E-cclesia: Universally Composable Self-Tallying Elections

Anonymous Author(s)

ABSTRACT

The technological advancements of the digital era paved the way for the facilitation of electronic voting (e-voting) in the promise of efficiency and enhanced security. In standard e-voting designs, the tally process is assigned to a committee of designated entities called *talliers.* Naturally, the security analysis of any e-voting system with tallier designation hinges on the assumption that a subset of the talliers follows the execution guidelines and does not attempt to breach privacy. As an alternative approach, Kiayias and Yung [PKC '02] pioneered the *self-tallying elections* (STE) paradigm, where the post-ballot-casting (tally) phase can be performed by any interested party, removing the need for tallier designation.

In this work, we explore the prospect of decentralized e-voting where security is preserved under concurrent protocol executions. In particular, we provide the first comprehensive formalization of STE in the *universal composability* (UC) framework introduced by Canetti [FOCS '01] via an ideal functionality that captures required security properties such as voter privacy, eligibility, fairness, one-voter one-vote, and verifiability. We provide a concrete instantiation, called E-cclesia , that UC realizes our functionality. The design of E-cclesia integrates several cryptographic primitives such as signatures of knowledge for anonymous eligibility check, dynamic accumulators for scalability, time-lock encryption for fairness and *anonymous broadcast channels* for voter privacy. For the latter primitive, we provide the first UC formalization along with a construction based on mix-nets that utilises layered encryption, threshold secret sharing and equivocation techniques.

Finally, we discuss deployment and scalability considerations for E-cclesia . We present preliminary benchmarks of the key operations (in terms of computational cost) of the voting client and demonstrate the feasibility of our proposal with readily available cryptographic tools for mid-sized elections (∼100,000 voters).

CCS CONCEPTS

• **Security and privacy → Cryptography**; **Privacy-preserving protocols**; *Distributed systems security.*

KEYWORDS

e-voting, anonymous broadcast, universally composable security

1 INTRODUCTION

In democratic societies, a wide spectrum of people with respect to their beliefs and social status can participate equally in shaping decisions that affect governance both at a national or smaller (e.g., unions, corporations, academic institutes, associations) scale. The means for achieving this is voting, which is essential so that governance can operate in a transparent and fair way. With the technological advancements of the digital era, *electronic voting* (e-voting) has been introduced, promising better efficiency, enhanced security and superior transparency than traditional voting.

Arguably, one critical aspect of e-voting design is to determine the level of centralisation desired (or feasible), given that conflicts naturally arise between scalability on the one hand, and security and privacy on the other. In principle, election tasks such as setup, registration, vote collection, tally, and result announcement, can be carried out in one of the three following manners in terms of decentralization: (i) completely centralized by a single authority, (ii) by a committee of designated entities, or (iii) fully decentralized such that the voters themselves are responsible for performing the task. Depending on the election setting, decentralization may be a requirement for some task, but considered impractical for another one. For instance, distributing trust for voter privacy during tallying is often highly recommended, yet one cannot expect a direct consensus among voters on a large scale election setup.

In this work, we explore the prospect of decentralized tallying. Since taking the centralized approach often results in privacy and robustness attacks [7, 67] on the authorities that constitute single points of failure, most state-of-the-art e-voting systems include a committee of designated parties called *talliers* in charge of tallying the result of the election (e.g., [2, 18, 20, 22, 23, 25, 26, 39, 44, 60]). In what we call *tallier designation* e-voting, security hinges on the assumption that a subset of the talliers follows the execution guidelines and does not attempt to breach privacy. Indeed, in any e-voting system of this type, privacy is trivially violated if all the talliers collude in order to jointly retrieve the voters' preferences (typically, by combining their partial decryption keys). Moreover, in the case where the voters post their votes to a publicly accessible *bulletin board* (e.g., [2, 20, 25, 26, 39, 44]), then partial results can be leaked during the ballot casting period (*fairness* violation) under full tallier collusion. Hence, while distributing trust among talliers strengthens the system w.r.t. privacy and robustness, the introduction of assumptions regarding the tallier corruption threshold to argue about security cannot be considered ideal. In fact, real world examples indicate that designated tallying authorities can be the weak link in the system's overall performance and security, either due to benign errors or by becoming high priority targets of attackers (e.g., the cases of the tallying machines in Georgia, USA [50], and Estonia [4]). Towards overcoming the limitations present in tallier designation e-voting, we explore the potential of an alternative approach expressed by the *self-tallying elections* (STE) paradigm [43]. Namely, an e-voting system is self-tallying if the post-ballot-casting (tally) phase can be performed by any interested party. Designing STE systems that satisfy a list of standard e-voting security properties such as eligibility, fairness, voter privacy, and verifiability raises a number of challenges. Specifically, the main challenges of STE are (i) guaranteeing that no voter (or coalition of voters) can boycott the election; (ii) no intermediate results are being leaked during the ballot casting phase (fairness); (iii) no vote can be linked back to the voter that cast it (voter privacy). Unfortunately, the existing STE proposals [30, 34, 36–38, 42, 43, 48, 49, 51, 54, 64, 66, 69] lack formal treatment and/or suffer from limitations such as being susceptible to abort, in the sense that there is a moment in the execution where the participation of all active voters is required for tally to take place, or requiring a trusted party to be involved during voting so that fairness can be achieved.

Contributions: We tackle all the aforementioned design challenges by deploying a formal framework where security is preserved under concurrent executions. In particular, we present E-cclesia , an STE protocol that constitutes a fine-grained integration of fundamental and special cryptographic tools (e.g., signatures of knowledge, non-interactive commitments, dynamic accumulators, time-lock encryption, anonymous broadcast), and is provably secure in Canetti's *universal composability model* (UC) [13] (cf. Appendix A).

We summarize in more details our contributions below.

1. UC formalization of STE and modular design. We formalize the concept of STE through the ideal functionality $\mathcal{F}_{\mathsf{STE}}$. Our functionality captures correctness and standard e-voting properties such as eligibility, fairness, voter privacy, one-voter one-vote, and verifiability. Specifically, we only allow eligible voters to vote no more than once. Regarding privacy and fairness, we only leak to the simulator the length of the message. In the casting phase, the voters' identity remains hidden. The ballots are opened only after the end of the casting phase and thus we guarantee fairness. Finally, during the tally phase, anyone can retrieve and/or verify the election result which the adversary cannot alter or drop, thus correctness is satisfied.

We break down $\mathcal{F}_{\mathsf{STE}}$ into two smaller modules named $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$. The functionality $\mathcal{F}_{\mathsf{elig}}$ is responsible for the *eligibility* part of $\mathcal{F}_{\mathsf{STE}}$ (e.g., credential generation and ballot authentication), while $\mathcal{F}_{\mathsf{vm}}$ is responsible for the *vote management* part of $\mathcal{F}_{\mathsf{STE}}$ (e.g., ballot generation, casting, and opening). Our modular approach facilitates easier future updates without the need for reproving the security of the whole STE protocol.

2. UC realization of $\mathcal{F}_{\mathsf{STE}}$: the E-cclesia protocol. We present E-cclesia , a self-tallying protocol that UC realizes $\mathcal{F}_{\mathsf{STE}}$. In its design, E-cclesia combines several cryptographic primitives such as time-lock encryption (TLE) to guarantee fairness, signatures of knowledge (SoK) and anonymous broadcast channels to guarantee eligibility, privacy, and the one-voter one-vote property, and dynamic accumulators for efficiency. E-cclesia relies on random oracles [53], common reference string [15], broadcast channels [31], and a global clock [5].

3. UC formalization and realization of anonymous broadcast. We provide the first UC treatment of the anonymous broadcast notion by introducing the ideal functionality $\mathcal{F}_{\mathsf{an.BC}}$ and a protocol based on mix-nets [18] that UC-realizes $\mathcal{F}_{\mathsf{an.BC}}$ in the presence of (plain) broadcast channels and a programmable random oracle. In our protocol, we split the messages into shares [62] and each share is layer encrypted and sent to a row of stratified mix network [29]. In order to achieve UC realization, we borrow techniques from non-committing encryption [53] for correct opening of the messages. In addition, we apply cover traffic to hide the senders' activity. The shares are randomly reordered by each layer of the mix servers and after some delay, they are broadcast to all parties, thus preventing timing attacks [40]. Although $\mathcal{F}_{\mathsf{an.BC}}$ and the protocol that UC realises it fit the purposes of E-cclesia , it is a novelty beyond the concept of STE and we believe it is of independent interest.

4. Benchmarking of (components of) E-cclesia . We evaluate the feasibility and scalability of E-cclesia . We propose a practical deployment strategy that introduces a computationally powerful (not trusted) server for the tallying of votes (i.e. TLE decryptions), alleviating the need for intensive computations from resource-constrained voting clients. Voters are only required to perform computationally less demanding verification of tallied ballots. As our preliminary benchmarks suggest, this verification as well as ballot generation are computationally reasonable and within reach. Specifically we instantiate the TLE scheme by Pietrzak's VDF [56] combined with AES encryption according to Rivest *et al.*'s scheme [58], and for the SoKs we use Groth's NIZK proof system [35], and we benchmark the computationally critical voters' operation: (a) the SoK signing operations performed by the voters for generating their eligible encrypted ballot, and (b) the verification of correct TLE decryptions and of SoK signatures performed by the voters to verify the final tally of the election. Our initial benchmarks indicate that with readily available state-of-the-art libraries these can be made practical and scalable at least up to mid-sized elections (~100,000 voters).

2 RELATED WORK

For a recap of the UC framework, cf. Appendix A.

Tallier designation e-voting. E-voting research spans over four decades [2, 18, 20–26, 39, 44, 60]. E-voting design faces the challenge of capturing (a reasonable subset of) security properties (e.g., eligibility, verifiability, fairness, voter privacy, receipt-freeness, coercion resistance) that may be conflicting. In the standard design approach, the execution of election processes such as setup, registration, vote collection, tally, and result announcement is assigned to designated entities. As already mentioned, under a full tallier collusion setting, tallier designation e-voting systems [2, 18, 20, 22, 23, 25, 26, 39, 44, 60] cannot preserve voter privacy, and not even fairness when vote collection is carried out via posting to a bulletin board [2, 20, 25, 26, 39, 44]. We stress that fairness is always violated if the talliers additionally collude with the vote collection authorities (e.g., ballot box) to retrieve the votes prior to the tally phase. On the contrary, E-cclesia satisfies fairness *unconditionally* w.r.t. corruption setting, i.e., it relies only on the security of the underlying cryptographic primitives (TLE).

Self-tallying voting. The STE notion was introduced in [43], and fairness was already pointed out as one of the challenges; the last voter can learn the (partial) election outcome before choosing their vote which may lead to the following issues: the last voter 1) *adapts* her vote according to the partial results, or 2) *aborts*, where an aborting voter prevents the other voters from performing tally. The construction in [43] addresses Issue 1 by considering a trusted party that casts a final "dummy vote", and Issue 2 by including an additional "recovery" round, yet in that round all remaining voters must participate. Subsequent works based on the ideas of [43] have the same limitations [30, 34, 64, 69]. In [42] and [51] (the latter presents an implementation of [37]), commitments are deployed to confront the adaptivity of the last voter. Moreover, any construction that relies on a recovery round [36, 42, 48] is susceptible to abort. Alternatively, enforcing financial incentives to achieve the necessary participation of all voters has been proposed in [51].

Regarding security modeling, we observe that in the literature, there is lack of a formal framework for the desired STE properties [30, 34, 36–38, 42, 43, 48, 49, 51, 54, 66, 69] (only ballot secrecy is formalized in [42, 48]). A more formal approach can be found in [64]. Specifically, the authors define an ideal functionality for e-voting that captures several properties such as correctness, eligibility, and privacy, and a separate definition for universal verifiability. The modeling in [64] has limitations, as the said functionality (a) allows a single voter to cause the whole election to abort; (b) does not capture timing attacks for all tally functions (e.g., individual handling of votes); (c) lacks detailed formal description (e.g., description of token handling in the UC framework); (d) considers the list of eligible voters as a fixed parameter, rather than an input to the execution that varies per session.

The use of TLE for constructing STE has been suggested in [48]. In [49], self-tallying voting is proposed as an application of homomorphic time-lock puzzles, without further security analysis.

Anonymous broadcast channels. The concept of anonymous broadcast was first studied in the context of DC-nets [19, 33, 65] that offer unconditional security but typically have limitations such as message drop due to collisions, vulnerability to jamming attacks, and/or quadratic complexity for broadcasting a single bit. Several anonymous broadcast protocols have been proposed, yet their analysis is under security models that do not support composition. The protocol in [34] is based on ideas of the STE construction in the same work. In [68], the authors build their protocol on top of the secure multiparty computation in [27]. In [47], the authors propose an anonymous broadcast implementation based on DC-nets. Moreover, the security analysis of the construction in [52] is inspired by the ideal/real world paradigm; however, the ideal functionality in [52] is not compatible with the UC setting (there is no environment that provides the parties with inputs over time).

3 PROTOCOL EXECUTION

In this section, we provide a concise description of the E-cclesia self-tallying elections (STE) protocol. First, we present the cryptographic building blocks that our protocol utilizes, the parties involved in an execution, and the intuitive security properties that it achieves. Throughout the paper, we use λ as security parameter.

3.1 Cryptographic building blocks

The E-cclesia protocol design encompasses a delicate integration of a set of cryptographic primitives. Below, we outline these building blocks' operations and refer the reader to the full description of the corresponding ideal functionalities.

– We assume the existence of a *global clock* that synchronizes all entities involved in the execution (cf. Figure 5 and [5] for the functionality $\mathcal{G}_{\mathsf{clock}}$). The time Cl increases when all entities are ready to advance in time and can be read by anyone upon request.

– A *random oracle* (RO) (cf. Figure 6 and [53] for the functionality $\mathcal{F}_{\mathsf{RO}}$) models the behavior of a randomly sampled function with some domain A and range B; The queries to the RO are responded with a random value in a consistent manner, i.e., querying for the same argument will result in the same response.

– A *common reference string* (CRS) (cf. Figure 7 and [13] for the functionality $\mathcal{F}_{\mathsf{CRS}}$) models a (structured) randomness r shared across all parties in the execution. Any party can obtain r from the CRS functionality upon request.

– We use the *broadcast* (BC) channel functionality $\mathcal{F}_{\mathsf{BC}}$ of [31] for message delivery in the pre-voting period (cf. Figure 8). This BC channel considers communication where *the sender is authenticated.*

– We introduce an *anonymous broadcast* channel, where a sender party P can broadcast a message M to all parties in the execution anonymously, i.e., without P's identity being disclosed. In Subsection 6.1, we formalize the notion (cf. Figure 2) and present a provably secure anonymous broadcast protocol based on mix-nets [18].

– We make use of *non-interactive commitments* (NICs) (cf.Figure 9 and [10] for the functionality $\mathcal{F}_{\mathsf{NIC}}$) that are (i) binding and (ii) trapdoor (thus, also hiding), such as the Pedersen scheme [55].

– We utilize *signatures of knowledge* (SoKs) (cf. Figure 10 and [17] for the functionality $\mathcal{F}_{\mathsf{SOK}}$), so that the voters prove their eligibility without revealing their identity. In SoKs, anyone (and only them) holding a witness w for a statement x in some language L is able to produce a signature $\sigma_{m,x,L}$ on a message m that verifies correctly.

– We deploy a secure *accumulator*, a primitive that allows the compact representation of a set of elements, that is *additive* (i.e., it supports only addition of elements to the set) and *positive* (i.e., it supports membership proofs that a certain element is in the set). We refer the reader to Subsection 6.2 for our formal UC treatment of accumulators that is along the lines of [6]. In our concrete protocol instantiation, we choose the secure hash-based accumulator construction in [57].

– To realize a secure STE construction, we turn to a special type of encryption, called *time-lock encryption* (TLE) (cf. Figure 11 and [3] for the functionality $\mathcal{F}_{\mathsf{TLE}}$). In TLE, the encryption algorithm takes as input a message m and some time difficulty τ_{dec} and outputs a ciphertext c. The decryption algorithm allows the decryption of c only after time τ_{dec} has elapsed. Decryption is available to any party who has a decryption witness $w_{\tau_{\mathsf{dec}}}$ that can be produced via some publicly known process (in our protocol, the witness is produced after the party has made a specific number of calls to a RO). In particular, we make use of the TLE construction in [3]. We denote the pair of encryption and decryption algorithms by $(e_{\mathcal{F}_{\mathsf{RO}}}, d_{\mathcal{F}_{\mathsf{RO}}})$, where $\mathcal{F}_{\mathsf{RO}}$ is the RO associated with the algorithms.

– To formally capture the parties' computational restrictions in the UC setting, we invoke a *wrapper functionality*, $\mathcal{W}_q$, (cf. Figure 12 and [3]) that is parameterized by a number of queries q and a random oracle. Informally, the wrapper restricts the access to the RO by allowing parties to call the RO only up to q number of times per round (clock tick).

3.2 Parties

An execution of the E-cclesia protocol comprises four election phases **Setup**, **Credential generation**, **Cast**, and **Tally**. The parties during the phases of a protocol execution are:

– The *setup authority* (SA) that is *active only prior to the voting period.* Specifically, during **Setup**, SA is responsible for providing the election parameters, that include the list of eligible voters, the set of valid election preferences, and the period of each election phase.

– The voters $V_1, \ldots, V_n$. Each voter engages as follows:

- In **Setup**, she receives the election parameters from SA.
- In **Credential generation**, if she is eligible, she interacts with SA to obtain a unique voting credential.

- In **Cast**, if she is eligible, she generates a ballot for her choice and broadcasts the ballot to all voters.
- In **Tally**, she computes the tally that corresponds to the set of ballots she received from other eligible voters.

In our threat model, the voters can be statically corrupted while SA remains honest.

3.3 Desired security properties

Intuitively, the security properties that E-cclesia satisfies (and any other STE system should satisfy) are the following:

(1) *Correctness*: Every honestly cast vote will be included in the tally set which is the same for all honest voters.
(2) *Eligibility*: Only eligible voters' votes will be included in the tally set of each honest voter.
(3) *Fairness*: During the **Cast** phase, no party can learn some partial result.
(4) *Voter Privacy*: The (honest) voters' identities cannot be linked to their votes.
(5) *One voter-one vote*: Only one vote per (eligible) voter can be included in the tally set of each honest voter.
(6) *Verifiability*: Every voter can verify that the result corresponds to the ballots broadcast in the **Cast** phase, subject to the eligibility and one voter-one vote properties [46].

The aforementioned six properties are formally captured via the description of our ideal STE functionality (cf. Section 4).

3.4 Protocol overview

An execution of E-cclesia considers two distinct ROs, denoted by $\mathcal{F}_{\mathsf{RO}}^1$ and $\mathcal{F}_{\mathsf{RO}}^2$. All parties have the description of an accumulator scheme Σ_{acc}. In addition, all voters have the description of a NIC scheme Σ_{NIC}, a SoK scheme Σ_{SoK}, a pair of the TLE encryption and decryption algorithms $(e_{\mathcal{F}_{\mathsf{RO}}^2}, d_{\mathcal{F}_{\mathsf{RO}}^2})$ of [3] and can access $\mathcal{F}_{\mathsf{RO}}^1, \mathcal{F}_{\mathsf{RO}}^2$. In the beginning of the execution, SA is given the set of eligible voters $\mathbf{V}_{\mathsf{elig}}$, the set of valid election preferences $\mathbf{O}$, and two time moments $t_{\mathsf{cast}}, t_{\mathsf{open}}$. The four phases are executed as follows:

Setup. First, SA checks that $\mathbf{V}_{\mathsf{elig}} \subseteq \mathbf{V}$ and $t_{\mathsf{cast}} < t_{\mathsf{open}}$. If both checks succeed, then it does (if not, it aborts):

(1) Given $t_{\mathsf{cast}}, t_{\mathsf{open}}$ and a pre-known value delay_cast, it sets $\vec{t} := (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})$ that specifies the beginning and the end of all subsequent election phases.
(2) It sets the voting parameters as $\mathsf{vote.par} := (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t})$.
(3) It broadcasts vote.par to all voters.
(4) It runs the generation of the initial accumulator value a_0.
(5) It sets the registration parameters as $\mathsf{reg.par} := (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t}, a_0)$.
(6) It broadcasts reg.par to all voters.

Upon receiving $(\mathsf{SA}, \mathsf{vote.par})$ and $(\mathsf{SA}, \mathsf{reg.par})$ from the (authenticated) broadcast channel, the voter V stores vote.par, reg.par.

Credential generation. The voter V reads the time Cl from the global clock and checks that the credential period is running. If so, then she does:

(1) She randomly samples a message cr from the commitment message space $\mathcal{M}$ and creates a commitment for cr, denoted by $\hat{\mathsf{cr}}$, using the randomness aux.
(2) She broadcasts $\hat{\mathsf{cr}}$ to all voters.
(3) Upon receiving $(V^*, \hat{\mathsf{cr}}^*))$ from the broadcast channel during the **Credential generation** phase, she stores the pair $(V^*, \hat{\mathsf{cr}}^*)$, as long as (i) $V^* \in \mathbf{V}_{\mathsf{elig}}$ and (ii) she has never received a similar message from V^* before.

Cast. Each (honest and eligible) voter V engages in the voting process once by generating and casting an authenticated ballot v for her preference $o \in \mathbf{O}$. Specifically, V reads the time Cl from the global clock and checks that the casting period is running. If so, then she does:

(1) She chooses a randomness r_1 and uses r_1 to produce RO queries $\{x_k\}_{k=1}^{p_2(\lambda)}$, for some polynomial $p_2(\cdot)$. These queries are necessary for the creation of a TLE "puzzle" which solution will lead to the ballot opening during **Tally** phase.
(2) She makes the queries $\{x_k\}_{k=1}^{p_2(\lambda)}$ to $\mathcal{F}_{\mathsf{RO}}^1$ that is wrapped by $\mathcal{W}_q$ and receives the responses $\{y_k\}_{k=1}^{p_2(\lambda)}$. These queries are used for the creation of a puzzle, associated with the vote encryption (see below). The fact that these queries are wrapped by the functionality wrapper $\mathcal{W}_q$ models the limited resources each party has in her disposal each round.
(3) She runs the encryption algorithm $e_{\mathcal{F}_{\mathsf{RO}}^1}$ on input the randomness r_1, the puzzle pairs $\{(x_k, y_k)\}_{k=1}^{p_2(\lambda)}$, and the time difficulty $t_{\mathsf{open}} - (\mathsf{Cl} + 1)$, and receives a TLE ciphertext c_1.
(4) She queries a different instantiation of RO $\mathcal{F}_{\mathsf{RO}}^2$[1] for r_1 and receives a response h. Then, she sets $c_2 \leftarrow h \oplus o$.
(5) She queries $\mathcal{F}_{\mathsf{RO}}^2$ for $r_1||o$ (where $||$ denotes concatenation of strings) and receives a response c_3.
(6) She sets the ballot as $v \leftarrow (c_1, c_2, c_3)$.
(7) She runs ballot authentication for v as follows:
(i) She computes the accumulator for all received credential commitments (including her own) according to the order these were received [2], by adding one commitment at a step and storing the intermediate accumulator values for each step. For ease of notation, assume that during the **Credential generation** phase, the voters $V_1, \ldots, V_{t_{\mathsf{max}}}$ broadcast the commitments $\hat{\mathsf{cr}}_1, \ldots, \hat{\mathsf{cr}}_{t_{\mathsf{max}}}$ and that for some k, $V = V_k$.
(ii) Upon completing the addition of all received commitments, she updates the accumulator witness for the commitment $\hat{\mathsf{cr}} = \hat{\mathsf{cr}}_k$ of her own credential cr and receives the new witness $w_k^{\hat{\mathsf{r}}\mathsf{c}_k}$.
(iii) She computes a SoK, σ, for the ballot v. The SoK is produced under the statement $x = (\mathsf{cr}, \alpha_{t_{\mathsf{max}}})$, where $\alpha_{t_{\mathsf{max}}}$ is the final accumulator value, which is computed *identically for all voters*, and the SoK witness $w = (\hat{\mathsf{cr}}, \mathsf{aux}, w_k^{\hat{\mathsf{cr}}_k})$.
(8) She anonymously broadcasts (v, cr, σ) to all voters.
(9) She stores any triple $(v^*, \mathsf{cr}^*, \sigma^*)$ she receives from the anonymous broadcast channel during the **Cast** phase.

Puzzle solving: Before completing her part in a round (clock tick), the voter V engages in the puzzle solving procedure for all the puzzles that are related to the ballots she has received from the anonymous broadcast channel. In particular, by parsing a ballot v^* that she has just received as (c_1^*, c_2^*, c_3^*), V can extract a puzzle included in

[1] We use a different instantiation of RO so that our encryption is equivocable.

[2] Note that the broadcast channel we use (cf. Figure 8 and [31]) guarantees that all voters received each other's credentials in the same chronological order.

c_1^* which solution will produce the TLE decryption witness $w^*_{t_{\mathsf{open}}}$, necessary for opening v^* at **Tally** phase. During the puzzle solving procedure, V makes oracle queries to the wrapped RO $\mathcal{F}^1_{\mathsf{RO}}$, under the restrictions that $\mathcal{W}_q$ imposes, i.e., the queries can be parallelized for all puzzles, but no more than q queries can be made per round. The idea is that each puzzle is created in chain-based manner, in the sense that the response of the i-th query becomes the $i+1$-th query, which implies that each puzzle will be solved *sequentially* after some well-defined time has elapsed (i.e., with time difficulty adjusted such that time t_{open} has been reached).

Tally. The voter V reads the time Cl from the global clock and checks that the tally period is running. If so, then she computes the tally by executing the following steps:

(1) For every triple $(v^*, \mathsf{cr}^*, \sigma^*)$ she has received during the **Cast** phase, she verifies the SoK σ^* for v^* under the statement $(\mathsf{cr}^*, \alpha_{t_{\max}})$. If the verification is successful, she adds $(v^*, \mathsf{cr}^*, \sigma^*)$ to the tally set. In this step, V discards any received triple such that the included credential does not correspond to any accumulated commitment value. However, note that after this step is completed, the tally set may contain multiple triples $(v_1^*, \mathsf{cr}_1^*, \sigma_1^*), \ldots, (v^*_{\mu^*}, \mathsf{cr}^*_{\mu^*}, \sigma^*_{\mu^*})$ that were broadcast by the same (dishonest) voter.

(2) She discards multiple triples by pairwise checking whether the received triples include credentials that match. Namely, for any two triples $(v^*, \mathsf{cr}^*, \sigma^*)$ and $(v^{**}, \mathsf{cr}^{**}, \sigma^{**})$ such that $\mathsf{cr}^* = \mathsf{cr}^{**}$, she discards the triple she received last out of those two. Clearly, after this pairwise check is completed, all except one of triples that correspond to the same credential will be removed from the tally set.

(3) After the tally set has been "filtered" regarding multiple triples, V decrypts every ballot $v^* = (c_1^*, c_2^*, c_3^*)$ of a triple $(v^*, \sigma^*, \mathsf{cr}^*)$ in the tally set as follows:
(i) She runs the TLE decryption algorithm $d_{\mathcal{F}^2_{\mathsf{RO}}}$ on input c_1^* and the corresponding decryption witness $w^*_{t_{\mathsf{open}}}$ and receives the output r_1^*.
(ii) She queries $\mathcal{F}^2_{\mathsf{RO}}$ for r_1^* and receives a response h^*. She extracts the election option as $o^* \leftarrow h^* \oplus c_2^*$.
(iii) She verifies the validity of o^* by first checking that $o^* \in \mathbf{O}$, and then querying $\mathcal{F}^2_{\mathsf{RO}}$ for $r_1^* || o^*$ and checking if the response matches c_3^*. If so, then she records o^* as valid.

(4) She returns as election tally the set of all options that have been recorded as valid during the execution of Steps 3(i)-(iii).

In Appendix D, we informally discuss the details that render our protocol secure w.r.t. the properties listed in Subsection 3.3.

4 THE $\mathcal{F}_{\mathsf{STE}}$ FUNCTIONALITY

In this section, we describe the functionality $\mathcal{F}_{\mathsf{STE}}$ which captures our security requirements for STE elections (correctness, eligibility, fairness, voter privacy, one voter-one vote, verifiability). The functionality $\mathcal{F}_{\mathsf{STE}}$ interacts with the setup authority SA, the voters in the set $\mathbf{V} = \{V_1, \ldots, V_n\}$ and the simulator $\mathcal{S}$. It is summarized in the next paragraphs and is formally presented in Figure 1.

The functionality is parameterized by SA, the set $\mathbf{V}$, an integer value $\mathsf{delay\_gen}$ which shows the number of rounds that are needed for the generation of the ballot, an integer value $\mathsf{delay\_cast}$ which shows how many rounds a message needs to reach its recipient from the time of casting, and the predicate $\mathtt{Status}$ that given the current time Cl, the time values that define the election, and an election phase, outputs $\top$ if that phase is active or $\bot$ otherwise.

In the **Setup** phase, the functionality registers the set of eligible voters $\mathbf{V}_{\mathsf{elig}}$, the set of valid election preferences $\mathbf{O}$, and the duration of the election (in the time vector $\vec{t}$), upon request from SA and the permission of $\mathcal{S}$.

The **Credential generation** phase is active for every Cl such that $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \top$. In this phase, each credential request from an eligible voter V is sent to $\mathcal{S}$. If $\mathcal{S}$ replies with $\mathtt{ready}$, then V is marked as ready to vote.

The **Cast** phase is active for every Cl such that $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \top$. In this phase, if a voter is ready to vote, $\mathcal{F}_{\mathsf{STE}}$ leaks to the simulator $\mathcal{S}$ the length of the vote and a fresh random value tag. The latter is necessary as we allow $\mathcal{S}$ to update this message with a ciphertext at later stages. Thus, $\mathcal{S}$ needs a reference point for updating that message without getting the message itself (preserving privacy).

Each time $\mathcal{F}_{\mathsf{STE}}$ receives a command message it follows the *delayed ballot generation and casting* subroutine. Specifically, $\mathcal{F}_{\mathsf{STE}}$ checks if by the time it received a cast ballot request from an honest voter V the time for ballot generation $\mathsf{delay\_gen}$ has elapsed. Then, it checks if the ballot can still be cast by executing the predicate $\mathtt{Status}$ for the current time Cl. If so, $\mathsf{F}_{\mathsf{STE}}$ includes that ballot both into the lists of cast ballots and the ballots pending for reception along with the current recording time Cl. Then, it checks for every ballot in the list of ballots pending for reception if $\mathsf{delay\_cast}$ time has elapsed. If so, it informs $\mathcal{S}$. All the ballots in the list of cast ballots will be accessible for tallying, as by the time of recording, the execution is still in the **Cast** phase, taking into consideration $\mathsf{delay\_cast}$. Observe that $\mathcal{S}$ might receive a vote before the **Tally** phase. This is not an issue as the **Cast** phase would be over.

Finally, the **Tally** phase is active for every Cl such that $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \top$. In this phase, the voter V requests the tally from $\mathcal{F}_{\mathsf{STE}}$ and receives the multi set of the cast ballots. Moreover, $\mathcal{S}$ can request the election outcome and receives it if $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \top$ or $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \bot$. The latter condition captures cases in which $\mathcal{S}$ might be able to learn the tally earlier from the **Tally** phase but still when the **Cast** phase would be over, meaning that *fairness* is preserved. In addition, V or $\mathcal{S}$ may execute verification by providing $\mathcal{F}_{\mathsf{STE}}$ with some multiset $\hat{\mathbf{T}}$ that replies by 1 or 0 depending on whether $\hat{\mathbf{T}}$ matches the tally multiset or not.

The predicate $\mathtt{Status} : \mathbb{N} \times (\mathbb{N})^3 \times \{\mathsf{Cred}, \mathsf{Cast}, \mathsf{Tally}\} \longrightarrow \{\top, \bot\}$ is defined as follows. Given the current time $\mathsf{Cl} \in \mathbb{N}$, the time vector $\vec{t} = (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast}) \in (\mathbb{N})^3$, and $\mathsf{A} \in \{\mathsf{Cred}, \mathsf{Cast}, \mathsf{Tally}\}$:

$$\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{A}) = \begin{cases} \top, & \mathsf{A} = \mathsf{Cred} \wedge \mathsf{Cl} < t_{\mathsf{cast}} \\ \top, & \mathsf{A} = \mathsf{Cast} \wedge t_{\mathsf{cast}} \leq \mathsf{Cl} < t_{\mathsf{open}} - \mathsf{delay\_cast} \\ \top, & \mathsf{A} = \mathsf{Tally} \wedge t_{\mathsf{open}} \leq \mathsf{Cl} \\ \bot, & \text{otherwise} \end{cases}$$

$\mathcal{F}_{\mathsf{STE}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$.

The functionality initializes as empty the lists of eligible voters' credentials L_{elig}, generated ballots L_{gball}, cast ballots

L_{cast}, pending for reception ballots L_{pend}, a list L_{adv} of the (dummy) parties that have submitted an $\textsc{Advance\_Clock}$ message for the current round, and a multiset $\mathbf{T}$. Upon receiving $(\text{sid}, \textsc{Corrupt}, \mathbf{V}_{\text{corr}})$ from $\mathcal{S}$, if $\mathbf{V}_{\text{corr}} \subseteq \mathbf{V}$, it fixes $\mathbf{V}_{\text{corr}}$ as the set of corrupted voters.

