Verifiable and Private Vote-by-Mail

Henri Devillez[1], Olivier Pereira[1,2], and Thomas Peters[1]

[1] UCLouvain – ICTEAM – Crypto Group, B-1348 Louvain-la-Neuve – Belgium
[2] Microsoft Research, Redmond, WA, USA
first.last@uclouvain.be

Abstract. Vote-by-mail is increasingly used in Europe and worldwide for government elections. Nevertheless, very few attempts have been made towards the design of verifiable vote-by-mail, and none of them come with a rigorous security analysis. Furthermore, the ballot privacy of the currently deployed (non-verifiable) vote-by-mail systems relies on procedural means that a single malicious operator can bypass.

We propose a verifiable vote-by-mail system that can accommodate the constraints of many of the large ballots that are common in Europe. Verifiability and privacy hold unless multiple system components collude to cheat on top of the postal channel. These security notions are expressed and analyzed in the simplified UC security framework.

Our vote-by-mail system only makes limited requirements on the voters: voters can verify their vote by copying and comparing short strings and verifying the computation of a single hash on a short input, and they can vote normally if they want to ignore the verification steps completely. Our system also relies on common cryptographic components, all available in the ElectionGuard verifiable voting SDK for instance, which limits the risks of errors in the implementation of the cryptographic aspects of the system.

Table of Contents

Verifiable and Private Vote-by-Mail 1
Henri Devillez, Olivier Pereira, and Thomas Peters
1 Introduction 3
1.1 A verifiable vote-by-mail protocol 3
1.2 Security Analysis 4
1.3 Related works 5
1.4 Structure of the Paper 6
2 Postal Voting Model 6
2.1 Overview 6
2.2 Parties 6
2.3 Postal Voting Protocol 7
3 Protocol overview 7
3.1 Ballot structure 7
3.2 Voting procedure 8
3.3 Tallying procedure 8
3.4 Verification procedure 9
4 Building blocks 9
4.1 Threshold encryption scheme 9
5 Protocol 10
5.1 Setup procedure 10
5.2 Voting Procedure (Figure 4) 11
5.3 Tallying procedure (Figure 5) 12
5.4 Verification procedure (Figure 6) 13
6 Security 15
6.1 Model 15
6.2 Ideal Functionality for postal voting 16
6.3 Secure Postal Voting 18

1 Introduction

The use of mail to return ballot keeps increasing as a mechanism for remote voting: it is offered to all the voters of several European countries, including Germany where its adoption grew from 29% in 2017 to 47% in the Bundestag Election of 2021,[3] or Switzerland where more than 90% of the ballots are mailed according to Swiss Post[4], and is also increasingly used in the rest of the world, including by 43% of the voters during the 2020 U.S. presidential election. This adoption clearly dwarfs the use of Internet voting, which is often considered as out-of-reach of current security technologies[5] and has only been consistently offered to the general public in Estonia.

Vote-by-mail is appealing for several reasons: it offers a way to vote to those who cannot or do not wish to attend a polling place and, in the remote voting setting (which remains weaker than in-person voting), it offers a simple solution to the cast-as-intended verifiability question that keeps plaguing Internet voting solutions and is typically either sidestepped as in France [19] or repeatedly broken as in Estonia or Switzerland [3,24,28].

It however raises important security concerns regarding the recorded-as-cast verifiability and vote privacy. Postal services (including foreign ones when voting from abroad) must be trusted to not (selectively) delay and/or open envelopes and/or modify or substitute ballots. Some states avoid using the postal system and deploy a network of official ballot drop boxes, but this can in-turn support the deployment of unauthorized ballot drop boxes,[6] offering new actors the possibility to tamper with a number of ballots. At a later step, the proper custody of the ballots when they reach return offices may be hard to guarantee, especially when ballots are returned over a relatively long period of time [25]. Here again, there are privacy and integrity concerns. Finally, we cannot ignore the challenges of voter impersonation, vote selling and coercion, which are inherent to remote voting and may be more or less important depending on the context [20].

Despite the importance of vote-by-mail and of its associated security risks, very few proposals have been made to offer verifiable vote-by-mail solutions, and none of them come with a systematic security analysis.

1.1 A verifiable vote-by-mail protocol

We propose a verifiable vote-by-mail protocol, together with a rigorous analysis of its security properties.

Design choices Our choices within a relatively large design space are motivated by various practical considerations that were developed as part of a study requested by the federal election office in Belgium.[7] The current paper refines and offers the first rigorous analysis of the voting system proposed there.

Hand-marked paper ballots We wish to keep intact the benefit of cast-as-intended verifiability offered by vote-by-mail. As such, we seek for a design in which voters can hand-mark their ballot and submit it in a way that is similar to traditional vote-by-mail experiences. Our scheme remains compatible with the use of ballot marking devices, but this is not a requirement. This allows side-stepping the pitfalls related to the (lack of) auditability of ballot marking devices [2,10]. This is also beneficial for privacy as no potentially corrupted device linked to a voter can learn the content of a vote, which may become more important as private printers are also becoming less common.

Internet ballot delivery and verification Even if the vote goes by mail, there are steps that can take advantage of the Internet. In our case, voters download blank ballots from the internet and also use the internet to verify that their vote has been recorded without being altered. Internet ballot delivery has the

[3] `https://www.bundeswahlleiterin.de/en/info/presse/mitteilungen/bundestagswahl-2021/53_21_briefwahlbeteiligung.html`

[4] `https://www.post.ch/fr/notre-profil/actualites/2023/comment-la-suisse-est-devenue-le-pays-du-vote-postal`

[5] `https://nap.nationalacademies.org/catalog/25120/securing-the-vote-protecting-american-democracy`

[6] `https://elections.cdn.sos.ca.gov/ccrov/pdf/2020/october/20240jl.pdf`

[7] `https://elections.fgov.be/informations-generales/etude-sur-la-possibilite-dintroduire-le-vote-internet-en-belgique/`

advantage over mail delivery that it saves the physical ballots half of their trip – only the return is needed. The delays incurred by postal services often make it impossible for voters to receive and return their ballot before the closing of the polls, especially for voters living abroad and on different continents[20]. This choice may also support stronger forms of voter authentication in places where online authentication mechanisms are available – which may be safer than the use of the ability to access a physical mailbox as a means of authentication, assuming that voters have one. Still, our system is also compatible with ballot delivery by mail if it happens to be a good option for external reasons.

Distributed trust for integrity We do not trust the postal service for integrity. Of course, we cannot guarantee that it will not alter or suppress ballots. But our system makes it observable if a ballot is not delivered or if it is altered.

Furthermore, even in collusion with the postal service, no single entity can break the integrity of the election. Still, integrity can be broken by a coalition of other actors.

This integrity model may appear to be weaker than what is demanded by some definitions of end-to-end verifiability (see discussions in [6]), which require that no collusion of parties should be able to violate the integrity of the election result without a voter (or an observer) being able to detect it. Here, we would argue that such a requirement cannot be satisfied in most practical cases and that, as such, our integrity model does not substantially weaken the best form of verifiability that we can hope for. Indeed, in most government elections, lists of voters are not public, which makes it impossible to detect integrity violations resulting from ballot stuffing – election officials and observers with specific access rights need to be trusted here. Similarly, in large scale paper elections, voters and observers must trust the ballot box chain-of-custody. It may still be a design goal to make the system as end-to-end verifiable as possible. But, as we will see below, by accepting that trust needs to be partially delegated, our approach leads to a system that is notably simpler than other solutions that offer a better compatibility with E2E, which may be a crucial quality for a correct deployment.

Distributed trust for privacy, except for the post Apart from the postal service, no single party can break ballot privacy alone. The trust in the postal service for privacy seems unavoidable when ballots are mailed: depending on the context, the postal service may be able to identify who sends a ballot (even if our ballots do not contain any identifying mark, the post office may see who drops an envelope), and the postal service would likely be able to carefully open an envelope, read its content, then close it back (or place the ballot in a new envelope). However, such a link is unlikely to be available for a return office that receives anonymous envelopes.

Limited coercion resistance Remote voting has inherent limitations regarding coercion: if a coercer can obtain a voter's ballot, then he can just submit it himself. Still, a voter who follows our voting instructions gets no useful receipt. And, the verification process is also designed in such a way that, if the voter is instructed by an external coercer to send information after submitting his vote, the voter can do so in such a way that all the verification steps will be consistent with the coercer requirements. In other words, the verifiability of our system does not worsen its limited coercion resistance compared to traditional vote by mail – even though the coercion resistance strategy may require a bit more work. This differs from various Internet voting systems where a voter who keeps the random coins used to encrypt his ballot is then able to prove how he voted to a third party thanks to the election bulletin board.

Classical cryptographic primitives Verifiable voting protocols may quickly become quite sophisticated and gather a number of complex cryptographic primitives that have never been standardized and for which well tested and maintained implementations do not exist. This has been the source of numerous failures in the deployment of Internet voting systems in the past. As such, we aimed at designing our protocol in such a way that it only requires relatively well known primitives for which various tested implementations exist: on top of TLS, all our primitives are implemented in systems like Helios [1], Belenios [15] and in the ElectionGuard open source SDK [5].

1.2 Security Analysis

We propose a security analysis by specifying our protocol and analyzing it in the simplified UC (SUC) framework [12]. This is a significant departure from the previous verifiable vote-by-mail proposals, which were never supported by a systematic security analysis (see the related works section below).

The choice of the SUC framework is appealing for several reasons. First, voting is a task that is naturally expressed as an ideal functionality that, in its simplest form, receives votes and returns the election result. Requiring that a real protocol emulates that functionality is a convenient way of expressing security. Though numerous game-based security definitions for voting exist and may be easier to prove, several SoK papers [8,14] illustrate the difficulty to assess whether they capture the desired security notions. Furthermore, proving that these game-based definitions imply security in the sense of emulating an ideal functionality has been repeatedly used as a way to argue about their meaningfulness [8,9,16]. Our vote-by-mail setting is very different of the one considered in these game-based definitions, which makes them hard to reuse in our case, and we then directly target definitions based on an ideal functionality.

Second, the choice of using the SUC framework instead of the traditional UC framework is motivated by its simplicity (which was a design goal) and its suitability in our setting: the simplicity of the SUC framework results from the choice to restrict its possible uses to protocols executed between a set of parties chosen in advance and that do not support the dynamic creation of participants.

Our security analysis nevertheless remains a challenging task: in terms of modeling, we need to find a proper abstraction for paper ballots and their properties. Furthermore, the relatively large number of entities participating in the protocol and the associated corruption model require dealing with many different cases. We did our best to keep the analysis modular: our proof starts from a base simulator, which is refined to handle the various corruption cases.

1.3 Related works

A large proportion of the (often partially) verifiable voting systems that use the mail are not actually vote-by-mail: voters receive a ballot by mail, which typically contains various codes, and the voters use these codes to cast their ballot through the Internet. These systems include the original code voting proposal by Chaum [13] and many follow-up works, including some that have been actually deployed in government elections, including Remotegrity [31], the Norwegian voting system [23] and the Swiss one [30].

In 2013, Benaloh et al. proposed a first verifiable postal voting system [7]. In their protocol, voters access and complete their ballot on their computer, which then prints a human readable version of the vote together with an encrypted and signed version of it. That printed ballot is then submitted by mail. Because of the presence of the signature, anyone opening an envelope may be able to violate the ballot privacy. It also does not offer protection against a malicious device and a malicious voting channel: the voting device could print a human-readable vote that matches the voter intent and encrypt and sign a different vote intent. A mail adversary can then change the human-readable part and keep the signed encrypted record, so that no auditing step would make it possible to detect the inconsistency. A protocol that shares these same features was proposed recently proposed by McMurtry et al.[26], which offers a higher probability to detect vote manipulations and slightly better receipt-freeness. By comparison, our protocol supports the submission of ballots that do not contain any identity, guarantees privacy if the voting device is corrupted and can still guarantee integrity when both the voting device and the postal channel are corrupted.

In a different approach, Strobe [4] offers en elegant solution to the vote-by-mail problem, based on the same cryptographic primitives that we are using. A central benefit of Strobe is its compatibility with an E2E verifiable voting process that could be used by voters voting in-person at the same election for instance. An important constraint of Strobe is its requirement that authorities produce two ciphertexts for each voter encrypting and proving the validity of every possible voter choice, which may quickly become prohibitive depending on the ballot style. A malicious authority is also only detected with probability at most one half, while our design can easily make it much larger, typically above $1-2^{-10}$. The Strobe design has been further refined in RemoteVote and SafeVote [17]. These protocols support a much broader class of ballot styles, but also bring additional complexities: all the voters need to interact with a server as part of the voting process in order to decide how a spoiling process needs to be executed. Besides, as in Strobe, the verification of spoiled ballots requires the use of a honest device to perform heavy (public) cryptographic operations. In contrast, all the voter verification steps of our protocol can mainly be performed “in the head": they are just string comparisons, except for a single hash value of a random token printed on the empty ballot. This printed token can only be 20-character long with 6-bit encoding characters and is independent of the votes.

As indicated above, none of the works discussed here offer a systematic security analysis of the properties of the proposed protocols.

1.4 Structure of the Paper

Section 2 introduces the structure of our postal voting protocol. Section 4 describes our cryptographic building blocks. Section 3 offers a high-level description of our voting protocol, followed by a full description in Section 5. We introduce our postal voting ideal functionality and prove that our protocol emulates that functionality in Section 6.

2 Postal Voting Model

2.1 Overview

We consider a remote election in a vote-by-mail setting. The election is overseen by a party called the elections authorities. This party sets the parameters of the election p, which consists among other things in the list of valid voters, the list of candidates for which one can vote and a list of talliers. These parameters are given to each participant at the beginning of the election.

