv and outputs a list of pairs of ciphertexts L_{out} and a proof π that L_{out} is correctly computed. L_{out} and π are computed as such:
  1. $L_{out}[i][0] = \mathsf{Rand}(\mathsf{pk}_1, L_{in}[\sigma(i)][0]^{sv}; r_{i,1})$
  2. $L_{out}[i][1] = \mathsf{Rand}(\mathsf{pk}_2, L_{in}[\sigma(i)][1]; r_{i,2})$
  3. $\pi \leftarrow \mathsf{Prove}(\{(x = (L_{in}, L_{out}, \mathsf{pk}_1, \mathsf{pk}_2),$
     $w = (\sigma, \mathsf{sv}, \left[[r_{i,j}]_{1\le j\le 2}\right]_{1\le i\le n})) : \phi(x,w)\})$
     where ϕ is the conjunction of item 1 and 2.
- VerifyMixSetup takes as input two lists of ciphertexts and a proof π and outputs $\mathsf{Verify}(\pi)$ (leaving implicit that L_{out} is correctly computed).

In the security proof of the protocol, we need the proof system to be complete, sound and zero-knowledge and the Rand algorithm of the encryption scheme to have the randomizability property: for any key pair $(\mathsf{pk}, \mathsf{sk})$ and any ciphertext c, $\mathsf{Rand}(\mathsf{pk}, c)$ and $\mathsf{Enc}(\mathsf{pk}, \mathsf{Dec}(\mathsf{sk}, c))$ have indistinguishable distributions. In particular, the ElGamal cryptosystem satisfies this property perfectly, since those two distributions are the same.

For the tally, we use a decryption mixnet consisting of a pair of algorithms $(\mathsf{MixTally}, \mathsf{VerifyMixTally})$ also composed of an ElGamal cryptosystem and a NIZK proof system.

- MixTally takes as input a list of lists of ciphertexts $L_{in} = \left[[C_{i,l}]_{1\le l\le k}\right]_{1\le i\le n}$, a list of public keys $\left[\mathsf{pk}_j\right]_{1\le j\le m}$, a secret key sk_1, a permutation σ, and outputs a list of tuples of ciphertext L_{out} and a proof π that L_{out} is correctly computed. L_{out} and π are computed as such:
  1. $C'_i \leftarrow \left[\mathsf{Rand}([\mathsf{pk}_j]_{1\le j\le m}, C_{\sigma(i),l})\right]_{1\le l\le k}$
  2. $\mathsf{DF}_i \leftarrow \left[\mathsf{PartDec}(\mathsf{sk}_1, C'_{i,l})\right]_{1\le l\le k}$
  3. $L_{out} \leftarrow \left[(C'_{i,l}[0], C'_{i,l}[1]\mathsf{DF}_{i,l}^{-1})\right]_{1\le l\le k}$
  4. $\pi \leftarrow \mathsf{Prove}(\{(x = (L_{in}, L_{out}, \left[\mathsf{pk}_j\right]_{1\le j\le m}), w = (\sigma, \left[[r_{i,l}]_{1\le l\le k}\right]_{1\le i\le n})) : \phi(x,w)\})$ where ϕ is the conjunction of item 1 through 3.

- VerifyMixTally takes as input a proof π and outputs $\mathsf{Verify}(\pi)$ (leaving implicit that $(L_{in}, L_{out}, [\mathsf{pk}_j]_{1\le j\le m})$ is correctly computed).

Again, we want the proof system to be complete, sound and zero-knowledge and the Rand algorithm to satisfy the randomizability property.

F.3 Formal description of the protocols

This section describes more precisely the specifications of all the algorithms of the voting protocol, which is needed for the security proofs.

F.3.1 Parameters Generation

SetupParams proceeds in two parts. First, $\mathsf{GrpSetup}(2^\lambda)$ is run and outputs $(\mathbb{G}, p, g)$, which is used in all the cryptographic primitives. Then, the encoding in $\mathbb{G}$ of the voting options is created. Each of the $\mathsf{v}_1, \ldots, \mathsf{v}_n$ are generated uniformly at random from $\mathbb{G}$ as distinct group elements. The output of SetupParams is $\mathsf{pp} = ((\mathbb{G}, p, g), (\mathsf{v}_1, \ldots, \mathsf{v}_n))$.

F.3.2 Key Generation

In the SetupKey protocol, all the parties generate their keys with GenKey and broadcast them. In an adversarial setting, we consider that the corrupted parties are the last parties to broadcast their keys, hence the adversary can choose them adaptively. We also set $\mathsf{pk}_{\mathsf{CC}} = \prod_{j=1}^{m} \mathsf{pk}_j$ and $\mathsf{pk}_{\mathsf{EL}} = \mathsf{pk}_{\mathsf{CC}}\mathsf{pk}_{\mathsf{TC}}$.

F.3.3 Setup

Figure 12 and Figure 13 describe how the credentials $(\mathsf{upk}_{\mathsf{id}}, [\mathsf{ccsk}_{j,\mathsf{id}}]_{1\le j\le m}, \mathsf{usk}_{\mathsf{id}})$ are generated for each voter. If there is any abort in the execution of a party, the party stops the execution of the algorithm and outputs $\mathsf{success} = 0$ only. Otherwise, it outputs its regular outputs and $\mathsf{success} = 1$.

F.3.4 CreateBallot

The CreateBallot algorithm involves two parties, the voter and the Client. The voter's process is simple: parse the $\mathsf{usk}_{\mathsf{id}} = (\mathsf{LC}, \star, \star, \star)$ and enter both the LC and the voting intention v in the Client. Then, Figure 14 describes the algorithm used by the Client to create a ballot.

F.3.5 ProcessVote

Figure 15 and Figure 16 describe the ProcessVote procedure. In case of an abort, the CC still appends the unfinalized ballot (b) or $(\mathsf{b}, \mathsf{b}^{proc})$ to its bulletin board, in order to detect other executions of the ProcessVote with the same $\mathsf{pk}_{\mathsf{id}}$, which could break the privacy if not aborted early. These ballots will be filtered out later on.

As opposed to the description of the protocol in Section 3.4, we assume for the sake of simplicity that the voter has only one attempt to type their AC value rightfully in the Client, otherwise their ballot is dropped. This simplification helps to have a simpler protocol to reason with. All the proofs could be adapted without any issue to a scenario where voters have several attempts at typing their AC, at the cost of slightly improving the probability of success of the adversary at the verifiability experiment.

F.3.6 VerifyVote

VerifyVote takes as input a voter's secret credentials $\mathsf{usk}_{\mathsf{id}}$, a transcript Γ_{id} of the ProcessVote algorithm, which consists in a tuple $(\left[\mathsf{RC}^{Sent}[i]\right]_{1\le i\le k}, \mathsf{FC}^{Sent})$, and a vote v. VerifyVote first parses $\mathsf{usk}_{\mathsf{id}}$ as $(\star, [\mathsf{RC}[i]]_{1\le i\le n}, \star, \mathsf{FC})$ and returns 1 if $\mathsf{RC}^{Sent}[i] = \mathsf{RC}[\mathsf{v}[i]]$ and $\mathsf{FC}^{Sent} = \mathsf{FC}$, or 0 otherwise.

F.3.7 FinalizeBB

In the FinalizeBB protocol, each CC takes as input their own view of the bulletin board BB_i, removes all the invalid ballots, and signs it. The final BB is composed from BB_1 (from which all the invalid ballots are removed) and the signatures of all the CCs.

F.3.8 IsValid

The IsValid algorithm takes as input the public credentials $\mathsf{upk}_{\mathsf{id}}$ of a voter and a ballot $(\mathsf{b}, \mathsf{b}^{\mathsf{proc}}, \mathsf{b}^{\mathsf{final}})$. It parses $\mathsf{upk}_{\mathsf{id}}$ as $(\star, \mathsf{pk}_{\mathsf{id}}, \star, c_{\mathsf{AC}}^{Setup})$, b as $(\mathsf{upk}_{\mathsf{id}}^{\mathsf{b}}, \star, \star, \star, \mathsf{sig}^{\mathsf{b}})$, and $\mathsf{b}^{\mathsf{final}}$ as $(c_{\mathsf{AC}}^{\mathsf{Vote}}, \star, \star, \mathsf{pet_trans})$.

VerifyVote outputs 1 if and only if

- $\mathsf{upk}_{\mathsf{id}} = \mathsf{upk}_{\mathsf{id}}^{\mathsf{b}}$.
- $\mathsf{Verify}(\mathsf{b}, \mathsf{sig}^{\mathsf{b}}) = 1$.
- $\mathsf{Verify}(c_{\mathsf{AC}}^{\mathsf{Vote}}, c_{\mathsf{AC}}^{\mathsf{Setup}}, \mathsf{pet_trans}) = 1$.
- All the zero-knowledge proofs associated in the ballot are valid.

Additionally, IsValid can take as input $\mathsf{upk}_{I\mathcal{D}}$ and a bulletin board BB, and returns 1 if and only if for every ballot in BB, there is $\mathsf{upk}_{\mathsf{id}} \in \mathsf{upk}_{I\mathcal{D}}$ such that $\mathsf{IsValid}(\mathsf{upk}_{\mathsf{id}}, \mathsf{b})$ outputs 1 and there is no $\mathsf{upk}_{\mathsf{id}} \in \mathsf{upk}_{I\mathcal{D}}$ such that for two ballots $\mathsf{b}_1, \mathsf{b}_2$ in BB, $\mathsf{IsValid}(\mathsf{upk}_{\mathsf{id}}, \mathsf{b}_1) = \mathsf{IsValid}(\mathsf{upk}_{\mathsf{id}}, \mathsf{b}_2) = 1$.

F.3.9 Tally

Figure 17 describes the execution of the tally algorithm.

F.3.10 VerifyTally

VerifyTally takes as input the bulletin board BB, and the proof of tally Π. The algorithm consists in running the VerifyMixTally algorithm on each step of the mixnet.