Each time the functionality receives a command message it executes the *delayed ballot generation and casting* procedure as described below:

Delayed ballot generation and casting: Upon receiving $(\text{sid}/\textsc{sid}_C, \mathcal{I}, \text{input})$ from $V \in \mathbf{V} \setminus \mathbf{V}_{\text{corr}}$, where $\mathcal{I} \in \{\textsc{Gen\_Cred}, \textsc{Cast}, \textsc{Update}, \textsc{Cast\_Check}, \textsc{Advance\_Clock}, \textsc{Read\_Clock}, \textsc{Tally}\}$, if $\mathbf{V} \setminus \mathbf{V}_{\text{corr}} \subseteq L_{\text{adv}}$, it sends $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\text{clock}}$ to proceed to the next round. Upon receiving $(\textsc{sid}_C, \textsc{Advanced\_Clock}, \mathcal{F}_{\text{vm}})$ from $\mathcal{G}_{\text{clock}}$, it reads the time Cl from $\mathcal{G}_{\text{clock}}$ and does:

1. For every tuple $(V, v, o, \text{tag}, \mathsf{Cl}', 1)$ in L_{gball} such that $\mathsf{Cl} - \mathsf{Cl}' \geq \text{delay\_gen}$, if $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cast}) = \top$ it adds $(V, v, o, \mathsf{Cl}, 1)$ to L_{cast} and (v, V, Cl) to L_{pend} .
2. For every triple $(v^*, V^*, \mathsf{Cl}^*) \in L_{\text{pend}}$ such that $\mathsf{Cl} - \mathsf{Cl}^* = \text{delay\_cast}$, it sends $(\text{sid}, \textsc{Cast\_Ballot}, v^*)$ to $\mathcal{S}$ only if $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cast}) = \bot$. Then, it removes $(v^*, V^*, \mathsf{Cl}^*)$ from L_{pend}.
3. It sets L_{adv} as empty.

Then, it executes $(\text{sid}/\textsc{sid}_C, \mathcal{I}, \text{input})$ as described below.

- Upon receiving $(\text{sid}, \textsc{Election\_Info}, \mathbf{V}_{\text{elig}}, \mathbf{O}, t_{\text{cast}}, t_{\text{open}})$ from SA for the first time, if $\mathbf{V}_{\text{elig}} \subseteq \mathbf{V}$ and $t_{\text{cast}} < t_{\text{open}}$, it forwards the message to $\mathcal{S}$. Upon receiving $(\text{sid}, \textsc{Election\_Info\_OK}, \mathbf{V}_{\text{elig}}, \mathbf{O}, t_{\text{cast}}, t_{\text{open}})$ from $\mathcal{S}$, it sets $\vec{t} \leftarrow (t_{\text{cast}}, t_{\text{open}}, \text{delay\_cast})$ and reg.par $:= (\mathbf{V}_{\text{elig}}, \mathbf{O}, \vec{t})$.
- Upon receiving $(\text{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\text{elig}}$ for the first time, it reads the time Cl from $\mathcal{G}_{\text{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cred}) = \top$, it sends $(\text{sid}, \textsc{Gen\_Cred}, V)$ to $\mathcal{S}$. Upon receiving $(\text{sid}, \textsc{Gen\_Cred}, V, \mathtt{ready})$ from $\mathcal{S}$, if $V \notin \mathbf{V}_{\text{corr}}$, it adds $(V, \mathtt{ready}, 1)$ to L_{elig}. Else, it adds $(V, \mathtt{ready}, 0)$ to L_{elig}.
- Upon receiving $(\text{sid}, \textsc{Cast}, o)$ from $V \in \mathbf{V}_{\text{elig}} \setminus \mathbf{V}_{\text{corr}}$ for the first time such that $(V, \mathtt{ready}, 1) \in L_{\text{elig}}$ and $o \in \mathbf{O}$, it reads the time Cl from $\mathcal{G}_{\text{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cast}) = \top$ it does:
  1. It picks tag $\overset{\$}{\leftarrow}$ TAG and it inserts the tuple $(V, \text{Null}, o, \text{tag}, \mathsf{Cl}, 1) \rightarrow L_{\text{gball}}$.
  2. It sends $(\text{sid}, \textsc{Gen\_Ballot}, \text{tag}, \mathsf{Cl}, 0^{|o|})$ to $\mathcal{S}$(*Ballot privacy*). Upon receiving the token back from $\mathcal{S}$, it returns $(\text{sid}, \textsc{Casting})$ to V.
- Upon receiving $(\text{sid}, \textsc{Update}, \{(v_j, \text{tag}_j)\}_{j=1}^{p(\lambda)})$ from $\mathcal{S}$ for all $v_j \neq \text{Null}$, if there is a tuple $(\cdot, v_j, \cdot, \cdot, \cdot, 1)$ in L_{gball} or if there are $j, j^* \in [1, p(\lambda)]$ such that $v_j = v_{j^*}$ it returns $(\text{sid}, \textsc{Update}, \{(v_j, \text{tag}_j)\}_{j=1}^{p(\lambda)}, \bot)$ to $\mathcal{S}$. Else, it updates each tuple $(V, \text{Null}, o_j, \text{tag}_j, \mathsf{Cl}_j, 1)$ to $(V, v_j, o_j, \text{tag}_j, \mathsf{Cl}_j, 1)$ in L_{gball}.
- (*One-voter-one-vote*) Upon receiving $(\text{sid}, \textsc{Cast}, v, V)$ from $\mathcal{S}$ for $V \in \mathbf{V}_{\text{corr}}$, for the first time, it reads the time Cl from $\mathcal{G}_{\text{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cast}) = \top$ and there is a tuple $(V, \mathtt{ready}, 0) \in L_{\text{elig}}$ (*eligibility*), it adds $(V, v, \cdot, \mathsf{Cl}, 0)$ to L_{cast}.
- Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from a voter $V \in \mathbf{V} \setminus \mathbf{V}_{\text{corr}}$, if $P \notin L_{\text{adv}}$, it adds P to L_{adv} and forwards $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\text{clock}}$ on behalf of P.
- Upon receiving $(\textsc{sid}_C, \textsc{Read\_Clock})$ from a voter $V \in \mathbf{V} \setminus \mathbf{V}_{\text{corr}}$, it reads the time Cl from $\mathcal{G}_{\text{clock}}$ and returns $(\textsc{sid}_C, \textsc{Read\_Clock}, \mathsf{Cl})$ to P.
- Upon receiving $(\text{sid}, \textsc{Tally})$ from a voter $V \in \mathbf{V} \setminus \mathbf{V}_{\text{corr}}$, it reads time Cl from $\mathcal{G}_{\text{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Tally}) = \top$, it does:
  1. If $\mathbf{T} = \emptyset$, for every tuple $(V^*, v, \cdot, \mathsf{Cl}, 0) \in L_{\text{cast}}$ it sends $(\text{sid}, \textsc{Opening}, V^*, v)$ to $\mathcal{S}$. Upon receiving $(\text{sid}, \textsc{Opening}, V^*, v, o)$ from $\mathcal{S}$, if $o \in \mathbf{O}$, then it updates the tuple as $(V^*, v, o, \mathsf{Cl}, 0)$ in L_{cast}. Finally, it sets the tally multiset as $\mathbf{T} \leftarrow \{o \in \mathbf{O} | (V^*, \cdot, o, \cdot, \cdot) \in L_{\text{cast}}\}$.
  2. It returns $(\text{sid}, \textsc{Tally}, \mathbf{T})$ to V.
- (*Fairness*)Upon receiving $(\text{sid}, \textsc{Tally})$ from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{\text{Clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cred}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Cast}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Tally}) = \bot$ or $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Tally}) = \top$, it returns to $\mathcal{S}$ all pairs (v, o) such that $(V, v, o, \text{tag}, \mathsf{Cl}^*, 1) \in L_{\text{gball}} \wedge (V, v, o, \mathsf{Cl}', 1) \in L_{\text{cast}}$ for some ballot generation time and casting time Cl^* and Cl', respectively.
- (*Verifiability*)Upon receiving $(\text{sid}, \textsc{Verify}, \hat{\mathbf{T}})$ from a voter $V \in \mathbf{V} \setminus \mathbf{V}_{\text{corr}}$, it reads Cl from $\mathcal{G}_{\text{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \text{Tally}) = \top$, it does:
  1. If $\mathbf{T} = \emptyset$, it computes the tally multiset as if it received a $(\text{sid}, \textsc{Tally})$ command.
  2. If $\hat{\mathbf{T}} = \mathbf{T}$, it returns $(\text{sid}, \textsc{Verify}, \hat{\mathbf{T}}, 1)$ to V. Else, it returns $(\text{sid}, \textsc{Verify}, \hat{\mathbf{T}}, 0)$ to V.

Figure 1: The self-tallying election functionality $\mathcal{F}_{\text{STE}}$.

5 MODULAR DESIGN

Taking advantage of the compositionality of the UC framework, we break $\mathcal{F}_{\text{STE}}$ down into two smaller modules: (i) the *eligibility* functionality $\mathcal{F}_{\text{elig}}$ that handles the credential generation, ballot authentication of eligible voters and ballot verification, and (ii) the *vote management* functionality $\mathcal{F}_{\text{vm}}$ that handles the ballot generation, casting, and opening. In Appendix E.1,E.2,E.3 we formally present these two functionalities along with a simple protocol that UC realizes $\mathcal{F}_{\text{STE}}$ in the $(\mathcal{F}_{\text{elig}}, \mathcal{F}_{\text{vm}})$-hybrid model. The advantage of following this approach is the ability to change the underlying cryptographic primitives in future instantiations of E-cclesia without arguing about the security of a whole protocol that UC realizes $\mathcal{F}_{\text{STE}}$. For example, to replace a primitive that is related to the vote management part of the protocol, it suffices to define a smaller protocol that UC realizes $\mathcal{F}_{\text{vm}}$ (instead of $\mathcal{F}_{\text{STE}}$). This approach makes both the proof and the modeling simpler, as it supports easier future updates of the underlying primitives due to the technological advancements of each era (an advanced hash-function) or design choices (threshold encryption instead of time-lock encryption).

In the rest of this section, we provide an overview of the command interface of $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$, as well as the theorem stating the realization of $\mathcal{F}_{\mathsf{STE}}$ in the $(\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model.

The functionality $\mathcal{F}_{\mathsf{elig}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_cast}, \mathtt{Status})$: (cf. Figure 13) It records the set of corrupted voters $\mathbf{V}_{\mathsf{corr}}$ provided by $\mathcal{S}$.

- Upon receiving $(\mathsf{sid}, \textsc{Eligible}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from SA, if the parameters are valid, it requests and receives algorithms $\mathtt{GenCred}$, $\mathtt{AuthBallot}$, $\mathtt{VrfyBallot}$, $\mathtt{UpState}$, and state St_{gen} from $\mathcal{S}$. Then, it sends the registration parameters $(\mathsf{sid}, \textsc{Elig\_Par}, (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t} := (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast}), St_{\mathsf{gen}}))$ to all voters and $\mathcal{S}$.
- Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$ once during the **Credential generation** phase, it computes a credential triple $(\mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}) \leftarrow \mathtt{GenCred}(1^{\lambda}, \mathsf{reg.par})$. It sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}}, \mathsf{sender})$ to V and $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}})$ to all other voters in $\mathbf{V} \setminus \{V\}$ and $\mathcal{S}$. The functionality allows $\mathcal{S}$ to generate credential triples on behalf of eligible corrupted voters.
- Upon receiving $(\mathsf{sid}, \textsc{Auth\_Ballot}, v)$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$ during the **Cast** phase, it runs ballot authentication by computing $\sigma \longleftarrow \mathtt{AuthBallot}(v, \mathsf{cr}, St_{\mathsf{fin}}, \mathsf{reg.par}, \mathsf{aux})$, where St_{fin} is generated by the $\mathtt{UpState}$ algorithm. It returns $(\mathsf{sid}, \textsc{Auth\_Ballot}, v, \sigma)$ to V. The functionality allows $\mathcal{S}$ to authenticate eligible corrupted voters' ballots.
- Upon receiving $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma} = (\mathsf{cr}, \sigma))$ from $V \in \mathbf{V}$, it runs ballot verification by computing $x \leftarrow \mathtt{VrfyBallot}(v, \vec{\sigma}, St_{\mathsf{fin}}, \mathsf{reg.par})$. If cr is recorded and v has been honestly authenticated via $\vec{\sigma}$, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma}, 1)$ to V. If $x = 1$ and v has not been authenticated via $\vec{\sigma}$, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma}, \perp)$ to V and halts. If $x = 1$ and there is an honest ballot $v' \neq v$ authenticated via $\vec{\sigma}' = (\mathsf{cr}, \sigma')$, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma}, \perp)$ to V and halts. Else, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma}, x)$ to V.
- Upon receiving $\big(\mathsf{sid}, \textsc{Link\_Ballots}, (v_1, (\mathsf{cr}_1, \sigma_1)), (v_2, (\mathsf{cr}_2, \sigma_2))\big)$ from $V \in \mathbf{V}$, it returns $\big(\mathsf{sid}, \textsc{Link\_Ballots}, (v_1, (\mathsf{cr}_1, \sigma_1)), (v_2, (\mathsf{cr}_2, \sigma_2)), x\big)$ to V, where $x = 1$ if $\mathsf{cr}_1 = \mathsf{cr}_2$ and v_1, v_2 have been authenticated via $(\mathsf{cr}_1, \sigma_1)$ and $(\mathsf{cr}_2, \sigma_2)$, respectively, and $x = 0$, otherwise.

The functionality $\mathcal{F}_{\mathsf{vm}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$: (cf. Figure 14) It records the set of corrupted voters $\mathbf{V}_{\mathsf{corr}}$ provided by $\mathcal{S}$.

- Upon receiving $(\mathsf{sid}, \textsc{Election\_Info}, \mathbf{O}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from SA for the first time, if parameters are valid, it sends the voting parameters $(\mathsf{sid}, \textsc{Election\_Info}, (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t} := (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})))$ to SA and $\mathcal{S}$.
- Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, o)$ from $V \notin \mathbf{V}_{\mathsf{corr}}$ for the first time, if $o \in \mathbf{O}$, it records the time that V submitted the request for selection o associating it with some random tag, and asks from $\mathcal{S}$ to generate a ballot for $0^{|o|}$, i.e., by disclosing only the length of o. It returns $(\mathsf{sid}, \textsc{Generating})$ to V. The functionality allows $\mathcal{S}$ to generate ballots on behalf of the corrupted voters for selections of its choice.
- Upon receiving $(\mathsf{sid}, \textsc{Update}, \{(v_j, \mathsf{tag}_j)\}_{j=1}^{p(\lambda)})$ from $\mathcal{S}$, it associates each ballot v_j with the preference o_j of V that is recorded under the same tag_j.
- Upon receiving $(\mathsf{sid}, \textsc{Retrieve})$ from $V \notin \mathbf{V}_{\mathsf{corr}}$, it returns $(\mathsf{sid}, \textsc{Retrieve}, (o, v))$ to V, if ballot v is associated with the selection o of V that was recorded at least delay_gen time earlier. Else, it returns $(\mathsf{sid}, \textsc{Retrieve}, \perp)$ to V.
- Upon receiving $(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma})$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$ during the **Cast** phase, if there is a ballot v associated with a selection o of V that was recorded at least delay_gen time earlier, then it marks $(v, \vec{\sigma})$ as "pending" to be cast on behalf of V. Otherwise, it returns $(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma}, \perp)$ to V. The functionality allows casting of any corrupted voters' ballots during the **Cast** phase.
- The functionality forwards the requests of all honest parties to $\mathcal{G}_{\mathsf{clock}}$ and monitors the $\textsc{Advance\_Clock}$ messages forwarded during each round. If all honest parties have made an $\textsc{Advance\_Clock}$ request for the current round, it sends an $\textsc{Advance\_Clock}$ request for itself to proceed to the next round. Then, for every M^* pending to be cast on behalf of V^* for delay_cast time, it sends $(\mathsf{sid}, \textsc{Cast\_Ballot}, M^*, \mathsf{sender})$ to V^* and $(\mathsf{sid}, \textsc{Cast\_Ballot}, M^*)$ to all voters in $\mathbf{V} \setminus \{V^*\}$ and $\mathcal{S}$.
- Upon receiving $(\mathsf{sid}, \textsc{Open}, v)$ from any party $P \in \mathbf{V} \cup \{\mathcal{S}\}$ during the **Tally** phase, if v is associated with an honestly recorded selection o, it sends $(\mathsf{sid}, \textsc{Open}, v, o)$ to P. Besides, the corrupted voters' ballots are opened as $\mathcal{S}$ instructs.
- Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{S}$ during either (i) a "waiting" period that neither credential generation, casting nor tally happens, or (ii) during the **Tally** phase, it provides $\mathcal{S}$ with all the honestly cast ballots and their associated selections.

The protocol $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$: is presented in Figure 15. Its purpose is to combine the two interfaces of $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$ in order to build a complete hybrid protocol that realizes $\mathcal{F}_{\mathsf{STE}}$. We prove the following theorem in Appendix E.3.2.

THEOREM 5.1. *The protocol* $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$ *described in Figure 15 UC-realizes* $\mathcal{F}_{\mathsf{STE}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$ *in the* $(\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}})$*-hybrid model.*

Subsequently, we provide concrete realizations of $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$ which results in E-CCLESIA, the first instantiation of $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}}$.

6 E-CCLESIA AS A UC REALIZATION OF $\mathcal{F}_{\mathsf{STE}}$

In Section 5, we showed how we can realize $\mathcal{F}_{\mathsf{STE}}$ by using the two functionality sub-modules $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$. In this section, we present UC realizations Π_{elig} (cf. Subsection 6.3) and Π_{vm} (cf. Subsection 6.4) for the functionalities $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$, respectively, in hybrid models. Given our realizations of UC secure anonymous broadcast (cf. Subsection 6.1) and accumulator (cf. Subsection 6.2), and results from the UC literature, we argue that each of the deployed hybrid functionalities can be realized via a subset of $\{\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{BC}}, \mathcal{W}_q, \mathcal{G}_{\mathsf{clock}}\}$. Thus, we conclude that the UC description of E-CCLESIA, $\Pi_{\text{E-CCLESIA}}$, UC-realizes $\mathcal{F}_{\mathsf{STE}}$ in the $(\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{BC}}, \mathcal{W}_q, \mathcal{G}_{\mathsf{clock}})$-hybrid model (cf. Subsection 6.5).

6.1 Realizing UC anonymous broadcast

In this subsection, we provide a formalization of the notion of anonymous broadcast and a UC realization that is based on mix-nets [18]. We stress that our approach guarantees a high level of sender anonymity, thus supporting resistance against *timing attacks* [1, 45].

The ideal functionality $\mathcal{F}_{\mathsf{an.BC}}^{\ell,B,p}$: The functionality $\mathcal{F}_{\mathsf{an.BC}}^{\ell,B,p}$ is presented in Figure 2. The parameter ℓ determines the communication delay from the moment that a message is transmitted till the moment it is received by all parties. In addition, $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ is parameterized by B, which is a bound on the number of messages that each party can broadcast per round. We stress that this bound appears to be necessary, otherwise the functionality would become unrealistic. Namely, in any real-world protocol, if the maximum number of messages that the environment can instruct a sender party to broadcast in some round is *unknown*, then any attempt to create a "cover traffic" effect via the transmission of (indistinguishable) dummy messages would fail. Thus, the sender's broadcast rate would be revealed to an adversary that observes the entire network (*global adversary*) and anonymity, as determined by a functionality that does not consider a bound B, could easily be broken. Finally, our functionality is parameterized by a polynomial $p(\cdot)$ that sets an upper bound on the length of the messages that are allowed to be broadcast. Like B, this bound seems to be necessary, otherwise in any realization attempt, the length of the dummy messages would not be able to support a cover traffic effect over actual messages of unknown variable length.

Overall, $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ aims to capture the highest possible level of sender anonymity by fully hiding the sender's activity apart from the fact that it has not broadcast more than B messages per round, and that the message length is bounded by $p(\lambda)$.

The Anonymous Broadcast functionality $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}(\mathbf{P})$.

The functionality initializes as empty a list L_{pool} of messages pending to be broadcast, and a list L_{adv} of the (dummy) parties that have submitted an $\textsc{Advance\_Clock}$ message for the current round. In addition, it sets a flag status as 0 and for every party $P \in \mathbf{P}$, it sets a counter count_P also as 0. Let $\mathbf{P}_{\mathsf{corr}} \subseteq \mathbf{P}$ be the set of corrupted parties.

Every time the functionality receives a command message from a party $P \in \mathbf{P}$, it executes the procedure *Setup or Broadcast* as described below and then executes the command message according to its description.

Setup or Broadcast: Upon receiving $(\mathsf{sid}/\textsc{sid}_C, \mathcal{I}, \mathsf{input})$ from $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$, where $\mathcal{I} \in \{\textsc{Broadcast}, \textsc{Advance\_Clock}, \textsc{Read\_Clock}\}$, if status $= 0$, it sends $(\mathsf{sid}, \textsc{Setup}, P)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Setup\_No}, P)$ from $\mathcal{S}$, it halts. Else, upon receiving $(\mathsf{sid}, \textsc{Setup\_OK}, P)$ from $\mathcal{S}$, it sets status = 1.

Next, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}} \subseteq L_{\mathsf{adv}}$, it sends $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$ to proceed to the next round. Upon receiving $(\textsc{sid}_C, \textsc{Advanced\_Clock}, \mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}})$ from $\mathcal{G}_{\mathsf{clock}}$, it does:

(1) It randomly chooses a permutation $\pi \overset{\$}{\leftarrow} \{1, \ldots, |L_{\mathsf{pool}}|\}$, where $|L_{\mathsf{pool}}|$ is the number of elements in L_{pool}.
(2) It reorders the entries in L_{pool} w.r.t. π, i.e., $L_{\mathsf{pool}} \leftarrow \pi(L_{\mathsf{pool}})$.
(3) For every triple $(M^*, P^*, \mathsf{Cl}^*) \in L_{\mathsf{pool}}$ such that $\mathsf{Cl} - \mathsf{Cl}^* = \ell + 1$, it anonymously broadcasts M^* to $P_1, \ldots, P_n$ and $\mathcal{S}$ as follows: it sends $(\mathsf{sid}, \textsc{Broadcast}, M^*, \mathsf{sender})$ to P^* and $(\mathsf{sid}, \textsc{Broadcast}, M^*)$ to all other parties in $\mathbf{P} \setminus \{P^*\}$ and $\mathcal{S}$. Then, it removes $(M^*, P^*, \mathsf{Cl}^*)$ from L_{pool}. For the triples of the form $(\mathsf{tag}, P^*, \mathsf{Cl}^*)$ it does the same except that it first requests from $\mathcal{S}$ the broadcast message M^* that corresponds to tag.
(4) It sets L_{adv} as empty and for every $P^* \in \mathbf{P}$ it resets count_{P^*} as 0.

Subsequently, it executes $(\mathsf{sid}/\textsc{sid}_C, \mathcal{I}, \mathsf{input})$ as described below.

- Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, M)$ from a party $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$, if $\mathsf{count}_P = B$ or $|M| > p(\lambda)$ or $P \in L_{\mathsf{adv}}$, it ignores the message. Otherwise, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$, it adds (M, P, Cl) to L_{pool} and increases count_P by 1.
- Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, \mathsf{tag}, \hat{P})$ from $\mathcal{S}$ on behalf of a party $\hat{P} \in \mathbf{P}_{\mathsf{corr}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and adds $(\mathsf{tag}, \hat{P}, \mathsf{Cl})$ to L_{pool}.
- Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from a party $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$, if $P \notin L_{\mathsf{adv}}$, it adds P to L_{adv} and forwards $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$ on behalf of P.
- Upon receiving $(\textsc{sid}_C, \textsc{Read\_Clock})$ from a party $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and returns $(\textsc{sid}_C, \textsc{Read\_Clock}, \mathsf{Cl})$ to P.

Figure 2: The anonymous broadcast functionality $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$.

The protocol $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$: Our mix-net-based construction is built upon a special case of a $m \times \ell$ stratified mix-net architecture [29]; For $j \in [m]$, there is a cascade of ℓ mix servers $\mathsf{MX}_{j,1} \rightarrow \cdots \rightarrow \mathsf{MX}_{j,k} \rightarrow \cdots \rightarrow \mathsf{MX}_{j,\ell}$. The input to the server $\mathsf{MX}_{j,1}$ is encrypted via ℓ-level layered encryption. Let $\mathbf{MX} = \{\mathsf{MX}_{j,k}\}_{j \in [m], k \in [\ell]}$ be the set of all mix servers.

The protocol execution is initialized by the first activated party broadcasting (via functionality $\mathcal{F}_{\mathsf{BC}}$) a "setup" message to all mix servers. In turn, every $\mathsf{MX}_{j,k}$ generates a pair of a secret and a public key $(\mathsf{sk}_{j,k}, \mathsf{pk}_{j,k})$ and broadcasts $\mathsf{pk}_{j,k}$ to all parties.

Subsequently, anonymous broadcast of messages is carried out. To achieve the high level of sender anonymity required by $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$, our design encompasses the following techniques:

1. *Padding:* Only messages of length up to $p(\lambda)$ will be broadcast. To achieve transmission of messages of equal length standard padding is used. Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, M)$ from $\mathcal{Z}$, if the sender party P has not already received B $\textsc{Broadcast}$ commands from $\mathcal{Z}$ for the current round and if the message M has length $|M| < p(\lambda)$, then M is padded (e.g., with leading zeros) so that $|M| = p(\lambda)$. For notation simplicity, we will still write the padded message as M and will clarify when necessary.

2. *Equivocation:* The sender applies the equivocation technique of [53] on the padded message M that utilizes a random oracle $H(\cdot)$ (modeled as $\mathcal{F}_{\mathsf{RO}}$ in the UC setting) for producing the pair $(r, H(r) \oplus M)$, where r is some randomness. In the security proof, this step allows the simulator that controls $\mathcal{F}_{\mathsf{RO}}$ to emulate message

transmission and produce a consistent view to the adversary, even if it receives the real broadcast messages from $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ with delay ℓ.

3. *Share-wise transmission:* To be transmitted to the mix-net, the pair $(r, H(r) \oplus M)$ is first split into m shares via Shamir's (t, m)-threshold secret sharing (TSS) scheme [62], where t is the threshold of shares required for recovering the secret. Each share $[(r, H(r) \oplus M)]_j$, $j \in [m]$, intended for the j-th cascade, is encrypted into ℓ layers as $\mathsf{PKE.Enc}\big(\mathsf{pk}_{j,1}, \ldots, (\mathsf{PKE.Enc}(\mathsf{pk}_{j,\ell}, (\mathsf{tag}, [(r, H(r)\oplus M)]_j)))\big)$, where tag is a random tag common for all shares of $(r, H(r) \oplus M)]$. By utilising (t, m)-TSS, we achieve fault-tolerance (up to a fixed threshold $m - t$) against fail-stop failures and totally hide $(r, H(r) \oplus M)$ from a coalition of up to $t - 1$ corrupted exit servers.

4. *Cover traffic and batch transmission:* Each party creates a cover traffic effect by transmitting as many dummy ciphertexts to each input (first layer) server, so that the bound B is reached. The party transmits all real and dummy ciphertexts together right before completing her round, i,e, when it receives a $(\textsc{sid}_C, \textsc{Advance\_Clock})$ command from $\mathcal{Z}$. By applying cover traffic and batch transmission as above, the protocol provides resistance against timing attacks.

5. *Anonymous routing:* Each input server of the m cascades receives the corresponding encrypted share of the message $(r, H(r) \oplus M)$, where all encrypted shares are accompanied by the same random tag. In each layer, the servers remove one layer of encryption and in the beginning of the next round, they randomly permute the (encrypted) shares they received in the current round. By permuting the shares, the knowledge that the global adversary has on the activation sequence of the senders during a round (inherent in the UC framework) is neutralized. Then, $\mathsf{MX}_{j,k}$ forwards the pool of permuted encrypted shares to $\mathsf{MX}_{j,k+1}$, for $k = 1, \ldots, \ell - 1$. In the final ℓ-th layer, the exit server of each cascade decrypts and obtains the shuffled shares of this cascade in plaintext. In the beginning of the next round and upon randomly permuting the shares, the exit server broadcasts the shares to all parties. The whole anonymization process imposes an aggregate delay ℓ (1 clock tick per layer). Moreover, the mix servers discard all ciphertexts that they have received before.This step guarantees protection against *replay attacks*, where the adversary eventually links an honest message to its original sender by retransmitting its original encryption a distinct number of times. Note that the security of the underlying encryption scheme implies that no message will be honestly encrypted twice in an identical manner (i.e., using the same randomness) except from some $\mathsf{negl}(\lambda)$ probability, so the servers can safely discard repeated ciphertexts.

6. *Message recovery:* Upon receiving at least t broadcast shares, every recipient can reconstruct the pair $(r, H(r) \oplus M)$ from the shares that are linked to the same tag. Then, the recipient will query the random oracle on r, obtain $H(r)$, and finally recover the message as $M \leftarrow H(r) \oplus (H(r) \oplus M)$ and remove the pads.

In terms of communication infrastructure, authentication is required from a sender party to an input server and from a server at the k-th layer to the server of the $k+1$-th layer of the same cascade. Broadcast is required at initialization, and during execution only at the final layer where the exit servers send the decrypted shares to all recipients. In our protocol description, to avoid inserting an extra hybrid message authentication functionality (such as the one in [14]) we make use of the authenticated broadcast functionality $\mathcal{F}_{\mathsf{BC}}$ of [31] (cf. Figure 8) that is sufficient for all communications.

The protocol $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$ is formally presented in Figure 17 (cf. Appendix F.1). Its design enables defense against adversaries that can (i) observe the whole network traffic (global adversary), (ii) corrupt parties, and (iii) corrupt up to a threshold of mix servers (specified by m, ℓ, t), in a *fail-stop* manner, i.e., the corrupted server follows the protocol semi-honestly, and can additionally abort at any time. We prove the following theorem in Appendix F.2.

Theorem 6.1. *Let m, ℓ, t, B be non-negative integers such that $m, \ell, B \geq 1$ and $t \leq m$. Let $p(\cdot)$ be some polynomial. Let Σ_{PKE} be a public key encryption scheme that is IND-CPA secure. Then, the protocol $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}(\mathbf{P}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}})$ described in Figure 17 over Σ_{PKE} UC-realizes $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}(\mathbf{P})$ in the $(\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model against all adversaries that (i) are global, (ii) can corrupt parties, and (iii) can corrupt mix servers in a fail-stop manner according to the following restrictions:*

(1) *For every $j \in [m]$, there is at least a $k_j \in [\ell]$ such that MX_{j,k_j} is honest (i.e., in every cascade, not all mix servers are corrupted).*
(2) *$\big|\{j \mid \exists k \text{ such that } \mathsf{MX}_{j,k} \text{ is corrupted}\}\big| \leq m - t$ (i.e, there are at least t cascades with no corrupted mix servers).*
(3) *$\big|\{j \mid \mathsf{MX}_{j,\ell} \text{ is corrupted}\}\big| < t$ (i.e., the number of corrupted exit servers is less than t).*

6.2 Realizing a universally composable accumulator without trusted party

As mentioned in Section 3, we utilize *signatures of knowledge* so that the voters prove their eligibility without revealing their identity. To achieve scalability by eliminating the dependency between the signature size and the voting population we introduce dynamic accumulators in our construction. Below, we present our ideal accumulator functionality $\mathcal{F}_{\mathsf{acc}}$ that is in the spirit of [6] adjusted to our scenario. Then, we introduce the protocol Π_{acc} that follows the command interface of $\mathcal{F}_{\mathsf{acc}}$. In addition, the instantiation of the accumulator scheme in [57] allows the execution of Π_{acc} without the involvement of a trusted party such as a CRS. The full version of this section along with the formal description of Π_{acc} and the proof of UC realization can be found in Appendix G.

The functionality $\mathcal{F}_{\mathsf{acc}}(\mathbf{P})$:

▪ Upon receiving $(\mathsf{sid}, \textsc{Setup})$ by any uncorrupted party, the functionality requests the accumulator algorithms Gen, Update, WitUp, VerStatus from $\mathcal{S}$. Then, $\mathcal{F}_{\mathsf{acc}}$, records the party as "ready", meaning that is allowed to use the functionality for calls in the future (e.g. insert an element in the accumulator or update a witness) and generates the accumulator's parameters by executing Gen.

▪ Upon receiving $(\mathsf{sid}, \textsc{Update}, \alpha, x)$, it allows every ready party to update their local accumulator value α by inserting the element x. Specifically, $\mathcal{F}_{\mathsf{acc}}$ executes the algorithm Update and obtains the updated accumulated value α^*, the witness w^x of the element x and the update message upmsg, which can be used for updating the outdated witnesses of previous elements that are part of the accumulator. Then, it returns $(\alpha^*, x, w^x, \mathsf{upmsg})$ to the party.

▪Upon receiving $(\mathsf{sid}, \textsc{Wit\_Up}, \alpha_{\mathsf{old}}, \alpha_{\mathsf{new}}, x, w_{\mathsf{old}}, (\mathsf{upmsg}_{\mathsf{old+1}}, \ldots, \mathsf{upmsg}_{\mathsf{new}}))$ from a ready uncorrupted party, it updates the outdated witness w_{old}. More precisely, the functionality executes the function `WitUp` with input the old witness of the element x, w^x, a series of update messages upmsg that occurred from previous updates, the old accumulator value α_{old} when x first inserted, and the most recent one α_{new}. The result is the updated witness w^{x*} which is returned back to the party.

▪Upon receiving $(\mathsf{sid}, \textsc{Ver\_Status}, \alpha, \mathtt{VerStatus'}, x, w)$, the functionality verifies if the given element x is part of the accumulator α by executing `VerStatus` and using the provided witness w. In case the verification succeeds but x is not part of α then a forgery has been occurred and $\mathcal{F}_{\mathsf{acc}}$ outputs the special symbol $\perp$.