Ballots are generated by an independent server. When a voter asks the server for a ballot, the voter authenticates (with an electronic ID for example) and receives a blank ballot if they have not received any ballot yet.

The voter can then print the ballot, fill it by hand and send it by mail to a postal voting election office. There might be additional materials associated to the ballot that should not be sent. That includes resources that could be used in a vote verification procedure.

During the tallying phase, the election office interacts with the talliers to compute the result of the election that can be published once every ballot has been counted. Moreover, the election office interacts with a verification server that can be used by voters to verify that their vote has been correctly counted as intended without any loss of vote privacy.

2.2 Parties

The proposed election system includes the following independent parties:

- EA: The election authorities setups and manages the execution of the election. Other parties report to the election authorities if they notice an irregular behavior in the protocol.
- BI: The ballots issuer server generates fresh ballots and issues them to voters after proper identification.
- ID: The voters participate in the election with identities $\mathsf{ID} = \{\mathsf{id}_1, \ldots, \mathsf{id}_\mathsf{l}\}$. Each voter can request a fresh ballot from BI, ballot that they can print, mark with their vote, and send by mail to the election office. They can later check that their ballot has been correctly recorded by interacting with the verification server VS.
- CB: The ballot collecting box collects the envelopes of the ballots sent by mail. It shuffles these envelopes, making them anonymous, before delivering them to the election office EO.
- EO: The election office counts the votes of the election and computes the result of the election in collaboration with the talliers. It also interacts with the verification server VS in order to make the auditing data available.
- T: The talliers $\mathsf{T} = \{\mathsf{T}_1, \ldots, \mathsf{T}_\mathsf{n}\}$ compute the result of the election, jointly with EO.
- VS: The verification server provides the voters with the data they need to verify that their vote has been correctly recorded and tallied.

All the parties but the voters are called the *official parties.*

2.3 Postal Voting Protocol

Definition 1 (Postal Voting Protocol). *A* Postal Voting Protocol $\mathcal{V} = (\mathsf{Setup}, \mathsf{Vote}, \mathsf{Tally}, \mathsf{VerifyVote})$ *is a tuple of protocols:*

1. Setup is a distributed protocol initiated by EA and jointly executed by the official parties, with as input the election parameters p that contain the list of voters, candidates and talliers.
2. Vote is a protocol executed by the voters. Given a blank ballot b°, the auxiliary information aux and the vote intention of the voter, it returns an interpretable ballot $\mathsf{b}^{\bullet}$ encoding the vote intention and a receipt rcpt used in the verification process.
3. Tally is a protocol executed by the election office EO and the talliers after having received the ballots. At the end of this protocol, each tallier and the election office sends to EA the result of the election r and send the proofs of individual verifiability Π to the verification server.
4. $\mathsf{VerifyVote}$ is a protocol executed by the verification server VS and voters willing to verify their vote. At the end of the protocol, the voters output honest if they are convinced that their vote has been correctly counted and cheat otherwise. In that case, they report the inconsistency to the election authorities.

We do not discuss formally what we expect from those algorithms yet. This will be captured in the security analysis through an ideal functionality.

During the execution of the protocol, parties can notice inconsistencies that should not occur during an honest execution of the protocol. In that case, they can report a complaint by sending a message cheat to the election authorities EA.

Voters send their completed ballot $\mathsf{b}^{\bullet}$ by mail. This postal channel is unreliable and asynchronous, hence ballots could be modified or never arrive at CB. Since CB sends the ballots to EO at some point, ballots received after the tally has started are discarded without even being ever opened. We consider that all the other communication channels between the parties are secure, that is they are confidential and authentic. In particular the set of identities $\mathsf{ID} = \{\mathsf{id}_1, \dots, \mathsf{id}_l\}$ is known by all the official parties, and communications related to any voter's identity id cannot be misidentified. If the interface that shows b° to voters is corrupt, for simplicity we assume that BI is corrupt, and the same for VS in $\mathsf{VerifyVote}$.

3 Protocol overview

To help getting some intuition about our protocol, we first give an overview of the different parts of the process. The protocol is designed for approval voting elections. By approval voting, we mean that given a list of m candidates or questions, a voter can choose to approve any subset of these candidates or questions.

3.1 Ballot structure

During the setup of the election, the ballot issuer BI crafts all the ballots. These ballots consist in three different components that can be printed on paper.

The first component is a simple list of the candidates of the race with a box for each candidate that can be checked to show the approval (see Figure 1). This sheet will be marked by the voter with their voting intentions and sent to CB. This sheet is also designed to be easily scanned to tally the result electronically by EO.

Moreover, there is an unique voting token tk included at the bottom of the selection sheet, which is a relatively short sequence of characters. This voting token is used to identify the ballot and to ensure that the ballot is tied to a real voter who authenticated to the ballot issuer server.

The second component is the code sheet. This component is similar to the first component with the following modification. The box that can be checked next to each candidate is replaced by two codes corresponding to the two possible choices. These codes can be printed on the ballot as a sequence of alphanumeric symbols.

The third and final component is the note sheet. This is a blank list without the names of the candidates. It contains $H(\mathsf{tk})$ at the bottom of the sheet. During the casting process, the voter is expected to write the codes associated to their choice in this note sheet. The hash of the voting token, is also written on this note sheet.

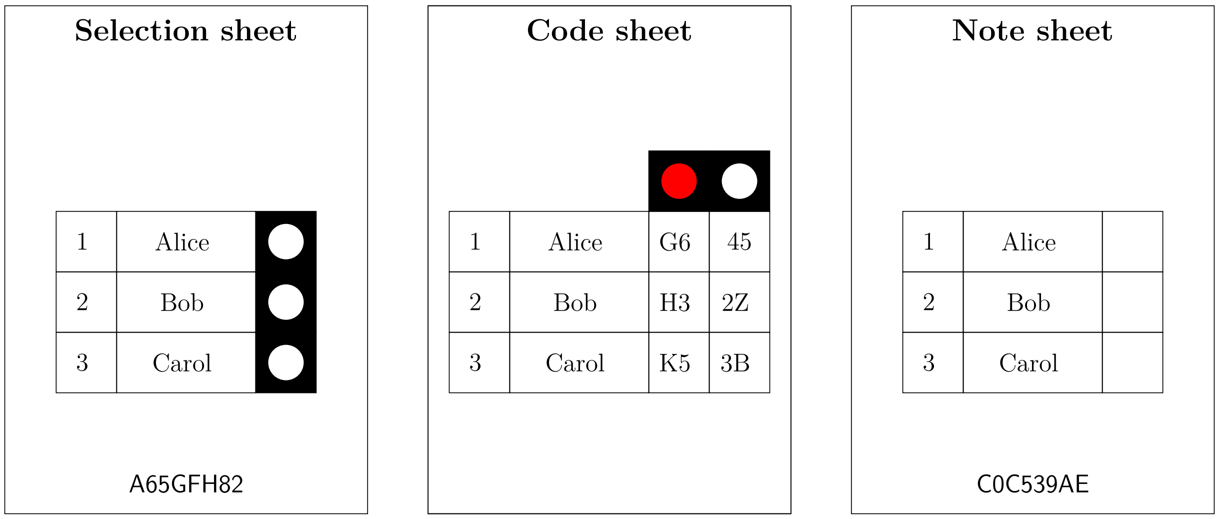


Fig. 1: Example of a blank ballot.

3.2 Voting procedure

The voting procedure of a voter is quite intuitive. Given a paper ballot describe above, a voter selects the candidates they approve by checking the corresponding box and copies the chosen codes of the code sheet in the note sheet. The voter can check that tk and $H(\mathsf{tk})$ match, for instance, by taking a picture of tk with a dedicated mobile application that returns hash values, or any other means. They then destroy the code sheet and send the selection sheet by mail to the ballot collecting box. At a later stage, it will be possible for the voter to verify that their ballot has been correctly counted using these codes.

An interesting and desirable feature of this paper voting protocol is that the ballots are interpretable. That is, a voter can recover their vote from the ballot without any other information and does not have to trust a mechanism hiding their vote, like in most onsite elections.

3.3 Tallying procedure

At the other end of the postal service, the ballot collecting box receives all the envelopes. These envelopes are open and shuffled, then the election office EO proceeds to the tally. Since the ballots are interpretable, the election office can directly compute the result.

To prove to the voter that their vote has been counted correctly, the election office EO and the talliers $\mathsf{T}_1, \ldots, \mathsf{T}_\mathsf{n}$ proceed to compute the codes of the voter's selection. These codes have been encrypted by the ballot issuer using a threshold encryption scheme at the ballot generation.

The election office then send to the talliers the selections of the ballot for each voting token. These talliers, who possess the shares of the private key of the threshold encryption scheme, then compute together the decryption of the code corresponding to the voter's choice. To compute these codes, they all have to agree on the choice of the voter and then on the result of the election. These codes are finally sent to the verification server who will interact with the voters.

3.4 Verification procedure

The procedure is also quite intuitive: The voter sends the hash of their voting token to the verification server and receives a code for each option. If these codes match the codes they have written on the note sheet, then he has good evidence that their vote has been correctly counted.

A malicious election office could always try and guess the codes of a voter while modifying their vote. We can control the probability of success of this adversary with the length of these codes: a longer code will be less likely to be guessed, at the cost of usability. 16 bits seems to be a good trade-off between security and usability.

4 Building blocks

The postal voting protocol we propose relies on common cryptographic primitives: a semantically secure t-out-of-n threshold encryption scheme with a distributed key generation, and a hash function H modeled as a random oracle. The computation of any cryptographic component is always made by official parties; voters are only assumed to read, write, and compare (short) strings. We rely on the usual definition of negligible function ε in a security parameter λ as function in $\lambda^{-\omega(1)}$. An overwhelming probability in λ is a probability in $1 - \lambda^{-\omega(1)}$.

4.1 Threshold encryption scheme

A t-out-of-n threshold encryption scheme is a public-key encryption scheme whose secret key is shared between n parties. At least t of them have to cooperate to decrypt a ciphertext but strictly less than t colluding parties cannot learn anything about the content of a ciphertext.

Definition 2. *A threshold encryption scheme is a tuple of algorithms* $(\mathsf{TGen}, \mathsf{TEnc}, \mathsf{TPartDec}, \mathsf{TCombine})$ *with the following properties. The key generation algorithm* $\mathsf{TGen}(1^\lambda, \mathsf{n}, \mathsf{t})$ *takes as input a security parameter* λ*, the number* n *of participating parties, and the number* t *of parties required to decrypt a ciphertext. It returns a public key* pk *and a tuple of* n *secret keys* $\mathsf{sk}_1, \ldots, \mathsf{sk}_\mathsf{n}$*. On input a public key* pk *and a message* m *in the message space* $\mathcal{M}$*, the probabilistic encryption algorithm* $\mathsf{TEnc}(\mathsf{pk}, \mathsf{m})$ *outputs a ciphertext* c*. The partial decryption algorithm* $\mathsf{TPartDec}(\mathsf{pk}, \mathsf{sk}_\mathsf{i}, \mathsf{c})$ *outputs a decryption share* d_i *of a ciphertext* c *given a secret key* sk_i*. Eventually, given the public key* pk*, a ciphertext* c *and a set of* t *decryption shares* $\mathsf{TCombine}(\mathsf{pk}, \mathsf{c}, (\mathsf{d}_{\mathsf{i}_1}, \ldots, \mathsf{d}_{\mathsf{i}_\mathsf{t}}))$ *outputs a message.*

Correctness requires that a honest run of these algorithms for any message m and a resulting ciphertext c leads $\mathsf{TCombine}$ to output back m with overwhelming probability. Following Damgard and Jurik [18] and Fouque, Poupard and Stern [21], a threshold encryption scheme is semantically secure if the underlying public-key encryption is indistinguishable against chosen-plaintext attacks (CPA-secure, for short), and if the partial decryption algorithm is simulatable (a.k.a. decryption simulatability).

More precisely, for the CPA security: for all integers $0 < \mathsf{t} \leq \mathsf{n}$, for any PPT adversary $\mathcal{A}$, and any index set $\mathcal{C} \subset [\mathsf{n}] = \{1, \ldots, \mathsf{n}\}$ with $|\mathcal{C}| < \mathsf{t}$ and $\mathcal{H} = [\mathsf{n}] \setminus \mathcal{C}$, the advantage $|\Pr[\mathsf{Exp}^{(\mathsf{t},\mathsf{n})\text{-cpa}}_{\mathcal{A},\mathcal{H},\mathcal{C}}(\lambda)] - 1/2|$ in the experiment in Figure 2 is negligible. Regarding decryption simulatibilty, there must exist a PPT algorithm TSim such that, for any PPT adversary $\mathcal{A}$, for any message m, and any $\mathsf{t}, \mathsf{n}, \mathcal{C}, \mathcal{H}$ as above, the advantage $|\Pr[\mathsf{Exp}^{(\mathsf{t},\mathsf{n})\text{-sim}}_{\mathcal{A},\mathcal{H},\mathcal{C}}(\lambda)] - 1/2|$ in the experiment in Figure 2 is negligible. We assume $\mathsf{TPartDec}$ outputs a non-interactive zero-knowledge proof that the decryption share is computed honestly. $\mathsf{TCombine}$ verifies those proofs and output $\perp$ if a decryption share is invalid.