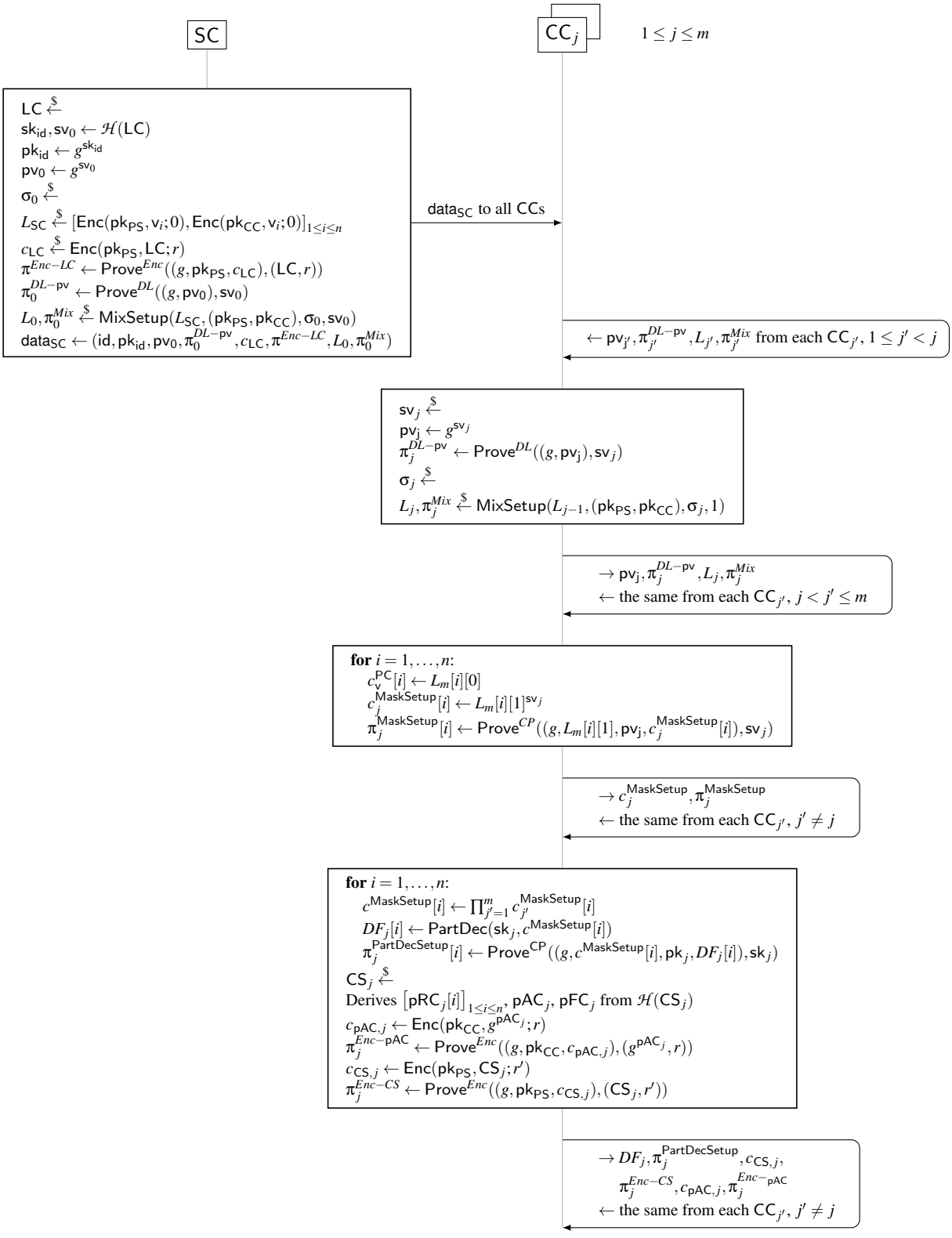


Figure 12: Setup, part 1: LC generation, mixing, additive masking decryption and code generation. The protocol is run concurrently for every $\mathsf{id} \in \mathcal{ID}$. For readability, the subscripts of the values associated to id omit id, *e.g.,* pv_0 denotes $\mathsf{pv}_{0,\mathsf{id}}$. The generic component CC_j stands for each of the control components, and each block between a loopy arrow is interpreted as a sequential execution of the code in the box by CC_1 to CC_m, which depends on the communication from the previous party.

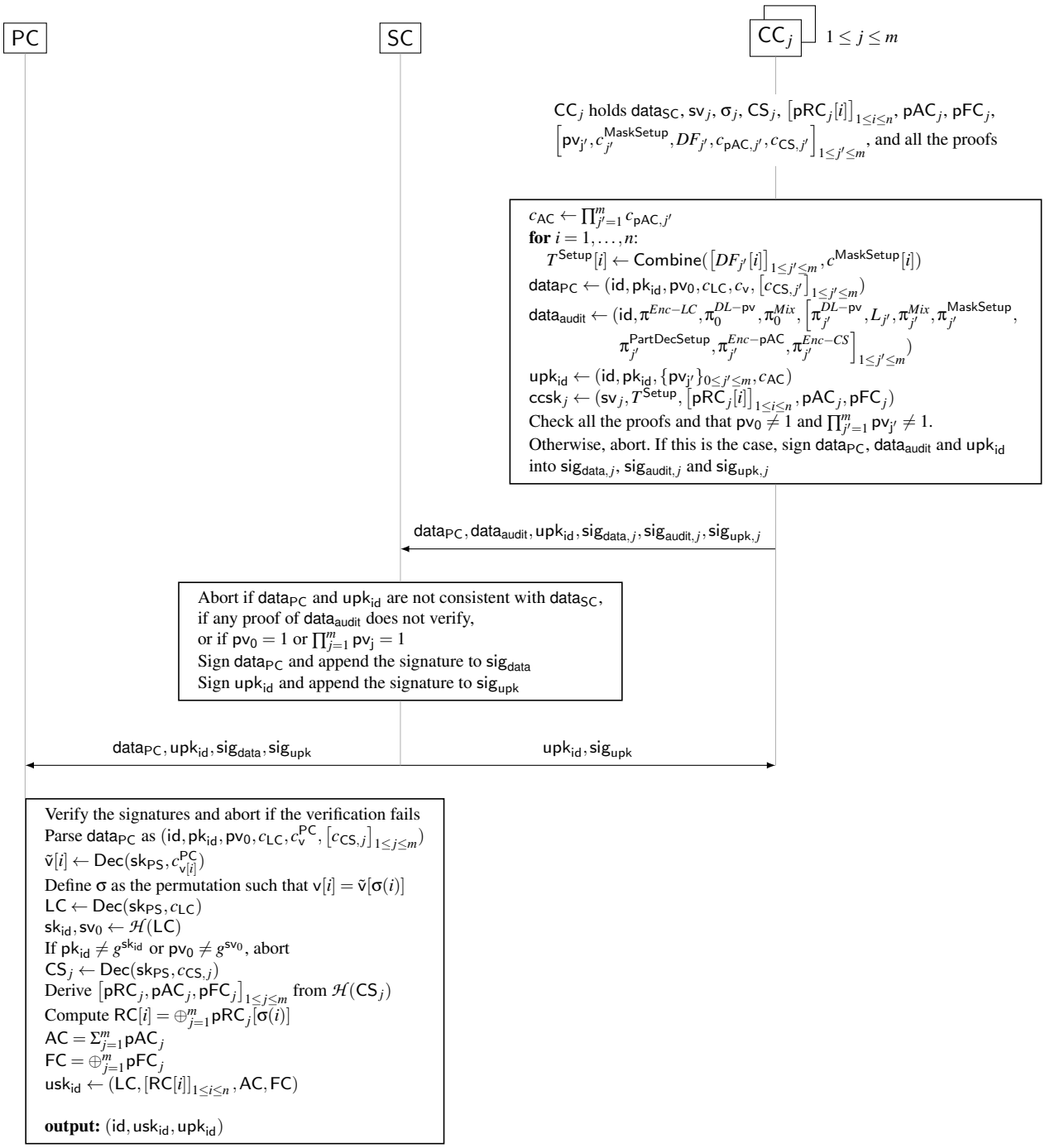


Figure 13: Setup, part 2: assembly of the voting material, signatures, and printing.

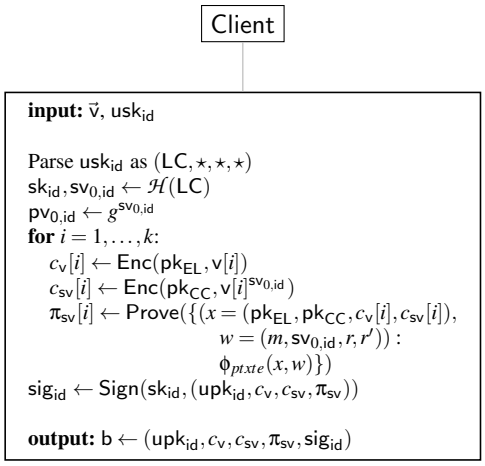


Figure 14: CreateBallot. ϕ_{ptxte} is the relation of $\mathsf{ZKP}_{\mathsf{ptxte}}$ defined in Appendix C.2, applied to $x = (\mathsf{pk}_{\mathsf{EL}},\mathsf{pk}_{\mathsf{CC}},c_{\mathsf{v}}[i],c_{\mathsf{sv}}[i])$ and $w = (m,\mathsf{sv}_{0,\mathsf{id}},r,r')$.

F.4 Security proof

To conduct the security analysis, we fix how to perform the tally. There are two main verifiable tally techniques based on mixnets:

- Either the Talliers sequentially mix the ballots and then decrypt the resulting list of ballots. This approach allows to easily account for any threshold k out of n.
- Or each Tallier sequentially verifiably mix the ballots and decrypt them, one after the other. The final Tallier returns the list of votes in clear. This approach assumes a fixed threshold n out of n where n is the number of Talliers.

In what follows, we chose the second option because it corresponds to the Swiss setting. However, our proof could easily be adapted to the first case.

F.4.1 Verifiability

Theorem 1. *For any k and $d > k$, the protocol has (p,k,d)-verifiability if* PC *and one of the online CC are honest, for $p = \binom{d}{k}(p_{\mathsf{RC}}+1)\,q^{d-k}$, where $q = \max(p_{\mathsf{RC}},p_{\mathsf{AC}},p_{\mathsf{FC}}) + n\,p_{\mathsf{AC}}$ and n is the total number of voters.*

Proof.

Game_0**:** This game is the original verifiability game $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{A}}(\lambda)$. In this proof, we are going to build a sequence of games $(\mathsf{Game}_{i,\mathcal{A}}(\lambda))_{i=0}^{N}$, transitioning from $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{A}}(\lambda)$ to a final game which reduces to the $\mathsf{Exp}^{\mathsf{guess}}_{\mathcal{A},k,d,|\mathcal{ID}|}()$ experiment of 8. Since the probability of success of any unbounded adversary at the guessing experiment is bounded by $\binom{d}{k}(p_{\mathsf{RC}}+1)\,q^{d-k}$, as proven in Appendix E, this reduction will conclude the proof. In the following, we abbreviate $\mathsf{Game}_{i,\mathcal{A}}(\lambda)$ as Game_i.