The functionality $\mathcal{F}_{\mathsf{acc}}$ is presented formally in Figure 18. The protocol Π_{acc} is presented in Figure 19 and builds upon an accumulator scheme $\Sigma_{\mathsf{acc}} = (\mathsf{Gen}, \mathsf{Update}, \mathtt{WitUp}, \mathtt{VerStatus})$. We prove that the protocol Π_{acc}, when instantiated by an accumulator scheme that satisfies the security properties in [57], UC realizes $\mathcal{F}_{\mathsf{acc}}$ as stated in the next theorem (cf. Appendix G.4).

Theorem 6.2. *The protocol* $\Pi_{\mathsf{acc}}(\mathbf{P}, \mathcal{F}_{\mathsf{CRS}}^{\mathsf{Gen}}, \Sigma_{\mathsf{acc}})$ *decribed in Figure Figure 19 UC-realizes* $\mathcal{F}_{\mathsf{acc}}(\mathbf{P})$ *in the* $\mathcal{F}_{\mathsf{CRS}}^{\mathsf{Gen}}$*-hybrid model if and only if* $\Sigma_{\mathsf{acc}} = (\mathsf{Gen}, \mathsf{Update}, \mathtt{WitUp}, \mathtt{VerStatus})$ *satisfies Correctness (cf. Definition G.1) and Soundness (cf. Definition G.2).*

Moreover, if Σ_{acc} *is instantiated with the scheme in [57], then* $\Pi_{\mathsf{acc}}(\mathbf{P}, \mathcal{F}_{\mathsf{CRS}}^{\mathsf{Gen}}, \Sigma_{\mathsf{acc}})$ *UC-realizes* $\mathcal{F}_{\mathsf{acc}}(\mathbf{P})$ *without trusted party.*

6.3 Realizing $\mathcal{F}_{\mathsf{elig}}$ via accumulators, SoK, and non-interactive commitments

We present the protocol Π_{elig} that UC realizes $\mathcal{F}_{\mathsf{elig}}$. In Π_{elig}, the SA sets up: (i) the accumulator functionality $\mathcal{F}_{\mathsf{acc}}$ used for accumulating the commitments of all voters' credentials; (ii) the non-interactive commitment functionality $\mathcal{F}_{\mathsf{NIC}}$ responsible for generating the voters' credentials; (iii) the signature of knowledge functionality $\mathcal{F}_{\mathsf{SOK}}$ used by each eligible voter to authenticate her ballot.

$\Pi_{\mathsf{elig}}(\mathbf{V}, \mathsf{SA}, \mathcal{F}_{\mathsf{acc}}, \mathcal{F}_{\mathsf{NIC}}, \mathcal{F}_{\mathsf{SOK}}, \mathcal{F}_{\mathsf{BC}}, \mathsf{delay\_cast}, \mathtt{Status})$.

All parties have hard-coded the predicates `Status` and the value delay_cast. Each voter V maintains the list that contains information related to the accumulation of elements L_{info}^V and the list of authenticated committed credentials L_{cred}^V both initially as empty. If at any point a hybrid functionality returns an error or $\perp$, the party forwards the message to $\mathcal{Z}$.

▪Upon receiving $(\mathsf{sid}, \textsc{Eligible}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from $\mathcal{Z}$, if $\mathbf{V}_{\mathsf{elig}} \subset \mathbf{V}$, SA does:

1. It sends $(\mathsf{sid}, \textsc{Setup})$ to $\mathcal{F}_{\mathsf{acc}}$.
2. Upon receiving $(\mathsf{sid}, \textsc{Setup}, \mathsf{shared\_params})$ from $\mathcal{F}_{\mathsf{acc}}$, it stores shared_params and sends $(\mathsf{sid}, \textsc{Com\_Setup\_Ini})$ to $\mathcal{F}_{\mathsf{NIC}}$. Upon receiving $(\mathsf{sid}, \textsc{Com\_Setup\_End}, \mathsf{OK})$ from $\mathcal{F}_{\mathsf{NIC}}$, SA sends $(\mathsf{sid}, \textsc{Setup})$ to $\mathcal{F}_{\mathsf{SoK}}$. Upon receiving $(\mathsf{sid}, \textsc{Algorithms}, \mathtt{Sign}, \mathtt{Verify})$ from $\mathcal{F}_{\mathsf{SoK}}$, it sets $St_{\mathsf{gen}} = \alpha_0$ (extracted from shared_params) and sets $\vec{t} \leftarrow (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})$, and $\mathsf{reg.par} \leftarrow (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t}, St_{\mathsf{gen}})$.
3. It sends $(\mathsf{sid}_{\mathsf{all}}, \textsc{Broadcast}, \mathsf{reg.par})$ to $\mathcal{F}_{\mathsf{BC}}$ for $\mathsf{sid}_{\mathsf{all}} = (\mathsf{sid}, \mathsf{SA} \cup \mathbf{V})$. Upon receiving $(\mathsf{sid}_{\mathsf{all}}, \textsc{Broadcast}, \mathsf{reg.par})$ from $\mathcal{F}_{\mathsf{BC}}$, it returns $(\mathsf{sid}, \textsc{EligPar}, \mathsf{reg.par})$ to $\mathcal{Z}$.

▪Upon receiving $(\mathsf{sid}_{\mathsf{all}}, \textsc{Broadcast}, (\mathsf{SA}, \mathsf{reg.par}))$ from $\mathcal{F}_{\mathsf{BC}}$, V stores reg.par and sets her status to 'Cred'.

▪Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $\mathcal{Z}$ for the first time, V reads Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \top$, then V does:

1. She picks a random cr from the message space $\mathcal{M}$ and sends $(\mathsf{sid}, \textsc{Com\_Commit\_Ini}, \mathsf{cr})$ to $\mathcal{F}_{\mathsf{NIC}}$. Upon receiving $(\mathsf{sid}, \textsc{Com\_Commit\_End}, \hat{\mathsf{cr}}, \mathsf{aux})$ from $\mathcal{F}_{\mathsf{NIC}}$, if $\hat{\mathsf{cr}} \notin D$, she repeats this step until it does. She stores $(\mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux})$.
2. She sends $(\mathsf{sid}_{\mathsf{all}}, \textsc{Broadcast}, \hat{\mathsf{cr}})$ to $\mathcal{F}_{\mathsf{BC}}$. Upon receiving $(\mathsf{sid}_{\mathsf{all}}, \textsc{Broadcast}, (V, \hat{\mathsf{cr}}))$ from $\mathcal{F}_{\mathsf{BC}}$, she appends the pair $(V, \hat{\mathsf{cr}})$ to L_{cred}.

▪Upon receiving $(\mathsf{sid}_{\mathsf{all}}, \textsc{Broadcast}, (V^*, \hat{\mathsf{cr}^*}))$ from $\mathcal{F}_{\mathsf{BC}}$, V reads Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \top$ and $V^* \in \mathbf{V}_{\mathsf{elig}}$, then V appends $(V^*, \hat{\mathsf{cr}^*})$ to L_{cred}.

▪Upon receiving $(\mathsf{sid}, \textsc{Auth\_Ballot}, v)$ from $\mathcal{Z}$, V reads Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \top$, then V does:

1. For all pairs $(V_1, \hat{\mathsf{cr}}_1), \ldots, (V, \hat{\mathsf{cr}} = \hat{\mathsf{cr}}_k), \ldots, (V_{t_{\max}}, \hat{\mathsf{cr}}_{t_{\max}})$ (in that order), V sends $(\mathsf{sid}, \textsc{Update}, \alpha_{t-1}, \hat{\mathsf{cr}}_t)$ to $\mathcal{F}_{\mathsf{acc}}$. Upon receiving $(\mathsf{sid}, \textsc{Update}, \alpha_{t-1}, \hat{\mathsf{cr}}_t, \alpha_t, w_t^{\hat{\mathsf{cr}}}, \mathsf{upmsg}_t)$ from $\mathcal{F}_{\mathsf{acc}}$, she appends the tuple $(V_t, \alpha_t, \hat{\mathsf{cr}}_t, w_t^{\hat{\mathsf{cr}}}, \mathsf{upmsg}_t)$ to L_{info}.
2. She sends $(\mathsf{sid}, \textsc{Wit\_Up}, \alpha_k, \alpha_{t_{\max}}, \hat{\mathsf{cr}}_k, w_k^{\hat{\mathsf{cr}}_k}, (\mathsf{upmsg}_{k+1}, \cdots, \mathsf{upmsg}_{t_{\max}}))$ to $\mathcal{F}_{\mathsf{acc}}$ where $t_{\max}$ the last element in the list L_{info}.
3. Upon receiving $(\mathsf{sid}, \textsc{Wit\_Up}, \alpha_k, \alpha_{t_{\max}}, \hat{\mathsf{cr}}_k, w_k^{\hat{\mathsf{cr}}_k}, (\mathsf{upmsg}_{k+1}, \cdots, \mathsf{upmsg}_{t_{\max}}), w_{\alpha_{t_{\max}}}^{\hat{\mathsf{cr}}_k})$ from $\mathcal{F}_{\mathsf{acc}}$, V sets $St_{fin} = \alpha_{t_{\max}}$ and sends $(\mathsf{sid}, \textsc{Sign}, v, (\mathsf{cr}, \alpha_{t_{\max}}), (\hat{\mathsf{cr}}_k, w_{\alpha_{t_{\max}}}^{\hat{\mathsf{cr}}_k}, \mathsf{aux}))$ to $\mathcal{F}_{\mathsf{SOK}}$.
4. Upon receiving $(\mathsf{sid}, \textsc{Sign}, v, (\mathsf{cr}, \alpha_{t_{\max}}), (\hat{\mathsf{cr}}_k, w_{\alpha_{t_{\max}}}^{\hat{\mathsf{cr}}_k}, \mathsf{aux}), \sigma)$ from $\mathcal{F}_{\mathsf{SOK}}$, V returns $(\mathsf{sid}, \textsc{Auth\_Ballot}, v, \sigma)$ to $\mathcal{Z}$.

▪Upon receiving $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma} = (\mathsf{cr}, \sigma))$ from $\mathcal{Z}$, V sends $(\mathsf{sid}, \textsc{Verify}, v, (\mathsf{cr}, St_{\mathsf{fin}}), \sigma)$ to $\mathcal{F}_{\mathsf{SOK}}$ and returns to $\mathcal{Z}$ whatever it receives.

▪ Upon receiving $(\mathsf{sid}, \textsc{Link\_Ballots}, (v_1, \vec{\sigma}_1 = (\mathsf{cr}_1, \sigma_1)), (v_2, \vec{\sigma}_2 = (\mathsf{cr}_2, \sigma_2)))$ from $\mathcal{Z}$, then V does:

1. She sends $(\mathsf{sid}, \textsc{Verify}, v, \mathsf{cr}_j, \sigma_j)$ for both $j = 1, 2$ to $\mathcal{F}_{\mathsf{SOK}}$. If for both $j = 1, 2$ $\mathcal{F}_{\mathsf{SOK}}$ returns $(\mathsf{sid}, \textsc{Verify}, v, \mathsf{cr}_j, \sigma_j, 1)$, she checks if $\mathsf{cr}_1 = \mathsf{cr}_2$ and sets $b = 1$. If for both j $\mathcal{F}_{\mathsf{SOK}}$ returns $(\mathsf{sid}, \textsc{Verify}, v, \mathsf{cr}_j, \sigma_j, 1)$ and $\mathsf{cr}_1 \neq \mathsf{cr}_2$, she sets $b = 0$.
2. She returns $(\mathsf{sid}, \textsc{Link\_Ballots}, (v_1, \vec{\sigma}_1), (v_2, \vec{\sigma}_2), b)$ to $\mathcal{Z}$.

Figure 3: The eligibility protocol Π_{elig}.

We provide a proof of the following theorem in Appendix H.

THEOREM 6.3. *The protocol* $\Pi_{elig}(\mathbf{V}, \mathsf{SA}, \mathcal{F}_{\text{acc}}, \mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{SOK}}, \mathcal{F}_{\text{BC}}, \text{delay\_cast}, \texttt{Status})$ *described in Figure 3 UC-realizes* $\mathcal{F}_{\text{elig}}(\mathbf{V}, \mathsf{SA}, \text{delay\_cast}, \texttt{Status})$ *in the* $(\mathcal{F}_{acc}, \mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{SOK}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$*-hybrid model.*

6.4 Realizing $\mathcal{F}_{\text{vm}}$ via time-lock encryption and anonymous broadcast

In this Subsection, we construct a real-world protocol Π_{vm} that UC-realizes the vote management functionality $\mathcal{F}_{\text{vm}}$ via the time-lock encryption functionality $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$ as introduced and UC-realized in the $(\mathcal{G}_{\text{clock}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{RO}})$-hybrid model in [3] for the case where the leakage function is defined as $\text{leak}(\text{Cl}) = \text{Cl} + 1$. We provide a proof that Π_{vm} UC-realizes $\mathcal{F}_{\text{vm}}$ in the $\{\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{an.BC}}^{\ell,1,p}, \mathcal{G}_{\text{clock}}\}$-hybrid model, where $\ell = \text{delay\_cast} - 1$ and $p(\lambda)$ is the length of a pair of a ballot v and authentication data $\vec{\sigma}$.

$\underline{\Pi_{\text{vm}}(\mathbf{V}, \mathsf{SA}, \mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{an.BC}}^{\ell,1,p}, \texttt{Status}).}$

- Upon receiving $(\text{sid}, \textsc{Election\_Info}, \mathbf{V}_{\text{elig}}, t_{\text{cast}}, t_{\text{open}})$ for the first time from $\mathcal{Z}$, if $\mathbf{V}_{\text{elig}} \subseteq \mathbf{V}$ and $t_{\text{cast}} < t_{\text{open}}$, SA sets $\text{delay\_cast} \leftarrow \ell + 1$, $\vec{t} \leftarrow (t_{\text{cast}}, t_{\text{open}}, \text{delay\_cast})$ and sends $(\text{sid}_{\text{all}}, \textsc{Broadcast}, \text{vote.par} = (\mathbf{V}_{\text{elig}}, \mathbf{O}, \vec{t}))$ to $\mathcal{F}_{\text{BC}}$, where $\text{sid}_{\text{all}} = (\text{sid}, \mathsf{SA} \cup \mathbf{V})$.
- Upon receiving $(\text{sid}_{\text{all}}, \textsc{Broadcast}, (\mathsf{SA}, \text{vote.par}))$ from $\mathcal{F}_{\text{BC}}$, V stores $(\mathbf{V}_{\text{elig}}, \mathbf{O}, \vec{t})$.
- Upon receiving $(\text{sid}, \textsc{Gen\_Ballot}, o)$ from $\mathcal{Z}$, V does:
  (1) If this is the first time receiving this command, $o \in \mathbf{O}$, and $V \in \mathbf{V}_{\text{elig}}$, V sends $(\text{sid}, \textsc{Enc}, o, t_{\text{open}})$ to $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$. Upon receiving $(\text{sid}, \textsc{Encrypting})$ from $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$, V sends $(\text{sid}, \textsc{Generating})$ to $\mathcal{Z}$.
  (2) Else, V returns to $\mathcal{Z}$ $(\text{sid}, \textsc{Gen\_Ballot}, o, \perp)$.
- Upon receiving $(\text{sid}, \textsc{Retrieve})$ from $\mathcal{Z}$, V sends $(\text{sid}, \textsc{Retrieve})$ to $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$. Upon receiving $(\text{sid}, \textsc{Retrieve}, (o, v, t_{\text{open}}))$ from $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$, she records the tuple $(V, v, o, 1)$ and sends $(\text{sid}, \textsc{Retrieve}, (o, v))$ to $\mathcal{Z}$.
- Upon receiving $(\text{sid}, \textsc{Cast}, v, \vec{\sigma})$ from $\mathcal{Z}$, V reads the time Cl from $\mathcal{G}_{\text{clock}}$. If $\texttt{Status}(\text{leak}(\text{Cl}), \vec{t}, \text{Cast}) = \top$, V does:
  (1) She sends $(\text{sid}, \textsc{Retrieve})$ to $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$. Upon receiving $(\text{sid}, \textsc{Retrieve}, (o', v', t_{\text{open}}))$ from $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$ she records the tuple $(V, v', o', 1)$.
  (2) If there is a tuple of the form $(V, v, \cdot, 1)$ stored and it is the first time receiving this command-message, V sends $(\text{sid}, \textsc{Broadcast}, (v, \vec{\sigma}))$ to $\mathcal{F}_{\text{an.BC}}^{\ell,1,p}$.
  (3) Else, V returns $(\text{sid}, \textsc{Cast}, v, \vec{\sigma}, \perp)$ to $\mathcal{Z}$.
- Upon receiving $(\text{sid}, \textsc{Broadcast}, (v, \vec{\sigma}))$ from $\mathcal{F}_{\text{an.BC}}^{\ell,1,p}$, V^* stores the tuple $(v, \vec{\sigma})$ to $L_{\text{cast}}^{V^*}$.
- Upon receiving $(\text{sid}, \textsc{Open}, v^*)$ from $\mathcal{Z}$, if there is a tuple $(v^*, \vec{\sigma}^*) \in L_{\text{cast}}^{V}$, V sends $(\text{sid}, \textsc{Dec}, v^*, t_{\text{open}})$ to $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$.
  (1) Upon receiving $(\text{sid}, \textsc{Dec}, v^*, t_{\text{open}}, o^*)$ from $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$, V returns the message $(\text{sid}, \textsc{Open}, v^*, o^*)$ to $\mathcal{Z}$.
  (2) Upon receiving $(\text{sid}, \textsc{Dec}, v^*, t_{\text{open}}, \perp)$ from $\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}$, V returns the message $(\text{sid}, \textsc{Open}, v^*, \perp)$ to $\mathcal{Z}$.

Figure 4: The vote management protocol Π_{vm}.

We provide a proof of the following theorem in Appendix I.

THEOREM 6.4. *The protocol* $\Pi_{\text{vm}}(\mathbf{V}, \mathsf{SA}, \mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{an.BC}}^{\ell,1,p}, \texttt{Status})$ *described in Figure 4 UC-realizes* $\mathcal{F}_{\text{vm}}(\mathbf{V}, \mathsf{SA}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$ *in the* $(\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{an.BC}}^{l,1,p}, \mathcal{G}_{\text{clock}})$*-hybrid model, where* $\text{leak}(\text{Cl}) = \text{Cl} + 1$, $\text{delay\_cast} = l + 1$*, and* $p(\lambda)$ *is the length of a pair of a ballot* v *and authentication data* $\vec{\sigma}$.

6.5 A UC realization of $\mathcal{F}_{\text{STE}}$

In this subsection, we conclude our formal reasoning about the UC realization of $\mathcal{F}_{\text{STE}}$. We make the following two observations:

(1) $\mathcal{F}_{\text{elig}}(\mathbf{V}, \mathsf{SA}, \text{delay\_cast}, \texttt{Status})$ *can be realized in the* $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$*-hybrid model.* This is a corollary of Theorem 6.3 and the facts that (i) $\mathcal{F}_{\text{acc}}$ can be realized in the standard model (cf. Theorem 6.2), (ii) $\mathcal{F}_{\text{NIC}}$ can be realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model (cf. [10] and Theorem C.4), and (iii) $\mathcal{F}_{\text{SOK}}$ can be realized in the $\mathcal{F}_{\text{CRS}}$-hybrid model (cf. [17] and Appendix C.2). Let $\tilde{\Pi}_{\text{elig}}(\mathbf{V}, \mathsf{SA}, \mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}}, \text{delay\_cast}, \texttt{Status})$ be the UC realization of $\mathcal{F}_{\text{elig}}$ that derives from Π_{elig} by replacing $\mathcal{F}_{\text{acc}}, \mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{SOK}}$ with their realizations.

(2) $\mathcal{F}_{\text{vm}}(\mathbf{V}, \mathsf{SA}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$ *can be realized in the* $(\mathcal{W}_q(\mathcal{F}_{\text{RO}}^*), \mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$*-hybrid model.* This is a corollary of Theorem 6.4 and the facts that (i) $\mathcal{F}_{\text{TLE}}$ can be realized in the $(\mathcal{W}_q(\mathcal{F}_{\text{RO}}^*), \mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$-hybrid model (cf. [3]), and (ii) $\mathcal{F}_{\text{an.BC}}$ can be realized in the $(\mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$-hybrid model (cf. Theorem 6.1). Let $\tilde{\Pi}_{\text{vm}}(\mathbf{V}, \mathsf{SA}, \mathcal{W}_q(\mathcal{F}_{\text{RO}}^*), \mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{BC}}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$ be the UC realization of $\mathcal{F}_{\text{vm}}$ that derives from Π_{vm} by replacing $\mathcal{F}_{\text{TLE}}, \mathcal{F}_{\text{an.BC}}$ with their realizations.

By the above two observations and Theorem 5.1 we get the following concluding theorem.

THEOREM 6.5. *Let* $\Pi_{E\text{-}\textsc{cclesia}}$ *be the protocol that derives from* $\Pi_{\text{STE}}^{\mathcal{F}_{\text{elig}}, \mathcal{F}_{\text{vm}}}(\mathsf{SA}, \mathbf{V}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$ *by replacing (i)* $\mathcal{F}_{\text{elig}}$ *with* $\tilde{\Pi}_{\text{elig}}(\mathbf{V}, \mathsf{SA}, \mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}}, \text{delay\_cast}, \texttt{Status})$ *and (ii)* $\mathcal{F}_{\text{vm}}$ *with* $\tilde{\Pi}_{\text{vm}}(\mathbf{V}, \mathsf{SA}, \mathcal{W}_q(\mathcal{F}_{\text{RO}}^*), \mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{BC}}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$. $\Pi_{E\text{-}\textsc{cclesia}}$ *UC-realizes* $\mathcal{F}_{\text{STE}}(\mathbf{V}, \mathsf{SA}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$ *in the* $(\mathcal{W}_q(\mathcal{F}_{\text{RO}}^*), \mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$*-hybrid model.*

7 PERFORMANCE CONSIDERATIONS

In this section, we turn to the question of feasibility and scalability of E-cclesia and present our preliminary results in this direction.

We focus here on the overheads incurred as a direct consequence of decentralisation, i.e. the broadcast channels and the tallying of the election.[3] Of course, requiring that each voter computes the tally themselves amounts to requiring that each voter breaks the computational puzzles underlying each ballot. But the computational cost on the voters' end associated with this might not be realistic for all voters. We suggest the deployment of a computationally powerful (not trusted) server, that will perform and announce the computationally expensive operation of TLE decrypting each ballot. This alleviates the need for intensive computations from resource-constrained voting clients. Lighter clients will not compute the tally themselves, instead they will verify the announced puzzle solutions. And anyone can now benefit from this verifiability so that they don't need to re-break puzzles that others have already broken. Furthermore, as a pragmatic decentralised solution for the implementation of the broadcast channels we suggest the use of a distributed ledger, i.e. messages broadcast to all voters in the specification of E-cclesia will be posted on a public distributed ledger such as Ethereum or Tezos.

As described in Sections 3 and 6, to vote, each voter needs to perform the following computations: (a) generate a credential and its commitment, (b) accumulate all credential commitments in a Merkle tree, (c) encrypt her ballot with a TLE scheme, and (d) sign the encrypted ballot with a SoK of the corresponding accumulated credential. To compute the tally, for each ballot each voter needs to (e) verify the SoK, and (f) decrypt the ballot.

We present our implementation and benchmarks of concrete instantiations of the key cryptographic operations E-cclesia voters need to perform, namely time-lock encryption and signatures of knowledge of accumulated credentials. By utilizing a consumer laptop computer for benchmarking (2.5 GHz Intel 7300HQ processor, 16 GB 2133 MHz RAM), we provide evidence of the practicality of self-tallying in small to mid-sized elections (up to 100K voters).

Ballot time-lock encryption. We instantiate the TLE scheme by Pietrzak's VDF [56] combined with AES encryption according to Rivest *et al.*'s scheme [58] (formally analyzed in [59] and [41]). Our instantiation allows fast verification of a solution to the time-lock puzzle. This way, voters are able to open ballots cooperatively, and off-load the computationally heavy task of solving the underlying TLE puzzles to more powerful devices. This approach also allows efficient puzzle generation using the RSA trapdoor. Pietrzak presents a non-interactive proof for solutions to such puzzles. The size of proofs and verification time is logarithmic with the difficulty parameter t, which is proportional to the desired decryption time. Our instantiation of the TLE primitive can be proven to UC realise $\mathcal{F}_{\mathsf{TLE}}$ in the generic group model and the RO modelal by adapting the proof of UC security of Astrolabous in [3], similarly to the UC treatment of Rivest *et al.*'s scheme proposed in [8].

We benchmark TLE solution verification according to two approaches (i) using Pietrzak's construction; (ii) when voters voluntarily reveal the trapdoor (factorization of modulus). We choose $t = 2^{42}$ as a representative difficulty based on the findings of the VDF Alliance FPGA Contest [63]. The results are tabulated in Table 1. Since puzzle solutions can be verified independently, the total verification time scales linearly with the number of ballots, and can be parallelized for improved performance.

Ballots	Pietrzak	Trapdoor	SoK
10	0.277s	0.321s	0.0239s
100	2.44s	2.33s	0.358s
1000	25.3s	23.1s	2.94s
10000	4m	3m47s	23.1s
100000	40m9s	37m55s	3m53s

Table 1: TLE solution verification and SoK verification time (total)

Voters	Tree height	Tree hashing (s)	Signing (s)
10	4	0.000275	0.0858
100	7	0.00166	0.126
1000	10	0.0135	0.168
10000	14	0.139	0.351
100000	17	1.36	1.61

Table 2: SoK signing time (for each voter)

SoK of accumulated credential. For SoK, we use the `gnark` [9] Go library which implements Groth's NIZK proof system [35], due to the relative maturity, easy-of-use, and performance of the library. We expect that other schemes and implementations will have comparable performance[4], hence our benchmarks support the feasibility of deploying SoKs in our STE design. Using the library, we develop a circuit which verifies the knowledge of an accumulated credential, using the Merkle branch and the opening of the credential commitment as witness.

We benchmark SoK verification time with respect to the number of ballots, and tabulate the results in Table 1. Similarly to TLE puzzle solution verification, this scales linearly with the number of ballots.

We also benchmark signing time, which is affected by the height of the Merkle tree used for credential commitment accumulation. Additionally, we benchmark the Merkle root computation time, which is needed as input for the signing and verification procedures, and scales linearly with the number of accumulated elements. We tabulate the results in Table 2.

Concluding remarks. Given that the SoK signing process dominates the ballot creation time, we observe that the computational cost for producing an authenticated ballot is very low. Regarding the overhead for performing tally, we find that the total TLE and SoK verification time is low for small-sized elections and reasonable for mid-sized ones (~100K voters).

REFERENCES

[1] Timothy G. Abbott, Katherine J. Lai, Michael R. Lieberman, and Eric C. Price. 2007. Browser-Based Attacks on Tor. In *Privacy Enhancing Technologies*, Nikita Borisov and Philippe Golle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–199.

[2] Ben Adida. 2008. Helios: Web-based Open-Audit Voting. In *USENIX security symposium*. 335–348.

[3] Other low level primitives mobilised in E-cclesia are considered standard in the e-voting literature.

[4] See performance comparison with `bellman`: https://docs.gnark.consensys.net/en/latest/#gnark-is-fast

[3] Myrto Arapinis, Nikolaos Lamprou, and Thomas Zacharias. 2021. Astrolabous: A Universally Composable Time-Lock Encryption Scheme. In *Advances in Cryptology – ASIACRYPT 2021*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 398–426.

[4] Charles Arthur. 2014. Estonian e-voting shouldn't be used in European elections, say security experts. https://www.theguardian.com/technology/2014/may/12/estonian-e-voting-security-warning-european-elections-research

[5] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO*. 324–356.

[6] Foteini Badimtsi, Ran Canetti, and Sophia Yakoubov. 2020. Universally Composable Accumulators. In *Topics in Cryptology – CT-RSA 2020*, Stanislaw Jarecki (Ed.). Springer International Publishing, Cham, 638–666.

[7] Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach. 2004. Hack-a-Vote: Security Issues with Electronic Voting Systems. *IEEE Security & Privacy* 2, 1 (2004), 32–37. https://doi.org/10.1109/MSECP.2004.1264851

[8] Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. 2021. TARDIS: A Foundation of Time-Lock Puzzles in UC. In *Advances in Cryptology – EUROCRYPT 2021*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer International Publishing, Cham, 429–459.

[9] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. 2022. *ConsenSys/gnark: v0.7.0*. https://doi.org/10.5281/zenodo.5819104 https://github.com/ConsenSys/gnark.

[10] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. 2016. UC Commitments for Modular Protocol Design and Applications to Revocation and Attribute Tokens. In *CRYPTO*.

[11] Jan Camenisch, Anja Lehmann, Gregory Neven, and Alfredo Rial. 2014. Privacy-Preserving Auditing for Attribute-Based Credentials. In *Computer Security - ESORICS 2014*, Mirosław Kutyłowski and Jaideep Vaidya (Eds.). Springer International Publishing, Cham, 109–127.

[12] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *CRYPTO*. 61–76.

[13] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*.

[14] Ran Canetti. 2004. Universally composable signature, certification, and authentication. In *IEEE CSWF*. 219–233.

[15] Ran Canetti, Rafael Pass, and Abhi Shelat. 2007. Cryptography from Sunspots: How to Use an Imperfect Reference String. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*. 249–259. https://doi.org/10.1109/FOCS.2007.70

[16] Barbara Carminati. 2009. *Merkle Trees*. Springer US, Boston, MA, 1714–1715. https://doi.org/10.1007/978-0-387-39940-9_1492

[17] Melissa Chase and Anna Lysyanskaya. 2006. On Signatures of Knowledge. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology* (Santa Barbara, California) *(CRYPTO'06)*. Springer-Verlag, Berlin, Heidelberg, 78–96. https://doi.org/10.1007/11818175_5

[18] David Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* 24, 2 (1981), 84–88. https://doi.org/10.1145/358549.358563

[19] David Chaum. 1988. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptol.* 1, 1 (1988), 65–75. https://doi.org/10.1007/BF00206326

[20] David Chaum. 2001. SureVote: Technical Overview. In *Proceedings of the Workshop on Trustworthy Elections (WOTE)*.

[21] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily Shen, Alan T. Sherman, and Poorvi L. Vora. 2009. Scantegrity II: end-to-end verifiability by voters of optical scan elections through confirmation codes. *IEEE Trans. Information Forensics and Security* 4, 4 (2009), 611–627.

[22] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. 2016. D-DEMOS: A Distributed, End-to-End Verifiable, Internet Voting System. In *ICDCS*. 711–720.

[23] Michael Clarkson, Stephen Chong, and Andrew Myers. 2008. Civitas: A secure remote voting system. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[24] Josh D. Cohen and Michael J. Fischer. 1985. A Robust and Verifiable Cryptographically Secure Election Scheme (Extended Abstract). In *FOCS*. 372–382.

[25] Véronique Cortier, P. Gaudry, and S. Glondu. 2019. Belenios: A Simple Private and Verifiable Electronic Voting System. In *Foundations of Security, Protocols, and Equational Reasoning (Lecture Notes in Computer Science, Vol. 11565)*, Guttman J.and Landwehr C.and Meseguer J.and Pavlovic D. (Ed.). Springer, Cham.

[26] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. 1997. A Secure and Optimally Efficient Multi-Authority Election Scheme. In *EUROCRYPT*, Walter Fumy (Ed.). 103–118.

[27] Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. 2012. Breaking the o (nm) bit barrier: Secure multiparty computation with a static adversary. *CoRR, abs/1203.0289* (2012).

[28] A. De Santis and G. Persiano. 1992. Zero-knowledge proofs of knowledge without interaction. In *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*. 427–436.

[29] Claudia Díaz, Steven J. Murdoch, and Carmela Troncoso. 2010. Impact of Network Topology on Anonymity and Overhead in Low-Latency Anonymity Networks. In *PETS*. 184–201.

[30] Jérôme Dossogne and Frédéric Lafitte. 2013. Blinded Additively Homomorphic Encryption Schemes for Self-Tallying Voting *(SIN '13)*. Association for Computing Machinery, New York, NY, USA, 173–180. https://doi.org/10.1145/2523514.2523542

[31] Juan A. Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. 2011. Adaptively secure broadcast, revisited. In *PODC*. 179–186.

[32] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *J. Comput. System Sci.* 28, 2 (1984), 270–299. https://doi.org/10.1016/0022-0000(84)90070-9

[33] Philippe Golle and Ari Juels. 2004. Dining cryptographers revisited. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 456–473.

[34] Jens Groth. 2004. Efficient Maximal Privacy in Boardroom Voting and Anonymous Broadcast. In *FC*.

[35] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–326.

[36] Gang Han, Yannan Li, Yong Yu, Kim-Kwang Raymond Choo, and Nadra Guizani. 2020. Blockchain-Based Self-Tallying Voting System with Software Updates in Decentralized IoT. *IEEE Netw.* 34, 4 (2020), 166–172. https://doi.org/10.1109/MNET.001.1900439

[37] Feng Hao, Peter Y. A. Ryan, and Piotr Zielinski. 2010. Anonymous voting by two-round public discussion. *IET Inf. Secur.* 4, 2 (2010), 62–67. https://doi.org/10.1049/iet-ifs.2008.0127

[38] Farid Javani and Alan T. Sherman. 2020. BVOT: Self-Tallying Boardroom Voting with Oblivious Transfer. *CoRR* abs/2010.02421 (2020). arXiv:2010.02421 https://arxiv.org/abs/2010.02421

[39] Ari Juels, Dario Catalano, and Markus Jakobsson. 2005. Coercion-resistant electronic elections. In *Proc. of the ACM workshop on privacy in the electronic society*. 61–70.

[40] Selva Karthik. 2020. Introduction to Timing Attacks! https://medium.com/spidernitt/introduction-to-timing-attacks-4e1e8c84b32b

[41] Jonathan Katz, Julian Loss, and Jiayu Xu. 2020. On the Security of Time-Lock Puzzles and Timed Commitments. In *Theory of Cryptography*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer International Publishing, Cham, 390–413.