In our postal voting system, we replace the centralized algorithm TGen by a distributed protocol Π_{TGen} realizing TGen in a generic way (through a MPC protocol). At the end of this protocol, each party T_i receives their private key sk_i. Since the encryption and the decryption simulator algorithms are non-interactive, all the above notions still hold in our system when a malicious adversary participates in the key generation. Finally, we stress that the encryption algorithm does not have to come with any additional verifiability

```
$\mathsf{Exp}^{(t,n)\text{-cpa}}_{\mathcal{A},\mathcal{H},\mathcal{C}}(\lambda)$

1 : $\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_\mathsf{n} \leftarrow\$ \mathsf{TGen}(1^\lambda, \mathsf{n}, \mathsf{t})$
2 : $\mathsf{m}_0, \mathsf{m}_1, \mathsf{st} \leftarrow\$ \mathcal{A}(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in\mathcal{C}})$
3 : $\mathsf{b} \leftarrow\$ \{0,1\}$
4 : $\mathsf{b}' \leftarrow\$ \mathcal{A}(\mathsf{st}, \mathsf{TEnc}(\mathsf{pk}, \mathsf{m}_\mathsf{b}))$
5 : $\textbf{return } \mathsf{b}' == \mathsf{b}$
```

```
$\mathsf{Exp}^{(t,n)\text{-sim}}_{\mathcal{A},\mathcal{H},\mathcal{C},\mathsf{m}}(\lambda)$

1 : $\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_\mathsf{n} \leftarrow\$ \mathsf{TGen}(1^\lambda, \mathsf{n}, \mathsf{t})$
2 : $\mathsf{c} \leftarrow\$ \mathsf{TEnc}(\mathsf{pk}, \mathsf{m})$
3 : $\mathsf{b} \leftarrow\$ \{0,1\}$
4 : $\text{if } \mathsf{b} = 0 : \mathsf{b}' \leftarrow\$ \mathcal{A}(\{\mathsf{sk_i}\}_{\mathsf{i}\in\mathcal{C}}, \{\mathsf{TPartDec}(\mathsf{sk_i}, \mathsf{c})\}_{\mathsf{i}\in\mathcal{H}})$
5 : $\text{if } \mathsf{b} = 1 : \mathsf{b}' \leftarrow\$ \mathcal{A}(\{\mathsf{sk_i}\}_{\mathsf{i}\in\mathcal{C}}, \mathsf{TSim}(\mathsf{pk}, \mathcal{H}, \mathsf{c}, \mathsf{m}))$
6 : $\textbf{return } \mathsf{b}' == \mathsf{b}$
```

Fig. 2: IND-CPA experiment (left) and Decryption simulatability experiment (right). $\mathcal{C}$ and $\mathcal{H}$ denote the sets of corrupted and honest parties.

notions. Therefore, we can use a plain ElGamal whose decryption is made threshold following the TDH constructions [29]. This gives the decryption simulatability, but we keep the CPA encryption algorithm (CCA-security is useless here). This centralize threshold encryption scheme can be decentralized by following [22] (based on [27]), for instance. We note that decryption share validity is enforced in [29] thanks to a non-interactive Chaum-Perdersen zero-knowledge proof of language membership (in the random oracle model). Such proofs allow determining which decryption shares to use in TCombine.

We stress that even if the encryption and the decryption are very efficient (only Π_{TGen} is costly, but it is run prior the election days), voters will never have to compute any of them. Given a blank ballot, the only cryptographic function that they could verify in our voting system is the computation of a single hash value.

5 Protocol

In this section, we present the formal description of the postal voting protocols. We use a Python-like notation for sets and dictionaries. We also define $\mathsf{list}(\mathsf{n})$ as a list of n entries and $\mathsf{list}(\mathsf{n}, \mathsf{m})$ as a list of n lists, each of them containing m entries. Those entries are initialized with the value $\perp$. We extract the i-th bit of an integer x as $\mathsf{x.bits(i)}$. We recall that all the channels are confidential and authentic, except the postal channel, and that the voters are identified by $\mathsf{ID} = \{\mathsf{id}_1, \ldots, \mathsf{id}_\mathsf{l}\}$.

5.1 Setup procedure

In our protocol, the setup procedure serves two purposes: the election key generation procedure and the ballots generations. More formally, Setup is a pair of algorithms (KeyGen, GenerateBallot) that takes as input the parameters of the election p. p consists in $(1^\lambda, \mathsf{m}, \mathsf{n}, \mathsf{t}, \mathsf{ID}, \mathsf{p\_limit})$, where λ is the security parameter. m and n are respectively the number of candidates (or questions) and talliers. t is the minimum number of talliers that have to be honest for the protocol to be verifiable. Regarding p_limit, a voter in our protocol might believe that their vote is correctly counted while this is not the case. This is because the protocol is based on a return code which gives confidence to the voter that their vote has been cast correctly if they receive in return the right code. An adversary could always try to guess this code and convince the voter in this way. We thus also include an upper bound on the probability of such a failure in the verification process, defined as p_limit.

At the end of the setup phase, the ballot issuer BI generated for each voter the blank ballots and the auxiliary information needed to vote. The trustees have the secret key materials that they will keep to make sure that their contribution is necessary for the ballot verification process to succeed. The election office EO and the verification server VS both also receive auxiliary information for the ballot verification process.

Election key generation (Figure 3a) The election key generation protocol $\mathsf{KeyGen}(1^\lambda, \mathsf{n}, \mathsf{t})$ is run by the talliers under the supervision of EA. It takes as input the number n of talliers and threshold number t of talliers. It outputs the public key pk of the election (and all the validity proofs) to all the involved parties, and EA sends it to BI. Moreover, this protocol also results in each tallier T_i creating and keeping their partial private key sk_i.

The protocol KeyGen is described in Figure 3a. Essentially, the talliers together run the secure distributed protocol Π_{TGen} of the threshold encryption scheme.

Ballots generation (Figure 3b) We define the ballot generation algorithm $\mathsf{GenerateBallot}(\lambda, \mathsf{pk}, \mathsf{m}, \mathsf{ID}, \mathsf{p\_limit})$ run by the ballot issuer BI. It takes as input the election key pk computed by KeyGen, the number of candidates m and the $\mathsf{p\_limit}$ parameter. The algorithm is described in Figure 3b in the box of the ballot issuer. It outputs a blank ballot $\mathsf{b}^{\circ}_{\mathsf{id}}$ and auxiliary information $\mathsf{aux}_{\mathsf{id}}$ for each $\mathsf{id} \in \mathsf{ID}$. It also outputs to every tallier, the election office and the verification server respectively the values $\mathsf{aux}_{\mathsf{T}}$, $\mathsf{aux}_{\mathsf{E}}$ and $\mathsf{aux}_{\mathsf{V}}$ that will be defined next.

Essentially, BI generates for each voter a random voting token tk which constitutes b°. Voters have different voting tokens with overwhelming probability. Hashes of those tokens are sent to the election offices EO, which constitutes $\mathsf{aux}_{\mathsf{E}}$. EO will later check if a voting token is valid by checking if its hash is in $\mathsf{aux}_{\mathsf{E}}$.

BI also generates additional information $\mathsf{b}^{\circ}_{\mathsf{id}}$ for the voter with identity $\mathsf{id} \in \mathsf{ID}$. For each possible choice of the candidates, BI generates two different codes c^0 and c^1. Those codes are such that their first bits are distinct and their remaining trailing bit-strings are distinct as well. The first bit of the code c^{j} is actually computed as follows: $\mathsf{tk}[\mathsf{i}] \oplus \mathsf{j}$. This link between the voting token and the codes is a protection against a corrupt BI. Those codes forms the auxiliary information of the voter $\mathsf{aux}_{\mathsf{id}}$ and are only needed for the verification procedure.

The length of trailing bits of the codes is $\mathsf{b} = \lceil \log_2(1 + \frac{1}{\mathsf{p\_limit}}) \rceil$. Given one code, we want the probability for an adversary to guess the other code to be bounded by $\mathsf{p\_limit}$. This is because if the adversary modifies the choice of one voter, it will receive the code of this other choice and will have to guess the other code to convince the voter that their vote has been correctly counted, even if it is false. Since the trailing bits r^0 and r^1 of the codes are independent up to their inequality, we have that:

$$\Pr[\mathsf{r}^0 = \mathsf{r}] = \begin{cases} 0 & \mathsf{r}^1 = \mathsf{r} \\ \frac{1}{2^{\mathsf{b}}-1} & \mathsf{r}^1 \neq \mathsf{r} \end{cases}$$

for any string r of b bits. Our choice of b lead to this bound of $\mathsf{p\_limit}$. Moreover, as the codes are independent for different choices (i.e., candidates), a cheating adversary who guesses k codes will not be caught with a probability bounded by $\mathsf{p\_limit}^{\mathsf{k}}$.

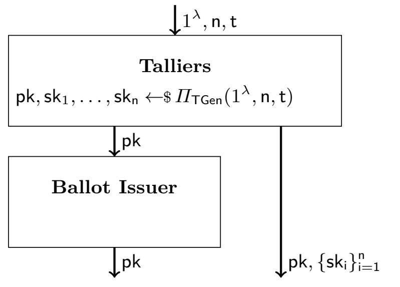
Furthermore, BI generates a way for the talliers to retrieve the codes of the voter's choice. To do so, for each ballot it generates an encryption $\mathsf{e}^{\mathsf{j}}_{\mathsf{i}}$ of the code $\mathsf{c}^{\mathsf{j}}_{\mathsf{i}}$ using the public key pk of the threshold encryption scheme and creates the tuple $(\mathsf{i}, \mathsf{e}^0_{\mathsf{i}}, \mathsf{e}^1_{\mathsf{i}}, \mathsf{H}(\mathsf{tk}||\mathsf{i}||0), \mathsf{H}(\mathsf{tk}||\mathsf{i}||1))$. Instead of including tk directly in the tuple, we use $\mathsf{H}(\mathsf{tk}||\mathsf{i}||\mathsf{j})$ to hide voting patterns. In this way, the talliers cannot link if two votes come from the same ballot. Moreover they will be able to detect if EO will ask them to decrypt both codes on a tuple, which is forbidden. Those tuples are shuffled and define the auxiliary information of the talliers $\mathsf{aux}_{\mathsf{T}}$.

Finally, VS receives from BI the link between the voters identity id and the hash $H(\mathsf{tk}_{\mathsf{id}})$ of their voting token, which constitutes $\mathsf{aux}_{\mathsf{V}}$. (Jumping ahead, VS will later receive from EO the decrypted codes associated to the voter's choice. EO will get these codes from the talliers and know the choices on each counted ballot, but not the link to the voter identities. In contrary, VS knows the voter identities linked to $H(\mathsf{tk}_{\mathsf{id}})$ and will get the codes for verification, but not the choices.)

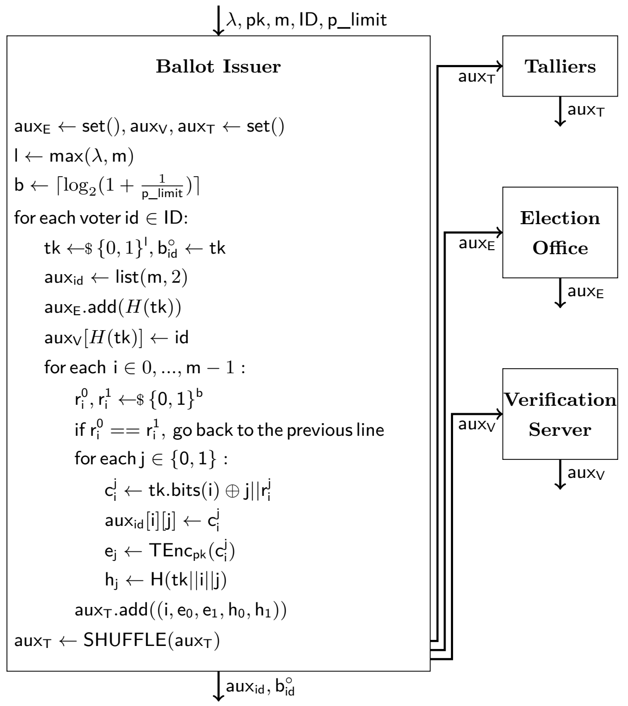
5.2 Voting Procedure (Figure 4)

In the voting procedure, a voter first obtains a ballot by authenticating and proving their identity to BI. They then run the $\mathsf{Vote}(\mathsf{v}, \mathsf{b}^{\circ}, \mathsf{aux})$ algorithm, which takes as input the voting intention v, a blank ballot b° and some auxiliary information aux and a voting intention v. In our approval voting case, v is a list of m bits. This algorithm outputs a completed ballot $\mathsf{b}^{\bullet}$ and a receipt rcpt. To cast a vote, the voter simply sends $\mathsf{b}^{\bullet}$ to the collecting box.