Strategy of the proof:

(a) In the preliminary games, we extract the zero-knowledge proofs and perform some preparatory transitions.

(b) Then, we show that if $\mathcal{A}$ modified a ballot of an honest voter, then with overwhelming probability, it does not receive the return code associated to the real voting intention of the voter. This ensures that the adversary must guess this code to convince the voter their vote is correctly recorded.

(c) Then, we hide from the adversary the AC codes it could retrieve if it were computationally unbounded. Especially, we replace the content of $c_{\mathsf{pAC},j_H,\mathsf{id}}$ value with a random value.

(d) Finally, we completely hide from the adversary the seed of the codes generating pRC, pAC and pFC. To do so, we replace the content of c_{CS,j_H} with a random value and bound the probability of the adversary to guess the codes seed from the random oracle.

As a reminder, the adversary in this game can ask a polynomially bounded number of $O\mathsf{vote}(\mathsf{id},\mathsf{v})$ and $O\mathsf{processVote}(\mathsf{id},\mathsf{b})$ requests.

Game_1**:** We define Game_1 as Game_0, but we now extract (with the knowledge extractor of the proof system) all the corrupted CCs secret keys in the SetupKey procedure. Similarly, we extract in the Setup procedure all the $\mathsf{sv}_{j,\mathsf{id}}$, the content of $c_{\mathsf{pAC},j}$ and $c_{\mathsf{CS},j}$ for each $j \neq j_H$ and each $\mathsf{id} \in \mathcal{ID}$.

In the OprocessVote oracle, we extract the content of the encryptions in the c_{v} and c_{sv} parts of the ballot. We write $\tilde{V}$ the list of extracted votes (in c_{v}) for such ballots that also appear in the BB of the tally. Also, we extract the content of each of the $c^{\mathsf{Vote}}_{\mathsf{AC}}$ ciphertext computed by the adversary.

This transition is guaranteed by the knowledge-soundness property of the proof system.

Game_2**:** We modify in this game the winning condition of the adversary. In the last "if" statement of the experiment, we remove the existential statement $\exists V_{\mathcal{A}}$ and replace it with an instance $V_{\mathcal{A}}$ defined as follows.

In the BB outputted by the FinalizeBB protocol, for each ballot associated to the public key $\mathsf{upk}_{\mathsf{id}}$ of an $\mathsf{id} \in C$, we add in $V_{\mathcal{A}}$ the extracted vote of the ballot processed in the ProcessVote protocol. Since this multi-set of votes is one particular instance of the possible $V_{\mathcal{A}}$ considered in the previous game, the probability of the adversary for the previous game is bounded by the probability of winning this game.

Game_3**:** In this transition, we modify how we compute all the zero-knowledge proofs. Instead of using the Prove algorithm, we now simulate each proof with the zero-knowledge simulator.

The two games are indistinguishable because of the zero-knowledge property of the proof system.

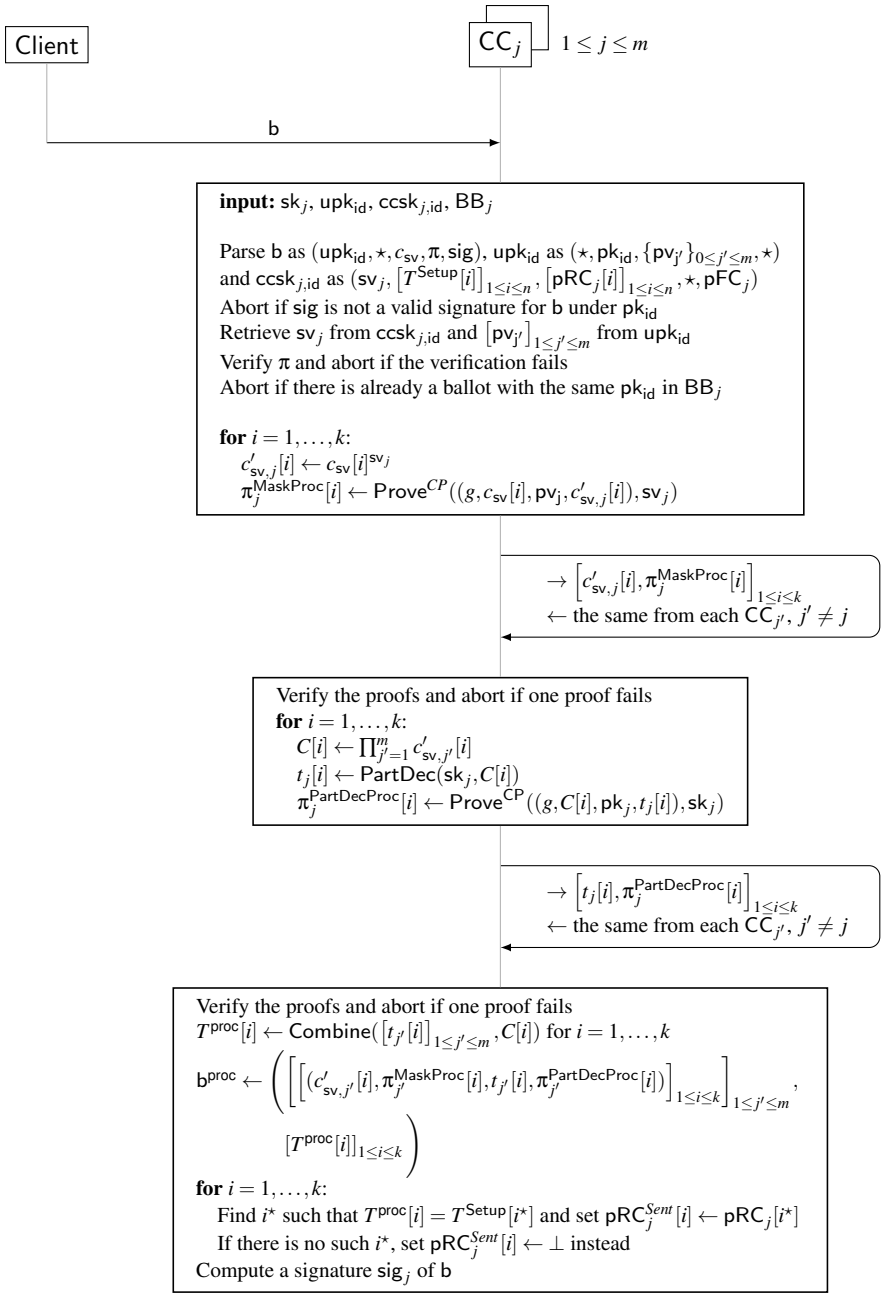


Figure 15: ProcessVote, part 1: additive masking decryption, retrieval of the return codes and signature of the ballot. At the beginning of the protocol, the Client sends to all the CCs the ballot b it received as input.

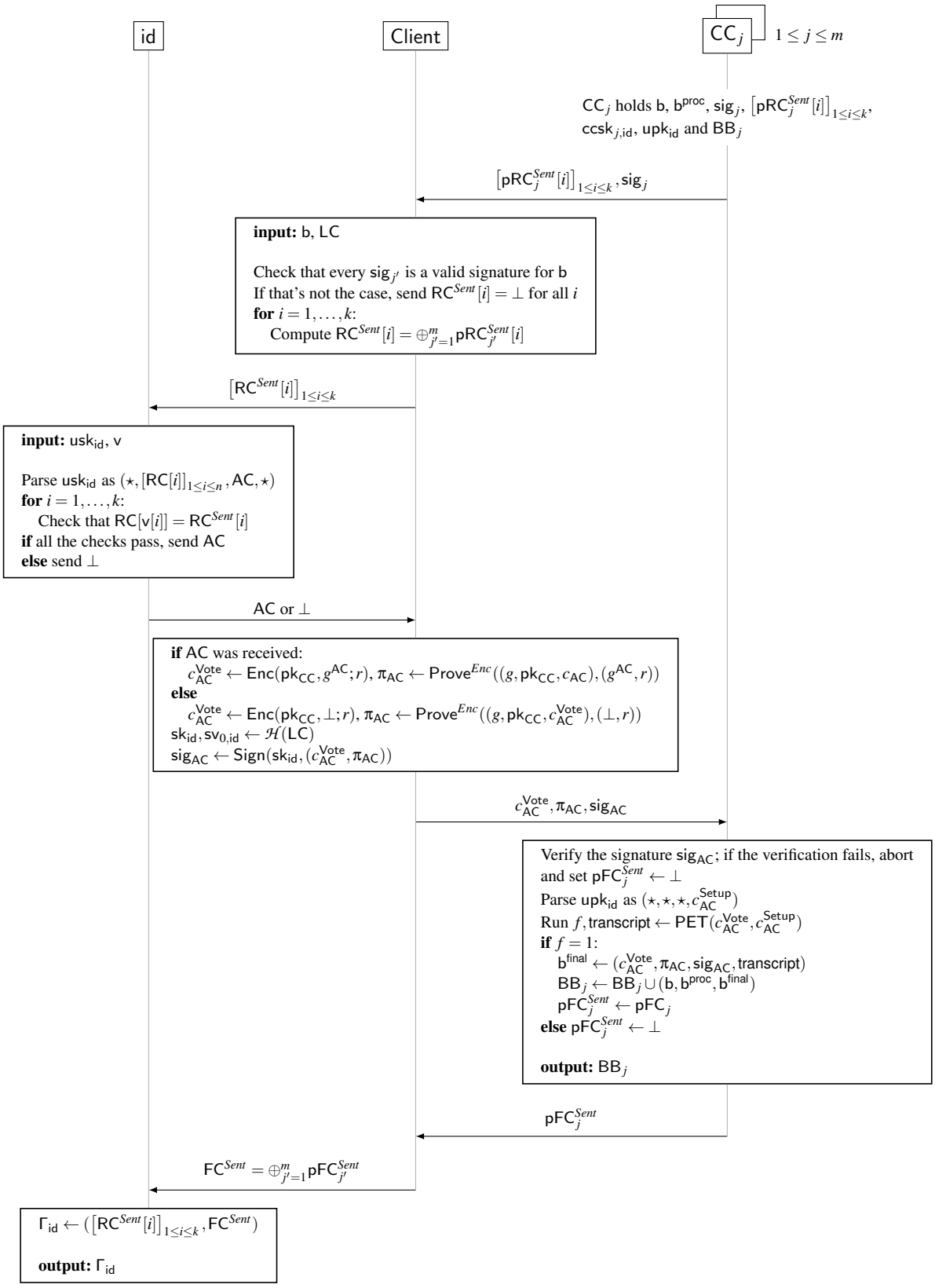


Figure 16: ProcessVote, part 2: checking of the return codes, plaintext equality test and finalization of the ballot.