[42] Dalia Khader, Ben Smyth, Peter Y. A. Ryan, and Feng Hao. 2012. A Fair and Robust Voting System by Broadcast. In *EVOTE*. 285–299.

[43] Aggelos Kiayias and Moti Yung. 2002. Self-tallying elections and perfect ballot secrecy. In *PKC*.

[44] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. 2015. DEMOS-2: Scalable E2E Verifiable Elections without Random Oracles. In *CCS*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). 352–363.

[45] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Neal Koblitz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–113.

[46] Steve Kremer, Mark Ryan, and Ben Smyth. 2010. Election Verifiability in Electronic Voting Protocols. In *ESORICS*. 389–404.

[47] Dragan Lazic and Charlie Obimbo. 2013. Anonymous Broadcast Messages. *International Journal of Advanced Computer Science and Applications* 4 (01 2013). https://doi.org/10.14569/IJACSA.2013.041206

[48] Yannan Li, Willy Susilo, Guomin Yang, Yong Yu, Dongxi Liu, Xiaojiang Du, and Mohsen Guizani. 2022. A Blockchain-Based Self-Tallying Voting Protocol in Decentralized IoT. *IEEE Trans. Dependable Secur. Comput.* 19, 1 (2022), 119–130. https://doi.org/10.1109/TDSC.2020.2979856

[49] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. 2019. Homomorphic Time-Lock Puzzles and Applications. In *CRYPTO*. 620–649.

[50] Joseph Marks. 2022. Are voting machines too vulnerable to hacking? Georgia's having that debate. https://www.washingtonpost.com/politics/2022/02/02/are-voting-machines-too-vulnerable-hacking-georgia-having-that-debate/

[51] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In *FC*. 357–375.

[52] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. 2014. Secure Anonymous Broadcast. In *DISC*.

[53] Jesper Buus Nielsen. 2002. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case. In *CRYPTO*.

[54] Somnath Panja, Samiran Bag, Feng Hao, and Bimal Kumar Roy. 2020. A Smart Contract System for Decentralized Borda Count Voting. *IEEE Trans. Engineering Management* 67, 4 (2020), 1323–1339. https://doi.org/10.1109/TEM.2020.2986371

[55] Torben Pryds Pedersen. 1992. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–140.

[56] Krzysztof Pietrzak. 2018. Simple Verifiable Delay Functions. In *10th Innovations in Theoretical Computer Science Conference (ITCS 2019) (Leibniz International*

Proceedings in Informatics (LIPIcs), Vol. 124)*, Avrim Blum (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 60:1–60:15. https://doi.org/10.4230/LIPIcs.ITCS.2019.60

[57] Leonid Reyzin and Sophia Yakoubov. 2016. Efficient Asynchronous Accumulators for Distributed PKI. In *Proceedings of the 10th International Conference on Security and Cryptography for Networks - Volume 9841*. Springer-Verlag, Berlin, Heidelberg, 292–309. https://doi.org/10.1007/978-3-319-44618-9_16

[58] R. L. Rivest, A. Shamir, and D. A. Wagner. 1996. *Time-lock Puzzles and Timed-release Crypto.* Technical Report. Cambridge, MA, USA.

[59] Lior Rotem and Gil Segev. 2020. Generically Speeding-Up Repeated Squaring Is Equivalent to Factoring: Sharp Thresholds for All Generic-Ring Delay Functions. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 481–509.

[60] Peter Y. A. Ryan and Steve A. Schneider. 2006. Prêt-à-voter with re-encryption mixes. In *ESORICS*.

[61] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 704–737.

[62] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[63] Supranational. 2020. VDF Alliance FPGA Contest Round 2 Results. https://github.com/supranational/vdf-fpga-round2-results

[64] Alan Szepieniec and Bart Preneel. 2015. New Techniques for Electronic Voting. *USENIX Journal of Election Technology and Systems (JETS)*. https://www.usenix.org/conference/jets15/workshop-program/presentation/szepieniec

[65] Michael Waidner and Birgit Pfitzmann. 1989. The Dining Cryptographers in the Disco - Underconditional Sender and Recipient Untraceability with Computationally Secure Serviceability (Abstract). In *EUROCRYPT*, Jean-Jacques Quisquater and Joos Vandewalle (Eds.). 690.

[66] Qingle Wang, Chaohua Yu, Fei Gao, Haoyu Qi, and Qiaoyan Wen. 2016. Self-tallying quantum anonymous voting. *Phys. Rev. A* 94 (Aug 2016), 022333. Issue 2. https://doi.org/10.1103/PhysRevA.94.022333

[67] Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. 2012. Attacking the Washington, D.C. Internet Voting System. In *FC*.

[68] Mahdi Zamani, Jared Saia, Mahnush Movahedi, and Joud Khoury. 2013. Towards Provably-Secure Scalable Anonymous Broadcast. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI 13)*.

[69] Gongxian Zeng, Meiqi He, Siu Ming Yiu, and Zhengan Huang. 2021. A Self-Tallying Electronic Voting Based on Blockchain. *Comput. J.* (2021).

A UNIVERSAL COMPOSABLE FORMALISM

The *Universal Composability (UC)* paradigm introduced by Canetti in [13], which is the state-of-the-art cryptographic model for arguing about the security of protocols when run under concurrent sessions. In the UC framework, the parties engage in a protocol session (labeled by a unique session ID, sid) modeled as interactive Turing Machines (ITMs) that communicate in the presence of an *adversary* ITM $\mathcal{A}$ that may control some of the parties. The protocol execution is scheduled by an *environment* ITM $\mathcal{Z}$ that provides parties with inputs and may interact arbitrarily with $\mathcal{A}$. The intuition here is that (i) $\mathcal{Z}$ captures the external "observer" that aims to break security by interacting with the protocol interface during session sid, while (ii) $\mathcal{A}$ plays the role of the "insider" that helps $\mathcal{Z}$ via any possible information it can obtain through engaging in the session in the back-end of the current execution.

The UC security of a protocol Π follows the *real-world/ideal-world indistinguishability* approach. Namely, security is captured via a special *ideal protocol* that has the same interface as Π that $\mathcal{Z}$ interacts with, but now the parties are "dummy", in the sense that they only forward their inputs provided by $\mathcal{Z}$ to an *ideal functionality* $\mathcal{F}$. The functionality $\mathcal{F}$ is in the center of the back-end (i.e., the ideal protocol has a star topology) and *does not interact* with $\mathcal{Z}$ directly. The ideal functionality $\mathcal{F}$ formalizes a trusted party carrying out the task that Π intends to realize (e.g., secure communication, key agreement, authentication, etc.). The functionality $\mathcal{F}$ interacts with the adversary present in the ideal protocol, usually called a *simulator* $\mathcal{S}$, and this interaction results in a "minimum leakage of information" that determines the ideal level of security that *any protocol* realizing the said task should satisfy (not only Π). For instance, if $\mathcal{F}$ formalizes an ideal secure channel, then the minimum leakage could be the ciphertext length. In case that $\mathcal{Z}$ gives an input to a corrupted party P in the ideal world, the functionality $\mathcal{F}$ passes that message to $\mathcal{S}$ and returns back to P whatever it receives from $\mathcal{S}$. In both executions, if a party has the token and halts, then by convention the token is passed to the environment. We say that the real-world protocol is UC-secure if no environment $\mathcal{Z}$ can distinguish its execution from the one of the ideal protocol managed by $\mathcal{F}$. More formally, let $\mathrm{EXEC}^{\Pi}_{\mathcal{Z},\mathcal{A}}$ denote an execution of a real-world protocol Π in the presence of the adversary $\mathcal{A}$ scheduled by an environment $\mathcal{Z}$, and $\mathrm{EXEC}^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}$ denote an execution of the ideal protocol managed by $\mathcal{F}$ in the presence of a simulator $\mathcal{S}$, again scheduled by $\mathcal{Z}$. The UC security of Π is defined as follows.

Definition A.1 (UC realization [13]). The protocol Π is said to *UC-realize* the ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$, the random variables $\mathrm{EXEC}^{\Pi}_{\mathcal{Z},\mathcal{A}}$ and $\mathrm{EXEC}^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}$ are computationally indistinguishable. More formally:

$$\left|\Pr[\mathrm{EXEC}^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}(\lambda)=1]-\Pr[\mathrm{EXEC}^{\Pi}_{\mathcal{Z},\mathcal{A}}(\lambda)=1]\right|=\mathsf{negl}(\lambda)$$

Composition and modularity. Perhaps the most prominent feature of the UC paradigm is the preservation of security of a protocol that runs concurrently with other protocol instances, or as a subroutine of another (often more complex) execution. In particular, assume a protocol Π that UC-realizes an ideal functionality $\mathcal{F}$ according to Definition A.1, and is used as a subroutine of a "larger" protocol $\tilde{\Pi}$. Then, UC guarantees that if we replace any instance of Π with $\mathcal{F}$, we obtain a "hybrid" protocol, denoted by $\tilde{\Pi}^{\Pi\rightarrow\mathcal{F}}$, that enjoys the same security as $\tilde{\Pi}$. Namely, if $\tilde{\Pi}$ UC-realizes some ideal functionality $\tilde{\mathcal{F}}$, then so does $\tilde{\Pi}^{\Pi\rightarrow\mathcal{F}}$.

The power of composition facilitates the design and analysis of complex cryptographic schemes with a *high-degree of modularity*. Namely, the scheme's formal description can be over the composition of ideal modules that are concurrently executed as subroutines. When a protocol Π using the functionalities $\mathcal{F}_1,\ldots,\mathcal{F}_k$ UC-realizes a functionality $\mathcal{F}$, we say that it does so in the $\{\mathcal{F}_1,\ldots,\mathcal{F}_k\}$*-hybrid model* and we write $\Pi^{\mathcal{F}_1,\ldots,\mathcal{F}_k}$ to clearly denote the hybrid functionalities. For instance, an e-voting system Π_{vote} can be described using the ideal functionalities $\mathcal{F}_{\mathsf{sc}}$, $\mathcal{F}_{\mathsf{auth}}$ and $\mathcal{F}_{\mathsf{BB}}$ that formalize the notions of a secure channel, an authenticated channel, and a Bulletin Board, respectively. In this case, we say that Π_{vote} is UC-secure in the $\{\mathcal{F}_{\mathsf{sc}},\mathcal{F}_{\mathsf{auth}},\mathcal{F}_{\mathsf{BB}}\}$*-hybrid model* and we write $\Pi_{\mathsf{vote}}^{\mathcal{F}_{\mathsf{sc}},\mathcal{F}_{\mathsf{auth}},\mathcal{F}_{\mathsf{BB}}}$ to clearly denote the hybrid functionalities. Furthermore, composition allows us to extend secure modular design into multiple ($\mathsf{poly}(\lambda)$ many) layers, since a protocol that uses a hybrid functionality as a subroutine may itself be the subroutine of another protocol of an "upper layer" until we reach the level of the root ideal protocol (in our example, an ideal e-voting functionality $\mathcal{F}_{\mathsf{vote}}$).

B COMMON FUNCTIONALITIES

We formally present the cryptographic building blocks listed in Subsection 3.1. We stress than in the UC framework, hybrid functionalities may capture more than the abstraction of a UC-secure real-life protocol. Indeed, they can also capture theoretical assumption, as a trusted source of randomness, or supposition about the network structure, as First In First Out channels. Notice that those specific functionalities may be *global*, meaning that they may act as shared states across multiple instances and be accessed by functionalities that do not belong to the current session.

The global clock functionality. This global clock (cf. [5]) can be read at any moment by any involved entity. For each session, the clock advances only when all the involved parties and functionalities in the session make an advance request.

The Global Clock functionality $\mathcal{G}_{\mathsf{clock}}(\mathbf{P}, \mathbf{F})$.

The functionality manages the set $\mathbf{P}$ of registered identities, i.e., parties $P = (\mathsf{pid}, \mathsf{sid})$ and the set $\mathbf{F}$ of registered functionalities (with their session identifier) $(\mathcal{F}, \mathsf{sid})$. For every sid, let $\mathbf{P}_{\mathsf{sid}} = \{(\cdot, \mathsf{sid}) \in \mathbf{P}\} \cap \{P \in \mathbf{P} \mid P \text{ is honest}\}$ and $\mathbf{F}_{\mathsf{sid}} = \{(\cdot, \mathsf{sid}) \in \mathbf{F}\}$.

For each session sid, the functionality initializes the clock variable $\mathsf{Cl}_{\mathsf{sid}} \leftarrow 0$ and the set of advanced entities per round as $L_{\mathsf{sid.adv}} \leftarrow \emptyset$.

- Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from $P \in \mathbf{P}_{\mathsf{sid}}$, if $P \notin L_{\mathsf{sid.adv}}$, then it adds P to $L_{\mathsf{sid.adv}}$. If $L_{\mathsf{sid.adv}} = \mathbf{P}_{\mathsf{sid}} \cup \mathbf{F}_{\mathsf{sid}}$, then it updates $\mathsf{Cl}_{\mathsf{sid}} \leftarrow \mathsf{Cl}_{\mathsf{sid}} + 1$, resets $L_{\mathsf{sid.adv}} \leftarrow \emptyset$ and forwards $(\textsc{sid}_C, \textsc{Advanced\_Clock}, P)$ to $\mathcal{A}$.
- Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \mathsf{sid}) \in \mathbf{F}$, if $(\mathcal{F}, \mathsf{sid}) \notin L_{\mathsf{adv}}$, then it adds $(\mathcal{F}, \mathsf{sid})$ to $L_{\mathsf{sid.adv}}$. If $L_{\mathsf{sid.adv}} = \mathbf{P}_{\mathsf{sid}} \cup \mathbf{F}_{\mathsf{sid}}$, then it updates $\mathsf{Cl}_{\mathsf{sid}} \leftarrow \mathsf{Cl}_{\mathsf{sid}} + 1$, resets $L_{\mathsf{sid.adv}} \leftarrow \emptyset$ and sends $(\textsc{sid}_C, \textsc{Advanced\_Clock}, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\textsc{sid}_C, \textsc{Read\_Clock})$ from any participant (including the environment on behalf of a party, the adversary, or any ideal (shared or local) functionality), it sends $(\textsc{sid}_C, \textsc{Read\_Clock}, \mathsf{Cl}_{\mathsf{sid}})$ to this participant, where sid is the sid of the calling instance.

Figure 5: The global clock functionality $\mathcal{G}_{\mathsf{clock}}(\mathsf{P}, \mathsf{F})$ interacting with the parties of the set P, the functionalities of the set F, the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$.

The random oracle functionality. The random oracle functionality (cf. [53]) can be seen as a trusted source of random input. Given a query, she returns a random value. She also updates a local variable $L_{\mathcal{H}}$ in order to return the same value to similar queries. This functionality can be seen as the "idealisation" of a hash function.

The Random Oracle functionality $\mathcal{F}_{\mathsf{RO}}(A, B)$.

The functionality initializes a list $L_{\mathcal{H}} \leftarrow \emptyset$.

- Upon receiving $(\mathsf{sid}, \textsc{Query}, x)$ from any party P, if $x \in A$, then:
  (1) If there exists a pair $(x, h) \in L_{\mathcal{H}}$, it returns $(\mathsf{sid}, \textsc{Random\_Oracle}, x, h)$ to P.
  (2) Else, it picks $h \in B$ uniformly at random, and it inserts the pair (x, h) to the list $L_{\mathcal{H}}$. Then, it returns $(\mathsf{sid}, \textsc{Random\_Oracle}, x, h)$ to P.

Figure 6: The random oracle functionality $\mathcal{F}_{\mathsf{RO}}$ with respect to a domain A and a range B.

The common reference string functionality. This functionality (cf. [13]) draws a single random string r over an uniform distribution of strings, and then she delivers it upon request.

Note that the functionality waits for the simulator's permission before sending back r to the party, and leak P_i's identity to $\mathcal{S}$. This is often called *public delayed output* in the literature. The intuition is the following: a concrete instantiation of a CRS might be a string on a website. As the string is chosen by the website maintainer, it can be seen by other parties as a random string. Moreover, accessing the website may take some time because of the network, and may leak your IP address. Both time and leakage are captured by the public delayed output.

The Common Reference String functionality $\mathcal{F}^{D}_{\mathsf{CRS}}$.

The functionality initializes a waiting list $L_{\mathsf{wait}} \leftarrow \emptyset$.

- Upon receiving $(\mathsf{sid}, \mathsf{CRS})$ from a party P, if no value r is recorded, it samples r in D, adds P to L_{wait} and sends $(sid, \textsc{Allow}, P)$ to $\mathcal{S}$.
- Upon receiving $(\mathsf{sid}, \textsc{Allowed}, P)$ from $\mathcal{S}$. If $P \in L_{\mathsf{wait}}$, it sends $(\mathsf{sid}, \mathsf{CRS}, r)$ to P and $\mathcal{S}$ and removes P from L_{wait}.

Figure 7: The CRS functionality $\mathcal{F}_{\mathsf{CRS}}$ interacting with the simulator $\mathcal{S}$, parameterized by distribution D.

The broadcast functionality. We use the (authenticated) broadcast functionality $\mathcal{F}_{\mathsf{BC}}$ in [31]. The realization of $\mathcal{F}_{\mathsf{BC}}$ in [31] utilizes the certification functionality in [14], which in turn, can be realized by deploying a certification authority and digital signatures. In our setting, the role of the certification authority can be played by the setup authority (cf. Subsection 3.2) that is active prior to the voting period.

The Broadcast functionality $\mathcal{F}_{BC}(\mathbf{P})$.

- Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, M)$ from $P_i \in \mathbf{P}$, it sends and $(\mathsf{sid}, \textsc{Broadcast}, P_i, M)$ to all parties $P_1, \ldots, P_n$ and $\mathcal{S}$.

Figure 8: The broadcast functionality $\mathcal{F}_{\mathsf{BC}}$ interacting with the parties in $\mathbf{P} = \{P_1, \ldots, P_n\}$ and the simulator $\mathcal{S}$.

The non-interactive commitment functionality. We provide the non-interactive commitment (NIC) functionality $\mathcal{F}_{\mathsf{NIC}}$, as introduced in [10]. As shown in [10], $\mathcal{F}_{\mathsf{NIC}}$ can be realized by using a standard commitment scheme that is binding and has a trapdoor, such as the Pedersen NIC scheme [55]. Namely, a party that (i) commits to a message cm cannot open the commitment to different valid message cm′, and (ii) there is a trapdoor information tk that allows the creation of commitments that can be opened to any message (this implies that the commitment scheme is hiding, i.e., the commitment does not reveal any information about the original message).

The non-interactive commitment functionality $\mathcal{F}_{\mathsf{NIC}}$.

The functionality is parameterized by system parameters sp. The following `COM.TrapCom`,`COM.TrapOpen` and `COM.Verify` are ppt algorithms.

▪ Upon receiving (sid, Com_Setup_Ini) from a party P_i, it does:

1. If (sid, cparcom, `COM.TrapCom`, `COM.TrapOpen`, `COM.Verify`, ctdcom) is already stored, it includes P_i in the set **P**, and sends a delayed output (sid, Com_Setup_End, OK) to P_i.
2. Otherwise, it proceeds to generate a random ssid, stores (ssid, P_i) and sends (sid, Com_Setup_Req, ssid) to $\mathcal{S}$.

▪ Upon receiving (sid, Com_Setup_Alg, ssid, m) from $\mathcal{S}$, it does:

1. If no pair (ssid, P_i) for some P_i is stored, it aborts.
2. It deletes record (ssid, P_i).
3. If (sid, cparcom, `COM.TrapCom`, `COM.TrapOpen`, `COM.Verify`, ctdcom) is already stored, it includes P_i in the set **P** and sends (sid, Com_Setup_End, OK) to P_i.
4. Otherwise, it proceeds as follows:
   (a) It parses m as (cparcom, `COM.TrapCom`, `COM.TrapOpen`, `COM.Verify`, ctdcom).
   (b) It stores (sid, cparcom, `COM.TrapCom`, `COM.TrapOpen`, `COM.Verify`, ctdcom) and initializes both an empty table $\mathsf{Tbl}_{\mathsf{com}}$ and an empty set **P**.
   (c) It includes P_i in the set **P** and sends (sid, Com_Setup_End, OK) to P_i.

▪ Upon receiving (sid, Com_Validate_Ini, ccom) from a party P_i, it does:

1. If $P_i \notin \mathbf{P}$, it aborts.
2. It parses ccom as (ccom′, cparcom′, `COM.Verify`′).
3. It sets $v \leftarrow 1$, if cparcom′ = cparcom and `COM.Verify`′ = `COM.Verify`. Otherwise, it sets $v \leftarrow 0$.
4. It sends (sid, Com_Validate_End, v) to P_i.

▪ Upon receiving (sid, Com_Commit_Ini, cm) from any honest party P_i, it does:

1. If $P_i \notin \mathbf{P}$ or if cm $\notin \mathcal{M}$, where $\mathcal{M}$ is defined in cparcom, it aborts.
2. It computes (ccom, cinfo) ← `COM.TrapCom`(sid, cparcom, ctdcom).
3. If there is an entry [ccom, cm′, copen′, 1] in $\mathsf{Tbl}_{\mathsf{com}}$ such that cm ≠ cm′, it aborts.
4. It computes copen ← `COM.TrapOpen`(sid, cm, cinfo).
5. If `COM.Verify`(sid, cparcom, ccom, cm, copen) ≠ 1, it aborts.
6. It appends [ccom, cm, copen, 1] to $\mathsf{Tbl}_{\mathsf{com}}$.
7. It sets ccom ← (ccom, cparcom, COM.Verify).
8. It sends (sid, Com_Commit_End, ccom, copen) to P_i.

▪ Upon receiving (sid, Com_Verify_Ini, ccom, cm, copen) from any honest party P_i, it does:

1. If $P_i \notin \mathbf{P}$ or if cm $\notin \mathcal{M}$ or if copen $\notin \mathcal{R}$, where $\mathcal{M}$ and $\mathcal{R}$ are defined in cparcom, it aborts.
2. It parses ccom as (ccom′, cparcom′, `COM.Verify`′).
3. If cparcom′ ≠ cparcom or `COM.Verify`′ ≠ `COM.Verify`, it aborts.
4. If there is an entry [ccom′, cm, copen, u] in $\mathsf{Tbl}_{\mathsf{com}}$, it sets $v \leftarrow u$.
5. Else, it proceeds as follows:
   (a) If there is an entry [ccom′, cm′, copen′, 1] in $\mathsf{Tbl}_{\mathsf{com}}$ such that cm ≠ cm′, it sets $v \leftarrow 0$.
   (b) Else, it proceeds as follows:
      (i) It sets v ← `COM.Verify`(sid, cparcom, ccom′, cm, copen).
      (ii) It appends [ccom′, cm, copen, v] to $\mathsf{Tbl}_{\mathsf{com}}$.
6. It sends (sid, Com_Verify_End, v) to P_i.

Figure 9: The non-interactive commitment functionality $\mathcal{F}_{\mathsf{NIC}}$ interacting with the simulator $\mathcal{S}$.

The signature of knowledge functionality. A signature of knowledge (SoK) allows any party who can prove a public statement to sign a message without revealing anything except that the statement is true. A *signature of knowledge* scheme consists of two algorithms, `Sign` and `Verify`. The algorithm `Sign` allows anyone holding a witness w for a statement x in some language L such that $M_L(x, w) = 1$, where M_L is the relation for L, to produce a signature $\sigma_{m,x,L}$ on a message m. The algorithm `Verify` verifies if a signature σ on message m with statement x is valid. The latter implies that the signer is aware of a witness w such that $M_L(x, w) = 1$.

In the UC framework, the notion of SoK is captured by the following functionality (cf. [17]):

The Signature of Knowledge functionality $\mathcal{F}_{\mathsf{SOK}}(L)$.

▪ Upon receiving (sid, Setup) from any party P, it verifies that sid = (M_L, sid′) for some sid′. If not, then it ignores the request. Else, if this is the first time that (sid, Setup) was received, then it sends (sid, Setup) to $\mathcal{S}$. Upon receiving (sid, Algorithms, `Verify`, `Sign`, `SimSign`, `Extract` from $\mathcal{S}$, where `Sign`, `SimSign` and `Extract` are descriptions of PPT TMs, and `Verify` is a description of a deterministic polynomial time TM, it stores these algorithms. It sends (sid, Algorithms, `Sign`, `Verify`) to P.

▪ Upon receiving (sid, Sign, m, x, w) from P, it checks that $M_L(x, w) = 1$. If not, it ignores the requests. Else, it computes σ ← `SimSign(m,x)` and checks that

$\mathtt{Verify}(m, x, \sigma) = 1$. If so, then it records the entry (m, x, σ) and sends $(\mathsf{sid}, \textsc{Signature}, m, x, \sigma)$ to P. Else, it sends $(\mathsf{sid}, \textsc{Completeness\_Error})$ to P and halts.

▪ Upon receiving $(\mathsf{sid}, \textsc{Verify}, m, x, \sigma)$ from some party V, if (m, x, σ') is stored for some σ', then it sends $(\mathsf{sid}, \textsc{Verified}, m, x, \sigma, \mathtt{Verify}(m, x, \sigma))$ to V. Else, it computes $w \leftarrow \mathtt{Extract}(m, x, \sigma)$; if $M_L(x, w) = 1$, it sends $(\mathsf{sid}, \textsc{Verified}, m, x, \sigma, \mathtt{Verify}(m, x, \sigma))$ to V. Else, if $\mathtt{Verify}(m, x, \sigma) = 0$, it sends $(\mathsf{sid}, \textsc{Verified}, m, x, \sigma, 0)$ to V. Else, it sends $(\mathsf{sid}, \textsc{Unforgeability\_Error})$ to V and halts.

Figure 10: The signature of knowledge functionality $\mathcal{F}_{\mathsf{SOK}}$ for language L interacting with the simulator $\mathcal{S}$.

The time-lock encryption functionality. We use the *time-lock encryption* (TLE) functionality from [3] to guarantee that no intermediate results leak before the tally phase.

The time-lock encryption functionality $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak},\mathsf{delay}}$.

It initializes the list of recorded messages/ciphertexts L_{rec} as empty and defines the tag space TAG.

▪ Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{P}_{\mathsf{corr}})$ from $\mathcal{S}$, it records the corrupted set $\mathbf{P}_{\mathsf{corr}}$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Enc}, m, \tau)$ from $P \notin \mathbf{P}_{\mathsf{corr}}$, it reads the time Cl and does:

(1) If $\tau < 0$, it returns $(\mathsf{sid}, \textsc{Enc}, m, \tau, \perp)$ to P.
(2) It picks tag $\overset{\$}{\leftarrow}$ TAG and it inserts the tuple $(m, \mathsf{Null}, \tau, \mathsf{tag}, \mathsf{Cl}, P) \rightarrow L_{\mathsf{rec}}$.
(3) It sends $(\mathsf{sid}, \textsc{Enc}, \tau, \mathsf{tag}, \mathsf{Cl}, 0^{|m|})$ to $\mathcal{S}$. Upon receiving the token back from $\mathcal{S}$ it returns $(\mathsf{sid}, \textsc{Encrypting})$ to P.

▪ Upon receiving $(\mathsf{sid}, \textsc{Update}, \{(c_j, \mathsf{tag}_j)\}_{j=1}^{p(\lambda)})$ from $\mathcal{S}$, for all $c_j \neq \mathsf{Null}$ it updates each tuple $(m_j, \mathsf{Null}, \tau_j, \mathsf{tag}_j, \mathsf{Cl}_j, P)$ to $(m_j, c_j, \tau_j, \mathsf{tag}_j, \mathsf{Cl}_j, P)$

▪ Upon receiving $(\mathsf{sid}, \textsc{Retrieve})$ from P, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and it returns $(\mathsf{sid}, \textsc{Encrypted}, \{(m, c \neq \mathsf{Null}, \tau)\}_{\forall (m,c,\tau,\cdot,\mathsf{Cl}',P) \in L_{\mathsf{rec}} : \mathsf{Cl} - \mathsf{Cl}' \geq \mathsf{delay}})$ to P.

▪ Upon receiving $(\mathsf{sid}, \textsc{Dec}, c, \tau)$ from $P \notin \mathbf{P}_{\mathsf{corr}}$:

(1) If $\tau < 0$, it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau, \perp)$ to P. Else, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and:
  (a) If $\mathsf{Cl} < \tau$, it sends $(\mathsf{sid}, \textsc{Dec}, c, \tau, \textsc{More\_Time})$ to P.
  (b) If $\mathsf{Cl} \geq \tau$, then
    – If there are two tuples $(m_1, c, \tau_1, \cdot, \cdot, \cdot), (m_2, c, \tau_2, \cdot, \cdot, \cdot)$ in L_{rec} such that $m_1 \neq m_2$ and $c \neq \mathsf{Null}$ where $\tau \geq \max\{\tau_1, \tau_2\}$, it returns to P $(\mathsf{sid}, \textsc{Dec}, c, \tau, \perp)$.
    – If no tuple $(\cdot, c, \cdot, \cdot, \cdot, \cdot)$ is recorded in L_{rec}, it sends $(\mathsf{sid}, \textsc{Dec}, c, \tau)$ to $\mathcal{S}$ and returns to P whatever it receives from $\mathcal{S}$.
    – If there is a unique tuple $(m, c, \tau_{\mathsf{dec}}, \cdot, \cdot, \cdot)$ in L_{rec}, then if $\tau \geq \tau_{\mathsf{dec}}$, it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau, m)$ to P. Else, if $\mathsf{Cl} < \tau_{\mathsf{dec}}$, it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau, \textsc{More\_Time})$ to P. Else, if $\mathsf{Cl} \geq \tau_{\mathsf{dec}} > \tau$, it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau, \textsc{Invalid\_Time})$ to P.

▪ Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and returns $(\mathsf{sid}, \textsc{Leakage}, \{(m, c, \tau)\}_{\forall (m,c,\tau \leq \mathsf{leak}(\mathsf{Cl}),\cdot,\cdot,\cdot) \in L_{\mathsf{rec}}})$ to $\mathcal{S}$.

▪ Whatever message it receives from $P \in \mathbf{P}_{\mathsf{corr}}$, it forwards it to $\mathcal{S}$ and vice versa.

Figure 11: The functionality $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak},\mathsf{delay}}$ parameterized by λ, a leakage function leak**, a delay variable** delay **,interacting with simulator $\mathcal{S}$, parties in P, and global clock $\mathcal{G}_{\mathsf{clock}}$.**

The wrapper functionality. We recall the wrapper functionality $\mathcal{W}_q$ in [3], for the special case where the wrapped evaluation functionality is the random oracle $\mathcal{F}_{\mathsf{RO}}$. The wrapper $\mathcal{W}_q$ allows the parties to access $\mathcal{F}_{\mathsf{RO}}$ only up to q times per round (clock tick).

The wrapper functionality $\mathcal{W}_q(\mathcal{F}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{clock}}, \mathbf{P})$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{P}_{\mathsf{corr}})$ from $\mathcal{S}$, it records the corrupted set $\mathbf{P}_{\mathsf{corr}}$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Evaluate}, (x_1, \ldots, x_j))$ from $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$ it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and does:

(1) If there is not a list L^P it creates one, initially as empty. Then it does:
  (a) For every k in $\{1, \ldots, j\}$, it forwards the message $(\mathsf{sid}, \textsc{Evaluate}, x_k)$ to $\mathcal{F}_{\mathsf{RO}}$.
  (b) When it receives back all the corresponding oracle responses $y_1, \ldots, y_j$, it inserts the tuple-$(\mathsf{Cl}, 1) \in L^P$.
  (c) It sends $(\mathsf{sid}, \textsc{Evaluate}, ((x_1, y_1), \ldots, (x_j, y_j)))$ to P.
(2) Else if there is a tuple-$(\mathsf{Cl}, j_{\mathsf{c}}) \in L^P$ with $j_{\mathsf{c}} < q$, then it changes the tuple to $(\mathsf{Cl}, j_{\mathsf{c}} + 1)$, and repeats the above steps 1a,1c.
(3) Else if there is a tuple-$(\mathsf{Cl}^*, j_{\mathsf{c}}) \in L^P$ such that $\mathsf{Cl}^* < \mathsf{Cl}$, it updates the tuple as $(\mathsf{Cl}, 1)$, and repeats the above steps 1a,1b,1c.

▪ Upon receiving $(\mathsf{sid}, \textsc{Evaluate}, (x_1, \ldots, x_j))$ from $P \in \mathbf{P}_{\mathsf{corr}}$ it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and repeats steps 1,3 except that it maintains the same list, named L_{corr}, for all the corrupted parties.

Figure 12: The Functionality wrapper $\mathcal{W}_q$ parameterized by a number of queries q, functionality $\mathcal{F}_{\mathsf{RO}}$, $\mathcal{G}_{\mathsf{clock}}$ and the parties in P.

C REALIZING $\mathcal{F}_{\mathsf{NIC}}$ AND $\mathcal{F}_{\mathsf{SOK}}$ WITHOUT TRUSTED PARTY

C.1 Realizing universally composable non-interactive commitments without trusted party

In this section we present a protocol that UC realises $\mathcal{F}_{\mathsf{NIC}}$ given in Figure 9 in the CRS model. Fist we present the definition of a *non-interactive commitment scheme* and the security properties that need to satisfy in order to argue about its security. Next, we present a theorem that says that any construction that satisfy the presented security properties UC realises $\mathcal{F}_{\mathsf{NIC}}$. Finally, we present such construction and thus we have a concrete UC realization of $\mathcal{F}_{\mathsf{NIC}}$.