The Vote algorithm is described in Figure 4 in the box of the voter. Essentially, it proceeds as follows. b° is first parsed as a voting token tk and auxiliary information aux which consists in the codes $\{\mathsf{c}^{\mathsf{j}}_{\mathsf{i}}\}^{\mathsf{j} \in \{0,1\}}_{\mathsf{i} \in [\mathsf{m}]}$ for


1^λ, n, t
Talliers
$\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n \leftarrow\$ \Pi_{\mathsf{TGen}}(1^\lambda, \mathsf{n}, \mathsf{t})$
pk
Ballot Issuer
pk
$\mathsf{pk}, \{\mathsf{sk}_i\}_{i=1}^{n}$

(a) Key generation



```
$\mathsf{aux_E} \leftarrow \mathsf{set}(), \mathsf{aux_V}, \mathsf{aux_T} \leftarrow \mathsf{set}()$
$\mathsf{l} \leftarrow \mathsf{max}(\lambda, \mathsf{m})$
$\mathsf{b} \leftarrow \lceil \log_2(1 + \frac{1}{\mathsf{p\_limit}}) \rceil$
for each voter id $\in$ ID:
    $\mathsf{tk} \leftarrow\$ \{0,1\}^{\mathsf{l}}, \mathsf{b}^{\circ}_{\mathsf{id}} \leftarrow \mathsf{tk}$
    $\mathsf{aux_{id}} \leftarrow \mathsf{list}(\mathsf{m}, 2)$
    $\mathsf{aux_E}.\mathsf{add}(H(\mathsf{tk}))$
    $\mathsf{aux_V}[H(\mathsf{tk})] \leftarrow \mathsf{id}$
    for each $\mathsf{i} \in 0, ..., \mathsf{m} - 1$ :
        $\mathsf{r}_\mathsf{i}^0, \mathsf{r}_\mathsf{i}^1 \leftarrow\$ \{0,1\}^{\mathsf{b}}$
        if $\mathsf{r}_\mathsf{i}^0 == \mathsf{r}_\mathsf{i}^1$, go back to the previous line
        for each $\mathsf{j} \in \{0,1\}$ :
            $\mathsf{c}_\mathsf{i}^\mathsf{j} \leftarrow \mathsf{tk.bits(i)} \oplus \mathsf{j}||\mathsf{r}_\mathsf{i}^\mathsf{j}$
            $\mathsf{aux_{id}}[\mathsf{i}][\mathsf{j}] \leftarrow \mathsf{c}_\mathsf{i}^\mathsf{j}$
            $\mathsf{e_j} \leftarrow \mathsf{TEnc_{pk}}(\mathsf{c}_\mathsf{i}^\mathsf{j})$
            $\mathsf{h_j} \leftarrow \mathsf{H}(\mathsf{tk}||\mathsf{i}||\mathsf{j})$
        $\mathsf{aux_T}.\mathsf{add}((\mathsf{i}, \mathsf{e}_0, \mathsf{e}_1, \mathsf{h}_0, \mathsf{h}_1))$
$\mathsf{aux_T} \leftarrow \mathsf{SHUFFLE}(\mathsf{aux_T})$
```

(b) Ballot generation

Fig. 3: Key Generation and Ballot Generation procedures.

each choice i and each choice j. Given those codes, the voter verifies their validity. That is, the voter should check that the first bit of $\mathsf{c}_\mathsf{i}^\mathsf{j}$ is $\mathsf{tk}[\mathsf{i}] \oplus \mathsf{j}$ and that for the same candidate, the trailing bits of c_i^0 and c_i^1 are different strings. If this verification fails, the voter does not have any guarantee of verifiability and should report to the election office. It also check tk upon $H(\mathsf{tk})$.

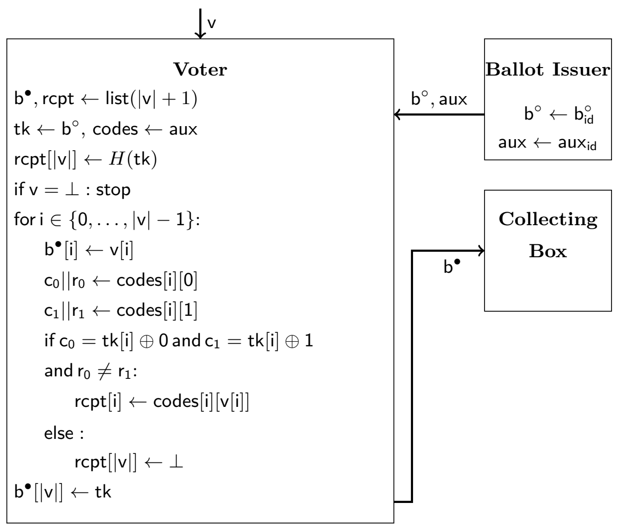
$\mathsf{b}^\bullet$ is then computed as the concatenation of the voting token tk and the list of choices v where $\mathsf{v_i}$ is equal to 1 if the voter approves candidate/question i and 0 otherwise. The voter also keeps a receipt rcpt consisting of $\mathsf{H}(\mathsf{tk})$ (pre-computed on the selection sheet and the note sheet as depicted in Figure 1, but this can be verified) and the codes $[\mathsf{c}_0^{\mathsf{v}_0}, \ldots, \mathsf{c}_{\mathsf{m}-1}^{\mathsf{v}_{\mathsf{m}-1}}]$ of the selected choices (that can be written on the note sheet). The voter then destroys the unused codes (by destroying the code sheet illustrated in Figure 1) and erases any local records of tk. $\mathsf{b}^\bullet$ is sent to CB through the postal channel (without any additional information about the sender).

5.3 Tallying procedure (Figure 5)

After the ballot collecting deadline, CB shuffles the received envelopes, opens them and transfers their content to EO. Now that all the involved parties have their inputs, the actual tally protocol can start.

The election office EO and the talliers $\mathsf{T}_1, \ldots, \mathsf{T_n}$ engage in the protocol $\mathsf{Tally}(\mathsf{ballots}, \mathsf{aux_E}, \mathsf{aux_T}, \mathsf{pk}, \{\mathsf{sk}_i\}_{i=1}^{t}, \mathsf{m}, \mathsf{t})$. EO takes as inputs the sets of ballots $\mathsf{ballots}$ received from CB, the election key pk, the auxiliary information $\mathsf{aux_E}$ received from the ballot issuer BI, the number of candidates or questions m and the threshold number of talliers t. Each tallier $\mathsf{T_j}$ takes as input the auxiliary information $\mathsf{aux_T}$ received from BI, the election public key pk and their own private key $\mathsf{sk_j}$. At the end of the protocol, EO outputs the result r of the election as well as the individual proofs of verifiability Π that contain the verified decrypted codes. $\mathsf{T_j}$ outputs the result $\mathsf{r_j}$ of the election.

The full tally procedure is described in Figure 5. The boxes of the tallier $\mathsf{T_j}$ and EO, and the interactions between them consist of the Tally protocol that we comment in the next paragraphs. The results of the election $\mathsf{r}, \mathsf{r}_1, \ldots, \mathsf{r_n}$ respectively output by $\mathsf{EO}, \mathsf{T}_1, \ldots, \mathsf{T_n}$ are sent to EA through their own secure channels. EA verifies



```
$b^\bullet, rcpt \leftarrow list(|v|+1)$
$tk \leftarrow b^\circ$, $codes \leftarrow aux$
$rcpt[|v|] \leftarrow H(tk)$
if $v = \perp$ : stop
for $i \in \{0, \dots, |v|-1\}$:
    $b^\bullet[i] \leftarrow v[i]$
    $c_0||r_0 \leftarrow codes[i][0]$
    $c_1||r_1 \leftarrow codes[i][1]$
    if $c_0 = tk[i] \oplus 0$ and $c_1 = tk[i] \oplus 1$
    and $r_0 \neq r_1$:
        $rcpt[i] \leftarrow codes[i][v[i]]$
    else :
        $rcpt[|v|] \leftarrow \perp$
$b^\bullet[|v|] \leftarrow tk$
```

Fig. 4: Voting procedure.

whether at least t of these r_j are consistent with r, and, if it is case, outputs r, and $\perp$ otherwise. Finally, EO sends the individual proofs of verifiability Π to VS that will be useful for the verification procedure.

Now, we turn back to the Tally protocol and provide additional explanation about how it works. First, EO computes the result of the election from all the ballots $b^\bullet$ in ballots. If the token of a ballot appears on another ballot or if the hash of the (printed) token in $b^\bullet$ does not correspond to any entry of aux_E, then the ballot is discarded. Note that this can be done electronically by scanning the ballots.

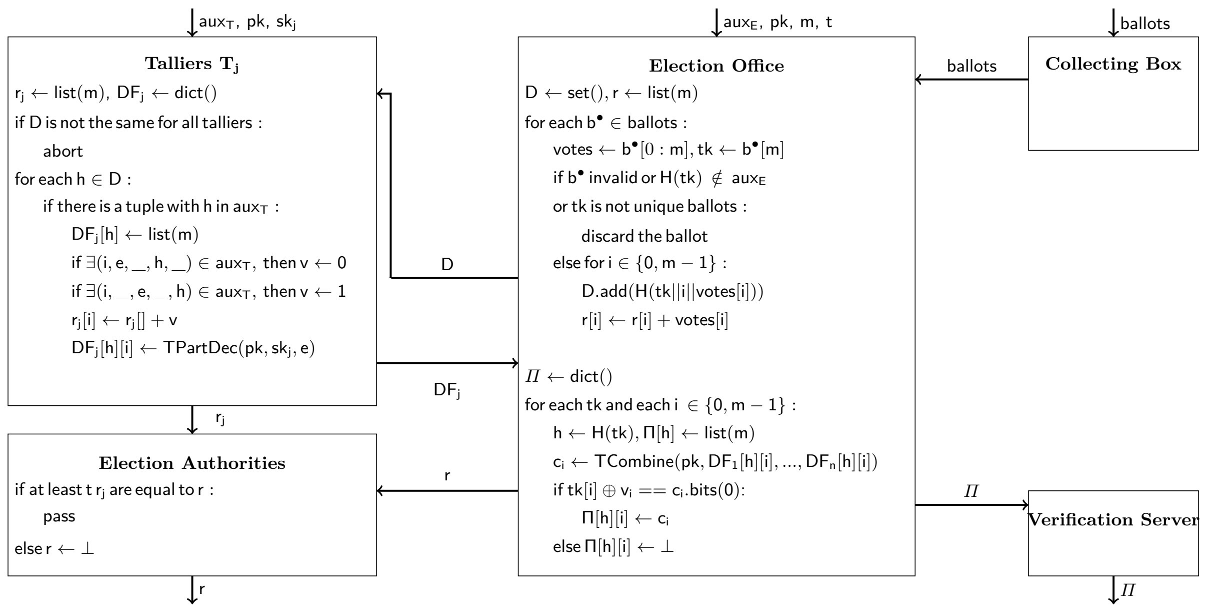
Afterward, EO interacts with the talliers to produce the proofs for the voters that their vote has been correctly counted. To do so, EO computes for each (non-discarded) ballot m values $\{H(tk||i||v_i)\}_1^m$ from the voting token tk, each choice i, and the vote $v_i \in \{0,1\}$ for this choice written on this ballot. All the elements of all these lists of values are then shuffled and sent to the talliers. As discussed before, this shuffling helps to hide the voting patterns to the talliers. Before computing the result, the talliers verify with each other that they have received the same list from the election office. Otherwise, they abort the election, and each T_j outputs $r_j = \perp$.

For each of those $h = H(tk||i||v_i)$, each tallier checks that they received a tuple $(i, e^0, e^1, h^0, h^1))$ in aux_T such that $h = h^{v_i}$. If this is not the case, the tuple is discarded. Otherwise, the tallier can recover the vote (i, v_i) associated to this value and add it to its own tally. Moreover, the tallier also computes the decryption factor of each ciphertext e in the tuple using its partial private key sk_j. T_j sends all these decryption factors DF_j to EO. EO can recover the code associated to each $H(tk||i||v_i)$ entry (since DF_j keeps the same order as the list D of shuffled tuples sent by EO and) by running TCombine (using the first t valid decryption factors it receives for example), but without ever knowing whether this code is the right one (because BI could have encrypted bad codes, for instance). Nevertheless, we assume that at least t lists of decryption factors have been computed honestly (and are detectable thanks to the validity decryption proofs they contain).

Given the list of codes computed by TCombine, EO can re-order them and gather the selection code c_i^j according to the structure of each ballot. Then, using the corresponding voting token tk, EO checks that the first bit of the codes is correctly computed as $tk[i] \oplus j$. If this is not the case, EO will not forward these codes to VS and the voter will ultimately report an error if he runs the verification procedure. The main intuition behind this first-bit check is to avoid a code-swapping attack. That is, a corrupt BI could have swapped the encrypted codes sent to the talliers. Then, if the vote is swapped to the other value before the tally, the voter would not see anything since the two swaps cancel each other.

5.4 Verification procedure (Figure 6)

After the elections, voters can verify that their vote has been counted correctly. To do so, they run the VerifyVote protocol, which is run with the verification server VS. The voter takes as input her receipt rcpt


$\mathsf{aux_T}$, pk, $\mathsf{sk_j}$
$\mathsf{aux_E}$, pk, m, t
ballots
ballots
D
$\mathsf{DF_j}$
$\mathsf{r_j}$
r
Π
r
Π

```
Talliers $T_j$
$r_j \leftarrow list(m)$, $DF_j \leftarrow dict()$
if D is not the same for all talliers :
    abort
for each $h \in D$ :
    if there is a tuple with h in $aux_T$ :
        $DF_j[h] \leftarrow list(m)$
        if $\exists(i, e, \_, h, \_) \in aux_T$, then $v \leftarrow 0$
        if $\exists(i, \_, e, \_, h) \in aux_T$, then $v \leftarrow 1$
        $r_j[i] \leftarrow r_j[] + v$
        $DF_j[h][i] \leftarrow TPartDec(pk, sk_j, e)$
```

```
Election Office
$D \leftarrow set(), r \leftarrow list(m)$
for each $b^\bullet \in ballots$ :
    $votes \leftarrow b^\bullet[0 : m], tk \leftarrow b^\bullet[m]$
    if $b^\bullet$ invalid or $H(tk) \notin aux_E$
    or tk is not unique ballots :
        discard the ballot
    else for $i \in \{0, m-1\}$ :
        $D.add(H(tk||i||votes[i]))$
        $r[i] \leftarrow r[i] + votes[i]$

$\Pi \leftarrow dict()$
for each tk and each $i \in \{0, m-1\}$ :
    $h \leftarrow H(tk), \Pi[h] \leftarrow list(m)$
    $c_i \leftarrow TCombine(pk, DF_1[h][i], ..., DF_n[h][i])$
    if $tk[i] \oplus v_i == c_i.bits(0)$:
        $\Pi[h][i] \leftarrow c_i$
    else $\Pi[h][i] \leftarrow \perp$
```

```
Collecting Box
```

```
Election Authorities
if at least t $r_j$ are equal to r :
    pass
else $r \leftarrow \perp$
```

```
Verification Server
```

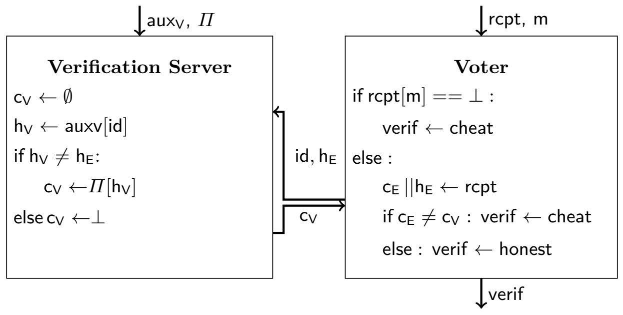
Fig. 5: Tallying procedure. The t decryption factors used in the TCombine algorithms are the first t decryption factors that have been received.

and the number of candidates m while the verification server takes as input the auxiliary information $\mathsf{aux_V}$ received from the ballot issuer and the proofs Π. At the end of the protocol, the voter outputs the string "cheat" if the verification fails or "honest" otherwise.