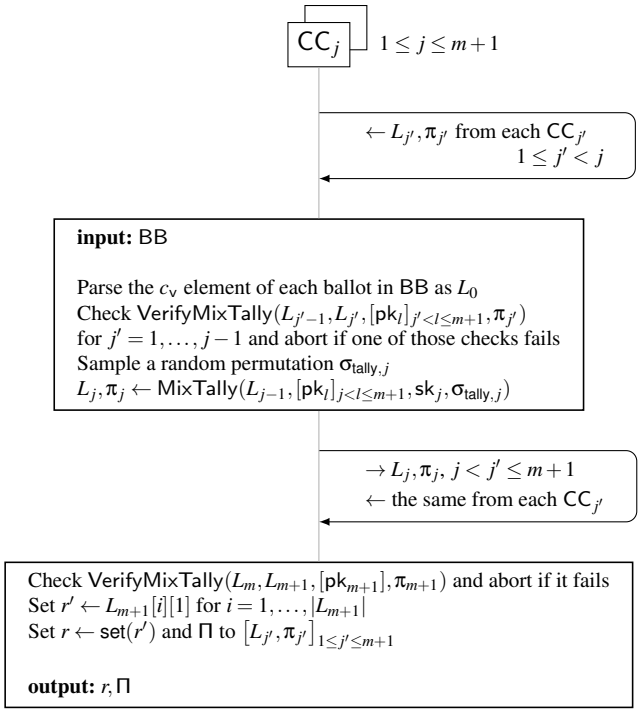


Figure 17: Tally. Here, the tally component is called CC_5 for the sake of succinctness, and $\mathsf{pk}_{\mathsf{TC}}$ is written as pk_5. The result is the set of the final votes (tuple of k elements) in L_5.

Game_4**:** In this game, we define a new failure event. In the Setup, if PC receives a $\mathsf{data}'_{\mathsf{PC}} \neq \mathsf{data}_{\mathsf{PC}}$ or $\mathsf{upk}'_{\mathsf{id}} \neq \mathsf{upk}_{\mathsf{id}}$ and a valid signature under pk_{j_H}, we abort the game. Likewise, if the final bulletin board contains a ballot with a signature sig_{j_H} that CC_{j_H} has not signed, we abort the game.

Since these values are signed by CC_{j_H}, the adversary would need to forge a signature under CC_{j_H}'s key for this event to happen. Hence, this transition is guaranteed by the unforgeability of the signature scheme.

This game ensures that the data handled by PC and TC_{EB} is coherent with the data generated and viewed by CC_{j_H} in the setup.

Game_5**:** In this game, we introduce the following event. This event happens if the finalized bulletin board BB is valid, if there is a ballot $(\mathsf{b}, \mathsf{b}^{\mathsf{proc}}, \mathsf{b}^{\mathsf{final}})$ in BB but CC_{j_H} never received b in a *O*processVote request.

This event does not happen by construction. Since the finalized bulletin board is the agreement of all the valid ballots of each CC, CC_{j_H} only signs a finalized BB that contains ballots it received in the execution of a ProcessVote algorithm.

From Game_6 to Game_8, we proceed to show that the adversary does not learn any code associated to a vote not included in a ballot that is processed.

Game_6**:** In this game, we introduce a new failure event. This event happens if at the end of the Setup, for at least one id and one i between 1 and n, $T^{\mathsf{Setup}}[i] \neq \mathsf{v}_{\sigma(i)}^{\mathsf{sv}_{0,\mathsf{id}} \sum_{j=1}^{m} \mathsf{sv}_{j,\mathsf{id}}}$.

First, the soundness of the mixnet guarantees with overwhelming probability that $L_0[i][1]$ contains an encryption of $\mathsf{v}_{\sigma(i)}^{\mathsf{sv}_{0,\mathsf{id}}}$. Then, the soundness of $\pi^{\mathsf{MaskSetup}}$ guarantees with overwhelming probability that for each j, $c_j^{\mathsf{MaskSetup}}[i]$ contains an encryption of $\mathsf{v}_{\sigma(i)}^{\mathsf{sv}_{0,\mathsf{id}} \mathsf{sv}_{j,\mathsf{id}}}$. Finally, the soundness of $\pi^{\mathsf{PartDecSetup}}$ and the correctness of the Combine algorithm guarantee with overwhelming probability that $T^{\mathsf{Setup}}[i] = \mathsf{v}_{\sigma(i)}^{\mathsf{sv}_{0,\mathsf{id}} \sum_{j=1}^{m} \mathsf{sv}_{j,\mathsf{id}}}$.

For the next games, we define a mapping φ_{id} that maps v to $\varphi_{\mathsf{id}}(\mathsf{v}) = \mathsf{v}^{\mathsf{sv}_{0,\mathsf{id}} \sum_{j=1}^{m} \mathsf{sv}_{j,\mathsf{id}}}$. It is worth noting that φ_{id} is injective, since the honest party checks in the setup that $\mathsf{pv}_{0,\mathsf{id}} \neq 1$ and $\prod_j^m \mathsf{pv}_{\mathsf{j},\mathsf{id}} \neq 1$.

Game_7**:** In this game, we introduce a new failure event. This event happens if in at least one *O*processVote request, for at least one i between 1 and n, $\mathsf{v}[i] \in \mathcal{V} \wedge T^{\mathsf{proc}}[i] \neq \varphi_{\mathsf{id}}(\mathsf{v}[i])$, where v is the extracted vote for this ballot and id is the associated voter.

This game is similar to Game_6. The soundness of π_{sv} guarantees that c_{sv} contains an encryption of $\mathsf{v}^{\mathsf{sv}_{0,\mathsf{id}}}$, the soundness of $\pi_j^{\mathsf{MaskProc}}$ guarantees that $c'_{\mathsf{sv},j}$ contains an encryption of $\mathsf{v}^{\mathsf{sv}_{0,\mathsf{id}} \mathsf{sv}_{j,\mathsf{id}}}$ and the soundness of $\pi^{\mathsf{PartDecProc}}$ guarantees that $T^{\mathsf{proc}}[i] = \mathsf{v}^{\mathsf{sv}_{0,\mathsf{id}} \sum_{j=1}^{m} \mathsf{sv}_{j,\mathsf{id}}}$ with overwhelming probability.

Game_8**:** In this game, we introduce the event concluding this part of the proof. This event happens if in one *O*processVote request, the adversary receives a value $\mathsf{pRC}_{\mathsf{v}',j_H,\mathsf{id}}$ in the list of $\left[\mathsf{pRC}_{j_H,\mathsf{id}}^{\mathsf{Sent}}\right]_{1\le i\le k}[i]$ such that v' is not one of the extracted vote of the corresponding ballot.

In the *O*processVote(id, b) request, CC_{j_H} retrieves for each index i the index $i^\star$ such that $T^{\mathsf{proc}}[i] = T^{\mathsf{Setup}}[i^\star]$ and send $\mathsf{pRC}_{j_H}^{\mathsf{Sent}}[i] = \mathsf{pRC}_{j_H}[i^\star]$ to the adversary (or $\perp$ if there is no such $i^\star$). Because of the events defined in Game_6 and Game_7, $\mathsf{pRC}_{j_H}^{\mathsf{Sent}}[i] = \mathsf{pRC}_{j_H}[\sigma(\mathsf{v}[i])]$ and this event never happens.

From now, the goal of the remaining transitions is to statistically hide all the codes, so that even an unbounded adversary can at best guess the codes at random. To do so, we aim at replacing the content of c_{CS,j_H} and c_{pAC,j_H} (computed in the setup) with random values thanks to a DDH reduction. To be able to make this reduction, we need to forget the keys sk_{j_H} and $\mathsf{sk}_{\mathsf{PS}}$, and thus we need to simulate in another way the decryption of various values throughout the protocol. First, we take care of sk_{j_H} from Game_9 to Game_{15}.

Game_9**:** In this game, we simulate the PartDec algorithm used in the Setup. Instead of computing $DF_{j_H}[i] \leftarrow \mathsf{PartDec}(\mathsf{sk}_{j_H}, c^{\mathsf{MaskSetup}}[i])$, we parse

$c^{\mathsf{MaskSetup}}[i]$ as an ElGamal ciphertext (c_1, c_2), and set $DF_{j_H}[i]$ to $c_2/(\varphi_{\mathsf{id}}(\mathsf{v}_{\sigma(i)})\prod_{j=1, j\neq j_H}^{m} c_1^{\mathsf{sk}_j})$.

The content of $c^{\mathsf{MaskSetup}}[i]$ is equal to $\varphi_{\mathsf{id}}(\mathsf{v}_{\sigma(i)})$ if the failure event described in Game_6 does not happen. Thus, we perfectly simulate our decryption factor such that the resulting Combine algorithm gives the same exact result, and the transition is perfect.

Game_{10}**:** In this game, we simulate the PartDec algorithm in the ProcessVote algorithm as well. For a ballot b given as input to the algorithm, for which we have extracted the vote v contained in it, we simulate our decryption factor to decrypt to $\varphi_{\mathsf{id}}(\mathsf{v})$.

Similarly, the failure event of Game_7 ensures that this transition is perfect.

Game_{11}**:** In this game, we simulate each of the plaintext equivalence tests on $c_{\mathsf{AC}}^{\mathsf{Vote}}$ and $c_{\mathsf{AC}}^{\mathsf{Setup}}$.

More precisely, we check if the $\mathsf{AC}^{\mathsf{Setup}}$ extracted at the setup phase is the same as the discrete log $\mathsf{AC}^{\mathsf{Vote}}$ of the content extracted from the $c_{\mathsf{AC}}^{\mathsf{vote}}$ received in the ProcessVote protocol.

If those two values are equal, we simulate the PET to succeed and output $f = 1$. Otherwise, we simulate the PET to fail and output $f = 0$.

This transition is guaranteed by the zero-knowledge property of the PET.