Non-interactive commitments: A *non-interactive commitment* scheme (NIC) consists of three algorithms name CSetup; generates the common parameters of the commitment scheme, Com; computes the commitment of a committed value and some auxiliary information necessary for verifying the opening of the commitment VfCom; verifies if a value and a commitment are correlated by providing the value the commitment and the auxiliary information given from Com.

We say that a NIC scheme is *hiding* when the commitment does not reveal nothing about the committed value. Similar, it is said *binding* if the commitment open to a unique value with high probability. Formally as presented in [11]:

Definition C.1 (Binding). A commitment scheme is *binding* if for any PPT adversary $\mathcal{A}$, it holds that:

$$\Pr\left[\begin{array}{l}\mathsf{par}_c \leftarrow \mathtt{CSetup}(1^\lambda); (\mathsf{com}, x, \mathsf{open}, x', \mathsf{open}') \leftarrow \mathcal{A}(\mathsf{par}_c) : \\ (x, x') \in \mathcal{M}^2 \wedge 1 = \mathtt{VfCom}(\mathsf{par}_c, \mathsf{com}, x, \mathsf{open}) \wedge \\ 1 = \mathtt{VfCom}(\mathsf{par}_c, \mathsf{com}, x', \mathsf{open}') \wedge x \neq x'\end{array}\right] \leq \nu(\lambda)$$

Definition C.2 (Hiding). A commitment scheme is *hiding* if for any PPT adversary $\mathcal{A}$ it holds that:

$$\Pr\left[\begin{array}{l}\mathsf{par}_c \leftarrow \mathtt{CSetup}(1^\lambda); (x_0, x_1, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{par}_c); \\ b \hookleftarrow \mathcal{U}[0,1]; (\mathsf{com}, \mathsf{open}) \leftarrow \mathtt{Com}(\mathsf{par}_c, x_b); \\ b' \leftarrow \mathcal{A}(\mathsf{com}) : (x_0, x_1) \in \mathcal{M}^2 \wedge b = b'\end{array}\right] \leq \frac{1}{2} + \nu(\lambda)$$

where in the above definitions par_c is the parameters of the NIC scheme, $\mathcal{M}$ is the message space, $\mathcal{U}[0,1]$ the uniform distribution on $[0,1]$, and com and open are the resulting commitment and its opening respectively and ν is a negligible function.

Trapdoor commitments: With a special type of NIC, which is called *trapdoor* NIC, one can build dummy commitments that are not related to any message. Then, by using the trapdoor information (if it is available to the user) can produce a proof of valid opening to any value. This special type of commitment is very useful in proving security properties (e.g in UC the simulator can equivocate by using the trapdoor information) but at the same time the security breaches if someone knows the trapdoor information (e.g binding properties does not hold). Formally as presented in [11].

Definition C.3 (Trapdoor NIC). There exist polynomial-time algorithms CSimSetup, ComOpen and TrapOpen, where CSimSetup on input 1^λ outputs parameters par_c with trapdoor td_c such that: (1) par_c are indistinguishable from those produced by CSetup, and, (2) for any $x, x' \in \mathcal{M}$ holds:

$$\left|\Pr\left[\begin{array}{l}(\mathsf{par}_c, \mathsf{td}_c) \leftarrow \mathtt{CSimSetup}(1^\lambda); \\ (\mathsf{com}, \mathsf{open}') \leftarrow \mathtt{Com}(\mathsf{par}_c, x'); \\ \mathsf{open} \leftarrow \mathtt{ComOpen}(\mathsf{par}_c, \mathsf{td}_c, x, x', \mathsf{open}') : \\ 1 = \mathcal{A}(\mathsf{par}_c, \mathsf{td}_c, \mathsf{com}, \mathsf{open})\end{array}\right] - \right.$$
$$\left. - \Pr\left[\begin{array}{l}(\mathsf{par}_c, \mathsf{td}_c) \leftarrow \mathtt{CSimSetup}(1^\lambda); \\ (\mathsf{com}, \mathsf{open}) \leftarrow \mathtt{Com}(\mathsf{par}_c, x)); \\ 1 = \mathcal{A}(\mathsf{par}_c, \mathsf{td}_c, \mathsf{com}, \mathsf{open})\end{array}\right]\right| \leq \nu(\lambda)$$

Below, we present the theorem as presented in [10, full version, Theorem 4] that links NIC's security definition with the UC realization of $\mathcal{F}_{\mathsf{NIC}}$.

Theorem C.4. *The construction Π_{NIC} [10] UC realizes $\mathcal{F}_{NIC}$ in the $\mathcal{F}_{CRS}^{\mathtt{CSetup}}$-hybrid model if the underlying NIC scheme* (CSetup, Com, VfCom) *is binding and trapdoor according to Definitions C.1 and C.3, respectively.*

A binding and trapdoor NIC: In [10], it is shown that Pedersen's NIC scheme [55] satisfies both binding and trapdoor, and thus we have a concrete instantiation of a protocol Π_{NIC} that UC realizes $\mathcal{F}_{\mathsf{NIC}}$. The lemma is given bellow.

Lemma C.5. *The Pedersen's non-interactive commitment scheme* (CSetup, Com, VfCom) *is binding and trapdoor as long as the discrete logarithm problem is hard.*

C.2 Realizing universally composable signature of knowledge without trusted party

A signature of knowledge (SoK) allows any party who can prove a public statement to sign a message without revealing anything except that the statement is true. A *signature of knowledge* scheme consists of two algorithms, Sign and Verify. The algorithm Sign allows anyone holding a witness w for a statement x in some language L such that $M_L(x, w) = 1$, where M_L is the relation for L, to produce a signature $\sigma_{m,x,L}$ on a message m. The algorithm Verify verifies if a signature σ on message m with statement x is valid. The latter implies that the signer is aware of a witness w such that $M_L(x, w) = 1$.

In the UC framework, the notion of SoK is captured by functionality $\mathcal{F}_{\mathsf{SOK}}$ [17] (cf. Figure 10).

In E-cclesia, we use signatures of knowledge for authenticating the eligible ballots. Specifically, eligible voters in the casting phase sign their ballots with the knowledge that they belong to the eligibility list without revealing their actual identity. This step ensures that the privacy of the voter is preserved, as the ballot and the voter's identity cannot be linked.

UC realization of $\mathcal{F}_{\mathsf{SOK}}$. We outline the realization of $\mathcal{F}_{\mathsf{SOK}}$ provided in [17]. The supported language is the *universal language* U_p, where for some polynomial p, the statement x would contain a description of a TM M and an instance x' such that $x \in U_p$ iff there exists w such that $M(x', w)$ halts and accepts in time at most $p(|x|)$. The protocol design builds upon (i) a SoK scheme Σ and (ii) the $\mathcal{F}_{\mathsf{CRS}}$ functionality (cf. Fig. 7), parameterized by the distribution of Σ parameters' generation. In [17, Theorem 2.2], it is shown that the said protocol UC-realizes $\mathcal{F}_{\mathsf{SOK}}(U_p)$ in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid model if and only if Σ satisfies a gamed-based definition called *SimExt-security* [17, Definition 2.2]. Subsequently, the authors provide a construction of a SoK scheme with SimExt-security based on two main building blocks: CPA-secure dense cryptosystems [28, 32] and simulation-sound non-interactive zero-knowledge proofs [61]. The latter step completes the realization of $\mathcal{F}_{\mathsf{SOK}}(U_p)$.

D SECURITY PROPERTIES OF E-CCLESIA

We informally discuss the details that render our protocol secure w.r.t. the properties listed in Subsection 3.3.

1) *Correctness*: is achieved by the binding property of the commitment scheme (cf. Definition C.1), the correctness of all other cryptographic primitives, and the availability of the underlying broadcast network. In particular, all ballots will be delivered to all voters and a malicious party cannot create a different valid credential cr' for a broadcast commitment $\hat{\mathsf{cr}}$ that was created originally for the honest voter's credential cr.

2) *Eligibility*: is satisfied by the security of the accumulator (cf. Definition G.2), the unforgeability of the SoK scheme (cf. Figure 10), the binding property of the commitment scheme, and the fact that the voters store only the credential commitments that they received from the eligible voters during the **Credential generation** phase. In addition, network availability and synchronicity is essential, so that the voters agree on (i) the transition between election phases, and (ii) the order of the received commitments that are going to produce the final accumulated value. Given the above, at verification Step 1, the voter is ascertained that no invalid credential has been added to the accumulator.

3)*Fairness:* is achieved by the security of the TLE algorithms. Namely, no broadcast encrypted vote can be decrypted before the **Tally** phase begins (at time t_{open}). Therefore, no party can learn some partial result before that point. This is exactly what TLE offers, decryption by everyone when a specific time has been reached.

4) *Voter Privacy*: is preserved by the anonymity offered by the anonymous broadcast channel (cf. Figure 2), as well as the hiding property of the commitment scheme (cf. Definition C.2). In particular, upon receiving a credential cr during the **Cast** phase, a party cannot link cr to the corresponding commitment $\hat{\mathsf{cr}}$ this party recorded during the **Credential generation** phase.

5) *One voter-one vote*: is guaranteed by the multiple triple elimination Step 2, where the voter performs the pairwise check for possible matching credentials.

6) *Verifiability*: is supported by the security of the authenticated broadcast channel, the unforgeability of the SoK scheme, and the correctness of the TLE scheme.

E MODULAR DESIGN

E.1 The eligibility functionality $\mathcal{F}_{\mathsf{elig}}$

$\mathcal{F}_{\mathsf{elig}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_cast}, \mathtt{Status})$.

The functionality initializes the lists of eligible voters $L_{\mathsf{elig}} \leftarrow \emptyset$, of authenticated ballots of eligible voters $L_{\mathsf{auth}} \leftarrow \emptyset$, the value $St_{\mathsf{fin}} = 0$. Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{S}$, if $\mathbf{V}_{\mathsf{corr}} \subseteq \mathbf{V}$, it fixes $\mathbf{V}_{\mathsf{corr}}$ as the set of corrupted voters.

▪ Upon receiving $(\mathsf{sid}, \textsc{Eligible}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from SA, if $\mathbf{V}_{\mathsf{elig}} \subseteq \mathbf{V}$ and $t_{\mathsf{cast}} < t_{\mathsf{open}}$, it sends $(\mathsf{sid}, \textsc{Setup\_Elig}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Setup\_Elig}, \mathtt{GenCred}, \mathtt{AuthBallot}, \mathtt{VrfyBallot}, \mathtt{UpState}, St_{\mathsf{gen}})$ from $\mathcal{S}$, then:

(1) It sets $\vec{t} \leftarrow (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})$ and $\mathsf{reg.par} := (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t}, St_{\mathsf{gen}})$ as registration parameters.
(2) It sends $(\mathsf{sid}, \textsc{Elig\_Par}, \mathsf{reg.par})$ to all voters in $\mathbf{V}$ and $\mathcal{S}$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \top$, it executes the following steps:

(1) If there is no tuple $(V, \mathsf{cr}', \hat{\mathsf{cr}}', \mathsf{aux}', 1)$ in L_{elig}, it runs $(\mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}) \leftarrow \mathtt{GenCred}(1^\lambda, \mathsf{reg.par})$. If there are tuples $(\cdot, \mathsf{cr}, \cdot, \cdot, \cdot)$ or $(\cdot, \cdot, \hat{\mathsf{cr}}, \cdot, \cdot)$ in L_{elig} or $(\mathsf{cr}, \mathsf{rc}) = \perp$, it sends $(\mathsf{sid}, \textsc{Gen\_Cred}, \perp)$ to V and halts. Else, it adds $(V, \mathsf{cr}, \mathsf{rc}, \mathsf{aux}, 1)$ to L_{elig} after permission of $\mathcal{S}$ via delayed output with $(V, \hat{\mathsf{cr}})$ as information leakage.
(2) It sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}}, \mathsf{sender})$ to V and $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}})$ to all other voters in $\mathbf{V} \setminus \{V\}$ and $\mathcal{S}$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\mathsf{elig}} \cap \mathbf{V}_{\mathsf{corr}}$, it forwards the message $(\mathsf{sid}, \textsc{Gen\_Cred}, V)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux})$ from $\mathcal{S}$, it does:

(1) If there are no tuples $(V, \mathsf{cr}', \hat{\mathsf{cr}}', \mathsf{aux}', 0)$, $(\cdot, \mathsf{cr}, \cdot, \cdot, 1)$ or $(\cdot, \cdot, \hat{\mathsf{cr}}, \cdot, 1)$ in L_{elig}, then it adds $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 0)$ to L_{elig}.
(2) It sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}})$ to all voters in $\mathbf{V} \setminus \{V\}$ and $\mathcal{S}$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Auth\_Ballot}, v)$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$, then it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \top$, it executes the following steps:

(1) If $St_{\mathsf{fin}} = 0$, then it runs $St_{\mathsf{fin}} \leftarrow \mathtt{UpState}(St_{\mathsf{gen}}, \{\hat{\mathsf{cr}} | (\cdot, \cdot, \hat{\mathsf{cr}}, \cdot) \in L_{\mathsf{elig}}\})$.
(2) If there is a tuple $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 1) \in L_{\mathsf{elig}}$ but no $(V, v', \mathsf{cr}, \sigma', 1) \in L_{\mathsf{auth}}$, then it runs $\sigma \leftarrow \mathtt{AuthBallot}(v, \mathsf{cr}, St_{\mathsf{fin}}, \mathsf{reg.par}, \mathsf{aux})$. If $\mathtt{VrfyBallot}(v, \sigma, St_{\mathsf{fin}}, \mathsf{reg.par}) = 0$, it sends $(\mathsf{sid}, \textsc{Auth\_Ballot}, \perp)$ to V and halts. Else, it (i) adds $(V, v, \mathsf{cr}, \sigma, 1)$ to L_{auth}, and (ii) returns $(\mathsf{sid}, \textsc{Auth\_Ballot}, v, \vec{\sigma} = (\mathsf{cr}, \sigma))$ to V.

▪ Upon receiving $(\mathsf{sid}, \textsc{Auth\_Ballot}, V, v, \vec{\sigma} = (\mathsf{cr}, \sigma))$ from $\mathcal{S}$, if there is a tuple $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 0) \in L_{\mathsf{elig}}$, then it adds $(V, v, \mathsf{cr}, \sigma, 0)$ to L_{auth}. It returns $(\mathsf{sid}, \textsc{Auth\_Ballot}, V, v, \vec{\sigma})$ to V.

▪ Upon receiving $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, \vec{\sigma} = (\mathsf{cr}, \sigma))$ from $V \in \mathbf{V}$:

(1) It computes $x \leftarrow \mathtt{VrfyBallot}(v, (\mathsf{cr}, \sigma), St_{\mathsf{fin}}, \mathsf{reg.par})$.
(2) If there is cr such that there are tuples $(\cdot, \mathsf{cr}, \cdot, \cdot, \cdot, 1) \in L_{\mathsf{elig}}$ and $(\cdot, v, \mathsf{cr}, \sigma, 1) \in L_{\mathsf{auth}}$, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, (\mathsf{cr}, \sigma), 1)$ to V.
(3) If $x = 1$ and there is no cr such that there are tuples $(\cdot, \mathsf{cr}, \cdot, \cdot, \cdot, \cdot) \in L_{\mathsf{elig}}$ and $(\cdot, v, \mathsf{cr}, \sigma, \cdot) \in L_{\mathsf{auth}}$, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, (\mathsf{cr}, \sigma), \perp)$ to V and halts.

(4) If $x = 1$ and there are tuples $(\cdot, v, \mathrm{cr}, \sigma, 0)$, $(\cdot, v', \mathrm{cr}', \sigma', 1) \in L_{\mathsf{auth}}$ such that $\mathrm{cr} = \mathrm{cr}'$ and $v \neq v'$, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, (\mathrm{cr}, \sigma), \bot)$ to V and halts.

(5) Else, it sends $(\mathsf{sid}, \textsc{Ver\_Ballot}, v, (\mathrm{cr}, \sigma), x)$ to V.

▪ Upon receiving $\big(\mathsf{sid}, \textsc{Link\_Ballots}, (v_1, (\mathrm{cr}_1, \sigma_1)), (v_2, (\mathrm{cr}_2, \sigma_2))\big)$ from $V \in \mathbf{V}$, if there are tuples $(\cdot, v_1, \mathrm{cr}_1, \sigma_1, \cdot)$, $(\cdot, v_2, \mathrm{cr}_2, \sigma_2, \cdot) \in L_{\mathsf{auth}}$ such that $\mathrm{cr}_1 = \mathrm{cr}_2$, then it sets $x = 1$. If there are such tuples but $\mathrm{cr}_1 \neq \mathrm{cr}_2$, then it sets $x = 0$. Then, it sends $\big(\mathsf{sid}, \textsc{Link\_Ballots}, (v_1, (\mathrm{cr}_1, \sigma_1)), (v_2, (\mathrm{cr}_2, \sigma_2)), x\big)$ to V.

Figure 13: The eligibility functionality $\mathcal{F}_{\mathsf{elig}}$ parameterized by delay_cast, Status**, interacting with the voters in** V, SA**, and simulator $\mathcal{S}$.**

E.2 The vote management functionality $\mathcal{F}_{\mathsf{vm}}$

$\mathcal{F}_{\mathsf{vm}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$.

The functionality initializes as empty the lists of generated ballots L_{gball}, cast ballots L_{cast}, pending for reception ballots L_{pend}, and a list L_{adv} of the (dummy) parties that have submitted an $\textsc{Advance\_Clock}$ message for the current round. Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{S}$, if $\mathbf{V}_{\mathsf{corr}} \subseteq \mathbf{V}$, it fixes $\mathbf{V}_{\mathsf{corr}}$ as the set of corrupted voters.

Each time the functionality receives a command message it executes the *delayed ballot casting* procedure as described below:

Delayed ballot casting: Upon receiving $(\mathsf{sid}/\textsc{sid}_C, \mathcal{I}, \mathsf{input})$ from $V \in \mathbf{V} \setminus \mathbf{V}_{\mathsf{corr}}$, where $\mathcal{I} \in \{\textsc{Gen\_Ballot}, \textsc{Retrieve}, \textsc{Cast}, \textsc{Open}$ $, \textsc{Advance\_Clock}, \textsc{Read\_Clock}\}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathbf{V} \setminus \mathbf{V}_{\mathsf{corr}} \subseteq L_{\mathsf{adv}}$, it sends $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$ to proceed to the next round. Upon receiving $(\textsc{sid}_C, \textsc{Advanced\_Clock}, \mathcal{F}_{\mathsf{vm}})$ from $\mathcal{G}_{\mathsf{clock}}$, it does:

(1) For every triple $(M^*, V^*, \mathsf{Cl}^*) \in L_{\mathsf{pend}}$ such that $\mathsf{Cl} - \mathsf{Cl}^* = \mathsf{delay\_cast}$, it sends $(\mathsf{sid}, \textsc{Cast\_Ballot}, M^*, \mathsf{sender})$ to V^* and $(\mathsf{sid}, \textsc{Cast\_Ballot}, M^*)$ to all voters in $\mathbf{V} \setminus \{V^*\}$ and $\mathcal{S}$. Then, it removes $(M^*, V^*, \mathsf{Cl}^*)$ from L_{pend}.

(2) It sets L_{adv} as empty.

Then, it executes $(\mathsf{sid}/\textsc{sid}_C, \mathcal{I}, \mathsf{input})$ as follows.

▪ Upon receiving $(\mathsf{sid}, \textsc{Election\_Info}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from SA for the first time, if $\mathbf{V}_{\mathsf{elig}} \subseteq \mathbf{V}$ and $t_{\mathsf{cast}} < t_{\mathsf{open}}$ it sets $\vec{t} \leftarrow (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})$ and $\mathsf{vote.par} := (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t})$ as voting parameters and sends $(\mathsf{sid}, \textsc{Election\_Info}, \mathsf{vote.par})$ to SA and $\mathcal{S}$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, o)$ from $V \notin \mathbf{V}_{\mathsf{corr}}$, if $o \in \mathbf{O}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and does:

(1) If there is no tuple $\big(V, v', o', \mathsf{tag}', \mathsf{Cl}', 1\big) \in L_{\mathsf{gball}}$, it (i) picks $\mathsf{tag} \overset{\$}{\leftarrow} \mathsf{TAG}$ and it inserts the tuple $\big(V, \mathsf{Null}, o, \mathsf{tag}, \mathsf{Cl}, 1\big) \rightarrow L_{\mathsf{gball}}$, (ii) sends $(\mathsf{sid}, \textsc{Gen\_Ballot}, \mathsf{tag}, \mathsf{Cl}, 0^{|o|})$ to $\mathcal{S}$. Upon receiving the token back from $\mathcal{S}$, it returns $(\mathsf{sid}, \textsc{Generating})$ to V.

(2) Else, it returns $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, \bot)$ to V.

▪ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot})$ from $V \in \mathbf{V}_{\mathsf{corr}}$, it sends the message $(\mathsf{sid}, \textsc{Gen\_Ballot}, V)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, v, V)$ from $\mathcal{S}$, it sends $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, v)$ to V.

▪ Upon receiving $(\mathsf{sid}, \textsc{Update}, \{(v_j, \mathsf{tag}_j)\}_{j=1}^{p(\lambda)})$ from $\mathcal{S}$ for all $v_j \neq \mathsf{Null}$, if there is a tuple $(\cdot, v_j, \cdot, \cdot, \cdot, 1) \in L_{\mathsf{gball}}$ or if there are $j, j^* \in [1, p(\lambda)]$ such that $v_j = v_{j^*}$, it returns $(\mathsf{sid}, \textsc{Update}, \{(v_j, \mathsf{tag}_j)\}_{j=1}^{p(\lambda)}, \bot)$ to $\mathcal{S}$. Else, it updates each tuple $(V, \mathsf{Null}, o_j, \mathsf{tag}_j, \mathsf{Cl}_j, 1)$ to $(V, v_j, o_j, \mathsf{tag}_j, \mathsf{Cl}_j, 1)$.

▪ Upon receiving $(\mathsf{sid}, \textsc{Retrieve})$ from $V \notin \mathbf{V}_{\mathsf{corr}}$ it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and does:

(1) If there is a tuple $\big(V, v, o, \mathsf{tag}, \mathsf{Cl}', 1\big) \in L_{\mathsf{gball}}$ with $v \neq \mathsf{Null}$ and $\mathsf{Cl} - \mathsf{Cl}' \geq \mathsf{delay\_gen}$, it returns $(\mathsf{sid}, \textsc{Retrieve}, (o, v))$ to V.

(2) Else, it returns $(\mathsf{sid}, \textsc{Retrieve}, \bot)$ to V.

▪ Upon receiving $\big(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma}\big)$ from V, if $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$ it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \top$, it does:

(1) If there is no tuple $(V, v, \cdot, \cdot, \mathsf{Cl}', 1) \in L_{\mathsf{gball}}$ or $\mathsf{Cl} - \mathsf{Cl}' < \mathsf{delay\_gen}$, it returns $(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma}, \bot)$ to V.

(2) If there is no $(V, v', \vec{\sigma}', \mathsf{Cl}', 1) \notin L_{\mathsf{cast}}$, it adds $(V, v, \vec{\sigma}, \mathsf{Cl}, 1)$ to L_{cast} and $((v, \vec{\sigma}), V, \mathsf{Cl})$ to L_{pend}.

(3) If there is a tuple $(V, v', \vec{\sigma}', \mathsf{Cl}', 1)$ in L_{cast}, it returns $(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma}, \bot)$ to V.

▪ Upon receiving $\big(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma}, V\big)$ from $\mathcal{S}$, if $V \in \mathbf{V}_{\mathsf{corr}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \top$, it adds $(V, v, \vec{\sigma}, \mathsf{Cl}, 0)$ to L_{cast} and $((v, \vec{\sigma}), V, \mathsf{Cl})$ to L_{pend}.

▪ Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from a voter $V \in \mathbf{V} \setminus \mathbf{V}_{\mathsf{corr}}$, if $P \notin L_{\mathsf{adv}}$, it adds P to L_{adv} and forwards $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$ on behalf of P.

▪ Upon receiving $(\textsc{sid}_C, \textsc{Read\_Clock})$ from a voter $V \in \mathbf{V} \setminus \mathbf{V}_{\mathsf{corr}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and returns $(\textsc{sid}_C, \textsc{Read\_Clock}, \mathsf{Cl})$ to P.

▪ Upon receiving $(\mathsf{sid}, \textsc{Open}, v)$ from any party $P \in \mathbf{V} \cup \{\mathcal{S}\}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \top$, it does:

(1) If there is a tuple $(V, v, \vec{\sigma}, \cdot, \cdot) \in L_{\mathsf{cast}}$, and a *unique* $(\cdot, v, o, \cdot, ;1) \in L_{\mathsf{gball}}$, it sends $(\mathsf{sid}, \textsc{Open}, v, o)$ to P.

(2) Else, if there is a tuple $(V, v, \vec{\sigma}, \cdot, \cdot) \in L_{\mathsf{cast}}$ but there is no tuple $(V, v, o, , \cdot, \cdot, 1) \in L_{\mathsf{gball}}$, it sends $(\mathsf{sid}, \textsc{Open}, v)$ to $\mathcal{S}$. Then, it sends the reply it gets from $\mathcal{S}$ to P.

▪ Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{\mathsf{Clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \bot$ or $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \top$, then it returns to $\mathcal{S}$ all the triples $(v, o, 1)$ such that $(\cdot, v, o, \cdot, \cdot, 1) \in L_{\mathrm{gball}} \land (\cdot, v, \cdot, \cdot, 1) \in L_{\mathrm{cast}}$.

Figure 14: The vote management functionality $\mathcal{F}_{\mathsf{vm}}$ parameterized by delay_gen, delay_cast, Status**, interacting with the voters in $\mathbf{V}$,** SA **and simulator $\mathcal{S}$.**

E.3 The hybrid STE protocol $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}}}$

Below we present the protocol $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}}}$ that UC realizes $\mathcal{F}_{\mathsf{STE}}$ in the $(\mathcal{F}_{\mathsf{vm}}, \mathcal{F}_{\mathsf{elig}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model (cf. Theorem 5.1). The protocol can be distinct into four phases (similar to $\mathcal{F}_{\mathsf{STE}}$), **Setup**, **Credential generation**, **Cast**, **Tally**.

E.3.1 Protocol description. In the **Setup** phase, SA accepts the set of the eligible voters $\mathbf{V}_{\mathsf{elig}}$, the set of valid election preferences $\mathbf{O}$, and the times that define the duration of the election $(t_{\mathsf{open}}, t_{\mathsf{open}})$ from $\mathcal{Z}$. Then, SA calls both $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$ for setting up the parameters of the election.

Following up, in the **Credential generation** phase each voter V generates their credential upon request from $\mathcal{Z}$. Specifically, V calls $\mathcal{F}_{\mathsf{elig}}$ and either receives the public part of her credential or $\perp$, in case a credential request has been made in the past.

Next, in the **Cast** phase each voter V after receiving a cast ballot request from $\mathcal{Z}$, she generates her ballot by calling $\mathcal{F}_{\mathsf{vm}}$. If the time that is required for ballot generation, delay_gen, is equal to 0, then she retrieves her ballot in the same round from $\mathcal{F}_{\mathsf{vm}}$ and executes the *Cast* procedure. Specifically, she authenticates it by calling $\mathcal{F}_{\mathsf{elig}}$ and broadcasts it by calling $\mathcal{F}_{\mathsf{vm}}$. In any other case she returns Casting to $\mathcal{Z}$.

In case V receives a clock advancement command from $\mathcal{Z}$, she checks if her ballot is generated (e.g. time delay_gen has been elapsed) by sending Retrieve to $\mathcal{F}_{\mathsf{vm}}$. If this is the case, she executes the ***Cast*** procedure.

Finally, in the **Tally** phase, each voter V upon request from $\mathcal{Z}$ produces the election outcome. Specifically, each voter verifies if each one of the cast ballots is originated from eligible voters by calling $\mathcal{F}_{\mathsf{elig}}$. She keeps the ballots that pass the verification of $\mathcal{F}_{\mathsf{elig}}$ and drop the others. Next, for the remaining ballots, she check through $\mathcal{F}_{\mathsf{elig}}$ if more than one ballots are linked to the same voter (cf. Link_Ballots command message). If she found ballots that re-linked to the same voter (without knowing exactly which one), then she keeps the first one in the order they received them (note that the receiving order is the same for every voter). Last, for the remaining ballots she requests a ballot opening by issuing the command message Open to $\mathcal{F}_{\mathsf{vm}}$. The *tally* is the multiset of all ballot openings that are valid election preferences.

$\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$.

Setup.

- Upon receiving $(\mathsf{sid}, \textsc{Election\_Info}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from $\mathcal{Z}$, if $\mathbf{V}_{\mathsf{elig}} \subseteq \mathbf{V}$ and $t_{\mathsf{cast}} < t_{\mathsf{open}}$, SA sends $(\mathsf{sid}, \textsc{Setup\_Info}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{F}_{\mathsf{vm}}$. Else, SA returns $(\mathsf{sid}, \textsc{Election\_Info}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \perp)$ to $\mathcal{Z}$. Upon receiving $(\mathsf{sid}, \textsc{Election\_Info}, \mathsf{vote.par})$ from $\mathcal{F}_{\mathsf{vm}}$, SA sends $(\mathsf{sid}, \textsc{Eligible}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{F}_{\mathsf{elig}}$ which sends reg.par to all voters in $\mathbf{V}$. Upon receiving reg.par from $\mathcal{F}_{\mathsf{elig}}$, each voter $V \in \mathbf{V}$ stores reg.par as the registration parameters and initializes a multiset $\mathbf{T}$ as empty. She also sets $\vec{t} \leftarrow (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})$.

Credential generation. This phase is completely managed by $\mathcal{F}_{\mathsf{elig}}$.

- Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $\mathcal{Z}$, V sends $(\mathsf{sid}, \textsc{Gen\_Cred})$ to $\mathcal{F}_{\mathsf{elig}}$, which in turn sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathrm{cr}})$ to all voters in $\mathbf{V}$ (or sends $(\mathsf{sid}, \textsc{Gen\_Cred}, \perp)$ to V and halts).

Cast. Here, $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{F}_{\mathsf{elig}}$ combined carry out the ballot generation, authentication and casting tasks.

- Upon receiving $\big(\mathsf{sid}, \textsc{Cast}, o\big)$ from $\mathcal{Z}$, V executes the following steps:
  1. She sends $(\mathsf{sid}, \textsc{Gen\_Ballot}, o)$ to $\mathcal{F}_{\mathsf{vm}}$ which replies either with $(\mathsf{sid}, \textsc{Generating})$ or $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, \perp)$. In the second case, she forwards the message to $\mathcal{Z}$.
  2. If $\mathsf{delay\_gen} = 0$ she sends $(\mathsf{sid}, \textsc{Retrieve})$ to $\mathcal{F}_{\mathsf{vm}}$. Upon receiving $(\mathsf{sid}, \textsc{Retrieve}, (o, v))$ from $\mathcal{F}_{\mathsf{vm}}$ she does the *Cast* step as described below.
  3. In any other case, she returns $(\mathsf{sid}, \textsc{Casting})$ to $\mathcal{Z}$.
- Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from $\mathcal{Z}$, V sends $(\mathsf{sid}, \textsc{Retrieve})$ to $\mathcal{F}_{\mathsf{vm}}$. Upon receiving $(\mathsf{sid}, \textsc{Retrieve}, (o, v))$ from $\mathcal{F}_{\mathsf{vm}}$ she does the *Cast* step as described below.
  - ***Cast:*** She sends (sid, $\textsc{Auth\_Ballot}$, v) to $\mathcal{F}_{\mathsf{elig}}$ which replies with the authentication receipt for v as (sid, $\textsc{Auth\_Ballot}$, v, $\vec{\sigma}$) (or sends (sid, $\textsc{Auth\_Ballot}$, $\perp$) to V and halts). Finally, she sends $\big(\mathsf{sid}, \textsc{Cast}, v, \vec{\sigma}\big)$ to $\mathcal{F}_{\mathsf{vm}}$ which broadcasts the message to all voters in $\mathbf{V}$ after delay_cast rounds. In turn, the voters store the received pair $(v, \vec{\sigma})$.

Then, she sends $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$.

Tally. In order for the voter to perform self-tallying, she accesses $\mathcal{F}_{\mathsf{elig}}$ for ballot verification and linkability and $\mathcal{F}_{\mathsf{vm}}$ for ballot opening.

- Upon receiving a message $\big(\mathsf{sid}, \textsc{Tally}\big)$ from $\mathcal{Z}$, if $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \perp$, then V ignores the message. Otherwise, if $\mathbf{T} = \emptyset$, V executes the following steps:
  1. **For** every tuple $\big($sid, $\textsc{Cast\_Ballot}$, v, $\vec{\sigma}\big)$ she has obtained from $\mathcal{F}_{\mathsf{vm}}$, V sends (sid, $\textsc{Ver\_Ballot}$, v, $\vec{\sigma}$) to $\mathcal{F}_{\mathsf{elig}}$ which replies with (sid, $\textsc{Ver\_Ballot}$, v, $\vec{\sigma}$, x), where $x \in \{0, 1, \perp\}$.
     If there is any ballot verification request such that $\mathcal{F}_{\mathsf{elig}}$ replied with $x = \perp$, then V discards that ballot. Otherwise, she includes in her tally set all pairs $(v, \vec{\sigma})$ such that $\mathcal{F}_{\mathsf{elig}}$ replied with $x = 1$.
  2. V discards multiple ballots as follows: for every pair $(v, \vec{\sigma})$, $(v', \vec{\sigma}')$ in her tally set, she sends $\big($sid, $\textsc{Link\_Ballots}$, $(v, \vec{\sigma})$, $(v', \vec{\sigma}')\big)$ to $\mathcal{F}_{\mathsf{elig}}$. If she gets $\big($sid,

$\textsc{Link\_Ballots}, (v, \vec{\sigma}), (v', \vec{\sigma}'), 1\big)$ as a response, then she discards the ballot she received the last out of those two. Clearly, after this pairwise check is completed, all except one of ballots that are linked will be removed from the tally set, so that one voter-one vote is guaranteed.