The protocol is described in Figure 6. The voter authenticates to VS as id and sends the hash of its voting token that was stored on rcpt. VS then verifies that this hash correspond to the hash received from BI in $\mathsf{aux_V}$ for this id. In that case, it sends to the voter the tuple of the codes received from EO for that H(tk). Otherwise, it sends an error message verification_fails to the voter.

This check ensures that the voter received the right voting token from the ballot issuer. Indeed, $\mathsf{aux_V}$ contains the link between the voters identities and their voting token. This should be consistent with the voter's token received from the ballot issuer.

The voter then checks if the list of codes corresponds to the codes written on rcpt and outputs honest or cheat accordingly. It reports to the election authorities EA if this is not the case.



```
Verification Server
$c_V \leftarrow \emptyset$
$h_V \leftarrow auxv[id]$
if $h_V \neq h_E$:
    $c_V \leftarrow \Pi[h_V]$
else $c_V \leftarrow \perp$
```

```
Voter
if $rcpt[m] == \perp$ :
    $verif \leftarrow cheat$
else :
    $c_E || h_E \leftarrow rcpt$
    if $c_E \neq c_V$ : $verif \leftarrow cheat$
    else : $verif \leftarrow honest$
```

Fig. 6: Verification procedure

6 Security

6.1 Model

We introduce a simulation-based security definition in the simplified universally composable security model (SUC) [12] for the security of postal voting protocols. We assume basic familliary with this model (or even with the UC model) and refer the reader to the former reference for further details.

The SUC model is particularly suitable in our setting: it has been designed for protocols executed by a fixed set of parties, as it is our case, and avoids the numerous technical challenges that come in the UC framework due to the dynamic machine creation process.

As usual in this framework, we define a trusted third party which would give natural security guarantees if it happened to exist. This incorruptible trusted third party implements an ideal functionality (which is described below). This functionality helps the parties to achieve some secure computations while limiting the view of the adversary.

In this model, there is also an external environment $\mathcal{E}$ that controls whatever is external to the protocol. This include the scheduling of the communications between the parties and the handling of the inputs and outputs. In our case, $\mathcal{E}$ chooses the settings of the election (as EA's input) and the vote of the voters (as their input). $\mathcal{E}$ also receives the output of each party, specifically the result of the election and the verification output of the voters. Finally, $\mathcal{E}$ interacts arbitrarily with the adversary during the execution of the protocol.

As a reminder, a protocol is secure in this model if for any adversary $\mathcal{A}$ against the protocol, there exists an adversary $\mathcal{S}$ (also called the simulator) such that the environment $\mathcal{E}$ cannot distinguish between a real execution of the protocol with the adversary $\mathcal{A}$ and from an ideal execution of the protocol through the trusted third party with $\mathcal{S}$.

We assume that all the communication channels between the parties are secure such that the only thing that the adversary learn about the content of a message is its size[8]. Even though communication channels are not secure per se in the SUC model, we can compose our protocol with an instantiation of the Secure Message Transmission functionality proposed in [11]. In practice, we assume that all the communications happen through TLS channels (or equivalent), with properly certified keys.

To model the potential lack of security of the postal channel, we allow the adversary to corrupt the postal endpoint CB. It can thus read, drop and modify any ballot before the tally if the postal channel is corrupted.

Regarding the scheduling of the executions of the different entities and the communication between them, we use the same model as the SUC model and refer the reader to Section 2 of [12].

The execution of the protocol in the real world follows the description given in Section 5. We full describe here the instantiation of the protocol in the SUC model.

1. At the beginning of the execution, all parties and the environment starts with the security parameter 1^λ written on their security parameter tape. $\mathcal{E}$ is then activated and writes an input in EA's tape to start the protocol. If this input cannot be parsed as $(\mathsf{m}, \mathsf{t}, \mathsf{p\_limit})$ such that m is a non-zero integer, t is an integer between 0 and n and $\mathsf{p\_limit}$ is a floating number between 0 and 1, EA aborts the protocol.
   Otherwise, the official parties run the Setup protocol with input $\mathsf{p} = (1^\lambda, \mathsf{m}, \mathsf{n}, \mathsf{t}, \mathsf{ID}, \mathsf{p\_limit})$[9]. More precisely:
   (a) EA forwards p to each participating party.
   (b) The talliers run the KeyGen protocol together and obtain the public key pk and the secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_\mathsf{T}$. pk is sent to BI.
   (c) BI runs the $\mathsf{GenerateBallot}(\lambda, \mathsf{pk}, \mathsf{m}, \mathsf{ID}, \mathsf{p\_limit})$ algorithm and obtains $\{\mathsf{b}^\circ_\mathsf{id}, \mathsf{aux}_\mathsf{id}\}_{\mathsf{id} \in \mathsf{ID}}, \mathsf{aux}_\mathsf{E}, \mathsf{aux}_\mathsf{T}, \mathsf{aux}_\mathsf{V}$. The last three values are respectively sent to EO and VS.
   (d) These parties check the consistency of their auxiliary value. Precisely:
      i. EO aborts if aux_E cannot be parsed as $|\mathsf{ID}|$ distinct outputs of the hash function.
      ii. VS aborts if aux_V cannot be parsed as a mapping of outputs of the hash function into ID.

[8] Since the lengths of the messages in our protocol is bounded by a constant, we assume that the messages are padded so that the adversary does not learn anything.

[9] ID and n are known by EA because every party knows the set of identities of all participating parties.

When an official party has finished its execution of the setup procedure, it sends a confirmation message to EA. EA starts the voting phase when it has received a message from all official parties.

2. At the beginning the voting phase, EA sends to every voter in ID a message $\mathsf{voting\_starts}$. When a voter id receives this message:
   - Either their input tape contains a voting intention $\mathsf{v} \in \{0,1\}^{\mathsf{m}}$ and
     (a) Sends a message $\mathsf{retrieve\_ballot}$ to BI.
     (b) Upon receiving a ballot $\mathsf{b}^{\circ}, \mathsf{aux}$ from BI, runs the vote algorithm $\mathsf{b}^{\bullet}, \mathsf{rcpt} \leftarrow \mathsf{Vote}(\mathsf{v}, \mathsf{b}^{\circ}, \mathsf{aux})$
     (c) Sends $\mathsf{b}^{\bullet}$ to CB and keep rcpt in memory.
   - Either their input tape is empty or does not encode a valid vote and the voter does nothing.

   Similarly to the ideal execution, the adversary chooses when the voting phase finishes by sending a $\mathsf{stop\_vote}$ command to CB.[10]
3. When the voting phase is finished, CB shuffles the ballots it received during the voting phase and sends them to EO.
   Together with the talliers and EA, EO runs the $\mathsf{Tally}(\mathsf{ballots}, \mathsf{aux}_{\mathsf{E}}, \mathsf{aux}_{\mathsf{T}}, \mathsf{pk}, \{\mathsf{sk}_{\mathsf{i}}\}_{\mathsf{i}=1}^{\mathsf{n}})$ procedure. At the end of the algorithm, EA outputs the result r and sends it to every voter, which also outputs it. If the result has been voided (that is, $\mathsf{r} = \perp$), then every party aborts the protocol. Otherwise, EO outputs the proof of individual verifiability Π, which is sent to VS.
4. At the beginning of the verification phase, EA sends to every voter in ID a message $\mathsf{verification\_starts}$. When a voter id receives this message, they runs with VS the $\mathsf{VerifyVote}(\mathsf{rcpt}, \mathsf{aux}_{\mathsf{V}}, \Pi)$. The voter then outputs to $\mathcal{E}$ the result of the protocol (either cheat or honest).
5. Finally, the execution of the protocol ends when $\mathcal{E}$ outputs a bit.

6.2 Ideal Functionality for postal voting

In the definition of the ideal execution of the protocol, honest voters and the election authorities are replaced by so-called dummy voters and dummy EA. When a dummy voter id receives a $\mathsf{voting\_starts}(\mathsf{sid})$ message from the functionality, they send their input v to the functionality or they do nothing if they have not received any input from the environment. Also, when they receive the result of the election or the result of the verification procedure, they output this result to the environment (if the voter receives a message $\mathsf{nothing\_received}$ during the verification procedure, they either output honest if they have not send anything or cheat otherwise).

Likewise, when the dummy EA is first activated, it sends a command $\mathsf{Setup}(\mathsf{sid}, \mathsf{p})$ with the elections parameters $\mathsf{p} = (1^{\lambda}, \mathsf{m}, \mathsf{n}, \mathsf{t}, \mathsf{ID}, \mathsf{p\_limit})$ to the functionality, where $(\mathsf{m}, \mathsf{t}, \mathsf{p\_limit})$ is the content of its input tape. It also returns to the environment the message abort if it receives this message from the functionality. After sending a message abort or after the last verification message, the functionality stops its execution.

The adversary $\mathcal{S}$ (which we also call the simulator in the ideal execution) interacts with the functionality and simulates the talliers, BI, CB, EO and VS. Given an adversary $\mathcal{A}$ in the real execution, $\mathcal{S}$ tries to simulate $\mathcal{A}$ behavior such that $\mathcal{E}$ cannot distinguish them.

For ease of readability, the functionality has been broken into several chunks corresponding to the phases of the protocol.

Postal Voting functionality - Setup phase

- Upon receiving a command $\mathsf{setup}(\mathsf{sid}, \mathsf{p})$ from EA:
  - Parse p as $(1^{\lambda}, \mathsf{m}, \mathsf{n}, \mathsf{t}, \mathsf{ID}, \mathsf{p\_limit})$.
  - Send the command $\mathsf{setup}(\mathsf{sid}, \mathsf{p})$ to every voter and the simulator.
  - Generate for each voter id in ID a random distinct string handle in $\{0,1\}^{\star}$ and record $(\mathsf{sid}, \mathsf{id}, \mathsf{handle})$.
  - For each each voter id in ID, set $\mathsf{cast}[\mathsf{id}] \leftarrow []$ and $\mathsf{counted}[\mathsf{id}] \leftarrow []$.
  - Send to the simulator the records $(\mathsf{sid}, \mathsf{id}, \mathsf{handle})$ for each $\mathsf{id} \in \mathcal{C}$.

[10] In the SUC model, the adversary cannot directly communicate with the honest parties. We can solve this technical detail by introducing a new party (that we call the election scheduler) which is always corrupted and sends a message to CB when the voting phase finishes.

    - If the postal channel is corrupted, send all the records $(\mathsf{sid}, \mathsf{id}, \mathsf{handle})$ for each $\mathsf{id} \in \mathsf{ID}$.
- Upon receiving a command $\mathsf{abort}(\mathsf{sid})$, send abort to EA.
- Upon receiving a command $\mathsf{continue}(\mathsf{sid})$ from the simulator, enter the Voting phase.

Every voter has an handle hiding their identity. The functionality will later know the vote of each handle but the mapping between the handles and the voters' id will remain secret, hence preserving the privacy (except for the corrupted voters). The variables $\mathsf{cast}[\mathsf{id}]$ and $\mathsf{counted}[\mathsf{id}]$ respectively represent the voter's ballots that are cast by the voter and counted in the tallying procedure. These might be different because of the adversarial behavior.

Secure Postal Voting functionality - Voting phase

- When entering the Voting phase, send a $\mathsf{voting\_starts}(\mathsf{sid})$ command to every voter.
- Upon receiving a $\mathsf{vote}(\mathsf{sid}, \mathsf{id}, \mathsf{v})$ command from the voter with identity id:
    - If $\mathsf{v} \in \{0,1\}^{\mathsf{m}}$, append v to $\mathsf{cast}[\mathsf{id}]$ and $\mathsf{counted}[\mathsf{id}]$.
    - If the postal channel is corrupted, forward the command to the simulator.
- Upon receiving a $\mathsf{vote}(\mathsf{sid}, \mathsf{id}, \mathsf{v})$ command from the simulator, append v to $\mathsf{counted}[\mathsf{id}]$ if $\mathsf{v} \in \{0,1\}^{\mathsf{m}}$.
- Upon receiving a command $\mathsf{continue}(\mathsf{sid})$ from the simulator, enter the Tallying phase.

In the real world, CB stops collecting the ballots at a given date and time. Even though time is not modeled here, it is equivalent to give to the adversary the power to stop the voting phase since it controls all the asynchronicity of the ballots.