Game_{12}**:** We now create the following event. This event happens if at the tally, the adversary outputs a pair of lists L_j, L_{j+1} such that these two lists of ciphertexts do not contain the same multi-set of plaintexts.

Since the proof of correct mixing verifies (in the VerifyTally algorithm), this event happens with a probability bounded by the probability of breaking the soundness property of the proof system.

Game_{13}**:** In Game_{13}, we do not apply MixTally on L_{j_H-1} directly. Instead, we proceed as follows.

First, since BB is valid, it can be parsed as a list of ballots $\{b_i\}_{1\leq i\leq n_b}$, where n_b is the number of ballots. These ballots are crafted by the adversary, from which we have extracted a multi-set of votes $\tilde{V}$ in Game_1.

Then, because of Game_{12}, the content of the ciphertexts in L_{j_H-1} is equal to the content of $\tilde{V}$.

Instead of applying line 1 of the MixTally algorithm on $\{b_i\}_{1\leq i\leq n_b}$, we compute $C'_i \leftarrow \left[\mathsf{Enc}([\mathsf{pk}_{j'}]_{j_H\leq j'\leq m+1}, \tilde{V}[\sigma_{\mathsf{tally}}(i)][l]; r_{i,l})\right]_{1\leq l\leq k}$, where σ_{tally} is a random permutation.

Instead of rerandomizing a set of ciphertexts, we compute fresh encryptions. This transition is thus indistinguishable because of the perfect randomizability property of ElGamal.

Game_{14}**:** Finally, we simulate in this game the PartDec algorithm in the MixTally algorithm. Instead of running PartDec in line 2 of MixTally, we directly simulate the decryption of the value we have just encrypted in Game_{13}.

Again, the transition is perfect because we simply rewrite the same game.

At this point, we do not use the secret key sk_{j_H} to decrypt any ciphertext.

Game_{15}**:** In this game, we hide the content of the ciphertexts in the setup related to AC. More precisely, we replace with random values the content of c_{pAC,j_H} computed in the Setup algorithm, for every voter.

The transition is guaranteed by a direct reduction to DDH. Let us focus first on replacing the content of one ciphertext with a random value. In the reduction, we receive a tuple (g, a, b, c), and use respectively a as CC_{j_H}s' public key and (b, mc) as the ciphertext to be transformed into an encryption of a random element, where m is the original plaintext. If (g, a, b, c) is a DDH tuple, then (b, mc) is the ElGamal encryption used in Game_{14}, while if c is a random group element, then (b, mc) is uniformly random as well. We can replace the content of all the ciphertexts with an hybrid argument, at the cost of only one DDH reduction, thanks to the self-reducibility of DDH.

We still have to hide the code seeds CS_{j_H} from the adversary, and have to forget sk_{PS} to do so. This is the goal of Game_{16} through Game_{22}.

Game_{16}**:** Similarly to Game_{12}, we create the following event for the mixnet of the setup. This event happens if at the setup, the adversary outputs a pair of lists L_j, L_{j+1} such that these lists of ciphertexts are not permutations of one another (or in the case of L_0 to L_1, up to an exponentiation by $\mathsf{sv}_{0,\mathsf{id}}$).

Since the proof of correct mixing verifies (otherwise the success flag of the setup would be 0), this event happens with a probability bounded by the probability of breaking the soundness property of the proof system.

Game_{17}**:** Similarly to Game_{13}, we do not apply MixSetup on the list L_{j_H-1} of the Setup. Instead, we compute $L_{j_H}[i]$ as $(\mathsf{Enc}(\mathsf{pk}_{\mathsf{PS}}, \mathsf{v}_{\sigma_{\mathsf{setup}}(i)}), \mathsf{Enc}(\mathsf{pk}_{\mathsf{CC}}, \mathsf{v}^{\mathsf{sv}_0}_{\sigma_{\mathsf{setup}}(i)}; r_i))$, where σ_{Setup} is a random permutation.

Again, this transition is thus indistinguishable because of the perfect randomizability property of ElGamal.

Game_{18}**:** This game changes how PC retrieve all the codes. Instead of decrypting c_{CS,j_H}, PC now directly uses the CS_{j_H} value generated by CC_{j_H}.

The correctness of the decryption, along with the fact that PC receives $c_{\mathsf{CS}_{j_H}}$ computed by CC_{j_H} (because of Game_4), ensures that the codes are computed in the exact same way in both games.

Game_{19}**:** Similarly, we modify in this game which CS_j for $j \neq j_H$ PC uses to generate all the codes. Instead of decrypting $c_{\mathsf{CS},j}$, it now uses the seed extracted in Game_1.

The correctness of the decryption algorithm again guarantees that this game is a restatement of the previous game.

Game_{20}**:** This game changes which LC value PC uses in the Setup. Instead of computing the decryption of c_{LC} to recover LC, PC directly uses the LC extracted in Game_1.

The correctness of the decryption algorithm again guarantees that this game is a restatement of the previous game.

Game_{21}**:** In this game, we bring yet another modification on the way the PC operates in the setup. Instead of retrieving the permutation σ_{Setup} of the mixnet from $\mathsf{Dec}(\mathsf{sk}_{\mathsf{PC}}, c_{\mathsf{v}}^{\mathsf{PC}})$, we retrieve it from $\mathsf{Dec}(\mathsf{sk}_{CC}, L_m[i][1]^{\mathsf{sv}_0^{-1}})$.

In the previous game, σ is computed from $\mathsf{v}_{\sigma(i)} = \mathsf{Dec}(\mathsf{sk}_{\mathsf{PS}}, c_{\mathsf{v}}^{\mathsf{PC}}[i])$, where $c_{\mathsf{v}}^{\mathsf{PC}}[i]$ is defined from $L_m[i][0]$. Now, we compute σ from $L_m[i][1]^{\mathsf{sv}_0^{-1}}$, which is the same value because of the soundness of the mixnet (see Game_{16}) and the fact that $\mathsf{pv}_0 \neq 1$.

At this point, we do not use $\mathsf{sk}_{\mathsf{PS}}$ **anymore.**

Game_{22}**:** In this game, we hide the content of c_{CS,j_H}. More precisely, we replace with random values the content of c_{CS,j_H} computed in the Setup algorithm.

Similarly to Game_{15}, this is a direct DDH reduction.

Game_{23}**:** In this game, we introduce a new failure event. If we receive from the adversary a random oracle request for some code seed $\mathsf{CS}_{j_H,\mathsf{id}}$ of an honest voter id, then we abort.

At this point of the proof, the code seed of honest voters are completely independent of the view of the adversary (because of Game_{22}). Since they are drawn from $\{0,1\}^\lambda$, the adversary has a probability $\frac{|H|}{2^\lambda}$ of guessing an honest voter's code seed in its first RO request. If the adversary makes q such requests in the whole game, then we can bound the probability of this event by $q\frac{|H|}{2^\lambda}$ using the union bound, which is negligible.

Analysis of the last game:

For every $\mathsf{id} \in H$, the codes $\mathsf{pRC}_{*,j_H,\mathsf{id}}$, $\mathsf{pAC}_{j_H,\mathsf{id}}$ and $\mathsf{pFC}_{j_H,\mathsf{id}}$ are uniform and independent of the adversary's view, except for the potential codes given to the adversary in the $O\mathsf{processVote}(\mathsf{id})$ request. In particular, at the moment the adversary is guessing a code, the code it must match is independent of its entire view so far, and it holds even though the adversary can adaptively choose which voters to attack and how. Hence, each guess succeeds with a probability equal to a given probability p (which depends) on the code to guess, and the number of successes at guessing these codes can be analyzed with a binomial random variable. We will thus be able to reduce this final game to the purely statistical guessing experiment of Figure 8.

For the adversary to win the game, there must be no more than k in complaints and at least d modifications from $X = V_{\mathcal{A}} \cup V_{check}$ to $r = V_{\mathcal{A}} \cup V_H$ where V_H is the set of extracted votes associated to honest voters, thus at least d modifications between V_{check} and V_H. These modifications can have the form:

- **Add(x)**: add x to X.
- **Remove(x)**: remove x from X.
- **Modify(x, y)**: remove x from X and add y to X.

We can write V_H as $V_{check} \cup V_{add} \cup V_{mod} \setminus (V_{drop} \cup V'_{mod})$, where :

- V_{add} is the multi-set of extracted votes in BB associated to voters who have not voted.
- V_{drop} is the multi-set of votes in H_{voted} for which there is no ballot in BB.
- V_{mod} is the multi-set of extracted votes in BB that are associated to voters who have voted, but whose voting intentions is different. V'_{mod} is the multi-set of votes in H_{voted}, but whose ballot in BB is extracted into another vote.

and which explain modifications from V_{check} to V_H, using V_{add} as additions, V_{drop} as deletions and V_{mod} as modifications. Hence, to win the game, the adversary must produce sets V_{add}, V_{drop} and V_{mod} such that $|V_{add} + V_{drop} + V_{mod}| \geq d$ with no more than k complaints.

We now reduce this game to $\mathsf{Exp}^{\mathrm{guess}}_{\mathcal{A},k,d,|\mathcal{ID}|}()$. In the reduction, we build an adversary $\mathcal{A}^{\mathcal{G}}$ that plays $\mathsf{Exp}^{\mathrm{guess}}_{\mathcal{A},k,d,|\mathcal{ID}|}()$. In our construction, we have access to the adversary $\mathcal{A}$ playing Game_{23}, and will show that $\mathcal{A}^{\mathcal{G}}$'s probability of success is at least $\mathcal{A}$'s probability of success. In our reduction, we keep track of the sets V_{add}, V_{drop}, V_{mod} and $V_{complaint}$, and we keep the following invariant. At any point, the D value of $\mathcal{A}^{\mathcal{G}}$'s oracle is greater or equal than $|V_{add}| + |V_{drop}| + |V_{mod}|$, and the K value of $\mathcal{A}^{\mathcal{G}}$'s oracle is smaller or equal than $|V_{complaint}|$.

First, we receive from $\mathcal{A}$ the values H, C, j_H, and we execute the SetupParams, SetupKey and Setup algorithms (with $\mathcal{A}$). If the setup fails, we abort the game without any request, and both $\mathcal{A}$ and $\mathcal{A}^{\mathcal{G}}$ loses their experiment. However, we do not sample the values $\left[\mathsf{pRC}_{j_H}[i]\right]_{1\leq i\leq n}$, pAC_{j_H} and pFC_{j_H} in the setup. Because of Game_{22} and Game_{23}, these values are outputs of the hash function that can be lazily sampled during executions of the ProcessVote algorithm.