(3) **For** every pair $(v, \vec{\sigma})$ in the tally set, V sends $(\mathsf{sid}, \textsc{Open}, v)$ to $\mathcal{F}_{\mathsf{vm}}$, which replies with the opening $(\mathsf{sid}, \textsc{Open}, v, o)$. If $o \in \mathbf{O}$, then V adds o to the multi-set of all opened valid preferences (initialized as empty).

(4) Finally, she sets the tally result $\mathbf{T}$ as the multi-set of all opened valid preferences.

V returns $(\mathsf{sid}, \textsc{Tally}, \mathbf{T})$ to $\mathcal{Z}$.

- Upon receiving $(\mathsf{sid}, \textsc{Verify}, \hat{\mathbf{T}})$ from $\mathcal{Z}$, V reads Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Tally}) = \top$, she does:

  (1) If $\mathbf{T} = \emptyset$, she computes the tally multiset as if it received a $(\mathsf{sid}, \textsc{Tally})$ command.

  (2) If $\hat{\mathbf{T}} = \mathbf{T}$, she returns $(\mathsf{sid}, \textsc{Verify}, \hat{\mathbf{T}}, 1)$ to $\mathcal{Z}$. Else, she returns $(\mathsf{sid}, \textsc{Verify}, \hat{\mathbf{T}}, 0)$ to $\mathcal{Z}$.

Figure 15: Description of the protocol $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}}$ parameterized by $\mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status}$ in the $(\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model.

E.3.2 Security of $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}}$.

Theorem 5.1. *The protocol $\Pi_{\mathsf{STE}}^{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$ described in Figure 15 UC-realizes $\mathcal{F}_{\mathsf{STE}}(\mathsf{SA}, \mathbf{V}, \mathsf{delay\_gen}, \mathsf{delay\_cast}, \mathtt{Status})$ in the $(\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model.*

Proof. For every adversary $\mathcal{A}$ we construct a simulator $\mathcal{S}$ such that every environment $\mathcal{Z}$ cannot distinguish the real from the idea execution of the protocol. Below follows the description of $\mathcal{S}$.

$\mathcal{S}$ initializes as empty the lists of generated credentials $L^{\mathcal{S}}_{\mathsf{elig}}$, generated ballots $L^{\mathcal{S}}_{\mathsf{gball}}$, authenticated ballots $L^{\mathcal{S}}_{\mathsf{auth}}$, cast ballots $L^{\mathcal{S}}_{\mathsf{cast}}$, and operates as follows:

Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was $\mathcal{Z}$. Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{elig}}$, $\mathcal{S}$ forwards the same message as if it was from $\mathcal{Z}$ to $\mathcal{A}$. Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{\mathsf{STE}}$.

Upon receiving $(\mathsf{sid}, \textsc{Election\_Info}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from $\mathcal{F}_{\mathsf{STE}}$, $\mathcal{S}$ sets $\vec{t} \leftarrow (t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathsf{delay\_cast})$ and $\mathsf{vote.par} := (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t})$ as voting parameters and sends $(\mathsf{sid}, \textsc{Election\_Info}, \mathsf{vote.par})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$. Upon receiving the permission from $\mathcal{A}$, $\mathcal{S}$ sends $(\mathsf{sid}, \textsc{Setup\_Elig}, \mathbf{V}_{\mathsf{elig}}, \mathbf{O}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{elig}}$. Upon receiving $(\mathsf{sid}, \textsc{Setup\_Elig}, \mathtt{GenCred}, \mathtt{AuthBallot}, \mathtt{VrfyBallot}, \mathtt{UpState}, St_{\mathsf{gen}})$ from $\mathcal{A}$ then:

(1) It sets $\mathsf{reg.par} := (\mathbf{V}_{\mathsf{elig}}, \mathbf{O}, \vec{t}, St_{\mathsf{gen}})$ as registration parameters.

(2) It sends $(\mathsf{sid}, \textsc{Elig\_Par}, \mathsf{reg.par})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{elig}}$ on behalf of every corrupted party to $\mathcal{A}$. If $\mathcal{A}$ returns the token back for every corrupted party then $\mathcal{S}$ sends the message $(\mathsf{sid}, \textsc{Election\_Info\_OK}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{F}_{\mathsf{STE}}$.

Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred}, V)$ from $\mathcal{F}_{\mathsf{STE}}$ for $V \notin \mathbf{V}_{\mathsf{corr}}$, $\mathcal{S}$ does:

(1) It runs $(\mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}) \leftarrow \mathtt{GenCred}(1^{\lambda}, \mathsf{reg.par})$. If there are tuples $(\cdot, \mathsf{cr}, \cdot, \cdot, \cdot)$ or $(\cdot, \cdot, \hat{\mathsf{cr}}, \cdot, \cdot)$ in $L^{\mathcal{S}}_{\mathsf{elig}}$ or $(\mathsf{cr}, \hat{\mathsf{cr}}) = \perp$, it sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \perp)$ to $\mathcal{F}_{\mathsf{STE}}$. Else, it adds $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 1)$ to $L^{\mathcal{S}}_{\mathsf{elig}}$.

(2) It sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{elig}}$. If $\mathcal{A}$ allows the broadcast then $\mathcal{S}$ sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \hat{\mathsf{cr}})$ on behalf of every corrupted party as if it was $\mathcal{F}_{\mathsf{elig}}$ to $\mathcal{A}$. If $\mathcal{A}$ returns the token back for every corrupted party then $\mathcal{S}$ sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathtt{ready})$ to $\mathcal{F}_{\mathsf{STE}}$.

Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, \mathsf{tag}, \mathsf{Cl}, 0^{|o|})$ from $\mathcal{F}_{\mathsf{STE}}$, $\mathcal{S}$ inserts the tuple $\big(V, \mathsf{Null}, o, \mathsf{tag}, \mathsf{Cl}, 1\big) \rightarrow L^{\mathcal{S}}_{\mathsf{gball}}$ for some V previously unused such that there is a tuple $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 1)$ in $L^{\mathcal{S}}_{\mathsf{elig}}$, and sends the message $(\mathsf{sid}, \textsc{Gen\_Ballot}, \mathsf{tag}, \mathsf{Cl}, 0^{|o|})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$. Upon receiving the token back from $\mathcal{A}$ it returns whatever receives from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{STE}}$.

Upon receiving $(\mathsf{sid}, \textsc{Cast\_Ballot}, M)$ from $\mathcal{F}_{\mathsf{STE}}$ it does:

(1) It searches for a tuple $\big(V, M, o, \mathsf{tag}, \mathsf{Cl}, 1\big)$ in $L^{\mathcal{S}}_{\mathsf{gball}}$. For such a V, it picks the tuple $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 1)$ from $L^{\mathcal{S}}_{\mathsf{elig}}$ and updates it as $(V, \mathsf{cr}, \hat{\mathsf{cr}}, \mathsf{aux}, 1, \mathtt{used})$. Observe that, the relationship between cr and V are only known to $\mathcal{S}$.

(2) If $St_{\mathsf{fin}} = 0$, then it runs $St_{\mathsf{fin}} \leftarrow \mathtt{UpState}(St_{\mathsf{gen}}, \{\hat{\mathsf{cr}} | (\cdot, \cdot, \hat{\mathsf{cr}}, \cdot, \cdot) \in L^{\mathcal{S}}_{\mathsf{elig}}\})$.

(3) It runs $\sigma \leftarrow \mathtt{AuthBallot}(v, \mathsf{cr}, St_{\mathsf{fin}}, \mathsf{reg.par}, \mathsf{aux})$. If it holds that $\mathtt{VrfyBallot}(v, \sigma, St_{\mathsf{fin}}, \mathsf{reg.par}, \mathsf{aux}) = 0$, it sends $(\mathsf{sid}, \textsc{Auth\_Ballot}, \perp)$ to $\mathcal{F}_{\mathsf{STE}}$. Else, it adds $(V, v, \mathsf{cr}, \sigma, 1)$ to $L^{\mathcal{S}}_{\mathsf{auth}}$.

(4) It sends $(\mathsf{sid}, \textsc{Cast\_Ballot}, (v, \vec{\sigma} = (\mathsf{cr}, \sigma)))$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$.

Upon receiving $(\mathsf{sid}, \textsc{Update}, \{(v_j, \mathsf{tag}_j)\}_{j=1}^{p(\lambda)})$ from $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$, for all $v_j \neq \mathsf{Null}$, if there is a tuple $(\cdot, v_j, \cdot, \cdot, \cdot, 1)$ or if there are $j, j^* \in [1, p(\lambda)]$ such that $v_j = v_{j^*}$ it sends $(\mathsf{sid}, \textsc{Update}, \{(v_j, \mathsf{tag}_j)\}_{j=1}^{p(\lambda)}, \perp)$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$. Else, it updates each tuple $(\mathsf{Null}, o_j, \mathsf{tag}_j, \mathsf{Cl}_j, 1)$ to $(v_j, o_j, \mathsf{tag}_j, \mathsf{Cl}_j, 1)$. Then it forwards the message to $\mathcal{F}_{\mathsf{STE}}$. Upon receiving $(\mathsf{sid}, \textsc{Opening}, V^*, v)$ from $\mathcal{F}_{\mathsf{STE}}$, it sends $(\mathsf{sid}, \textsc{Open}, v)$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$. Upon receiving $(\mathsf{sid}, \textsc{Open}, v, o)$ from $\mathcal{A}$, it sends $(\mathsf{sid}, \textsc{Opening}, V^*, v, o)$ to $\mathcal{F}_{\mathsf{STE}}$.

Upon receiving $(\mathsf{sid}, \textsc{Tally})$ from $\mathcal{F}_{\mathsf{STE}}$ on behalf of $V \in \mathbf{V}_{\mathsf{corr}}$ it forwards the message to $\mathcal{A}$ as if it was V and replies back to $\mathcal{F}_{\mathsf{STE}}$ whatever it receives from $\mathcal{A}$.

Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{Z}$, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was $\mathcal{Z}$. Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cred}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Cast}) = \mathtt{Status}(\mathsf{Cl}, \vec{t}, \mathsf{Open}) = \perp$, $\mathcal{S}$ sends $(\mathsf{sid}, \textsc{Tally})$ to $\mathcal{F}_{\mathsf{STE}}$. Upon receiving all pairs (v, o) such that $(V, v, o, \mathsf{tag}, \mathsf{Cl}^*, 1) \in L_{\mathsf{gball}} \wedge (V, o, \mathsf{Cl}', 1) \in L_{\mathsf{cast}}$ from $\mathcal{F}_{\mathsf{STE}}$ where L_{gball} and L_{cast} lists that are maintained in $\mathcal{F}_{\mathsf{STE}}$, it returns them to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{vm}}$.

Observe that the distribution of messages in both execution are exactly the same. This completes the proof.

□

F THE ANONYMOUS BROADCAST PROTOCOL $\Pi_{\text{an.BC}}^{m,\ell,t,B,p}$

In Figure 16, we depict the aforementioned stratified mix-net architecture for the special case where $m = \ell = 3$.

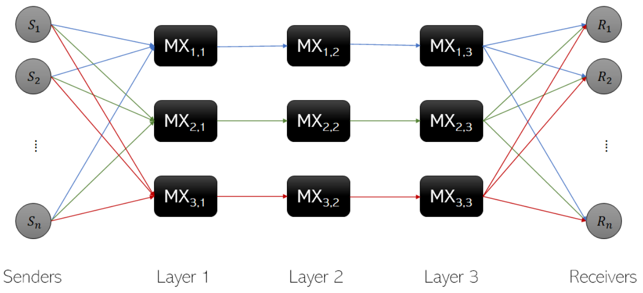

Figure 16: The anonymous broadcast protocol $\Pi_{\text{an.BC}}^{3,3,t,B,p}$ over a 3×3 stratified mix-net. The blue, green, and red arrows illustrate the routing of each of the three message shares.

F.1 Protocol description

The Anonymous BC protocol $\Pi_{\text{an.BC}}^{m,\ell,t,B,p}(\mathbf{P}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{RO}})$.

The hybrid protocol runs over an $m \times \ell$ stratified mix-net with mix servers $\mathsf{MX}_{j,k} \in \mathbf{MX}$, $j \in [m], k \in [\ell]$ as described above. It is parameterized by a public key encryption scheme $\Sigma_{\text{PKE}} = (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ and Shamir's (t, m)-TSS scheme Σ_{tss} [62]. Each party P initializes a counter count_P and a flag setup_P as 0. Upon receiving a command message from $\mathcal{Z}$ and if $\mathsf{setup}_P = 0$, P executes the procedure *Setup* as described below and then executes the command message based on her description.

Setup:

– Upon receiving $(\mathsf{sid}/\textsc{sid}_C, \mathcal{I}, \mathsf{input})$ from $\mathcal{Z}$, where $\mathcal{I} \in \{\textsc{Broadcast}, \textsc{Advance\_Clock}, \textsc{Read\_Clock}\}$, P sends $(\mathsf{sid}_{P,\mathbf{MX}}, \textsc{Broadcast}, \mathsf{setup})$ to $\mathcal{F}_{\text{BC}}$, where $\mathsf{sid}_{P,\mathbf{MX}} = (\mathsf{sid}, \{P\} \cup \mathbf{MX})$.

– Upon receiving $(\mathsf{sid}_{P,\mathbf{MX}}, \textsc{Broadcast}, P, \mathsf{setup})$ from $\mathcal{F}_{\text{BC}}$ for the first time, the mix server $\mathsf{MX}_{j,k}$ reads the time Cl from $\mathcal{G}_{\text{clock}}$ and initializes a local time clock variable as $\mathsf{Cl}_{j,k} \leftarrow \mathsf{Cl}$. It also initializes a list of received messages per round, $L_{\text{pool}}^{j,k}$ and a list of all received messages $L_{\text{rec}}^{j,k}$, as empty. Next, it runs $\mathsf{PKE.Gen}(1^\lambda)$ and obtains a pair of a secret and a public key $(\mathsf{sk}_{j,k}, \mathsf{pk}_{j,k})$. Then, it provides all parties with its public key by sending $(\mathsf{sid}_{\mathsf{MX}_{j,k},\mathbf{P}}, \textsc{Broadcast}, (\mathsf{setup}, \mathsf{pk}_{j,k}))$ to $\mathcal{F}_{\text{BC}}$, where $\mathsf{sid}_{\mathsf{MX}_{j,k},\mathbf{P}} = (\mathsf{sid}, \mathsf{MX}_{j,k} \cup \mathbf{P})$.

– Upon receiving $(\mathsf{sid}_{\mathsf{MX}_{j,k},\mathbf{P}}, \textsc{Broadcast}, \mathsf{MX}_{j,k}, (\mathsf{setup}, \mathsf{pk}_{j,k}))$ from $\mathcal{F}_{\text{BC}}$, P^* stores the pair $(\mathsf{MX}_{j,k}, \mathsf{pk}_{j,k})$. Once she has stored the public keys from all mix servers, P^* sets status_{P^*} to 1.

Subsequently, P executes $(\mathsf{sid}/\textsc{sid}_C, \mathcal{I}, \mathsf{input})$ as described below.

▪ Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, M)$ she reads the time Cl from $\mathcal{G}_{\text{clock}}$. If $\mathsf{status}_P = 1$, $P \in \mathbf{P}$ does:

1. If $\mathsf{count}_P = B$ or $|M| > p(\lambda)$ or the tuple $(\mathsf{Cl}, 1)$ is recorded, then she ignores the message. Else, she increases count_P by 1 and proceeds as follows.
2. She pads M so that $|M| = p(\lambda)$.
3. She randomly chooses a value r from some randomness space $\mathcal{R}$ and sends $(\mathsf{sid}, \textsc{Query}, r)$ to $\mathcal{F}_{\text{RO}}$. Upon receiving $(\mathsf{sid}, \textsc{Random\_Oracle}, r, h)$ from $\mathcal{F}_{\text{RO}}$, where $|h| = p(\lambda)$, she computes the pair $(r, h \oplus M)$.
4. She splits $(r, h \oplus M)$ into m shares $[(r, h \oplus M)]_1, \ldots, [(r, h \oplus M)]_m$.
5. She randomly chooses tag from a space TAG of exponential size with respect to the security parameter λ.
6. For $j = 1, \ldots, m$, she computes an ℓ-level layered encryption of $(\mathsf{tag}, [(r, h \oplus M)]_j)$ as

$$c_j \leftarrow \mathsf{PKE.Enc}\big(\mathsf{pk}_{j,1}, \ldots \\ \ldots, (\mathsf{PKE.Enc}(\mathsf{pk}_{j,\ell}, (\mathsf{tag}, [(r, h \oplus M)]_j))))$$

4. She stores tag in an, initially empty, list of transmitted tags L_{send}^P and stores $(\mathsf{pk}_{j,1}, c_j)$ in an, initially empty, list of pending ciphertexts L_{pool}^P.

▪ Upon receiving $(\textsc{sid}_C, \textsc{Advance\_Clock})$ from $\mathcal{Z}$, P reads the time Cl from $\mathcal{G}_{\text{clock}}$ and records the tuple $(\mathsf{Cl}, 1)$.Then she does:

1. While L_{pool}^P is not empty,
   (a) She picks the first pair $(\mathsf{pk}_{j^*,1}, c_{j^*})$ in L_{pool}^P.
   (b) She sends $(\mathsf{sid}_{P,\mathbf{MX}}, \textsc{Broadcast}, (\mathsf{transmit}, \mathsf{pk}_{j^*,1}, c_{j^*}))$ to $\mathcal{F}_{\text{BC}}$, where $\mathsf{sid}_{P,\mathbf{MX}} = (\mathsf{sid}, \{P\} \cup \mathbf{MX})$.
   (c) She removes $(\mathsf{pk}_{j^*,1}, c_{j^*})$ from L_{pool}^P.
2. She creates as many dummy ciphertexts as to cause a cover traffic effect, i.e., to broadcast exactly B times during the current round. Namely, for $b = 1, \ldots, B - \mathsf{count}_P$:
   (a) She chooses a random tag_b from space TAG.
   (b) She creates the pair $(r_b, h_b \oplus \mathsf{Null})$ for the special message 'Null' of length $p(\lambda)$ via $\mathcal{F}_{\text{RO}}$ as if it was an original message.
   (c) She splits $(r_b, h_b \oplus \mathsf{Null})$ into m shares $[(r_b, h_b \oplus \mathsf{Null})]_1, \ldots, [(r_b, h_b \oplus \mathsf{Null})]_m$.
   (d) For $j = 1, \ldots, m$:
      (i) She computes an ℓ-level layered ciphertext

$$c_{b,j} \leftarrow \mathsf{PKE.Enc}\big(\mathsf{pk}_{j,1}, \ldots, (\mathsf{PKE.Enc} \\ (\mathsf{pk}_{j,\ell}, (\mathsf{tag}_b, [(r_b, h_b \oplus \mathsf{Null})]_j))))$$

      (ii) She sends $(\mathsf{sid}_{P,\mathbf{MX}}, \textsc{Broadcast}, (\mathsf{transmit}, \mathsf{pk}_{j,1}, c_{b,j}))$ to $\mathcal{F}_{\text{BC}}$.

(3) She resets count_P as 0.
(4) She sends $(\textsc{sid}_C, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$ and completes her round.

■ Upon receiving either (i) $(\mathsf{sid}_{P,\mathbf{MX}}, \textsc{Broadcast}, P, (\mathsf{transmit}, \mathsf{pk}_{j,k}, c^*_{j,k}))$ from $\mathcal{F}_{\mathsf{BC}}$ (if $k = 1$), or (ii) $(\mathsf{sid}_{\mathbf{MX}}, \textsc{Broadcast}, \mathsf{MX}_{j,k-1}, (\mathsf{transmit}, c^*_{j,k}))$ from $\mathcal{F}_{\mathsf{BC}}$, where $\mathsf{sid}_{\mathbf{MX}} = (\mathsf{sid}, \mathbf{MX})$ (if $2 \leq k \leq \ell$), the server $\mathsf{MX}_{j,k}$ does:

(1) If $c^*_{j,k} \in L^{j,k}_{\mathsf{rec}}$, then it ignores the message. Else, it adds $c^*_{j,k}$ to $L^{j,k}_{\mathsf{rec}}$ and proceeds as follows.
(2) Consider that the $\ell - (k-1)$-level layered ciphertext $c^*_{j,k}$ is denoted as

$$c^*_{j,k} := \mathsf{PKE.Enc}\big(\mathsf{pk}_{j,k}, \ldots \ldots, (\mathsf{PKE.Enc}(\mathsf{pk}_{j,\ell}, (\mathsf{tag}^*, [C^*]_j)))\big) .$$

$\mathsf{MX}_{j,k}$ decrypts one layer using $\mathsf{sk}_{j,k}$ such that if $k < \ell$, the decryption results in a $\ell - k$-level layered ciphertext

$$c^*_{j,k+1} := \mathsf{PKE.Enc}\big(\mathsf{pk}_{j,k+1}, \ldots \ldots, (\mathsf{PKE.Enc}(\mathsf{pk}_{j,\ell}, (\mathsf{tag}^*, [C^*]_j)))\big) ,$$

whereas if $k = \ell$ (exit server), the decryption results in the plaintext pair $(\mathsf{tag}^*, [C^*]_j)$.
(3) It reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$.
(4) If $\mathsf{Cl} = \mathsf{Cl}_{j,k} + 1$ (i.e., the beginning of a new round occurred), then it does:
   (a) It parses $L^{j,k}_{\mathsf{pool}}$ as $\langle R_1, \ldots, R_{|L^{j,k}_{\mathsf{pool}}|}\rangle$, where $|L^{j,k}_{\mathsf{pool}}|$ is the size of $L^{j,k}_{\mathsf{pool}}$.
   (b) It performs a random permutation $\pi : [|L^{j,k}_{\mathsf{pool}}|] \longrightarrow [|L^{j,k}_{\mathsf{pool}}|]$ on the entries of $L^{j,k}_{\mathsf{pool}}$, i.e., it randomly reorders $L^{j,k}_{\mathsf{pool}}$ as $\langle R_{\pi(1)}, \ldots, R_{\pi(|L^{j,k}_{\mathsf{pool}}|)}\rangle$.
   (c) For $\kappa = 1, \ldots, |L^{j,k}_{\mathsf{pool}}|$:
      - If $k < \ell$ (no exit point), then $\mathsf{MX}_{j,k}$ sends $(\mathsf{sid}_{\mathbf{MX}}, \textsc{Broadcast}, (\mathsf{transmit}, R_{\pi(\kappa)}))$ to $\mathcal{F}_{\mathsf{BC}}$, where $R_{\pi(\kappa)}$ is an $\ell - k$-level layered ciphertext.
      - If $k = \ell$ (exit point), then the exit server $\mathsf{MX}_{j,\ell}$ sends $(\mathsf{sid}_{\mathsf{MX}_{j,\ell},\mathbf{P}}, \textsc{Broadcast}, (\mathsf{transmit}, R_{\pi(\kappa)}))$ to $\mathcal{F}_{\mathsf{BC}}$ where $\mathsf{sid}_{\mathsf{MX}_{j,\ell},\mathbf{P}} = (\mathsf{sid}, \{\mathsf{MX}_{j,\ell}\} \cup \mathbf{P})$ and $R_{\pi(\kappa)}$ is a pair of a tag and the linked share.
   (d) It resets $L^{j,k}_{\mathsf{pool}}$ as empty.
   (e) It advances its local time, i.e., it updates $\mathsf{Cl}_{j,k} \leftarrow \mathsf{Cl}$.
(5) It adds the decryption of $c^*_{j,k}$ (that is either $c^*_{j,k+1}$, if $k < \ell$, or $(\mathsf{tag}^*, [M^*]_j)$, if $k = \ell$) to $L^{j,k}_{\mathsf{pool}}$.

■ Upon receiving $(\mathsf{sid}_{\mathsf{MX}_{j,\ell},\mathbf{P}}, \textsc{Broadcast}, \mathsf{MX}_{j,\ell}, (\mathsf{transmit}, R))$ from $\mathcal{F}_{\mathsf{BC}}$, the party $P \in \mathbf{P}$ does:

(1) She parses R as $(\mathsf{tag}, [(r, C)]_j$ and checks if a triple $(\mathsf{tag}, \cdot, \mathsf{MX}_{j,\ell})$ is already recorded in L^P_{rec}. If so, she aborts.
(2) She adds $(\mathsf{tag}, [(r, C)]_j, \mathsf{MX}_{j,\ell})$ to an, initially empty, list of received messages L^P_{rec}.
(3) She checks if there are at least t tuples of the form $(\mathsf{tag}, [(r, C)]^*, \mathsf{MX}^*)$ in L^P_{rec}. If so, she reconstructs the pair (r, C) from these tuples, and removes every message $(\mathsf{tag}, \cdot, \cdot)$ from L^P_{rec}.
(4) She sends $(\mathsf{sid}, \textsc{Query}, r)$ to $\mathcal{F}_{\mathsf{RO}}$. Upon receiving $(\mathsf{sid}, \textsc{Random\_Oracle}, h)$ from $\mathcal{F}_{\mathsf{RO}}$, she recovers the message by computing $M \leftarrow h \oplus C$ and removing the pads.
(5) If $M = \mathsf{Null}$, then she takes no further action. Else, if $\mathsf{tag} \in L^P_{\mathsf{send}}$, then she returns $(\mathsf{sid}, \textsc{Broadcast}, M, \mathsf{sender})$ to $\mathcal{Z}$. Else, she returns $(\mathsf{sid}, \textsc{Broadcast}, M)$ to $\mathcal{Z}$.

Figure 17: The anonymous broadcast functionality $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}(\mathbf{P}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}})$ parameterized by the mix-net $m \times \ell$ stratified topology, the corruption threshold $t < k$, and the bound B.

F.2 Security analysis

.

Theorem 6.1. *Let m, ℓ, t, B be non-negative integers such that $m, \ell, B \geq 1$ and $t \leq m$. Let $p(\cdot)$ be some polynomial. Let Σ_{PKE} be a public key encryption scheme that is IND-CPA secure. Then, the protocol $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}(\mathbf{P}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}})$ described in Figure 17 over Σ_{PKE} UC-realizes $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}(\mathbf{P})$ in the $(\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model against all adversaries that (i) are global, (ii) can corrupt parties, and (iii) can corrupt mix servers in a fail-stop manner, according to the following restrictions:*

(1) *For every $j \in [m]$, there is at least a $k_j \in [\ell]$ such that MX_{j,k_j} is honest (i.e., in every cascade, not all mix servers are corrupted).*
(2) *$\big|\{j \mid \exists k \text{ such that } \mathsf{MX}_{j,k} \text{ is corrupted}\}\big| \leq m - t$ (i.e, there are at least t cascades with no corrupted mix servers)*[5].
(3) *$\big|\{j \mid \mathsf{MX}_{j,\ell} \text{ is corrupted}\}\big| < t$ (i.e., the number of corrupted exit servers is less than t).*

Proof. First, we construct a simulator that successfully emulates an execution of $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}(\mathbf{P}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}})$ (except from some negligible probability of failure). Then, we will reduce the protocol's security to the IND-CPA security of the underlying encryption scheme Σ_{PKE}.

Constructing the simulator $\mathcal{S}$. We define the simulator $\mathcal{S}$ that operates as follows:

- It emulates a real-world execution for $\mathcal{A}$, itself playing the role of the honest parties and $\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}}$ while acting as a proxy between $\mathcal{A}$ and the environment.
- It normally follows the protocol on behalf any honest mix server.

[5] This restriction can be removed if we consider only semi-honest adversaries where fail-stops do not happen.

- It manages the following data structures:(i) the list of random oracle graph pairs L_{RO}, (ii) the list of activity records L_{table}, (iii) the list of pending messages of corrupted parties L_{corr}, (iv) the list of allocated messages for honest parties L_{hon}, and (v) the list of all pending messages L_{pend}, all initialized as empty. Let $\mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$ be the set of the (emulated) honest parties.
- Upon receiving $(\mathsf{sid}, \textsc{Setup}, P)$ from $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$, if $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$, then it initiates the Setup procedure of $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$ by sending $(\mathsf{sid}_{P,\mathsf{MX}}, \textsc{Broadcast}, P, \mathsf{setup})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{BC}}$. If $\mathcal{A}$ allows completion of Setup, then $\mathcal{S}$ sends $(\mathsf{sid}, \textsc{Setup\_OK})$ to $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$, else it sends $(\mathsf{sid}, \textsc{Setup\_NO})$ to $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$. If $P \in \mathbf{P}_{\mathsf{corr}}$, then it allows the Setup of $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$, only if $\mathcal{A}$ allows the Setup of the emulated execution via the corrupted party P and the corrupted mix servers.
- The simulator upon receiving a broadcast request from $\mathcal{Z}$ for a corrupted party P it forwards the message to $\mathcal{A}$ as if it was $\mathcal{Z}$. Then, upon receiving the token back from $\mathcal{A}$ it randomly chooses a random tag from TAG, reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$, records the tuple $(\mathsf{tag}, P, \mathsf{Cl})$ and sends $(\mathsf{sid}, \textsc{Broadcast}, \mathsf{tag}, P)$ to $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$. When $\mathcal{S}$ receives from $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ a request for the message that corresponds to the value tag, it forwards it to $\mathcal{A}$ as if it was a corrupted mix server and returns back to $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ whatever receives from $\mathcal{A}$.
- Upon receiving $(\textsc{sid}_C, \textsc{Advanced\_Clock}, P)$ from $\mathcal{G}_{\mathsf{clock}}$, if $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$, then it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and emulates a transmission carried by P at time Cl as follows. For $b = 1, \dots, B$:

(1) It chooses a random tag_b from TAG. If tag_b has been reused in a transmission of a (either honest or corrupted) party during the period $[\mathsf{Cl} - \ell, \mathsf{Cl} - 1]$, then it aborts and simulations fails. This is because reuse of tags within this period causes obvious correctness errors during message reconstruction. Note that we can allow the reuse of tags that correspond to messages whose delivery time has passed without any correctness risks (delivered tagged shares are removed from the parties' lists of received messages).

(2) It randomly chooses a value r_b from the exponential-sized domain (query) space and a value C_b of length $p(\lambda)$ from the exponential-sized image (response) space of $\mathcal{F}_{\mathsf{RO}}$. If there is a pair $(r_b, \cdot)$ in L_{RO}, it aborts and simulation fails. The reason is that the value r_b should be fresh in order to be the medium for equivocation, i.e. the "decryption" of C_b to some desired message that $\mathcal{S}$ will receive at time $\mathsf{Cl} + \ell$.Else, it records the pair (r_b, empty) to L_{RO}.

(3) It splits (r_b, C_b) into m shares $[(r_b, C_b)]_1, \dots, [(r_b, C_b)]_m$.

(4) It adds $(r_b, C_b, \mathsf{Cl} + \ell)$ to L_{table}.

(5) For $j = 1, \dots, m$: it computes an ℓ-level layered encryption, $c_{b,j}$, of $(\mathsf{tag}_b, [(r_b, C_b)]_j)$ and sends $(\mathsf{sid}_{P,\mathsf{MX}}, \textsc{Broadcast}, (\mathsf{transmit}, \mathsf{pk}_{j,1}, c_{b,j}))$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\mathsf{BC}}$.

If $\mathcal{A}$ controls the entry server, then $\mathcal{S}$ forwards the message $(\mathsf{sid}_{P,\mathsf{MX}}, \textsc{Broadcast}, (\mathsf{transmit}, \mathsf{pk}_{j,1}, c_{b,j}))$ to $\mathcal{A}$ as if it was the corrupted server that corresponds to the public key $\mathsf{pk}_{j,1}$ and does whatever $\mathcal{A}$ instructs. Note that $\mathcal{A}$ can abort due to fail-stop behavior, so the next mix server will not receive the corresponding encrypted share. Despite that, the message can still be retrieved as enough shares are available based on threat model restriction *ii*) in the theorem statement.

In addition, $\mathcal{S}$ keeps track of the message flow in the simulated stratified mix-network and activates $\mathcal{A}$ when a message reaches a corrupted mix-server and does whatever $\mathcal{A}$ instructs. This is possible because $\mathcal{S}$ knows which servers are corrupted. Again, observe that $\mathcal{A}$ can abort but because of restriction 2), the message will still be retrievable.

- At any moment of the execution, $\mathcal{A}$ may submit queries for the emulated $\mathcal{F}_{\mathsf{RO}}$, so $\mathcal{S}$, who programs the RO, should respond consistently. Upon receiving a query x, $\mathcal{S}$ reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$, it responds by checking the following:

(1) If there is a triple $(r^*, C^*, \mathsf{Cl}^*) \in L_{\mathsf{table}}$ such that (i) $r^* = x$ and (ii) $\mathsf{Cl} < \mathsf{Cl}^*$. If so, then $\mathcal{A}$ has managed to guess (or extract by breaking the underlying crypto) a query necessary for equivocation *before* the expected message delivery time Cl^*, which means that when the parties will receive the messages at time Cl^* this tuple cannot be used for equivocation by $\mathcal{S}$, thus $\mathcal{S}$ aborts and simulation fails.

(2) If there is a triple $(r^*, C^*, \mathsf{Cl}^*) \in L_{\mathsf{table}}$ such that (i) $r^* = x$ and (ii) $\mathsf{Cl} \geq \mathsf{Cl}^*$, then this means that $\mathcal{A}$ makes the query after the expected message delivery time, so $\mathcal{S}$ can use this tuple for equivocation, as we will explain later.