Secure Postal Voting functionality - Tallying phase

- When entering the $\mathsf{tallying}$ phase:
    - Generate a list votes in the following way. For each $\mathsf{id} \in \mathsf{ID}$:
        * If $|\mathsf{counted}[\mathsf{id}]| = 1$, append $(\mathsf{handle}, \mathsf{counted}[\mathsf{id}])$ to votes.
        * If $|\mathsf{counted}[\mathsf{id}]| \neq 1$, append $\perp$ to votes.
    - Shuffle the list votes.
    - Send to the simulator a command $\mathsf{tally}(\mathsf{sid}, \mathsf{votes})$.
- Upon receiving a command $\mathsf{modify\_ballot}(\mathsf{sid}, \mathsf{handle}, \mathsf{new\_v})$ from the simulator:
    - If there exists some recorded $(\mathsf{sid}, \mathsf{id}, \mathsf{handle})$ update $\mathsf{counted}[\mathsf{id}]$ into $\mathsf{new\_v}$.
- Upon receiving a command $\mathsf{abort}(\mathsf{sid})$ from the simulator, send a message abort to EA.
- Upon receiving a command $\mathsf{continue}(\mathsf{sid})$ from the simulator:
    - Compute the result r of the election as follows:
        1. Set $\mathsf{r} = \mathsf{list}(\mathsf{m})$
        2. For each $\mathsf{id} \in \mathsf{ID}$ and $i \in \{0, \mathsf{m}\}$: if $|\mathsf{counted}[\mathsf{id}]| = 1$, set $\mathsf{r}[\mathsf{i}]$ to $\mathsf{r}[\mathsf{i}] + \mathsf{v}[\mathsf{i}]$ where v is the unique vote in $\mathsf{counted}[\mathsf{id}]$.
    - Send to the dummy EA and to every voter the result of the election.
    - Enter the Verification phase.

In the tallying phase, the ideal functionality computes the result of the election as the election office would do in our protocol. Even though the adversary learns the values of the ballots, it does not know to which voter each ballot is assigned, except if the postal channel is non-confidential. If several ballots are recorded for one voter, the simulator does not learn any of them. This presents attacks in which the simulators sends a distinct number of ballots for each voter in order to learn their voting intention.

Secure Postal Voting functionality - Verification phase

- When entering the verification phase, wait for receiving a command $\mathsf{identification\_type}(\mathsf{sid}, \mathsf{type})$ from the simulator.
- Then, upon receiving a command $\mathsf{verify}(\mathsf{sid}, \mathsf{voter}, \mathsf{verification\_type})$ from the simulator, set
    - id to voter if $\mathsf{identification\_type} = \mathsf{identification\_by\_id}$
    - id to the record corresponding to the handle voter
      if $\mathsf{identification\_type} = \mathsf{identification\_by\_handle}$

and send to the id one of the following answer:

- If verification_type = ballot_based, send:
  - nothing_received if cast[id] = counted[id] = []
  - cheat if cast[id] $\neq$ counted[id]
  - honest otherwise
- If verification_type = conditional_cheat(fail_vote), where fail_vote is a set of pair (candidate, choice), send:
  - cheat if for some $(\mathsf{i}, \mathsf{j}) \in$ fail_vote, $(\mathsf{i}, \mathsf{j}) \in$ counted[id] as well
  - follow the ballot_counted command otherwise
- If verification_type = verification_fail, send cheat.
- If verification_type = conditional_success(p_success), ignore the command if p_success > p_limit. Otherwise, send honest with probability p_success or cheat with probability $1 -$ p_success

In the verification phase, the adversary can choose the behavior of the ideal functionality for each voter. It can set the voter to return cheat (verification_fails request) or to output honest if and only if the ballot cast by the voter has been correctly counted (ballot_based request), which is the desired behaviour.

There is also a conditional_cheat request that makes the voter output cheat if their vote is in some given set. This is needed, because a malicious ballot issuer could encrypt different codes than the ones given to the voter. Depending on her choices, the voter would see or not the wrong codes which do not correspond to her auxiliary data.

Finally, the functionality can also set the voter to output honest with a probability at most p_limit, where this p_limit is a parameter of the election (with the verification_success request). This essentially means that a malicious adversary can only modify the vote of a voter without being detected with a controlled probability. Hence if a voter outputs honest, he knows that with probability at least $1-$p_limit, his vote has been correctly counted. However the converse is not true: if the voters outputs cheat, there is no information about whether their vote has been correctly counted or not.

This tolerance of ‘true-negative’ attacks is actually needed because of the nature of the return code protocol. Indeed, an adversary could always cheat and try to guess the correct code of a voter. Indeed when the guess is correct, the voter cannot detect the vote manipulation. However for a small p_limit, the adversary is much more likely to get caught if he tries to modify many votes. In our protocol, p_limit can be controlled by decreasing or increasing the length b of the codes.

6.3 Secure Postal Voting

From this description of the ideal world, we can finally state our security definition that our protocol realizes.

Definition 3 (Secure postal voting). *A Postal Voting Protocol $\mathcal{V}$ is secure if for any PPT adversary $\mathcal{A}$, there exists an ideal PPT adversary $\mathcal{S}$ such that for any PPT environment $\mathcal{E}$, the probability that the environment distinguishes between the execution of the real protocol and an interaction with the ideal postal voting functionality is negligible. In this game, at most one of these sets of official parties is corrupted: {EO and less than t tallier}, {BI} and {VS}. Any number of voters and the postal channel can be corrupted.*

Theorem 1. *The postal voting protocol π described in Section 5 is secure in the sense of Definition 3.*

To prove this result, we first start by reminding some technicalities of the SUC model. We then give the different simulators that realizes the theorem, for each scenario of corruption, and finally the proofs of indistinguishability between the ideal and the real worlds.

The participating parties are the parties from Section 2.2. Each of them is modeled as Turing machine and has a unique id. We consider a single protocol session, and define a single session identifier *sid* that contains the list of id’s of all the protocol participants[11]. Some of these parties are corrupted following a

[11] In the SUC model, the set of parties is known to all.

static setting. That is the set of corrupted parties is fixed by the adversary before the beginning of the protocol and is known by everyone.

Parties communicate together and with the functionality through a router, which is controlled by the adversary. When a party or the functionality wants to send a message to another party, the message is stored in the router and the adversary is notified about the identity of the sender and the receiver of the message. Then, the adversary can notify the router to deliver the message at any time and the message is written on some communication tape of the receiver. To fully show indistinguishability, the simulator needs to simulate this communication router as well.

The adversary has access to the communication tapes of the corrupted parties and has thus access every message they receive to the adversary. The adversary can also write on their communication tape in order to send messages on their behalves. As usual, we consider a dummy adversary controlled by the environment. The dummy adversary executes every message sent by the environment and forwards its view each time it is activated.

To prove the theorem, we have to show that $\mathcal{E}$'s view in the real world is indistinguishable from $\mathcal{E}$'s view in the ideal world. We respectively write $\mathsf{view}_{\mathcal{R}}$ and $\mathsf{view}_{\mathcal{I}}$ $\mathcal{E}$'s view in the real or ideal world.

More precisely, the view of the environment consists in the following: the communications of the honest parties ($\mathsf{view}_{\mathsf{COM}}$), the view of the corrupted voters ($\mathsf{view}_{\mathsf{V}}$), the input and output of EA ($\mathsf{io}_{\mathsf{EA}}$) and of each of the honest voters (io_{V}). We will detail how each part of this view is computed identically in both worlds.

Simulators

Generic case: $\mathcal{S}$. Regardless of which official party is corrupted, we first build a generic simulator $\mathcal{S}$. That is, we assume first that only voters may be corrupted and thus that BI, EO, CB, VS and the talliers are honest. This simulator will later be modified depending on which party is corrupted. We build the simulator $\mathcal{S}$ in the following way:

1. $\mathcal{S}$ invokes and simulates $\mathcal{A}, \mathsf{BI}, \mathsf{EO}, \mathsf{CB}, \mathsf{VS}$ and each of the talliers. $\mathcal{S}$ also creates a simulated router interacting with $\mathcal{A}$ that will be used to handle the communications between the participating and the simulated parties. $\mathcal{S}$ notifies $\mathcal{A}$ each time a party wants to communicate with another and delivers the message according to $\mathcal{A}$'s instructions. Also, $\mathcal{S}$ forwards messages between $\mathcal{E}$ and $\mathcal{A}$.
2. In the setup phase, $\mathcal{S}$ receives from the functionality the parameters of the election p and the handles corresponding to the corrupted voters. It simulates an execution of the Setup protocol and sends a command $\mathsf{continue}$ to the functionality.
3. During the voting phase, $\mathcal{S}$ simulates the view of the corrupted voters:
   - When a malicious voter authenticates to the simulated BI, $\mathcal{S}$ sends to $\mathcal{A}$ the ballot data $(\mathsf{b}^{\circ}_{\mathsf{id}}, \mathsf{aux}_{\mathsf{id}})$ of the corresponding malicious voter.
   - When $\mathcal{A}$ delivers a voting instruction to CB from a malicious voter that has the form of a valid voting ballot $\mathsf{b}^{\bullet} = \mathsf{v}|\mathsf{tk}_{\mathsf{id}}$ with a voting token $\mathsf{tk}_{\mathsf{id}}$ associated to an honest voter, then $\mathcal{S}$ sends the command $\mathsf{vote}(\mathsf{id}, \mathsf{v})$ to the functionality.
   - When the adversary sends a $\mathsf{continue}$ command to CB, $\mathcal{S}$ forwards the message to the functionality as well.
4. At the beginning of the tallying phase, $\mathcal{S}$ receives from the functionality the list of pairs of handles and votes. It randomly assigns to each honest handle the id of a different voter (abstentionist voters are associated to an handle without vote).
   Then it computes for each vote v of each handle the ballot $\mathsf{b}^{\bullet} = \mathsf{Vote}(\mathsf{b}^{\circ}_{\mathsf{id}}, \mathsf{v}, \mathsf{aux}_{\mathsf{id}})$ where $\mathsf{b}^{\circ}_{\mathsf{id}}$ and $\mathsf{aux}_{\mathsf{id}}$ are the values generated for the id assigned to the given handle. Using this list of ballots, $\mathcal{S}$ simulates an execution of the Tally protocol and obtain the result of the election r and the proofs of individual verification Π. At the end of the Tally protocol, $\mathcal{S}$ sends a $\mathsf{continue}$ command.
5. Finally in the verification phase, $\mathcal{S}$ sends to the functionality an $\mathsf{identification\_type}(\mathsf{sid}, \mathsf{identification\_by\_id})$ command and simulates the messages of the $\mathsf{VerifyVote}$ algorithm for every voter.
   - When the last message of the $\mathsf{VerifyVote}$ interaction is delivered to a voter id, $\mathcal{S}$ sends a command $\mathsf{verify}(\mathsf{sid}, \mathsf{id}, \mathsf{ballot\_based})$ to the functionality.

  - When interacting with a malicious voter, $\mathcal{S}$ executes the $\mathsf{VerifyVote}$ proto-col with $\mathsf{aux_V}$ and Π as input.

Corrupted postal channel corrupted case: $\mathcal{S}^{\mathsf{PC}}$. If the postal channel is corrupted, the corrupted CB is expected to receive the ballots in the same order as they are delivered. This include the well-formed ballots, but also the malformed ballots sent by the adversary. So in step 3 of the generic simulator, we add the following bullet point:

– When $\mathcal{A}$ delivers a vote command of the simulated router from a voter to CB, $\mathcal{S}^{\mathsf{PC}}$ delivers the corresponding command to the functionality and receives the content of the command $\mathsf{vote}(\mathsf{sid}, \mathsf{id}, \mathsf{v})$. Given this vote v and the public voting token $\mathsf{tk_{id}}$ of this voter, $\mathcal{S}^{\mathsf{PC}}$ then computes the corresponding ballot with $\mathsf{Vote}(\mathsf{v}, \mathsf{tk_{id}}, \mathsf{aux_{id}})$ and forwards it to the corrupted CB on behalf of id.

In step 4, $\mathcal{S}^{\mathsf{PC}}$ receives the mapping between the handles and the ids. So, it uses it in step 4 instead of the random mapping of the generic simulator. Also, in step 4, when receiving a message with the ballots from the corrupted CB:

– Instead of using the votes outputed by the simulators, $\mathcal{S}^{\mathsf{PC}}$ uses this list as an input of the Tally protocol.
– For each ballot $\mathsf{b}^\bullet$ on this list, $\mathcal{S}^{\mathsf{PC}}$ checks if the ballot has a valid voting token $\mathsf{tk_{id}}$ associated to a voter id. In that case, it recovers the vote and adds it to a list $\mathsf{counted}^{\mathcal{S}}[\mathsf{id}]$. Otherwise, it drops the ballot.
– $\mathcal{S}^{\mathsf{PC}}$ makes the functionality's $\mathsf{counted}^{\mathcal{S}}[\mathsf{id}]$ consistent with the corrupted CB's $\mathsf{counted}[\mathsf{id}]$ list with appropriate $\mathsf{modify\_ballot}$ commands.

All the other messages from CB are ignored.

Corrupted verification server case: $\mathcal{S}^{\mathsf{VS}}$. If VS is corrupted, $\mathcal{A}$'s view is augmented with the values $\mathsf{aux_V}$ in the setup phase and the verification data Π at the end of the tallying procedure. Thus, $\mathcal{S}^{\mathsf{VS}}$ sends these values (on behalf of BI and EO) respectively in step 2 during the $\mathsf{GenerateBallot}$ execution and in step 4 at the end of the Tally execution.