Then, for each of $\mathcal{A}$'s Ovote requests, we simulate Game_{23}'s Ovote oracle and do nothing particular. For each of $\mathcal{A}$'s $O\mathsf{processVote}(\mathsf{id})$ requests however, $\mathcal{A}^{\mathcal{G}}$ proceeds as one of the following cases. In each case, we call at most one request to $\mathcal{A}^{\mathcal{G}}$'s oracle, and track the effect of the request on the sets V_{add}, V_{drop}, V_{mod} and $V_{complaint}$. Each case is exclusive; that is, for a given case, none of the conditions of the previous cases applies to the case.

(i) **Case 1: There has been a prior** $O\mathsf{processVote}(\mathsf{id},\mathsf{b})$ **request for this** id. $\mathcal{A}^{\mathcal{G}}$ simulates Game_{23}'s oracle normally. In particular, the honest CC_{j_H} will abort the resulting ProcessVote protocol because of the duplicate ballot, as expected.

(ii) **Case 2: There has been no prior** $O\mathsf{vote}(\mathsf{id},\mathsf{v})$ **request for this** id. In that case, $\mathcal{A}$ is trying to cast a ballot for an absentee voter. $\mathcal{A}^{\mathcal{G}}$ executes the ProcessVote's protocol until we have to start the PET.

Instead of comparing $c_{\mathsf{AC}}^{\mathsf{Vote}}$ with $c_{\mathsf{AC}}^{\mathsf{Setup}}$, $\mathcal{A}^{\mathcal{G}}$ makes an $O\mathsf{guessAC}()$ request, and receives a bit β indicating if its guess succeeded. If it did, we simulate the PET to succeed as well. Otherwise, we simulate the PET to fail.

If the PET succeeds, we add the v of the ballot to V_{add}.

Note: it is possible that the content of $c_{\mathsf{AC}}^{\mathsf{Vote}}$, is not in the space of possible codes from which $\mathsf{AC}^{\mathsf{Setup}}$ is generated, and thus that the adversary has no hope to succeed the PET. In that case, we set the PET to fail, regardless of the β value. Even if $\mathcal{A}^{\mathcal{G}}$ succeeds the guess with its oracle, this keeps the invariant on D.

(iii) **Case 3: In the prior** $O\mathsf{processVote}(\mathsf{id},\mathsf{b})$ **request, the extracted vote** v' **from this ballot is different from the registered vote** v **in the** $O\mathsf{vote}(\mathsf{id},\mathsf{v})$ **request**. $\mathcal{A}^{\mathcal{G}}$ executes the ProcessVote algorithm as in Game_{23}, until we receive the return codes $\left[\mathsf{RC}_l^{\mathcal{A}}\right]_{1\le l\le n}$ from $\mathcal{A}$.

Instead of comparing the RC codes, $\mathcal{A}^{\mathcal{G}}$ makes an $O\mathsf{guessRC}(i)$ request, where i is the number of indices for which $\mathsf{v}[i]$ and $\mathsf{v}'[i]$ differ. From this request, $\mathcal{A}^{\mathcal{G}}$ receives a bit β indicating if its guess succeeded. If it did, we send AC to the Client, and otherwise, we send $\perp$. As the previous case, $\mathcal{A}$ could have sent codes that are in the space of possible options for these codes, and in that case we proceed as if we received a bit $\beta = 0$.

We then proceed with a normal execution of the rest of the protocol. If the PET succeeds, we add v' to V_{mod}, and if it fails, we add v to V_{drop}. In both cases, we also sample the value pFC_{j_H} from the RO realization of $H(\mathsf{CS}_{\mathsf{id}})$ and compute the voter's FC. If $\beta = 0$, or if $\mathcal{A}$ sends a code $\mathsf{FC}^{\mathcal{A}} \neq \mathsf{FC}$ to the voter, we add id to $V_{complaint}$.

(iv) **Case 4:** $\mathcal{A}$**'s interactions in the** ProcessVote **algorithms result in** b **to be dropped**. $\mathcal{A}^{\mathcal{G}}$ executes the ProcessVote algorithm as in Game_{23}, until we receive the list $\left[\mathsf{pFC}_j^{\mathcal{A}}\right]_{1\le j\le m}$ from the adversary.

Instead of comparing the FC codes, $\mathcal{A}^{\mathcal{G}}$ makes an $O\mathsf{guessFC}()$ request and receives a bit β indicating if its guess succeeded. Again, $\mathcal{A}$ could have sent a code out of the space of possible codes, and in that case we proceed as if we received a bit $\beta = 0$.

If $\beta = 0$, we add id to $|V_{complaint}|$. In any case, we add the vote v associated to the ballot to V_{drop}.

(v) **Case 5: All the other interactions** For the other interactions, we execute the ProcessVote algorithm as in Game_{23}.

All of these cases keep the invariant on the values D and K of $\mathcal{A}^{\mathcal{G}}$'s oracle, and consider all the cases where the adversary could contribute to V_{add}, V_{drop} or V_{mod}. Because of Game_5, it is enough to only look at the ballots appended to BB during ProcessVote requests.

Finally, $\mathcal{A}$ outputs its BB, and if it is valid, we compute the tally as in Game_{14}. In that process, the $\mathcal{A}$ could lose, by providing for example an invalid BB in the finalization of the bulletin board or a wrong proof in the tally. This does not change the analysis, since $\mathcal{A}$ loses in that case.

Because of our invariant, whenever $\mathcal{A}$ wins, $d \le |V_{add} \cup V_{drop} \cup V_{mod}| \le D$ and $K \le |V_{complaint}| \le k$. Also, because of the first case of $\mathcal{A}^{\mathcal{G}}$'s behavior for ProcessVote, each voter can be considered only once, and $\mathcal{A}^{\mathcal{G}}$ makes at most $|\mathcal{ID}|$ requests to its oracles. Hence, whenever $\mathcal{A}$ wins its game, $\mathcal{A}^{\mathcal{G}}$ wins its game as well, which concludes the proof.

□

F.4.2 Privacy

Theorem 4. *The protocol has privacy if* PC, TC_{EB} *and one of the* CC*s are honest.*

Proof.

Game_0**:** This game is the original privacy game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{priv}}(\lambda)$. In this proof, we are going to build a sequence of games $(\mathsf{Game}_{i,\mathcal{A}}(\lambda))_{i=0}^{N}$, transitioning from $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{priv}}(\lambda)$ to a game in which β is independent of the view of the adversary, which has thus a probability $\frac{1}{2}$ of winning the game. In the following, we abbreviate $\mathsf{Game}_{i,\mathcal{A}}(\lambda)$ as Game_i.

As a reminder, the adversary in this game can ask a polynomially bounded number of $O\mathsf{voteLR}(\mathsf{id},\mathsf{v}_0,\mathsf{v}_1)$, $O\mathsf{processVote}(\mathsf{id})$. As in the verifiability proof, we define a mapping φ_{id} mapping a group element x to $x^{sv_{0,\mathsf{id}}\sum_{j=1}^{m}\mathsf{sv}_{j,\mathsf{id}}}$.

Strategy of the proof: In this proof, we want to hide from the adversary all the content related to the selected voting option v_β.

- First, we run the same game hops as the verifiability proof, with a few modifications. The goal of this part is to hide the content of critical ciphertexts (such as the c_{v}), and we need almost the same steps to simulate the decryption of all the ciphertexts.
- Then, since v_β is used in the check $\mathsf{RC}[\mathsf{v}_\beta[i]] \stackrel{?}{=} \mathsf{RC}^{Sent}[i]$, we aim at making this check independent of v_β.
- Finally, v_β is still leaked in the values $T^{\mathsf{proc}} = \varphi_{\mathsf{id}}(\mathsf{v}_\beta)$ that the CCs decrypt. We make a last transition under DDH to make T^{proc} independent of v_β.

We start with the same sequence of game hops as the verifiability proof, with these few differences:

- Game_2 is not relevant to the privacy experiment, so it is skipped.
- Game_8 introduces an event which is not needed for the proof, so it is skipped.
- Game_{15} In this game, we hide the content of other ciphertexts. In particular, we need to replace with random values the content of the $[L_{j_H}[i][1]]_{1\le i\le n}$ ciphertexts in the mixnet of the setup (encrypting $\mathsf{v}^{\mathsf{sv}_0}$) and the content of the c_{v} and c_{sv} ciphertexts in the CreateBallot algorithm. This is done with the same DDH reduction as in the verifiability proof.
- Game_{21} is more tricky. In the verifiability proof, we retrieve the Setup's permutation σ_{Setup} from $\mathsf{Dec}(\mathsf{sk}_{\mathsf{CC}}, L_m[i][1])$ instead of the decryption of $c_{\mathsf{v}}^{\mathsf{PC}}[i]$.
  In our case, we replaced in Game_{15} the content of $[L_{j_H}[i][1]]_{1\le i\le n}$ with a random value, so the decryption of $[L_m[i][1]]_{1\le i\le n}$ does not contain a valid vote anymore.
  To reconstruct σ_{Setup}, we first compute σ_{pre} such that $\mathsf{v}[i] = \mathsf{Dec}(\mathsf{sk}_{\mathsf{CC}}, L_{j_H-1}[\sigma_{pre}(i)][1])$ and then compute σ_{post} such that $\mathsf{Dec}(\mathsf{sk}_{\mathsf{CC}}, L_{j_H}[i][1]) = \mathsf{Dec}(\mathsf{sk}_{\mathsf{CC}}, L_m[\sigma_{post}(i)][1])$.
  Finally, we set $\sigma = \sigma_{post} \circ \sigma_{j_H} \circ \sigma_{pre}$, where σ_{j_H} is the permutation by CC_{j_H} used to shuffle L_{j_H-1} in the mixnet.
- Game_{22} In this game, we hide again the content of ciphertexts. Here, we need to hide the content of $[L_{j_H}[i][0]]_{1\le i\le n}$. This is again a straightforward DDH reduction.

After this sequence of common games, we can continue with games particular to the privacy proof. Our first goal is to make the result of the check $\mathsf{RC}[\mathsf{v}[i]] \stackrel{?}{=} \mathsf{RC}^{Sent}[i]$ independent of v_β.