(3) If there is no triple $(x, \cdot, \cdot) \in L_{\mathsf{table}}$, then if there is a pair (x, y) in L_{RO}, then it responds with $(\mathsf{sid}, \textsc{Random\_Oracle}, x, y)$ to $\mathcal{A}$. Else, it chooses a random y^* from the image space of $\mathcal{F}_{\mathsf{RO}}$, adds (x, y^*) to L_{RO}, and responds with $(\mathsf{sid}, \textsc{Random\_Oracle}, x, y^*)$ to $\mathcal{A}$.

- In case that $\mathcal{S}$ receives a ciphertext instead of a message from $\mathcal{A}$ to be broadcast for a corrupted party P, it generates at random a tag and reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. Then it sends $(\mathsf{sid}, \textsc{Broadcast}, \mathsf{tag}, P)$ to $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$. When the functionality requests the actual message from $\mathcal{S}$ when the time has come, $\mathcal{S}$ either reconstructs the message or requests the message from $\mathcal{A}$ and returns to $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ whatever it receives.
- Upon receiving a sequence of messages $(\mathsf{sid}, \textsc{Broadcast}, M_1)$, $\dots, (\mathsf{sid}, \textsc{Broadcast}, M_{N_{\mathsf{Cl}}})$ from $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and creates a random permutation of the messages that will be assigned to honest senders by running the procedure below:

Allocation$(\mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}, B, L_{\mathsf{corr}}, (M_1, \dots, M_{N_{\mathsf{Cl}}}), \mathsf{Cl})$

(1) Set $L_{\mathsf{pend}} \leftarrow \big((M_1, \mathsf{Cl}) \dots, (M_{N_{\mathsf{Cl}}}, \mathsf{Cl})\big)$.

(2) While L_{corr} does not contain any $(\cdot, \mathsf{Cl})$ entry:

   (a) Pick the first message (M^*, Cl) in L_{corr}. This refers to a message M^* that was transmitted ℓ clock ticks earlier by some corrupted party.

   (b) Find the first j^* s.t. $(M_{j^*}, \mathsf{Cl}) = (M^*, \mathsf{Cl})$. Such occurrence is guaranteed since $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ broadcasts all messages from corrupted parties when instructed by $\mathcal{S}$.

   (c) Remove (M^*, Cl) from L_{corr} and (M_{j^*}, Cl) from L_{pend}.

(3) Apply padding so that all messages in L_{pend} are of length $p(\lambda)$.
(4) Repeatedly insert 'Null' messages to L_{pend} until the list contains exactly $|\mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}| \cdot B$ entries.
(5) Randomly permute the $|\mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}| \cdot B$ entries in L_{pend}.
(6) Set $L_{\mathsf{hon}} \leftarrow L_{\mathsf{pend}}$.

- It remains to explain how $\mathcal{S}$ handles RO queries that $\mathcal{A}$ makes after expected delivery time has passed and are necessary for message recovery (e.g., on behalf of a corrupted recipient that has reconstructed the equivocation pairs from the corresponding shares). Upon receiving a query x from $\mathcal{A}$, if there is a triple $(r^*, C^*, \mathsf{Cl}^*) \in L_{\mathsf{table}}$ such that (i) $r^* = x$ and (ii) $\mathsf{Cl} \geq \mathsf{Cl}^*$, then $\mathcal{S}$ picks the first pair formed as (M^*, Cl^*) in L_{hon}. The existence of such pair follows by the facts that (i) after the *Allocation* procedure for time Cl^* is completed, there are exactly $|\mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}| \cdot B$ pairs $(\cdot, \mathsf{Cl}^*) \in L_{\mathsf{hon}}$, and (ii) ℓ clock ticks earlier, exactly B triples $(\cdot, \cdot, \mathsf{Cl}^*) \in L_{\mathsf{table}}$ were created by $\mathcal{S}$ during transmission emulation of every party or every party $P \in \mathbf{P} \setminus \mathbf{P}_{\mathsf{corr}}$. By picking (M^*, Cl^*), $\mathcal{S}$ maps (r^*, C^*) to M^* in a 1-1 manner[6]. Next, $\mathcal{S}$ replies to the query $r^* = x$ by using equivocation as follows: if a pair (r^*, y) is already recorded in L_{RO}, then it responds with y, else it computes $h^* \leftarrow C^* \oplus M^*$ (observe that since C^* is random, h^* is also random), adds (r^*, h^*) to L_{RO} and responds with h^*. This ensures that $\mathcal{A}$ will recover the message $M^* \leftarrow h^* \oplus C^*$, as desired.

Analysis of the simulation. We observe that for every message M, the distribution

$$\{r \overset{\$}{\leftarrow} \mathsf{RO\_Domain}; h \overset{\$}{\leftarrow} \mathsf{RO\_Range} : (r, h \oplus M)\}$$

is identical to the distribution

$$\{r \overset{\$}{\leftarrow} \mathsf{RO\_Domain}; C \overset{\$}{\leftarrow} \mathsf{RO\_Range} : (r, C)\} \ .$$

Therefore, the equivocation that $\mathcal{S}$ applies by programming the RO allows for the perfect emulation of the honest transmission of any message M, unless the following happens: $\mathcal{S}$ aborts because at some point of the emulated transmission for an honesty party it chooses a randomness that is already in L_{RO}, or it chooses tag that was reused up to ℓ rounds earlier.

In addition, another possible deviation of $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$ from the perfect correctness that $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$ offers, is the event where a mix server discards a ciphertext that has received in the past, although it happened that this new ciphertext was honestly generated using the same randomness.

We denote the union of the above events by $\mathsf{Fail\_Repeat}$. Clearly, since the randomness and tag domains are of exponential size and the execution runs in polynomial time, we have that $\Pr[\mathsf{Fail\_Repeat}] = \mathsf{negl}(\lambda)$.

By threat model restriction (i) in the theorem statement, the existence of at least one honest mix server, hence of at least one permutation unknown to the adversary $\mathcal{A}$, in each of the m cascades, guarantees the unlinkability of all the shares of each transmitted value. The latter along with the random allocation of honest transmitted messages during the *Allocation* process, ensures that the said messages are broadcast in random order, just as in $\mathcal{F}^{\ell,B,p}_{\mathsf{an.BC}}$.

By threat model restriction (ii),the adversary $\mathcal{A}$, even when it acts in fail-stop manner, cannot block the routing of at least t shares per message. Thus, message reconstruction is always feasible on the recipient side.

It remains to show that simulation will not fail because $\mathcal{A}$ managed to query the RO for some randomness by recovering a transmitted value *before* all its tagged shares are eventually broadcast to the parties. Unless $\mathcal{A}$ simply guesses the randomness correctly (which happens only with $\mathsf{negl}(\lambda)$ probability), the latter could be achieved if $\mathcal{A}$ managed to break the underlying crypto. However, by the information theoretic security of Shamir's TSS and threat model restriction (iii), the shares that the corrupted exit servers obtain do not suffice for reconstructing the secret. Therefore, $\mathcal{A}$'s strategy should rely on breaking the security of the underlying encryption scheme, Σ_{PKE}. In the following, we show that with overwhelming probability this cannot happen, given that Σ_{PKE} is IND-CPA secure.

Reduction to IND-CPA security. We will reduce the security of $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$ to the security of a public key encryption scheme denoted by $\Sigma^{(z)}_{\mathsf{PKE}}$ that naturally derives from z "iterations" of Σ_{PKE}. Formally, for some z that is polynomial in λ, $\Sigma^{(z)}_{\mathsf{PKE}}$ is defined as follows:

- $\mathsf{PKE.Gen}^{(z)}(1^\lambda)$: run $\mathsf{PKE.Gen}(1^\lambda)$ z times and obtain the pairs of keys $(\mathsf{sk}_1, \mathsf{pk}_1), \ldots, (\mathsf{sk}_z, \mathsf{pk}_z)$. Set $\mathbf{sk} := (\mathsf{sk}_1, \ldots, \mathsf{sk}_z)$ and $\mathbf{pk} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_z)$.
- $\mathsf{PKE.Enc}^{(z)}(\mathbf{pk}, M := (M_1, \ldots, M_z))$: for $j \in [z]$, run $c_j \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_j, M_j)$ and set $c := (c_1, \ldots, c_z)$.
- $\mathsf{PKE.Dec}^{(z)}(\mathbf{sk}, c := (c_1, \ldots, c_z))$: for $j \in [z]$, run $M_j \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_j, c_j)$ and set $M := (M_1, \ldots, M_z)$.

By using a standard hybrid argument, in the following claim, we show that the security of Σ_{PKE} suffices for the security of $\Sigma^{(z)}_{\mathsf{PKE}}$.

Claim F.0.1. *Let z be polynomial in λ. If Σ_{PKE} is IND-CPA secure, then $\Sigma^{(z)}_{\mathsf{PKE}}$ is also IND-CPA secure.*

Proof of Claim F.0.1. We use a contradiction argument for the proof. Let us assume that $\mathcal{B}$ be an IND-CPA adversary against $\Sigma^{(z)}_{\mathsf{PKE}}$ that wins with probability $\frac{1}{2} + \beta(\lambda)$. For $j^* = 1, \ldots, z$, we construct an IND-CPA adversary $\mathcal{B}_{j^*}$ against Σ_{PKE} that emulates the IND-CPA game against $\Sigma^{(z)}_{\mathsf{PKE}}$ as follows:

(1) On input a public key pk, for $j \in [z] \setminus \{j^*\}$, it runs $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{PKE.Gen}^{(z)}(1^\lambda)$. Then, it sets $\mathsf{pk}_{j^*} \leftarrow \mathsf{pk}$. It provides $\mathcal{B}$ with $\mathbf{pk} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_z)$.
(2) It receives two distinct message vectors $(M^0_1, \ldots, M^0_z)$, $(M^1_1, \ldots, M^1_z)$ from $\mathcal{B}$.
(3) For $j < j^*$, it creates an encryption of M^1_j under pk_j, c^1_j.
(4) For $j = j^*$, it sends $(M^0_{j^*}, M^1_{j^*})$ to the IND-CPA challenger and receives an encryption, $c^b_{j^*}$, of $M^b_{j^*}$.
(5) For $j > j^*$, it creates an encryption of M^0_j under pk_j, c^0_j.
(6) It replies to $\mathcal{B}$ with $(c^1_1, \ldots, c^1_{j^*-1}, c^b_{j^*}, c^0_{j^*+1}, \ldots, c^0_z)$.

[6] Note that, implicitly by the description of the *Allocation* procedure, if (r^*, C^*) was created during the transmission of some party P^*, then P^* has been randomly assigned as presumed sender of M^* among the honest parties.

(7) It returns whatever $\mathcal{B}$ outputs.

By the description of $\mathcal{B}_{j^*}$, the following hold:

$$\begin{aligned}\Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=0] &= \Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=0]\\ \Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=1] &= \Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=1]\\ \Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=1] &= \Pr[\mathcal{B}_{j^*+1}(\mathsf{pk}) \to 1|b=0]\end{aligned}$$

Thus, by the assumption for $\mathcal{B}$, we have that

$$\begin{aligned}&\frac{1}{2} + \beta(\lambda) =\\ =&\Pr[\mathcal{B}(\mathbf{pk}) \to 1 \wedge b=1] + \Pr[\mathcal{B}(\mathbf{pk}) \to 0 \wedge b=0] \leq\\ \leq&\Big|\Pr[\mathcal{B}(\mathbf{pk}) \to 1 \wedge b=1] + \Pr[\mathcal{B}(\mathbf{pk}) \to 0 \wedge b=0]\Big| =\\ =&\frac{1}{2}\cdot\Big|\Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=1] + \Pr[\mathcal{B}(\mathbf{pk}) \to 0|b=0]\Big| =\\ =&\frac{1}{2}\cdot\Big|\Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=1] + 1 - \Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=0]\Big| \leq\\ \leq&\frac{1}{2} + \frac{1}{2}\cdot\Big|\Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=1] - \Pr[\mathcal{B}(\mathbf{pk}) \to 1|b=0]\Big| =\\ =&\frac{1}{2} + \frac{1}{2}\cdot\Big|\Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=0]\Big|.\end{aligned}$$

Therefore, we get that

$$\begin{aligned}&2\beta(\lambda) \leq\\ \leq&\Big|\Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=0]\Big| =\\ =&\Big|\Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=1] - \sum_{j^*=2}^{z}\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0] +\\ &+\sum_{j^*=2}^{z}\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0] - \Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=0]\Big| =\\ =&\Big|\Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=1] - \sum_{j^*=2}^{z}\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0] +\\ &+\sum_{j^*=1}^{z-1}\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=0]\Big| =\\ =&\Big|\Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_z(\mathsf{pk}) \to 1|b=0] -\\ &-\sum_{j^*=2}^{z-1}\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0] + \sum_{j^*=2}^{z-1}\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0] +\\ &+\Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_1(\mathsf{pk}) \to 1|b=0]\Big| =\\ =&\Big|\sum_{j^*=1}^{z}\big(\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0]\big)\Big| \leq\\ \leq&\sum_{j^*=1}^{z}\Big|\Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_{j^*}(\mathsf{pk}) \to 1|b=0]\Big|.\end{aligned}$$

By the above inequality and an averaging argument, we get that there exists a $j_0^* \in z$ such that

$$\Big|\Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 1|b=0]\Big| \geq \frac{2\beta(\lambda)}{z}.$$

We study the two following cases:

If it holds that

$$\Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 1|b=1] - \Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 1|b=0] \geq \frac{2\beta(\lambda)}{z},$$

then this directly implies that $\mathcal{B}_{j_0^*}$ wins the IND-CPA security game with probability at least $\frac{1}{2} + \frac{\beta(\lambda)}{z}$.

Else, if it holds that

$$\begin{aligned}&\Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 1|b=0] - \Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 1|b=1] \geq \frac{2\beta(\lambda)}{z} \Leftrightarrow\\ \Leftrightarrow&\Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 0|b=1] - \Pr[\mathcal{B}_{j_0^*}(\mathsf{pk}) \to 0|b=0] \geq \frac{2\beta(\lambda)}{z},\end{aligned}$$

then the adversary $\bar{\mathcal{B}}_{j_0^*}$, that operates exactly like $\mathcal{B}_{j_0^*}$ but flips $\mathcal{B}_{j_0^*}$'s output bit, wins the IND-CPA security game with probability at least $\frac{1}{2} + \frac{\beta(\lambda)}{z}$.

In any case, if $\beta(\cdot)$ is a non-negligible function, then we devise an adversary that wins the IND-CPA game against Σ_{PKE} with non-negligible distinguishing advantage $\frac{\beta(\lambda)}{z}$, which contradicts to the security of Σ_{PKE}.

(End of Proof of Claim F.0.1) ⊣

Reduction to the IND-CPA security of$\Sigma_{\mathsf{PKE}}^{(m-\hat{t})}$. We assume for the sake of contradiction (cf. Definition A.1), that there is an adversary $\mathcal{A}^*$ under the mix server corruption restrictions of the theorem statement, and an environment $\mathcal{Z}^*$ such that for some non-negligible function $\alpha(\cdot)$, it holds that

$$\begin{aligned}\Big|\Pr\big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda)=1\big] -\\ -\Pr\big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda)=1\big]\Big| \geq \alpha(\lambda).\end{aligned} \tag{1}$$

Let $T^*(\lambda)$, or simply T^*, be the running time of $\mathcal{Z}^*$. Let $\mathbf{MX}^*_{\mathsf{exit.corr}}$ be the set of exit mix servers that $\mathcal{A}^*$ corrupts. Let $\hat{t} = |\mathbf{MX}^*_{\mathsf{exit.corr}}|$, where $\hat{t} < t$. For notation simplicity assume that $\mathbf{MX}^*_{\mathsf{exit.corr}} = \{\mathsf{MX}_{1,\ell}, \ldots, \mathsf{MX}_{\hat{t},\ell}\}$. We construct a sequence of IND-CPA adversaries $\mathcal{D}_1, \ldots, \mathcal{D}_{T^*}$ against $\Sigma_{\mathsf{PKE}}^{(m-\hat{t})}$, where $\mathcal{D}_\tau$ operates as follows:

- On input $\mathbf{pk} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_{m-\hat{t}}) \leftarrow \mathsf{PKE.Gen}^{(m-\hat{t})}(1^\lambda)$, it emulates an execution of $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}(\mathbf{P}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{RO}})$ in the presence of $\mathcal{A}^*$ and $\mathcal{Z}^*$, by assigning $\mathsf{pk}_1, \ldots, \mathsf{pk}_{m-\hat{t}}$ as the public key of the honest exit server $\mathsf{MX}_{\hat{t}+1,\ell}, \ldots, \mathsf{MX}_{m,\ell}$, respectively. If $\mathcal{A}^*$ disallows the completion of the Setup procedure, then $\mathcal{D}_\tau$ stops the emulation and returns a random bit to the IND-CPA challenger. Otherwise, for any honest mix server that is not an exit server, it normally creates a pair of a public and a secret key by running $\mathsf{PKE.Gen}(1^\lambda)$.
- For some execution, let $M_1, \ldots, M_{T'}$ be the number of messages that are broadcast by the honest parties (including 'Null' messages), as scheduled by $\mathcal{Z}^*$. Here, T' is upper bounded by T^*B (this bound is reached in case $\mathcal{Z}^*$ always instructs parties to Advance_Clock). If $T' = 0$, then $\mathcal{D}_\tau$ stops the emulation and returns a random bit to the IND-CPA challenger. If $T' \geq 1$, then it emulates honest transmission of M_w, $w = 1, \ldots, T'$ as follows:
  ∗ If $w < \tau$, then it transmits M_w as the instructed honest party would do by following the steps in $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$.
  ∗ If $w = \tau$, then it does:
  (1) Like in $\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}$, it (i) chooses a random value r^0_τ and (ii) programs $\mathcal{F}_{\mathsf{RO}}$ by choosing a random value h_τ and recording the pair (r^1_τ, h_τ).

(2) It normally creates the shares of $(r^1_\tau, h_\tau \oplus M_\tau)$ according to Shamir's secret sharing scheme; namely, it chooses a random $t-1$ degree polynomial $f(x) = a_0 + a_1 x + \cdots + a_{t-1}x^{t-1}$ and sets each share as $[(r^1_\tau, h_\tau \oplus M_\tau)]_j = (j, f(j))$, $j \in [m]$. Here, $f(0) = a_0$ encodes $(r^1_\tau, h_\tau \oplus M_\tau)$.

(3) It creates the shares of another random pair (r^0_τ, C_τ) in a consistent way w.r.t. the shares that the corrupted exit servers will receive. For $j = \hat{t}+1, \ldots, t$, it chooses some random image values $\hat{y}_j$. Then, it computes the unique polynomial $\hat{f}(x) = \hat{a}_0 + \hat{a}_1 x + \cdots + \hat{a}_{t-1}x^{t-1}$ that is determined by the t points $(1, f(1)), \ldots, (\hat{t}, f(\hat{t}))$, $(\hat{t}+1, \hat{y}_{\hat{t}+1}), \ldots, (t, \hat{y}_t)$ via Lagrange interpolation. If the degree of $\hat{f}(x)$ is lower than $t-1$, then it stops the emulation and returns a random bit to the mIND-CPA challenger. Otherwise, it computes the points $(t+1, \hat{f}(t+1)), \ldots, (m, \hat{f}(m))$. It sets as (r^0_τ, C_τ) the value that $\hat{a}_0$ encodes. The m shares of (r^0_τ, C_τ) are formed as:

$$[(r^0_\tau, C_\tau)]_j = \begin{cases} (j, f(j)), & \text{if } 1 \leq j \leq \hat{t} \\ (j, \hat{f}(j)), & \text{if } \hat{t} < j \leq m \end{cases}$$

(4) It chooses a random tag tag_τ.

(5) It sends $\big((\mathsf{tag}_\tau, [(r^0_\tau, C_\tau)]_{\hat{t}+1}), \ldots, (\mathsf{tag}_\tau, [(r^0_\tau, C_\tau)]_m)\big)$ and $\big((\mathsf{tag}_\tau, [(r^1_\tau, h_\tau \oplus M_\tau)]_{\hat{t}+1}), \ldots, (\mathsf{tag}_\tau, [(r^1_\tau, h_\tau \oplus M_\tau)]_m)\big)$ as challenge lists of messages to the mIND-CPA challenger and receives a list of ciphertexts $c^b_{\hat{t}+1}, \ldots, c^b_m$.

(6) It creates ℓ-level layer encryptions $c_1, \ldots, c_{\hat{t}}$ for the tagged shares $(\mathsf{tag}_\tau, (1, f(1)), \ldots, (\mathsf{tag}_\tau, (\hat{t}, f(\hat{t}))$ (common for (r^0_τ, C_τ) and $(r^1_\tau, h_\tau \oplus M_\tau)$), respectively. These encryptions are intended for the corrupted exit servers $\mathsf{MX}_{1,\ell}, \ldots, \mathsf{MX}_{\hat{t},\ell}$.

(7) It creates ℓ-level layer encryptions $c_{\hat{t}+1}, \ldots, c_m$ that correspond to $c^b_{\hat{t}+1}, \ldots, c^b_m$ (i.e., it adds ($\ell - 1$ more layers per ciphertext). These encryptions are intended for the honest exit servers $\mathsf{MX}_{\hat{t}+1,\ell}, \ldots, \mathsf{MX}_{m,\ell}$ with public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_{m-\hat{t}}$.

(8) It normally transmits $((\mathsf{pk}_{1,1}, c_1), \ldots, (\mathsf{pk}_{1,m}, c_m))$ to the mix servers.

(9) When the honest exit server $\mathsf{MX}_{j,\ell}$ receives c^b_j (recall that $\mathcal{D}_\tau$ cannot decrypt this ciphertext in the emulation), it *always* broadcasts $(\mathsf{tag}_\tau, [(r^1_\tau, h_\tau \oplus M_\tau)]_j)$ to the parties.

* If $w > \tau$, then it acts like $\mathcal{S}$; namely, it (i) chooses random values r_w, C_w; (ii) splits (r_w, C_w) into m shares $[(r_w, C_w)]_1, \ldots, [(r_w, C_w)]_m$; (iii) transmits the shares associated with a random tag tag_w. During the execution, it responds to $\mathcal{A}^*$'s query r_w to $\mathcal{F}_{\mathsf{RO}}$ as if it was $\mathcal{S}$ in order to equivocate for the message M_w when message delivery comes.

- Like $\mathcal{S}$, if at any point of the execution it chooses a random value r_w multiple times, then it stops emulation and returns a random bit to the IND-CPA challenger.
- It responds to the IND-CPA challenger with whatever $\mathcal{Z}^*$ returns.

In the description of $\mathcal{D}_\tau$, emulation may stop if $\mathcal{A}^*$ does not even allow the beginning of the execution (completion of the Setup procedure fails) or if $\mathcal{Z}^*$ does not request the broadcast of any messages. Let Abstain denote the event that any of the above two events happen. Clearly, it holds that

$$\Big| \Pr\big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda) = 1 \big| \mathsf{Abstain}\big] - \\ - \Pr\big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda) = 1 \big| \mathsf{Abstain}\big] \Big| = 0,$$

i.e., $\mathcal{Z}^*$ has no distinguishing advantage when Abstain happens. Assuming that Eq. (1) holds and w.l.o.g., we may assume that $\Pr[\mathsf{Abstain}] = 0$, namely, the execution under $\mathcal{A}^*, \mathcal{Z}^*$ is never trivial.

Next, we analyze the case where $w = \tau$. By the information theoretic security of the (t, m)-threshold secret sharing scheme, we have that the $\hat{t} < t$ common shares $(1, f(1)), \ldots, (\hat{t}, f(\hat{t}))$ do not reveal any information about the values (r^0_τ, C_τ) and $(r^1_\tau, h_\tau \oplus M_\tau)$. In addition, by choosing $t - \hat{t}$ random points $(\hat{t}+1, \hat{y}_{\hat{t}+1}), \ldots, (t, \hat{y}_t)$ for the generation of $\hat{f}(x)$, we have that $\hat{f}(0) = \hat{a}_0$ is random so the pair (r^0_τ, C_τ) that $\hat{a}_0$ encodes is also random. As a result, the tagged shares $(\mathsf{tag}_\tau, [(r^1_\tau, h_\tau \oplus M_\tau)]_{\hat{t}+1}), \ldots, (\mathsf{tag}_\tau, [(r^1_\tau, h_\tau \oplus M_\tau)]_m)$ broadcast by the exit servers follow the same distribution as the tagged shares $(\mathsf{tag}_\tau, [(r^0_\tau, C_\tau)]_{\hat{t}+1}), \ldots, (\mathsf{tag}_\tau, [(r^0_\tau, C_\tau)]_m)$.

Observe that the transmission schedule in the emulation of $\mathcal{D}_\tau$ given that the IND-CPA challenge bit $b = 1$, is similar to the one of $\mathcal{D}_{\tau+1}$ given $b = 0$, with the following exception: for $w = \tau+1$, $\mathcal{D}_\tau$ finally broadcasts $(\mathsf{tag}_{\tau+1}, [(r_{\tau+1}, C_{\tau+1})]_1), \ldots, (\mathsf{tag}_{\tau+1}, [(r_{\tau+1}, C_{\tau+1})]_m)$ while $\mathcal{D}_{\tau+1}$ finally broadcasts $(\mathsf{tag}'_{\tau+1}, [(r^1_{\tau+1}, h_{\tau+1} \oplus M_{\tau+1})]_1), \ldots, (\mathsf{tag}'_{\tau+1}, [(r^1_{\tau+1}, h_{\tau+1} \oplus M_{\tau+1})]_m)$. Since these two message sequences follow the same distribution, we get that unless emulation stops (either for the same reasons that $\mathcal{S}$ fails or because a polynomial $\hat{f}(x)$ of degree less than $t-1$ was randomly chosen during the secret sharing process), the two algorithms behave similarly. Clearly, the probability that emulation stops is $\mathsf{negl}(\lambda)$, so it holds that for some negligible function $\delta_\tau(\cdot)$:

$$\big| \Pr[\mathcal{D}_\tau(\mathbf{pk}) \to 1 | b = 1] - \\ - \Pr[\mathcal{D}_{\tau+1}(\mathbf{pk}) \to 1 | b = 0] \big| \leq \delta_\tau(\lambda). \tag{2}$$

Following the same reasoning as above, we can deduce that the behavior of $\mathcal{D}_1$ given $b = 0$ is similar to the one of $\mathcal{S}$ with the difference that $\mathcal{D}_1$ finally broadcasts $(\mathsf{tag}_1, [(r^1_q, h_1 \oplus M_1)]_1), \ldots, (\mathsf{tag}_1, [(r^1_1, h_1 \oplus M_1)]_m)$ instead of the tagged shares of a random pair (r, C). So, we get that for some negligible function $\delta_0(\cdot)$:

$$\Big| \Pr\big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda) = 1\big] - \\ - \Pr[\mathcal{D}_1(\mathbf{pk}) \to 1 | b = 0] \Big| \leq \delta_0(\lambda). \tag{3}$$

Besides, by the description of $\mathcal{D}_{T'}$, we directly get that

$$\Big| \Pr\big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda) = 1\big] - \\ - \Pr[\mathcal{D}_{T'}(\mathbf{pk}) \to 1 | b = 1] \Big| = 0. \tag{4}$$

By Eq. (1), (2), (3), and (4), we get that

$$
\begin{aligned}
&\alpha(\lambda) \leq \\
\leq&\Big| \Pr\Big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda) = 1\Big] - \Pr\Big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda) = 1\Big]\Big| \leq \\
\leq&\Big| \Pr\Big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda) = 1\Big] - \sum_{\tau=1}^{T'} \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] + \\
&+ \sum_{\tau=1}^{T'} \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0] - \Pr\Big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda) = 1\Big]\Big| + \\
+&\Big| \sum_{\tau=1}^{T'} \Pr\Big[\Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] - \sum_{\tau=1}^{T'} \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0]\Big| \leq \\
\leq&\Big| \Pr\Big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda) = 1\Big] - \Pr[\mathcal{D}_{T'}(\mathbf{pk}) \rightarrow 1 | b = 1] - \\
&- \sum_{\tau=1}^{T'-1} \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] + \sum_{\tau=2}^{T'} \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0] + \\
&+ \Pr[\mathcal{D}_1(\mathbf{pk}) \rightarrow 1 | b = 0] - \Pr\Big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda) = 1\Big]\Big| + \\
+& \sum_{\tau=1}^{T'} \Big| \Pr\Big[\Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] - \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0]\Big| \leq \\
\leq&\Big| \Pr\Big[\mathsf{EXEC}^{\Pi^{m,\ell,t,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{A}^*}(\lambda) = 1\Big] - \Pr[\mathcal{D}_{T'}(\mathbf{pk}) \rightarrow 1 | b = 1]\Big| + \\
&+ \sum_{\tau=1}^{T'-1} \Big| \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] - \Pr[\mathcal{D}_{\tau+1}(\mathbf{pk}) \rightarrow 1 | b = 0]\Big| + \\
&+ \Big| \Pr[\mathcal{D}_1(\mathbf{pk}) \rightarrow 1 | b = 0] - \Pr\Big[\mathsf{EXEC}^{F^{\ell,B,p}_{\mathsf{an.BC}}}_{\mathcal{Z}^*,\mathcal{S}}(\lambda) = 1\Big]\Big| + \\
+& \sum_{\tau=1}^{T'} \Big| \Pr\Big[\Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] - \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0]\Big| \leq \\
\leq& 0 + \sum_{\tau=1}^{T'-1} \delta_\tau(\lambda) + \delta_0(\lambda) + \\
+& \sum_{\tau=1}^{T'} \Big| \Pr\Big[\Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] - \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0]\Big| = \\
=& \sum_{\tau=0}^{T'-1} \delta_\tau(\lambda) + \\
+& \sum_{\tau=1}^{T'} \Big| \Pr\Big[\Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 1] - \Pr[\mathcal{D}_\tau(\mathbf{pk}) \rightarrow 1 | b = 0]\Big|.
\end{aligned}
\tag{5}
$$

Thus, by Eq. (5) and an averaging argument, we have that there is a $\tau^* \in [T']$ such that

$$
\begin{aligned}
&\Big| \Pr\Big[\Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 1 | b = 1] - \Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 1 | b = 0]\Big| \geq \\
\geq& \frac{\alpha(\lambda) - \sum_{\tau=0}^{T'-1} \delta_\tau(\lambda)}{T'}.
\end{aligned}
$$

Since $\alpha(\cdot)$ is a non-negligible function, $\delta_0(\cdot), \ldots, \delta_{T'-1}(\cdot)$ are negligible functions, and T' is polynomial in λ, we have that $\gamma(\lambda) := \dfrac{\alpha(\lambda) - \sum_{\tau=0}^{T'-1} \delta_\tau(\lambda)}{T'}$ is a non-negligible function.

We study the two following cases:
If it holds that

$$
\begin{aligned}
\Pr\Big[&\Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 1 | b = 1] - \\
&- \Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 1 | b = 0] \geq \gamma(\lambda),
\end{aligned}
$$

then we directly get that $\mathcal{D}_{\tau^*}$ wins the IND-CPA game against $\Sigma^{(m-\hat{t})}_{\mathsf{PKE}}$ with probability $\frac{1}{2} + \frac{\gamma(\lambda)}{2}$.

Else, if it holds that

$$
\begin{aligned}
\Pr\Big[&\Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 1 | b = 0] - \\
&- \Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 1 | b = 1] \geq \gamma(\lambda) \Leftrightarrow \\
\Leftrightarrow \Pr\Big[&\Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 0 | b = 1] - \\
&- \Pr[\mathcal{D}_{\tau^*}(\mathbf{pk}) \rightarrow 0 | b = 0] \geq \gamma(\lambda),
\end{aligned}
$$

then the adversary $\bar{\mathcal{D}}_{\tau^*}$, that operates exactly like $\mathcal{D}_{\tau^*}$, but flips $\mathcal{D}_{\tau^*}$'s output bit, wins the IND-CPA game against $\Sigma^{(m-\hat{t})}_{\mathsf{PKE}}$ with probability $\frac{1}{2} + \frac{\gamma(\lambda)}{2}$.

In any case, we can devise an adversary that breaks the IND-CPA security of $\Sigma^{(m-\hat{t})}_{\mathsf{PKE}}$. By Claim F.0.1, this contradicts to the IND-CPA security of Σ_{PKE}. Therefore, Eq. (1) does not hold and the proof is complete.

□

G ACCUMULATORS

In this section, we present the ideal functionality $\mathcal{F}_{\text{acc}}$, the protocol Π_{acc} that UC realizes it and the proof of realization.

G.1 The ideal functionality $\mathcal{F}_{\text{acc}}$

The realization of the E-cclesia family, especially its eligibility feature, relies on a UC secure accumulator that is *additive* (i.e., it supports only addition of elements to the set) and *positive* (i.e., it supports membership proofs). In Subsection G.1.1, we present our ideal accumulator functionality $\mathcal{F}_{\text{acc}}$ that is in the spirit of [6] adjusted to our scenario. In Subsection G.2, we introduce the protocol Π_{acc} that follows the command interface of $\mathcal{F}_{\text{acc}}$. In Subsection G.4, we prove that Π_{acc} UC-realizes $\mathcal{F}_{\text{acc}}$, if the underlying accumulator scheme of Π_{acc} satisfies the standard correctness and soundness properties, such as the hash-based scheme in [57]. In addition, the initialization of the scheme in [57] allows the execution of Π_{acc} without the involvement of a trusted party such as a CRS.

G.1.1 The ideal functionality $\mathcal{F}_{\text{acc}}$. We present the ideal accumulator functionality $\mathcal{F}_{\text{acc}}$ that is inspired by the accumulator functionality in [6] with some modifications that fit our purposes. Most importantly, the accumulator's operations (e.g. addition) are managed by $\mathcal{F}_{\text{acc}}$ in a way that abstracts a real-world scenario where these operations are handled by the parties themselves in a local manner, instead of having an accumulator manager that is in control of a shared accumulator state.