In the verification phase, the corrupted VS also interacts with the honest voters through an execution of the $\mathsf{VerifyVote}$ algorithm. So in step 5, we modify $\mathcal{S}$ as such:

– When simulating the $\mathsf{VerifyVote}$ protocol of honest voters, send $H(\mathsf{tk_{id}})$ to the corrupted VS on behalf of id.
– When $\mathcal{A}$ delivers a message m from VS to an honest voter id:
  - If m is equal to the codes $\Pi[\mathsf{H}(\mathsf{tk_{id}})]$ computed by the Tally algorithm, send a $\mathsf{verify}(\mathsf{sid}, \mathsf{id}, \mathsf{ballot\_based})$ command to the functionality (as $\mathcal{S}$ would do).
  - Otherwise, if the postal channel is honest, use $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.

  In the other cases, $\mathcal{S}^{VS}$ knows the $\mathsf{cast}[\mathsf{id}]$ and $\mathsf{counted}[\mathsf{id}]$ values associated to id. Let c_i^{true} and $c_i^{\mathcal{A}}$ be respectively the i-th code of $\Pi[\mathsf{H}(\mathsf{tk_{id}})]$ and m.
  - If for some candidate i the choice is the same in $\mathsf{cast}[\mathsf{id}]$ and $\mathsf{counted}[\mathsf{id}]$ but $c_i^{true} \neq c_i^{\mathcal{A}}$, then $\mathcal{S}^{\mathsf{VS}}$ uses $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.
  - Otherwise, if for some candidate i the choice is different in $\mathsf{cast}[\mathsf{id}]$ and $\mathsf{counted}[\mathsf{id}]$ but $c_i^{true} = c_i^{\mathcal{A}}$, then $\mathcal{S}^{\mathsf{VS}}$ uses $\mathsf{verification\_fails}$ as the $\mathsf{verifica-\ tion\_type}$.
  - Otherwise, if $c_i^{true} \neq c_i^{\mathcal{A}}$ but both codes share either the same first bit or the same trailing bits , then $\mathcal{S}^{\mathsf{VS}}$ uses $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.
  - Otherwise, $\mathcal{S}^{\mathsf{VS}}$ uses $\mathsf{verification\_success}(\mathsf{p}^{\mathsf{k}})$ as the $\mathsf{verification\_type}$ where k is the number of different entries in $\mathsf{cast}[\mathsf{id}]$ and $\mathsf{counted}[\mathsf{id}]$ and p is equal to $\frac{1}{2^{\mathsf{b}}-1}$.

Finally if at any time $\mathcal{A}$ aborts the election on behalf of VS, $\mathcal{S}^{\mathsf{VS}}$ sends an abort message to the functionality as well. The other messages are ignored.

Corrupted election office and less than t corrupted talliers case: $\mathcal{S}^{\mathsf{EO}}$. If EO and less than t talliers are corrupted by the adversary, then $\mathcal{A}$'s view is augmented with the values $\mathsf{aux_E}$ and $\mathsf{aux_T}$ in the setup phase. Thus, $\mathcal{S}^{\mathsf{EO}}$ sends these values (on behalf of BI) in step 2 during the $\mathsf{GenerateBallot}$ execution.

At the end of the voting phase, the corrupted EO expects to receive from CB a shuffled list of ballots (if the postal channel is not corrupted). Similarly as before, $\mathcal{S}^{\mathsf{EO}}$ uses the voting token $\mathsf{tk_{id}}$ to craft ballots of the votes list received from the functionality. Additionally, $\mathcal{S}^{\mathsf{EO}}$ includes each malformed ballots sent by the adversary that were dropped in step 2, shuffles the list and sends it to EO.

In the tallying phase, the corrupted EO interacts with the honest talliers and sends them a list of values D to be tallied. When such a list has been sent to each honest talliers, they run a consensus protocol (with the corrupted talliers) as well to check that this value is the same for all of them. When the consensus algorithm terminates for one tallier, $\mathcal{S}^{\mathsf{EO}}$ executes the tallier's Tally algorithm on D and receives as output a decryption factor that is sent to EO and a result $\mathsf{r_j}$ that is kept in memory.

Also, for each value $\mathsf{h_{tk,i,v_i}} \in D$, the simulator computes the unique $(\mathsf{tk}, \mathsf{i}, \mathsf{v_i})$ such that $\mathsf{H}(\mathsf{tk}||\mathsf{i}||\mathsf{v_i}) = \mathsf{h_{tk,i,v_i}}$. If this tuple is not unique, $\mathcal{S}$ aborts. From this vote v and the handle associated to the voting token tk, $\mathcal{S}$ sends a $\mathsf{modify\_ballot}$ command to the functionality to ensure that for this id, $\mathsf{counted}[\mathsf{handle}] = \mathsf{v_i}$.

To conclude the tallying phase, if the malicious and the honest parties does not agree on the election result, $\mathcal{S}^{\mathsf{EO}}$ aborts the election. Otherwise, $\mathcal{S}^{\mathsf{EO}}$ sends the $\mathsf{continue}$ command to the functionality.

Finally, when the corrupted EO sends the codes Π^{EO} to VS, $\mathcal{S}$ launches the verification procedure with an $\mathsf{identification\_type}(\mathsf{sid}, \mathsf{identification\_by\_handle})$ command. Similarly to the corrupted VS case, $\mathcal{S}^{\mathsf{EO}}$ sends different $\mathsf{verify}(\mathsf{sid}, \mathsf{id}, \mathsf{verification\_type})$ requests depending on the values of the codes Π^{EO}.

First, for a given handle and its token tk, we define v^{true} either as the vote in votes associated to handle if the postal channel is not corrupted or as the vote cast by the voter associated to handle which can be tracked otherwise. Then, we define c_i^{true} as the code associated to i, v^{true} and tk in the ballot generation mechanism. Similarly, we define $\mathsf{v}^{\mathcal{A}}$ as the vote associated to tk in the list D sent to the honest talliers (or $\perp$ otherwise) and $c_i^{\mathcal{A}}$ as the i-th value of $\Pi^{\mathsf{EO}}[\mathsf{H}(\mathsf{tk})]$.

With these values in mind, $\mathcal{S}^{\mathsf{EO}}$ proceeds as follows in step 5:

- If $\mathsf{v}^{true} = \mathsf{v}^{\mathcal{A}}$ and for every candidate i, $c_i^{true} = c_i^{\mathcal{A}}$, then send a $\mathsf{verify}(\mathsf{sid}, \mathsf{handle}, \mathsf{ballot\_based})$ command to the functionality.
- If for some candidate i the choice is the same in v^{true} and $\mathsf{v}^{\mathcal{A}}$ but the corrupted $c_i^{true} \neq c_i^{\mathcal{A}}$, use $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.
- Otherwise if for some candidate the choice is different in v^{true} and $\mathsf{v}^{\mathcal{A}}$ but $c_i^{true} = c_i^{\mathcal{A}}$, use $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.
- Otherwise, if $c_i^{true} \neq c_i^{\mathcal{A}}$ but both codes share either the same first bit or the same trailing bits , then $\mathcal{S}^{\mathsf{VS}}$ uses $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$. $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.
- Otherwise, use $\mathsf{verification\_success}(\mathsf{p^k})$ as the $\mathsf{verification\_type}$ where k is the number of different entries in $\mathsf{votes}[\mathsf{handle}]$ and $\mathsf{counted}[\mathsf{handle}]$ and p is equal to $\frac{1}{2^{\mathsf{b}}-1}$ if $\mathsf{v}^{\mathcal{A}} \neq \perp$ or $\frac{1}{2^{\mathsf{b}}}$ otherwise.

Finally if at any time $\mathcal{A}$ aborts the election on behalf of EO or a corrupted tallier, $\mathcal{S}^{\mathsf{EO}}$ sends an abort message to the functionality as well. The other messages are ignored.

Corrupted ballot issuer case: $\mathcal{S}^{\mathsf{BI}}$. If BI is corrupted, the simulator does not compute the ballots and the auxiliary data anymore. Instead, it receives from $\mathcal{A}$ the sets $\mathsf{aux_T}, \mathsf{aux_E}$ and $\mathsf{aux_V}$ in the setup phase.

In the voting phase, $\mathcal{S}^{\mathsf{BI}}$ simulates for each honest voter id they request to retrieve their ballot to BI. In return, $\mathcal{S}^{\mathsf{BI}}$ receives from $\mathcal{A}$ an empty ballot $\mathsf{b}^{\circ}_{\mathsf{id}}$ and the auxiliary data $\mathsf{aux_{id}}$. Then:

In the tally phase, some ballots might have to be dropped because the votin token is not in $\mathsf{aux_E}$. More precisely, for every id such that $H(\mathsf{b}^{\circ}_{\mathsf{id}}) \notin \mathsf{aux_E}$, $\mathcal{S}^{\mathsf{BI}}$ sends a command $\mathsf{modify\_ballot}(\mathsf{sid}, \mathsf{id}, \perp)$.

Finally in the verification phase, BI might have computed the data sent to the honest voters and the data sent to the other parties maliciously. So for every voter id:

- If $(\mathsf{b}^{\circ}, \mathsf{aux})$ cannot be parsed as tk and 2 codes for each candidate i such that the first bit of each code is set by $\mathsf{tk}[\mathsf{i}]$, then $\mathcal{S}^{\mathsf{BI}}$ uses $\mathsf{verification\_fails}$ as the $\mathsf{verification\_type}$.
- Otherwise, if $(\mathsf{id}, H(\mathsf{b}^{\circ}_{\mathsf{id}})) \notin \mathsf{aux_V}$, then $\mathcal{S}^{\mathsf{BI}}$ uses $\mathsf{verification\_fails}$ as the $\mathsf{verifica\text{-} tion\_type}$.
- Otherwise, if for candidate i and choice j we have that the code $\mathsf{c_i^j}$ in $\mathsf{aux_{id}}$ is different from the decryption of the code corresponding to $\mathsf{H}(\mathsf{tk}||\mathsf{i}||\mathsf{j})$ in $\mathsf{aux_T}$, then $\mathcal{S}^{\mathsf{BI}}$ adds (i, j) to a set $\mathsf{fail\_vote}$. It then uses a $\mathsf{conditional\_cheat}(\mathsf{fail\_vote})$ as the $\mathsf{verification\_type}$.

– Otherwies, $\mathcal{S}$ uses $\mathsf{ballot\_based}$ as the $\mathsf{verification\_type}$ as in the generic simula-tor.

Again, all the other messages from BI are ignored.

Indistinguishability

Generic case:

Let F be the event that for two different tuples $(\mathsf{id}, \mathsf{i}, \mathsf{j})$ and $(\mathsf{id}', \mathsf{i}', \mathsf{j}')$ we have $\mathsf{H}(\mathsf{tk}_{\mathsf{id}}||\mathsf{i}||\mathsf{j}) = \mathsf{H}(\mathsf{tk}_{\mathsf{id}'}||\mathsf{i}'||\mathsf{j}')$ or $\mathsf{H}(\mathsf{tk}_{\mathsf{id}}) = \mathsf{H}(\mathsf{tk}_{\mathsf{id}'})$ (in the real world or in the simulated protocol). We will show that 1) F happens with negligible probability 2) If F does not happen, then the view of the environment is identical in both worlds.
Proof that F happens with negligible probability: With overwhelming probability, all the voting tokens are different since they are uniformly sampled from $\{0,1\}^{\lambda}$. Also, as H is modeled as a random oracle, there is a collision on the random values of the function with negligible probability.
Proof that $\mathsf{view}_{\mathcal{R}} = \mathsf{view}_{\mathcal{I}}$ if F does not happen: Regarding the $\mathsf{view}_{\mathsf{COM}}$ part, $\mathcal{S}$ perfectly simulates the execution flow of the real protocol. For the $\mathsf{view}_{\mathsf{V}}$ part, the ballot data $(\mathsf{b}^{\circ}, \mathsf{aux})$ of each malicious voters are computed identically as in the real world by the simulator. Also, the ballots cast and counted for each malicious voters are correctly tracked by the functionality. If the voter sent only one vote, the tally algorithm will correctly recover the corresponding codes. Otherwise, no code will be retrieved, as in the real world. This discussion regarding $\mathsf{view}_{\mathsf{COM}}$ and $\mathsf{view}_{\mathsf{V}}$ will also hold for the other simulators when more parties are corrupted, thus this will not be discussed anymore.

We then look at the $\mathsf{io}_{\mathsf{EA}}$ part of $\mathcal{E}$'s view, especially EA's output. In both the real world and the ideal world, there is no abort since the official parties are all honest. Also, the functionality correctly records the set of ballots in the $\mathsf{counted}$ variable and the result of the election is computed similarly because the tallying procedure followed by the functionality is the same as what EO is doing in the real world (discarding ballots if the voting token appears more than once). Moreover, as every tallier is honest, they will agree with EO on the result they send to EA.

Let us finish with the output of the voters io_{V}. In the real world, an honest voter returns honest if and only if:

– On the ballot they receive, the first bit of each code is consistent with the voting token.
– On the verification procedure, they send the value h that the verification server has received from BI in $\mathsf{aux}_{\mathsf{V}}$.
– On the verification procedure, they receive from VS the codes associated to their selection in aux.

As all the official parties are honest, the first two bullet points are always verified. For the last point, the codes received from VS corresponds to the decrypted codes of $\mathsf{aux}_{\mathsf{T}}$ for the choices counted by EO. These decrypted codes are the same as the codes sent by BI in the aux data. This holds because again, all the official parties are honest.

Hence an honest voter returns honest if and only if the vote they sent is the same as the vote counted by EO, which is also the case in the ideal world because the functionality consistently tracks the ballots cast and counted for every voter. Thus the io_{V} part of $\mathcal{E}$'s view is identical again in both worlds.

We now see how we need to modify the simulator if more parties are corrupted.