Game_{24}**:** First, we introduce the following failure event. If there is a $O\mathsf{processVote}(\mathsf{id})$ in which the adversary sends a ballot b' different from the ballot b_β computed in the $O\mathsf{voteLR}$ request for this id, and if the ciphertext $c_{\mathsf{AC}}^{\mathsf{Vote}}$ we send to the adversary does not contain $\perp$, then we abort the game.

Indeed, for this event to happen, the Client must receive a valid signature for b_β under pk_{j_H}, otherwise the Client will encrypt $\perp$ in $c_{\mathsf{AC}}^{\mathsf{Vote}}$. Because CC_{j_H} is only processing one ballot for a given $\mathsf{upk}_{\mathsf{id}}$, it only computes a signature for b', and refuse to process b_β for any following ProcessVote request. Hence, this event happens with negligible probability because of the existential unforgeability of the signature scheme.

Game_{25}**:** In this game, we modify the check of the return codes $\mathsf{RC}[\mathsf{v}[i]] \stackrel{?}{=} \mathsf{RC}^{Sent}[i]$ in the ProcessVote algorithm. Instead, we compare $\mathsf{RC}^{Sent}[i]$ with $\oplus_j^m \mathsf{pRC}_j[i^\star]$, for every i, and output AC if they are all equal. This condition is equivalent to the adversary sending honest pRC_j^{Sent}, hence the adversary does not learn anything from the check of the return codes.

This transition is a restatement of the previous game, because of Game_{24}.

There is one last part of the protocol where v_β appears in the view of the adversary: in the $T^{\mathsf{Vote}} = \varphi_{\mathsf{id}}(\mathsf{v}_\beta)$ value. In Game_{26} and Game_{27}, we proceed to hide this last occurrence of v_β.

Game_{26}**:** In the SetupParams algorithm, we change the way we generate the voting options v_i. Instead of sampling them at random, we generate a trapdoor $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$ and set $\mathsf{v}_i = g^{\alpha_i}$.

This transition is simply a programming of the ROM.

Game_{27}**:** In this game, we modify how each of the φ_{id} is computed. Instead of computing $\varphi(x) = x^{\mathsf{sv}_{0,\mathsf{id}} \sum_{j=1}^m \mathsf{sv}_{j,\mathsf{id}}}$, we now sample each φ as a random function.

This transition is guaranteed by DDH, with an hybrid argument. Let us look first at a given id and a given v. Intuitively, we receive from the DDH challenger a tuple $(g, a = g^x, b = g^y, c = g^z)$, and want to replace $\varphi_{\mathsf{id}}(\mathsf{v}) = \mathsf{v}^{\mathsf{sv}_0 \sum_{j=1}^m \mathsf{sv}_j}$ with c. If $z \neq xy$, then c is a random value as wanted, and if $z = xy$, we want to compute $\mathsf{v}^{\mathsf{sv}_0 \sum_{j=1}^m \mathsf{sv}_j}$ from a and b, while keeping the other $\varphi_{\mathsf{id}}(\mathsf{v})$ values consistent with a and b (that is, using the same v_i and sv_{j_H}).

This is a 2D hybrid argument, because we have to iterate over the voting options $[\mathsf{v}_i]_{1\le i\le n}$ and over the voters $\mathsf{id} \in I\mathcal{D}$ (for the sake of simplicity, we assume we can iterate id from 1 to $|I\mathcal{D}|$). In the $(i^\star, \mathsf{id}^\star)$-th hybrid, we receive a tuple (g, a, b, c), and set respectively $\mathsf{v}_{i^\star}$ as a, $\mathsf{pv}_{\mathsf{jH},\mathsf{id}^\star}$ as b, and

- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to a random element if $\mathsf{id} < \mathsf{id}^\star \vee (\mathsf{id} = \mathsf{id}^\star \wedge i < i^\star)$.
- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to $c^{\mathsf{sv}_{0,\mathsf{id}}} a^{\mathsf{sv}_{0,\mathsf{id}} \sum_{j\neq j_H} \mathsf{sv}_{j,\mathsf{id}}}$ if $\mathsf{id} = \mathsf{id}^\star \wedge i = i^\star$.
- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to $(b \prod_{j\neq j_H} \mathsf{pv}_{j,\mathsf{id}})^{\mathsf{sv}_{0,\mathsf{id}} \alpha_i}$ if $\mathsf{id} = \mathsf{id}^\star \wedge i > i^\star$.
- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to $\mathsf{v}_i^{\mathsf{sv}_{0,\mathsf{id}} (\sum_j \mathsf{sv}_{j,\mathsf{id}})}$, if $\mathsf{id} > \mathsf{id}^\star$.

If (g, a, b, c) is a DH tuple, then $\varphi_{\mathsf{id}^\star}(\mathsf{v}_{i^\star})$ is computed as in the previous hybrid (because $c^{\mathsf{sv}_{0,\mathsf{id}}} a^{\mathsf{sv}_0 \sum_{j\neq j_H} \mathsf{sv}_{j,\mathsf{id}^\star}} = \mathsf{v}_{i^\star}^{\mathsf{sv}_{0,\mathsf{id}^\star} \sum_j \mathsf{sv}_{j,\mathsf{id}^\star}}$), and otherwise, it is a uniformly random group element. The $(i^\star, \mathsf{id}^\star)$-th hybrid constraints two slices: the values $\varphi_{\mathsf{id}_{\mathsf{id}}}(\mathsf{v}_{i^\star})$ for $\mathsf{id} \neq \mathsf{id}^\star$ and the values $\varphi_{\mathsf{id}^\star_{\mathsf{id}}}(\mathsf{v}_i)$ for $i \neq i^\star$. Luckily, we can compute the former thanks to the programming of the v_i in Game_{26}, and the latter thanks to the extracted sv_j. This argument is graphically represented in Figure 18.

By iterating over these hybrids, first along the first coordinate then along the second one, each hybrid is indistinguishable from the previous one under DDH, and thus Game_{27} is indistinguishable from Game_{26}. This transition can be seen

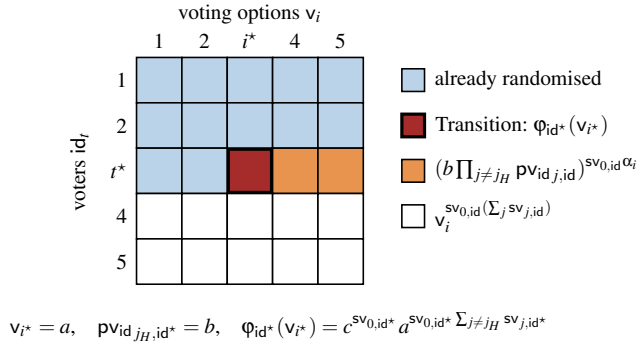


Figure 18: Graphical representation of the 2D hybrid argument.

as a MDDH instance [23], which is shwon to be tight because of the self-reducibility of the DDH tuple.

Conclusion. Now, each v_β is completely independent of the view of the adversary, and it has a probability $\frac{1}{2}$ of guessing β.

□

Theorem 5. *The protocol has privacy if* PC, TC_{EB} *and* SC *are honest.*

Proof.

Game_0**:** This game is the original privacy game $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(\lambda)$. Again, we are going to build a sequence of games $(\mathsf{Game}_{i,\mathcal{A}}(\lambda))_{i=0}^{N}$, transitioning from $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(\lambda)$ to a game in which β is independent of the view of the adversary, which has thus a probability $\frac{1}{2}$ of winning the game. In the following, we abbreviate $\mathsf{Game}_{i,\mathcal{A}}(\lambda)$ as Game_i.

$O\mathsf{processVote}(\mathsf{id})$. As in the other proofs, we define a mapping φ_{id} mapping a group element x to $x^{sv_{0,\mathsf{id}}\sum_{j=1}^{m}\mathsf{sv}_{j,\mathsf{id}}}$.

Strategy of the proof: The strategy of the proof is identical to the one CC honest case. The main difference is the values we are hiding are different: instead of hiding the coding seed CS_{j_H}, we need to hide the login code LC.

For the sake of clarity, we rewrite explicitly all the games.

Game_1**:** We define Game_1 as Game_0, but we now extract (with the knowledge extractor of the proof system) all the corrupted CCs secret keys in the SetupKey procedure. Similarly, we extract in the Setup procedure all the $\mathsf{sv}_{\mathsf{id}}$, the content of $c_{\mathsf{pAC},j}$ and $c_{\mathsf{CS},j}$ for each j and each $\mathsf{id}\in\mathcal{ID}$.

In the $O\mathsf{processVote}$ oracle, we extract the content of the encryptions in the c_{v} and c_{sv} parts of the ballot. We write $\tilde{V}$ the list of extracted votes (in c_{v}) for such ballots that also appear in the BB of the tally. Also, we extract the content of each of the $c^{\mathsf{Vote}}_{\mathsf{AC}}$ ciphertext computed by the adversary.

This transition is guaranteed by the knowledge-soundness property of the proof system.

Game_2**:** In this transition, we modify how we compute all the zero-knowledge proofs. Instead of using the Prove algorithm, we now simulate each proof with the zero-knowledge simulator.

The two games are indistinguishable because of the zero-knowledge property of the proof system.

Game_3**:** In this game, we define a new failure event. In the Setup, if PC receives a $\mathsf{data}'_{\mathsf{PC}}\neq\mathsf{data}_{\mathsf{PC}}$ or $\mathsf{upk}'_{\mathsf{id}}\neq\mathsf{upk}_{\mathsf{id}}$ and a valid signature under pk_0, we abort the game.

Since these values are signed by SC, the adversary would need to forge a signature under SC's key for this event to happen. Hence, this transition is guaranteed by the unforgeability of the signature scheme.

This game ensures that the data handled by PC and TC_{EB} is coherent with the data generated and viewed by SC in the setup.

Game_4**:** We now create the following event. This event happens if at the tally, the adversary outputs for the mixnet a pair of lists L_j, L_{j+1} such that these two lists of ciphertexts do not contain the same multi-set of plaintexts.

Since the proof of correct mixing verifies (in the VerifyTally algorithm), this event happens with a probability bounded by the probability of breaking the soundness property of the proof system.

Game_5**:** In Game_5, we do not apply MixTally on L_{j_H-1} directly. Instead, we proceed as follows.

First, since BB is valid, it can be parsed as a list of ballots $\{b_i\}_{1\leq i\leq n_b}$, where n_b is the number of ballots. These ballots are crafted by the adversary, from which we have extracted a multi-set of votes $\tilde{V}$ in Game_1.