Postal channel corrupted

The event F happens with the same probability because it is independent of the view and actions of the postal channel. Note that this will be true for any corrupted party except the ballot issuer.
Proof that $\mathsf{view}_{\mathcal{R}} = \mathsf{view}_{\mathcal{I}}$ if F does not happen:

$\mathcal{E}$'s view is now augmented with the view of the corrupted CB, which is perfectly simulated thanks to the functionality's forwarding of all the vote commands.

Moreover, the result of the election is consistent in both world because of the $\mathsf{modify\_ballot}$ commands. If the adversary tampers with the list of ballots output by the corrupted CB, then the simulator updates the

$\mathsf{counted}$ variable of the voter accordingly. Thus the $\mathsf{io_{EA}}$ and $\mathsf{io_V}$ view will be the same in both worlds.

VS corrupted

First, the result of the election is computed similarly and independently of the action of VS. If the corrupted VS aborts the election in the real world, the simulator also abort the election in the ideal world. Hence $\mathsf{io_{EA}}$ is also the same in both worlds.

Then, $\mathcal{E}$'s is now augmented with the view of the corrupted VS $\mathsf{view_{VS}}$, which consist in $(\mathsf{aux_V}, \Pi, \{\mathsf{h_{E,id}}\}_{\mathsf{id}\in\mathcal{V}})$.

More precisely, $\mathcal{E}$'s view for each honest voter id is $(\mathsf{id}, \mathsf{v_{id}}, \mathsf{H}(\mathsf{tk_{id}}), \mathsf{codes}_{\mathsf{id}}^{\mathsf{received}}, \mathsf{codes}_{\mathsf{id}}^{\mathsf{sent}}, \mathsf{h_{E,id}}, \mathsf{out_{id}})$ where $\mathsf{codes}_{\mathsf{id}}^{\mathsf{received}}$ and $\mathsf{codes}_{\mathsf{id}}^{\mathsf{sent}}$ are respectively the codes received from EO and sent to the voters and $\mathsf{out_{id}}$ is the status (cheat or honest) of id (from $\mathsf{io_V}$). If the postal channel is corrupted, this view also includes $\mathsf{b}_{\mathsf{id}}^{\bullet}$.

In both words, $\mathsf{id}, \mathsf{v_{id}}$ and $\mathsf{codes}^{\mathsf{sent}}$ are identical as they are inputed by the environment. Also, the values $\mathsf{H}(\mathsf{tk_{id}})$ and $\mathsf{h_{E,id}}$ are identically distributed in both worlds since they come from the honest parties BI and voter id. If the postal channel is corrupted $\mathsf{b}_{\mathsf{id}}^{\bullet}$ is also identical in both world, as the simulator knows the mapping between ids and handles.

It remains to look at $\mathsf{codes}_{\mathsf{id}}^{\mathsf{sent}}$ and the output of the honest voters $\mathsf{out_{id}}$.

There are two case:

- *The postal channel is honest*:
  Let Q_{RO} be the set of values queried to the random oracle by the adversary in the voting phase and F_{RO} the event that for at least one voting token tk associated to some honest voter, $\mathsf{tk} \in Q_{RO}$.
  We argue that this event happens with negligible probability. If there are n honest voters and c corrupted voters, the first query hits a voting token with probability at most $\frac{\mathsf{n}}{2^{\mathsf{l}}-\mathsf{c}}$. Similarly, if the first query does not hit a voting token, the second does with probability bounded by $\frac{\mathsf{n}}{2^{\mathsf{l}}-\mathsf{c}-1}$ and so on. Hence, the probability that F_{RO} happens is bounded by $\frac{\mathsf{n}|\mathsf{Q_{RO}}|}{2^{\mathsf{l}}-\mathsf{c}-|\mathsf{Q_{RO}}|}$, which is negligible. As a consequence, the event F_{tk} that at some point in the voting phase $\mathcal{A}$ sends a ballot $\mathsf{b}^{\bullet}$ with the voting token tk of an honest voter happens with negligible probability.
  If F_{tk} do not happen, we have for every honest voter that $\mathsf{counted}[\mathsf{id}] = \mathsf{cast}[\mathsf{id}]$ and EO decrypts with the talliers the codes $\mathsf{codes}^{\mathsf{received}}$ expected by the voter in a successful verification procedure. If, the corrupted VS sends $\mathsf{codes}^{\mathsf{sent}} \neq \mathsf{codes}^{\mathsf{received}}$ to an honest voter, the voter will then output $\mathsf{out_{id}} = \mathsf{cheat}$ in both worlds. Otherwise, the voter will output $\mathsf{out_{id}} = \mathsf{honest}$.
- *The postal channel is corrupted*: In that case, $\mathsf{codes}^{\mathsf{received}}$ is computed identi-cally in both world since the map between $\mathsf{handles}$ and ids is known. Also, the new simulator $\mathcal{S}^{\mathsf{VS}}$ correctly tracks if the corrupted VS sends a code that will result in a cheat output by the voter in the real world. On the other way, it is also possible that the corrupted VS correctly guesses all the right codes of the voter's choices for a modified ballot. Since $\mathcal{A}$ knows the voting token of each voter through the corrupted postal channels, it knows as well the first bit of the code. Moreover, it knows that the trailing bits of the code are different from the trailing bits of the code of the other choice received from EO, hence it has a probability equal to $\frac{1}{2^{\mathsf{b}}-1}$ to guess it correctly.
  Finally, VS guesses all the right codes with probability $\mathsf{p}^{\mathsf{k}} \leq \mathsf{p\_limit}$, where k is the number of codes to guess. From the description of the $\mathsf{verification\_success}$ command, the dummy voter's output is indistinguishable from the same voter's output in the real world.

EO and talliers corrupted

The view of the adversary is perfectly simulated, as the $\mathsf{aux_E}$ values, the shuffling of the ballots and the decryption factors of the honest tallier are computed identically in both worlds. Again, the result of the election is identical in both world, since the $\mathsf{modify\_ballot}$ requests make sure that the $\mathsf{counted}$ values are consistent with the count of the honest tallier. It remains to look at $\mathsf{io_V}$.

Let us look first at voters who did not vote. Arguably, the probability that EO sends a value $\mathsf{h_{tk,i,v_i}}$ to the honest talliers for a token associated to such a voter is negligible, using the same argument as the F_{RO} event of the corrupted VS case. Hence, the honest tallier will not count a vote for this voter and the adversary will not receive any code for this voter. If the corrupted EO sends codes for this voter to the verification server, they will output cheat in both world.

For the other voters, we argue that for a given honest voter and a given choice for a candidate, the election office can guess the code with probability negligibly close to $p = \frac{1}{2^b-1}$ if the adversary send the corresponding value $h_{\mathsf{tk},i,\mathsf{v_i}}$ to the talliers or $\frac{1}{2^b}$ otherwise.

We proceed by a sequence of indistinguishable games (in both the real and ideal worlds) from the security of the threshold encryption scheme.

In the first game, we use the simulator TSim of the threshold encryption scheme to compute the decryption factor of the honest party. So instead of computing $\mathsf{TPartDec}(\mathsf{sk_t}, \mathsf{c})$ for each encryption c of the i-th code, we use $\mathsf{TSim}(\mathsf{pk}, \mathcal{H}, \mathsf{c}, \mathsf{m})$ where m is equal to $\mathsf{c_i^{v_i}}$. This transition is indistinguishable from the decryption simulatability property.

In the second game, we replace during the ballot generation the encryption of $\mathsf{c_i^0}$ by a random value. This transition is indistinguishable from the CPA security of the scheme. In the third game, we proceed similarly and replace the encryption of $\mathsf{c_i^1}$ by a random value.

As the code $\mathsf{c_i^{1-v_i}}$ is now independent from the adversary view (up to the fact that the first bit and the leading bits of the code are different from the code of the other choice), it has a probability $\frac{1}{2^b-1}$ or $\frac{1}{2^b}$ to guess the right code like in the corrupted VS case.

BI corrupted

Compared to the generic simulator, the adversary does not learn anything from the view of the ballot issuer.

Regarding $\mathsf{io_{EA}}$, the simulator still consistently tracks valid ballots and discards ballots which would be discarded as well in the real protocol. Moreover, an honest tallier aborts the election if it receives a value inconsistent with the value $\mathsf{aux_T}$ received from the corrupted BI both in the real and ideal world.

For $\mathsf{io_V}$, either the honest voter outputs cheat if the codes are not consistent with tk or if the code vote of their vote is different from the encrypted value in $\mathsf{aux_T}$, or else the output is the same as in the generic simulator.

Acknowledgment

Henri Devillez is a research fellow of the Belgian FRIA. Thomas Peters is research associate of the Belgian Fund for Scientific Research (F.R.S.-FNRS). This work has been supported in part by a Microsoft Research Award.

References

1. Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a university president using open-audit voting: Analysis of real-world use of helios. In *2009 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '09*. USENIX Association, 2009.
2. Philip B Stark Andrew W Appel, Richard A DeMillo. Ballot-marking devices cannot ensure the will of the voters. *Election Law Journal: Rules, Politics, and Policy*, 19(3):432–450, 2020.
3. Peter Rønne Anggrio Sutopo, Thomas Haines. On the auditability of the estonian ivxv system and an attack on individual verifiability. In *Workshop on Advances in Secure Electronic Voting*, 2023.
4. Josh Benaloh. Strobe-voting: Send two, receive one ballot encoding. In *E-Vote-ID 2021*, volume 12900 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2021.
5. Josh Benaloh and Michael Naehrig. Electionguard design specification version 2.0.0. `https://www.electionguard.vote/spec/`, August 2023.
6. Josh Benaloh, Ronald L. Rivest, Peter Y. A. Ryan, Philip B. Stark, Vanessa Teague, and Poorvi L. Vora. End-to-end verifiability. *CoRR*, abs/1504.03778, 2015.
7. Josh Benaloh, Peter Y. A. Ryan, and Vanessa Teague. Verifiable postal voting. In *Security Protocols XXI - 21st International Workshop*, volume 8263 of *Lecture Notes in Computer Science*, pages 54–65. Springer, 2013.

8. David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. Sok: A comprehensive analysis of game-based ballot privacy definitions. In *2015 IEEE Symposium on Security and Privacy*, pages 499–516. IEEE, 2015.
9. David Bernhard, Véronique Cortier, Olivier Pereira, and Bogdan Warinschi. Measuring vote privacy, revisited. In *the ACM Conference on Computer and Communications Security, CCS'12*, pages 941–952. ACM, 2012.
10. Matthew Bernhard, Allison McDonald, Henry Meng, Jensen Hwa, Nakul Bajaj, Kevin Chang, and J. Alex Halderman. Can voters detect malicious manipulation of ballot marking devices? In *2020 IEEE Symposium on Security and Privacy, SP 2020*, pages 679–694. IEEE, 2020.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.
12. Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.
13. David Chaum. Surevote: Technical overview. In *Proceedings of the Workshop on Trustworthy Elections, WOTE 2001*, 2001.
14. 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, SP 2016*, pages 779–798. IEEE Computer Society, 2016.
15. Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. Belenios: A simple private and verifiable electronic voting system. In *Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows*, volume 11565 of *Lecture Notes in Computer Science*, pages 214–238. Springer, 2019.
16. Véronique Cortier, Joseph Lallemand, and Bogdan Warinschi. Fifty shades of ballot privacy: Privacy against a malicious board. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020*, pages 17–32. IEEE, 2020.
17. Braden L. Crimmins, Marshall Rhea, and J. Alex Halderman. Remotevote and SAFE vote: Towards usable end-to-end verification for vote-by-mail. In *FC 2022 International Workshops*, volume 13412 of *LNCS*, pages 391–406. Springer, 2022.
18. Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC 2001*, pages 119–136. Springer, 2001.
19. Alexandre Debant and Lucca Hirschi. Reversing, breaking, and fixing the french legislative election e-voting protocol. In *32nd USENIX Security Symposium, USENIX Security 2023*, pages 6737–6752. USENIX Association, 2023.
20. Clara Faulí, Katherine Stewart, Federica Porcu, Jirka Taylor, Alexandra Theben, Ben Baruch, Frans Folkvord, Fook Nederveen, Axelle Devaux, and Francisco Lupiáñez-Villanueva. Study on the benefits and drawbacks of remote voting. `https://commission.europa.eu/strategy-and-policy/policies/justice-and-fundamental-rights/eu-citizenship/democracy-and-electoral-rights/studies/study-benefits-and-drawbacks-remote-voting_en`, 2018.
21. Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. Sharing decryption in the context of voting or lotteries. In *FC 2000*, pages 90–104. Springer, 2001.
22. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, 2007.
23. Kristian Gjøsteen. The norwegian internet voting protocol. In *VoteID 2011, Tallinn, Estonia, September 28-30, 2011, Revised Selected Papers*, volume 7187 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.
24. Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy, SP 2020*, pages 644–660, 2020.
25. Christian Killer and Burkhard Stiller. The swiss postal voting process and its system and security analysis. In *E-Vote-ID 2019*, volume 11759 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2019.
26. Eleanor McMurtry, Xavier Boyen, Chris Culnane, Kristian Gjøsteen, Thomas Haines, and Vanessa Teague. Towards verifiable remote voting with paper assurance. *CoRR*, abs/2111.04210, 2021.
27. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology - CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.
28. Olivier Pereira. Individual verifiability and revoting in the estonian internet voting system. In *FC 2022 International Workshops*, volume 13412 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2022.
29. Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *J. Cryptol.*, 15(2):75–96, 2002.
30. Swiss Post. The Swiss Post e-voting system. `https://gitlab.com/swisspost-evoting/`, 2023.
31. Filip Zagórski, Richard Carback, David Chaum, Jeremy Clark, Aleksander Essex, and Poorvi L. Vora. Remotegrity: Design and use of an end-to-end verifiable remote voting system. In *ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 441–457. Springer, 2013.