Then, because of Game_4, the content of the ciphertexts in L_{j_H-1} is equal to the content of $\tilde{V}$.

Instead of applying line 1 of the MixTally algorithm on $\{b_i\}_{1\leq i\leq n_b}$, we compute $C'_i \leftarrow \left[\mathsf{Enc}([\mathsf{pk}_{m+1}]),\tilde{V}[\sigma_{\mathsf{tally}}(i)][l];r_{i,l})\right]_{1\leq l\leq k}$, where σ_{Tally} is a random permutation.

Instead of rerandomizing a set of ciphertexts, we computed fresh encryptions. This transition is thus indistinguishable because of the perfect randomizability property of ElGamal.

Game_6**:** Finally, we simulate in this game the PartDec algorithm in the MixTally algorithm. Instead of running PartDec in line 2 of MixTally, we directly simulate the decryption of the value we have just encrypted in Game_5).

The transition is perfect because we simply rewrite the same game.

At this point, we do not use the secret key $\mathsf{sk}_{m+1}=\mathsf{sk}_{\mathsf{TC}_{EB}}$ **anymore.**

Game_7: In this game, we hide the content of the c_{v} ciphertexts computed in a CreateBallot algorithm. More precisely, we replace with random values the content of c_{v} computed in the CreateBallot algorithm, for every voter. This is done with the same DDH reduction as in Game_{15} of the verifiability proof.

We still have to hide the login code LC from the adversary. This is the goal of Game_8 through Game_{12}.

Game_8: Similarly to Game_4, we create the following event for the mixnet of the setup. This event happens if at the setup, the adversary outputs a pair of lists L_j, L_{j+1} such that these lists of ciphertexts are not permutations of one another (or in the case of L_0 to L_1, up to an exponentiation by $\mathsf{sv}_{0,\mathsf{id}}$).

Since the proof of correct mixing is verified (otherwise the success flag of the setup would be 0), this event happens with a probability bounded by the probability of breaking the soundness property of the proof system.

Game_9: This game changes which CS_j value PC uses to generate all the codes. Instead of decrypting $c_{\mathsf{CS},j}$, it now uses the seed extracted in Game_1.

The correctness of the decryption algorithm again guarantees that this game is a restatement of the previous game.

Game_{10}: This game changes which LC value PC uses in the Setup. Instead of computing the decryption of c_{LC} to recover LC, PC directly uses the LC generated by SC.

The correctness of the decryption, along with the fact that PC receives the encryption of LC computed by SC (because of Game_3), ensures that LC is computed in the exact same way in both games.

Game_{11}: In this game, we bring yet another modification on the way the PC operates in the setup. Instead of retrieving the permutation σ_{Setup} of the mixnet from $c_{\mathsf{v}}^{\mathsf{PC}}$, we retrieve it from $T_{\mathsf{id}}^{\mathsf{Setup}}[i]$.

In the previous game, σ is computed from $\mathsf{v}_{\sigma(i)} = \mathsf{Dec}(\mathsf{sk}_{\mathsf{PS}}, c_{\mathsf{v}}^{\mathsf{PC}}[i])$. Now, we compute σ such that $(\mathsf{v}_{\sigma(i)})^{\mathsf{sv}_0} = L_m[i][0]$ which is the same value because of the soundness of the mixnet (see Game_8).

At this point, we do not use $\mathsf{sk}_{\mathsf{PS}}$ anymore.

Game_{12}: In this game, we hide the content of $[L_0[i][1]]_{1\leq i\leq n}$ and c_{LC}. More precisely, we replace with random values the content of $[L_0[i][1]]_{1\leq i\leq n}$ and c_{LC} computed in the Setup.

Similarly to Game_7, this is a direct DDH reduction.

Game_{13}: In this game, we introduce a new failure event. If we receive from the adversary a random oracle request for some login code LC of an honest voter id, then we abort.

At this point of the proof, the login codes of honest voters are completely independent of the view of the adversary (because of Game_{12}). Since they are drawn from $\{0,1\}^\lambda$, the adversary has a probability $\frac{|H|}{2^\lambda}$ of guessing an honest voter's login code in its first RO request. If the adversary makes q such requests in the whole game, then we can bound the probability of this event by $q\frac{|H|}{2^\lambda}$ using the union bound, which is negligible.

Game_{14}: We introduce the following failure event. If there is an $O\mathsf{processVote}(\mathsf{id})$ in which the adversary sends a ballot b' different from the ballot b_β computed in the $O\mathsf{voteLR}$ request for this id, and if the signature $\mathsf{sig}_{\mathsf{id}}$ of this ballot is valid, then we abort the game.

Indeed, for this event to happen, the adversary must craft a valid signature for b under $\mathsf{pk}_{\mathsf{id}}$. However, the adversary only possesses a valid signature for b_β. Hence, this event happens with negligible probability because of the existential unforgeability of the signature scheme.

Game_{15}: In this game, we modify the check of the return codes $\mathsf{RC}[\mathsf{v}[i]] \stackrel{?}{=} \mathsf{RC}^{Sent}[i]$ in ProcessVote algorithm. Instead, we compare $\mathsf{RC}^{Sent}[i]$ with $\oplus_{j=1}^m \mathsf{pRC}_j[i^\star]$, for every i, and output AC if they are all equal. This condition is equivalent to the adversary sending honest pRC_j^{Sent}, hence the adversary does not learn anything from the check of the return codes.

This transition is a restatement of the previous game, because of the same argument as the transition in the Game Game_8 of the verifiability proof.

There is one last part of the protocol where v_β appears in the view of the adversary: in the $T^{\mathsf{Vote}} = \varphi_{\mathsf{id}}(\mathsf{v}_\beta)$ value. In Game_{17}, we proceed to hide this last occurrence of v_β.

Game_{16}: In the SetupParams algorithm, we change the way we generate the voting options v_i. Instead of sampling them at random, we generate a trapdoor $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$ and set $\mathsf{v}_i = g^{\alpha_i}$.

This transition is simply a programming of the ROM.

Game_{17}: In this game, we modify how each of the φ_{id} is computed. Instead of computing $\varphi(x) = x^{\mathsf{sv}_{0,\mathsf{id}}\sum_{j=1}^m \mathsf{sv}_{j,\mathsf{id}}}$, we now sample each φ as a random function.

This transition is guaranteed by DDH, with a 2D hybrid argument, because we have to iterate over the voting options $[\mathsf{v}_i]_{1\leq i\leq n}$ and over the voters $\mathsf{id} \in I\mathcal{D}$. This is the same reasoning as in the analogous Game_{27} for the case where one CC is honest.

In the $(i^\star, \mathsf{id}^\star)$-th hybrid, we receive a tuple (g,a,b,c), and set respectively $\mathsf{v}_{i^\star}$ as a, $\mathsf{pv}_{0,\mathsf{id}^\star}$ as b and

- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to a random element if $\mathsf{id} < \mathsf{id}^\star \vee (\mathsf{id} = \mathsf{id}^\star \wedge i < i^\star)$.
- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to $c^{\sum_{j=1}^m \mathsf{sv}_{j,\mathsf{id}}}$ if $\mathsf{id} = \mathsf{id}^\star \wedge i = i^\star$.
- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to $b^{\alpha_i \sum_{j=1}^m \mathsf{sv}_{j,\mathsf{id}}}$ if $\mathsf{id} = \mathsf{id}^\star \wedge i > i^\star$.

- $\varphi_{\mathsf{id}}(\mathsf{v}_i)$ to $\mathsf{v}_i^{\mathsf{sv}_{0,\mathsf{id}} \Sigma_j \mathsf{sv}_{j,\mathsf{id}}}$ if $\mathsf{id} > \mathsf{id}^\star$.

The $(i^\star, t^\star)$-th hybrid constraints two slices: the values $\varphi_{\mathsf{id}_t}(\mathsf{v}_{i^\star})$ for $t \neq t^\star$ and the values $\varphi_{\mathsf{id}_t^\star}(\mathsf{v}_i)$ for $i \neq i^\star$. Luckily, we can compute the former thanks to the programming of the v_i in Game_{26}, and the latter thanks to the extracted sv_j.

By iterating over these hybrids, first along the first coordinate then along the second one, each hybrid is indistinguishable from the previous one under DDH, and thus Game_{17} is indistinguishable from Game_{16}. Again, this transition is tight, because of the self-reducibility of the DDH tuple, as shown in [23].

Conclusion. Now, each v_β is completely independent of the view of the adversary, and it has a probability $\frac{1}{2}$ of guessing β.

Our main privacy theorem is as follows.

Theorem 3. *The protocol has vote privacy if* PC*,* TC*, and either* SC *or one of the online* CC*s are honest.*

It follows directly from Theorems 4 and 5 that the protocol has privacy if PC, TC_{EB} and either SC or one of the CCs are honest. This proves Theorems 3 when considering TC to be played by TC_{EB} and the CC. The general case where TC is left abstract and is assumed to be honest as a whole can be deduced from our proof, by simply considering distinct roles for CC in the voting phase and in the tally phase.

F.4.3 Eligibility

Theorem 2. *The protocol satisfies eligibility if* SC *and* PC *are honest.*

Proof.

Game_0: $\mathsf{Game}_{i,\mathcal{A}}(\lambda)$ is the original eligibility game $\mathsf{Exp}^{\mathsf{elig}}_{\mathcal{A}}(\lambda)$. In this proof, we are going to build a sequence of games $(\mathsf{Game}_{i,\mathcal{A}}(\lambda))_{i=0}^{N}$, transitioning from $\mathsf{Exp}^{\mathsf{elig}}_{\mathcal{A}}(\lambda)$ to a game which directly reduces to the unforgeability of the signature scheme. In the following, we abbreviate $\mathsf{Game}_{i,\mathcal{A}}(\lambda)$ as Game_i.

Since we want to reduce to the unforgeability of the signature scheme, we need to make sure that for each $\mathsf{id} \in A$, the voter's secret key $\mathsf{sk}_{\mathsf{id}}$ is not in the adversary's view. Since this view is generated from LC, we need to hide the content of c_{LC}.

We actually use the same sequence of games as the privacy proof with SC honest, up to Game_{13}. Not all games are necessary, in particular games Game_4 to Game_7. At the end of Game_{13}, the set of signing keys of absentee voters $\{\mathsf{sk}_{\mathsf{id}}\}_{\mathsf{id}\in A}$ is out of the view of the adversary, and we can reduce the game to an unforgeability experiment.