The functionality $\mathcal{F}_{\text{acc}}$ initializes a mapping $\mathbf{S}^P$ with $\mathbf{S}^P[0] = \emptyset$ for every honest party P, which maps the number of elements in the multiset or list to the actual multiset or list of the accumulated elements. In case the *quasi-commutativity* security property [12] is captured by $\mathcal{F}_{\text{acc}}$ then $\mathbf{S}^P$ maps elements to a multiset, else it maps them to a list. We make this distinction clear by indicating with red the text that corresponds to the version of $\mathcal{F}_{\text{acc}}$ which captures quasi-commutativity, and with blue the version that does not capture it.

Moreover, $\mathcal{F}_{\text{acc}}$ initializes as 0 the counter $\mathsf{t_P}$ which shows the total number of the added elements in the accumulated multiset or list for the party P. The counter $\mathsf{t_P}$ also indicates the number of operations of that specific time for a given accumulator and because we use only addition operations it coincides with the total number of elements in it. It initializes as empty the list L^P_{state} which contains tuples that include the accumulated value along with some auxiliary information (depending on the actual construction of the accumulator), the "update" message that is needed for updating the witnesses of the older elements for previous accumulated values, the set or list of elements of the previous accumulated value, the last accumulated value, the corresponding witness of that value and the total number of elements in the current accumulator. Moreover, the functionality initializes the shared parameters vector shared_params, that consists of the accumulation algorithms and a generated initialization triple, as $\emptyset$. Finally, it initializes a set $\mathbf{P}_{\text{ready}}$ of parties ready to engage as empty.

The simulator $\mathcal{S}$ provides the set of the corrupted parties $\mathbf{P}_{\text{corr}}$. Each time the functionality receives a command message from a corrupted party it handles it to the simulator and waits for its response and returns whatever receives from $\mathcal{S}$.

Upon receiving Setup from a party P, if no algorithms are stored, $\mathcal{F}_{\text{acc}}$ requests the accumulation algorithms from $\mathcal{S}$. Specifically, $\mathcal{S}$ returns the following PPT algorithms: (i) `Gen`, which generates the accumulator's parameters; (ii) `Update`, which updates the accumulated value after the addition of a new element along with other parameters essential for the other operations; (iii) `WitUp`, which updates the witness w^x_{old} for an element x to w^x_{new} after the addition of new elements in the accumulator; and (iv) `VerStatus`, which verifies if an element is part of the accumulated value by providing its witness. The functionality generates the initial accumulated value along with some auxiliary information by executing the function `Gen`. Then, it checks via the function `VerStatus` that the first accumulation value indeed corresponds to the empty set (if not, it sets shared_params to $\langle\bot\rangle$). Here, we capture the sound operation of the accumulator. It sets shared_params as the vector that contains the initialization values and the accumulator algorithms `Update`, `WitUp`, and `VerStatus`. Moreover, when P becomes "ready" to engage, if she is honest, then $\mathcal{F}_{\text{acc}}$ inserts the initial tuple in L^P_{state}. Here, we capture that each honest party shares the same view on the initial accumulated value. In addition, the fact that for each honest party P the functionality maintains P's own list illustrates that the accumulation operations take place locally rather than in a shared state setting.

Upon receiving Update along with an accumulated value α and an element x from an honest ready party P, the functionality checks if that party has previously recorded such an α, which means that P has obtained this accumulated value in a past interaction with $\mathcal{F}_{\text{acc}}$ as a result of another element addition. Here, we capture the fact that parties only accumulate elements in known accumulated values that have been obtained before, rather than arbitrary ones that the parties do not know their history with respect to their element representation. Next, $\mathcal{F}_{\text{acc}}$: (1) increases the counter $\mathsf{t_P}$ by 1, which means that a new element is accumulated; (2) computes the new accumulated value $\alpha_{\mathsf{t_P}}$, an auxiliary message $m_{\mathsf{t_P}}$, the witness $w^x_{\mathsf{t_P}}$ that x is part of the accumulator $\alpha_{\mathsf{t_P}}$, and an update message $\mathsf{upmsg}_{\mathsf{t_P}}$ that can be used to update witnesses of other values after the addition of x in the accumulator. Then, it stores the tuple $(\alpha_{\mathsf{t_P}}, m_{\mathsf{t_P}}, v_{\mathsf{t_P}}, \mathsf{upmsg}_{\mathsf{t_P}}, \mathsf{S}^P[\mathsf{t_P}-1], x, w^x_{\mathsf{t_P}}, \mathsf{t_P})$ in L^P_{state}, where the value $\mathsf{S}^P[\mathsf{t_P}-1]$ equals with the previous accumulated multiset or list without the element x, for tracking. Then, it verifies if the new accumulated state is generated correctly by executing the algorithm `VerStatus`. If not, then a *correctness* error occurred and $\mathcal{F}_{\text{acc}}$ returns this error message to P. If no error occurred, then it returns the new accumulated value along with the witness of x and the update message.

Upon receiving the command message Wit_Up from an honest ready party P, the functionality updates an old witness w_{old} for a given element x. Specifically, it accepts the old accumulated value α_{old} (e.g., before the addition of new elements by the time x was inserted), the target element x, its old witness, the target accumulator with which we want to make compatible the old witness, α_{new}, and some series of update messages that are the result of the addition of new elements into the accumulator. $\mathcal{F}_{\text{acc}}$ returns to P the updated witness for the element x for the accumulator α_{new} after it checks that the updated witness is compatible with the functions `VerStatus`. If it is not, it returns $\bot$ as correctness property has been breached.

Finally,upon receiving the verification command Ver_Status, from an honest party P, $\mathcal{F}_{\text{acc}}$ verifies if an element x with witness w is part of the accumulated value α. In case the verification returns true but the element is not accumulated into a recorded α then a *soundness* error occurred and the functionality returns an error message to the party. It is worth mentioning that the functionality not only searches for the input values into the stored data based for P, but for all honest parties. This means that the for every honestly generated values no forgery should occur (soundness). In contrast, the correctness property is only meaningful for each party individually. This is why the functionality returns a $\bot$ symbol in the previous cases by only considering each party's data base individually.

$\mathcal{F}_{\text{acc}}(\mathbf{P})$.

The functionality initializes the following for each party P: the mapping S^P from the number of accumulated elements to the accumulated *multiset*\list as $\mathsf{S}^P[0] = \emptyset$; a counter $\mathsf{t_P}$ that represents the number of elements in the accumulator as 0; the list of tuples L^P_{state} as empty, where each tuple contains (i) the accumulated value $\alpha_{\mathsf{t_P}}$, (ii) the auxiliary information $m_{\mathsf{t_P}}$ for verifying the membership of an element into the accumulator, (iii) the update message $\mathsf{upmsg}_{\mathsf{t_P}}$ for updating older witnesses, (iv) the *multiset\list* $\mathsf{S}^P[\mathsf{t_P}-1]$ of the previous accumulated value, (v) the new accumulated element x, (vi) its related witness w^x, and (vii) the counter $\mathsf{t_P}$. Moreover, the functionality initializes the shared parameters vector shared_params, that consists of the accumulation algorithms and a generated initialization triple, as $\emptyset$. Finally, it initializes a set $\mathbf{P}_{\text{ready}}$ of parties ready to engage as empty. Upon receiving (sid, Corrupt, $\mathbf{P}_{\text{corr}}$) from $\mathcal{S}$, if $\mathbf{P}_{\text{corr}} \subseteq \mathbf{P}$, it fixes $\mathbf{P}_{\text{corr}}$ as the set of corrupted parties.

■ Upon receiving (sid, Setup) from some party $P \notin \mathbf{P}_{\text{corr}}$ or (sid, Setup, P) from $\mathcal{S}$ for some party $P \in \mathbf{P}_{\text{corr}}$, it does:

(1) If shared_params = $\emptyset$, it executes the ***Generation*** procedure as follows:
   (a) It sends (sid, Gen) to $\mathcal{S}$. Upon receiving (sid, Gen, Gen, Update, WitUp, VerStatus) from $\mathcal{S}$, it stores the algorithms Gen, Update, WitUp, VerStatus.
   (b) It computes the initialization triple $(\alpha_0, m_0, v_0) \leftarrow \mathtt{Gen}(1^\lambda)$.
   (c) If $\mathtt{VerStatus}(\alpha_0, \text{Null}, v_0) = 1$, it sets shared_params := $\big((\alpha_0, m_0, v_0)$, Update, WitUp, VerStatus$\big)$. Otherwise, it sets shared_params := $\perp$.

(2) If $P \notin \mathbf{P}_{\text{ready}}$, it adds P to $\mathbf{P}_{\text{ready}}$.

(3) If $P \notin \mathbf{P}_{\text{corr}}$ and shared_params := $\langle(\alpha_0, m_0, v_0)$, Update, WitUp, VerStatus$\rangle$, then it appends the tuple $(\alpha_0, (m_0, v_0), \text{Null}, \text{Null}, \text{Null}, \text{Null}, 0)$ to L^P_{state} .

(4) It sends (sid, Setup, shared_params) to P or $\mathcal{S}$.

■ Upon receiving (sid, Update, α, x) from some party $P \in \mathbf{P}_{\text{ready}} \setminus \mathbf{P}_{\text{corr}}$, if there exists a tuple $(\alpha, m_{\mathrm{t_P}}, \mathsf{upmsg}_{\mathrm{t_P}}, \mathbf{S}^P[\mathrm{t_P} - 1], x', w^{x'}_{\mathrm{t_P}}, \mathrm{t_P})$ or $(\alpha, (m_0, v_0), \text{Null}, \text{Null}, \text{Null}, \text{Null}, 0)$ in L^P_{state}, it does:

(1) It increases the counter $\mathrm{t_P} \leftarrow \mathrm{t_P} + 1$.

(2) It computes $(\alpha_{\mathrm{t_P}}, m_{\mathrm{t_P}}, w^x_{\mathrm{t_P}}, \mathsf{upmsg}_{\mathrm{t_P}}) \leftarrow \mathtt{Update}(\alpha, m_{\mathrm{t_P}-1}, x)$. If $\mathrm{t_P} \neq 1$, it sets $\mathbf{S}^P[\mathrm{t_P} - 1] = \mathbf{S}^P[\mathrm{t_P} - 2] \cup \{x\}$. It adds $(\alpha_{\mathrm{t_P}}, m_{\mathrm{t_P}}, \mathsf{upmsg}_{\mathrm{t_P}}, \mathbf{S}^P[\mathrm{t_P} - 1], x, w^x_{\mathrm{t_P}}, \mathrm{t_P})$ to L^P_{state}.

(3) If $\mathtt{VerStatus}(\alpha_{\mathrm{t_P}}, x, w^x_{\mathrm{t_P}}) \neq 1$, it returns (sid, Update, $\alpha, x, \perp$) to P.

(4) If $\mathtt{VerStatus}(\alpha_{\mathrm{t_P}}, x, w^x_{\mathrm{t_P}}) = 1$, it returns (sid, Update, $\alpha, x, \alpha_{\mathrm{t_P}}, w^x_{\mathrm{t_P}}, \mathsf{upmsg}_{\mathrm{t_P}}$) to P.

■ Upon receiving (sid, Wit_Up, $\alpha_{\text{old}}, \alpha_{\text{new}}, x, w_{\text{old}}$, $(\mathsf{upmsg}_{\text{old}+1}, \ldots, \mathsf{upmsg}_{\text{new}})$) from party $P \in \mathbf{P}_{\text{ready}} \setminus \mathbf{P}_{\text{corr}}$, if there exist tuples $\{(\alpha_{\text{old}}, m_{\text{old}}, \mathsf{upmsg}_{\text{old}}, \mathbf{S}^P[\text{old} - 1], x, w_{\text{old}}, \text{old}), \cdots, (\alpha_{\text{new}}, m_{\text{new}}, \mathsf{upmsg}_{\text{new}}, \mathbf{S}^P[\text{new} - 1], x_{\text{new}}, w_{\text{new}}, \text{new})\}$ in L^P_{state} with new > old such that $x \in \mathbf{S}^P[\text{old} - 1] \cap \cdots \cap \mathbf{S}^P[\text{new} - 1]$, it does:

(1) It computes $w_{\text{new}} \leftarrow \mathtt{WitUp}(x, w_{\text{old}}, (\mathsf{upmsg}_{\text{old}+1}, \ldots, \mathsf{upmsg}_{\text{new}}))$.

(2) If $\mathtt{VerStatus}(\alpha_{\text{new}}, x, w_{\text{new}}) \neq 1$, it returns (sid, WitUp, $\alpha_{\text{old}}, \alpha_{\text{new}}, x, w_{\text{old}}, (\mathsf{upmsg}_{\text{old}+1}, \ldots, \mathsf{upmsg}_{\text{new}}), \perp$) to P.

(3) If $\mathtt{VerStatus}(\alpha_{\text{new}}, x, w_{\text{new}}) = 1$, it returns (sid, Wit_Up, $\alpha_{\text{old}}, \alpha_{\text{new}}, x, w_{\text{old}}, (\mathsf{upmsg}_{\text{old}+1}, \ldots, \mathsf{upmsg}_{\text{new}}), w_{\text{new}}$) to P.

■ Upon receiving (sid, Ver_Status, α, VerStatus′, x, w) from party $P \in \mathbf{P}_{\text{ready}} \setminus \mathbf{P}_{\text{corr}}$ or (sid, Ver_Status, α, VerStatus′, x, w, P) from $\mathcal{S}$ for some party $P \in \mathbf{P}_{\text{corr}}$, it does:

(1) If VerStatus′ = VerStatus and for some largest integer t_{P^*}, there exists a tuple $(\alpha, m_{t_{P^*}}, \mathsf{upmsg}_{t_{P^*}}, \mathbf{S}^{P^*}[t_{P^*} - 1], x_{t_{P^*}}, w_{t_{P^*}}, t_{P^*})$ in $L^{P^*}_{\text{state}}$ for some (honest) party P^* such that $(x_{t_{P^*}} \neq x) \vee (x \notin \mathbf{S}^{P^*}[t_{P^*} - 1])$ and $\mathtt{VerStatus}(\alpha, x, w) = 1$, it returns (sid, Ver_Status, $\alpha, x, w, \perp$) to P. Otherwise, it computes $\phi \leftarrow \mathtt{VerStatus'}(\alpha, x, w)$.

(2) It returns (sid, Ver_Status, α, x, w, ϕ) to P or $\mathcal{S}$.

■ Upon receiving any command message from party $P \in \mathbf{P}_{\text{corr}}$, it forwards it to $\mathcal{S}$. Upon receiving the token back from $\mathcal{S}$ on behalf of P, it returns whatever it receives back to P.

Figure 18: The accumulator functionality $\mathcal{F}_{\text{Acc}}(\mathbf{P})$ interacting with the parties in P and the simulator $\mathcal{S}$.

Remark 1 (Comparison between $\mathcal{F}_{\text{acc}}$ and the accumulator functionality in [6]). As already mentioned, $\mathcal{F}_{\text{acc}}$ can be seen as an adaptation of the generic accumulator functionality in [6] for our purposes. Specifically, we only consider additions and membership checks, so $\mathcal{F}_{\text{acc}}$ abstracts the class of additive and positive accumulators. Besides, we are interested in scenarios where, although there may be an agreement on the accumulated elements, the accumulated values' computation is done locally by each party. Thus, unlike the functionality in [6] that captures the maintenance of a shared accumulator state by an accumulator manager, our functionality $\mathcal{F}_{\text{acc}}$ handles an accumulator state for each honest party where the only shared data are the accumulator algorithms and the initial value and auxiliary information. Moreover, we do not require a mechanism that verifies if the Update operation was carried out correctly, as each party is responsible for updating their value locally. Other notable differences between the two functionalities are:

- The initial set (denoted as S_0 in [6]) is always $\emptyset$.
- Since the hash-based construction in [57] that will be used in our realization does not utilize a secret key, the Gen algorithm does not output a value sk in our case.
- In an Update request, $\mathcal{F}_{\text{acc}}$ also accepts an accumulated value α (besides x). This is because in our setting, there is no shared state and we allow "branches" in the history of accumulated multisets/lists. Thus, we provide α that serves as the starting point.
- In our e-voting use case, the users (voters) will obtain a unique credential in order to vote. So, generating multiple witnesses for the same element is an accumulator operation that will not take place and, unlike the functionality in [6], $\mathcal{F}_{\text{acc}}$ does not consider WitCreate requests.

G.2 The protocol Π_{acc}

The protocol $\Pi_{\text{acc}}(\mathbf{P}, \mathcal{F}^{\text{Gen}}_{\text{CRS}}, \Sigma_{\text{acc}})$ is presented in Figure 19 and, briefly, it operates as follows: Each party P has hard-coded the accumulator algorithms (Update, WitUp, VerStatus), and maintains the mapping $\mathbf{S}^P$ from the number of accumulated elements to the accumulated multiset or list and an operation counter t_P.

When a party receives a Setup command from $\mathcal{Z}$, she requests the setup parameters from the $\mathcal{F}_{\text{CRS}}$ functionality (cf. Figure 7), parameterized by the Gen algorithm. We stress that in our hash-based instantiation, we use the Gen algorithm that always returns the empty set. Thus, $\mathcal{F}_{\text{CRS}}$ can be realized in a decentralized manner (e.g., parties trivially return $\emptyset$ upon the Setup request from $\mathcal{Z}$).

If a party receives the Update command from $\mathcal{Z}$ along with an accumulated value α and an element x, she checks in her records if the value α has previously occurred. If not, she returns $\perp$ (like $\mathcal{F}_{\text{Acc}}$, honest parties deny accumulating values for previously unknown accumulators). Then, she updates the accumulator and her records, and returns the new accumulator value along with the witness for the value x.

Upon receiving a Wit_Up command along with the old accumulator value α_{old}, the accumulator value in which the new witness must be compatible with, α_{new}, the accumulated value x, the old witness for x and a series of update messages, the party P searches if in her database for these accumulator values, the value x and the series of update messages are already registered. If so, then she checks that there is an "update path" from the value α_{old} to α_{new}. This means that there is a chain of tuples from α_{old} to α_{new} with: (i) The operation counter t_P increases by one in each part-tuple of the chain; (ii) The data S^P are increasing in a progressive way in each tuple of the chain including previous values. If so, she computes and returns the new witness by using the `Wit_Up` function.

Finally, when P receives Ver_Status along with an accumulator α, a value x and a witness w, she checks that x is part of α by using w and function `Ver_Status`. Note that, these values are not necessary to be registered in the party's database, enabling us to cross-check values between parties. Then, P returns whatever she receives from `Ver_Status`.

$\Pi_{\text{acc}}(\mathbf{P}, \mathcal{F}_{CRS}^{\text{Gen}}, \texttt{Gen}, \texttt{Update}, \texttt{WitUp}, \texttt{VerStatus})$

Each party $P \in \mathbf{P}$ has hard-coded the accumulator algorithms (`Update,WitUp,VerStatus`) and has initialized the counter t_P as 0, the mapping from the number of accumulated elements to the accumulated list as S^P such that $\text{S}^P[0] = \emptyset$, and the list L^P_{state} as described in Subsection G.1.1.

- Upon receiving (sid, Setup) from $\mathcal{Z}$ for the first time, P does:
  1. She sends (sid, CRS) to $\mathcal{F}_{\text{CRS}}^{\text{Gen}}$. Upon receiving (sid, CRS, (α_0, m_0, v_0)) from $\mathcal{F}_{\text{CRS}}^{\text{Gen}}$, she appends the tuple $(\alpha_0, (m_0, v_0), \text{Null}, \text{Null}, \text{Null}, \text{Null}, 0)$ to L^P_{state}.
  2. She returns (sid, Setup, $\langle(\alpha_0, m_0, v_0)$, `Update` , `WitUp`, `VerStatus`$\rangle$) to $\mathcal{Z}$.
- Upon receiving (sid, Update, α, x) from $\mathcal{Z}$, if P has submitted a Setup request and there exists a tuple $(\alpha, m_{\text{t}_\text{P}}, \text{upmsg}_{\text{t}_\text{P}}, \text{S}^P[\text{t}_\text{P} - 1], x', w^{x'}_{\text{t}_\text{P}}, \text{t}_\text{P})$ or $(\alpha, (m_0, v_0), \text{Null}, \text{Null}, \text{Null}, \text{Null}, 0)$ in L^P_{state}, she does:
  1. She increases the counter $\text{t}_\text{P} \leftarrow \text{t}_\text{P} + 1$.
  2. She computes $(\alpha_{\text{t}_\text{P}}, m_{\text{t}_\text{P}}, w^x_{\text{t}_\text{P}}, \text{upmsg}_{\text{t}_\text{P}}) \leftarrow \texttt{Update}(\alpha, m_{\text{t}_\text{P}-1}, x)$, if $\text{t}_\text{P} \neq 1$ sets $\text{S}^P[\text{t}_\text{P} - 1] = \text{S}^P[\text{t}_\text{P} - 2] \cup \{x\}$. She appends $(\alpha_{\text{t}_\text{P}}, m_{\text{t}_\text{P}}, \text{upmsg}_{\text{t}_\text{P}}, \text{S}^P[\text{t}_\text{P} - 1], x, w^x_{\text{t}_\text{P}}, \text{t}_\text{P})$ to L^P_{state}.
  3. She returns (sid, Update, $\alpha, x, \alpha_{\text{t}_\text{P}}, w^x_{\text{t}_\text{P}}, \text{upmsg}_{\text{t}_\text{P}}$) to $\mathcal{Z}$.
- Upon receiving (sid, Wit_Up, $\alpha_{\text{old}}, \alpha_{\text{new}}, x, w_{\text{old}}$, $(\text{upmsg}_{\text{old}+1}, \ldots, \text{upmsg}_{\text{new}})$) from $\mathcal{Z}$, if P has submitted a Setup request and there exist tuples $\{(\alpha_{\text{old}}, m_{\text{old}}, v_{\text{old}}, \text{upmsg}_{\text{old}}, \text{S}^P[\text{old} - 1], x, w_{\text{old}}, \text{old}), \ldots, (\alpha_{\text{new}}, m_{\text{new}}, v_{\text{new}}, \text{upmsg}_{\text{new}}, \text{S}^P[\text{new} - 1], x_{\text{new}}, w_{\text{new}}, \text{new})\}$ in L^P_{state} with new > old such that $x \in \text{S}^P[\text{old} - 1] \cap \cdots \cap \text{S}^P[\text{new} - 1]$, she does:
  1. She computes $w_{\text{new}} \leftarrow \texttt{WitUp}(x, w_{\text{old}}, (\text{upmsg}_{\text{old}+1}, \ldots, \text{upmsg}_{\text{new}}))$.
  2. She returns (sid, Wit_Up, $\alpha_{\text{old}}, \alpha_{\text{new}}, x, w_{\text{old}}$, $(\text{upmsg}_{\text{old}+1}, \ldots, \text{upmsg}_{\text{new}}), w_{\text{new}}$) to $\mathcal{Z}$.
- Upon receiving (sid, Ver_Status, α, x, w) from $\mathcal{Z}$, if P has submitted a Setup request, she does:
  1. She computes $\phi = \texttt{VerStatus}(\alpha, x, w)$.
  2. She returns (sid, Ver_Status, α, x, w, ϕ) to $\mathcal{Z}$.

Figure 19: The accumulator protocol Π_{acc} for parties in P, parameterized by the accumulator algorithms `Gen`, `Update`, `WitUp`, `VerStatus`, and the common reference string functionality $\mathcal{F}_{\text{CRS}}$ w.r.t. the distribution $D = \{r : (\alpha_0, m_0, v_0) \leftarrow \texttt{Gen}(1^\lambda); r = (\alpha_0, m_0, v_0)\}$.

G.3 Definitions of secure accumulator

In [57], a Merkle-tree [16] is deployed to store the accumulated values. The $\texttt{Gen}(1^\lambda)$ procedure always returns an empty string. The witness that a value x has been accumulated is the path from the leaf of the Merkle-tree to the top hash. For more information and a detailed description of the actual construction we refer the reader to [57, Section 4].

The aforementioned hash-based construction in [57] is proven secure under a game-based framework that captures two security properties: *Correctness* [57, Definition 1] and *Soundness* [57, Definition 2]. Informally, Correctness states that for every accumulated element x we added at some point between a series of sequential additions in the accumulator, we get its witness w^x. After all additions have taken place, we update w^x by using the `WitUp` algorithm for each one of these additions after x. The property requires that the verification via `VerStatus` that x is part of the accumulator with the final witness w^x_t, where t indicates the total number of added elements, returns 1 with probability 1. Moreover, Soundness informally states that the adversary has negligible probability of succeeding in the following experiment: it adds to the accumulator an arbitrary set of elements. Then, it attempts to find an element and a witness of it such that: (i) that element is not part of the resulting accumulator; (ii) the `VerStatus` algorithm with input the resulting accumulator, that element and its witness returns true.

Below are the definitions of Correctness and Soundness as appear in [57].

Definition G.1 (Correctness). An accumulator (`Gen`, `Update`, `WitUp`, `VerStatus`) is *correct*, if an up-to-date witness w^x corresponding to value x can always be used to verify the membership of x in an up-to-date accumulator α. More formally, for all security parameters λ, all values x and additional sets of values $[y_1, \ldots, y_{t_x-1}]$, $[y_{t_x+1}, \ldots, y_t]$, it holds that

$$\Pr\left[\begin{array}{l} \alpha_0 \leftarrow \mathtt{Gen}(1^\lambda); \\ (\alpha_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathtt{Update}(\alpha_{i-1}, y_i) \text{ for } i \in [1, \ldots, t_x - 1]; \\ (\alpha_{t_x}, w_{t_x}^{x}, \mathsf{upmsg}_{t_x}) \leftarrow \mathtt{Update}(\alpha_{t_x-1}, x); \\ (\alpha_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathtt{Update}(\alpha_{i-1}, y_i) \text{ for } i \in [t_x + 1, \ldots, t]; \\ w_i^{x} \leftarrow \mathtt{WitUp}(x, w_{i-1}^{x}, \mathsf{upmsg}_i) \text{ for } i \in [t_x + 1, \ldots, t] : \\ \mathtt{VerStatus}(a_t, x, w_t^{x}) = 1 \end{array}\right] = 1.$$

Definition G.2 (Soundness). An accumulator (Gen, Update, WitUp, VerStatus) is *sound* (or *secure*), if it is hard to fabricate a witness w for a value x that has not been added to the accumulator. More formally, for any PPT stateful adversary $\mathcal{A}$ there exists a negligible function $\mu(\cdot)$ such that:

$$\Pr\left[\begin{array}{l} \alpha_0 \leftarrow \mathtt{Gen}(1^\lambda); t = 1; x_1 \leftarrow \mathcal{A}(1^\lambda, \alpha_0); \\ \textbf{while } x_t \neq \bot \\ \quad (\alpha_t, w_t^{x_t}, \mathsf{upmsg}_t) \leftarrow \mathtt{Update}(\alpha_{t-1}, x_t); \\ \quad t = t + 1; \\ \quad x_t \leftarrow \mathcal{A}(\alpha_{t-1}, w_{t-1}^{x_t-1}, \mathsf{upmsg}_{t-1}); \\ (x, w) \leftarrow \mathcal{A} : \\ x \notin \{x_1, \ldots, x_t\} \text{ and } \mathtt{VerStatus}(\alpha_{t-1}, x, w) = 1 \end{array}\right] \leq \mu(\lambda).$$

G.4 Proof of Theorem 6.2

We instantiate the accumulator algorithms Gen, Update, WitUp, VerStatus with the ones in the hash-based construction as presented in [57] with the exception that the syntax algorithms Add, MemWitUpOnAdd, VerMem in [57] are named Update, WitUp, VerStatus, respectively to match the syntax of [6]. From [57, Theorem 1], we get the following Lemma.

Lemma G.3. *The hash-based construction Σ_{acc} = (Gen, Update, WitUp, VerStatus) presented in [57], satisfies Correctness (cf. Definition G.1) and Soundness (cf. Definition G.2), as long as the underlying hash function is collision resistant.*

Armed with Lemma G.3 , we prove that the protocol Π_{acc}, when instantiated by an accumulator scheme that satisfies the security properties in [57], UC realizes $\mathcal{F}_{\text{acc}}$ as stated in the next theorem.

Theorem 6.2. *The protocol $\Pi_{\text{acc}}(\mathbf{P}, \mathcal{F}_{\text{CRS}}^{\text{Gen}}, \Sigma_{\text{acc}})$ described in Figure 19 UC-realizes $\mathcal{F}_{\text{acc}}(\mathbf{P})$ in the $\mathcal{F}_{\text{CRS}}^{\text{Gen}}$-hybrid model if and only if Σ_{acc} = (Gen, Update, WitUp, VerStatus) satisfies Correctness and Soundness.*

Moreover, if Σ_{acc} is instantiated with the accumulator scheme in [57], then $\Pi_{\text{acc}}(\mathbf{P}, \mathcal{F}_{\text{CRS}}^{\text{Gen}}, \Sigma_{\text{acc}})$ UC-realizes $\mathcal{F}_{\text{acc}}(\mathbf{P})$ without trusted party.

Proof. ($\Rightarrow$)For proving the first direction, let us assume that Σ_{acc} does not satisfy Correctness or Soundness. We construct an environment $\mathcal{Z}^*$ such that for a dummy adversary (cf. Definition A.1 and [13, section 4.3.2]) $\mathcal{A}_{\text{dummy}}$ and for every simulator $\mathcal{S}$ our $\mathcal{Z}^*$ distinguishes the real from the ideal execution with probability $\tilde{\beta}(\lambda)$ where λ the security parameter and $\tilde{\beta}$ a non-negligible function. Formally:

$$\left| \Pr\left[\mathsf{EXEC}_{\mathcal{Z}^*,\mathcal{S}}^{\mathcal{F}_{\text{acc}}}(\lambda) = 1\right] - \right. \\ \left. - \Pr\left[\mathsf{EXEC}_{\mathcal{Z}^*,\mathcal{A}_{\text{dummy}}}^{\Pi_{\text{acc}}}(\lambda) = 1\right] \right| \geq \tilde{\beta}(\lambda). \tag{6}$$

Specifically, let us assume that Soundness property is not satisfied. This means that there is an adversary $\mathcal{B}$ that wins the game in [57, definition 2, p.7] with probability greater than $\beta(\lambda)$, where $\beta()$ a non-negligible function. We construct $\mathcal{Z}^*$ as follows:

Initially, $\mathcal{Z}^*$ asks for the accumulator's algorithms by sending (sid, Retrieve) to an uncorrupted party. If $\mathcal{S}$ does not provide $\mathcal{F}_{\text{acc}}$ with the ones in Σ_{acc}, then $\mathcal{Z}^*$ can trivially distinct the two settings with probability 1 and this completes the proof for the first direction.

In the case that $\mathcal{S}$ provides $\mathcal{F}_{\text{acc}}$ with the same algorithms as in Σ_{acc}, $\mathcal{Z}^*$ does: It executes internally the adversary $\mathcal{B}$ as if it was the challenger of the Soundness property. Initially $\mathcal{Z}^*$ sends (sid, Setup) to an uncorrupted party P and gets back the initial accumulator value α_0. Then, $\mathcal{Z}^*$ provides α_0 to $\mathcal{B}$ as if it was the challenger of the game. For every $x_l \neq \bot$ element $\mathcal{Z}^*$ gets from $\mathcal{B}$ along with the current accumulated value α_l, it sends (sid, Update, α_l, x_l) to the same honest party P. Upon receiving (sid, Update, $\alpha_l, x_l, \alpha_{l+1}, w_{l+1}^{x_l}$, upmsg$_{l+1}$) from P, $\mathcal{Z}^*$ sends (α_{l+1}, $w_{l+1}^{x_l}$, upmsg$_{l+1}$) to $\mathcal{B}$ as if it was the challenger of the game and repeats until it receives an $x_l^* = \bot$ from $\mathcal{B}$. Let us assume that $\mathcal{B}$ sends $x_l^* = \bot$ after t_{fin} queries. In that case, $\mathcal{B}$ sends (x, w) to $\mathcal{Z}^*$. We know that this x is not previously queried but still the algorithm VerStatus($\alpha_{t_{\text{fin}}}, x, w$) will return 1 with probability greater that $\beta(\lambda)$ from our assumption. Based on that, $\mathcal{Z}^*$ sends (sid, Ver_Status, $\alpha_{\text{fin}}, x, w$) to any honest party P. Observe that, if we are in the ideal setting, with probability more than $\beta(\lambda)$, P will return (sid, Ver_Status, $\alpha_{\text{fin}}, x, w, \bot$) to $\mathcal{Z}^*$. On the contrary, if we are in the real setting, P will return (sid, Ver_Status, $\alpha_{\text{fin}}, x, w, x$), where $x \in \{0, 1\}$. As a result, $\mathcal{Z}^*$ will distinct the real from the ideal setting with probability more than $\beta(\lambda)$, thus non-negligible.

Lets us assume that the Correctness property is not satisfied. This means that there is an x and an additional set of values $[y_1, \ldots, y_{t_x-1}]$, $[y_{t_x+1}, \ldots, y_t]$ that after updating the witness of x with the most recent addition in the accumulator, the algorithm VerStatus(α_t, x, w_t^x) returns 0 with at least non-negligible probability $\beta'(\lambda)$ (where α_t the latest accumulated value, and w_t^x the updated witness of x). It is easy to observe that, if $\mathcal{Z}^*$ sends that x and the mentioned additions in the ideal execution, $\mathcal{F}_{\text{acc}}$ will return $\bot$ with probability $\beta'(\lambda)$. Specifically, if the number of added elements after x are 0, then $\mathcal{Z}^*$ sends (sid, Update, α, x) to an honest party P for a previously returned accumulated value α and $\mathcal{F}_{\text{acc}}$ returns (sid, Update, $\alpha, x, \bot$) message with probability at least $\beta'(\lambda)$. In the case that the added elements after x are not 0, $\mathcal{Z}^*$ adds the element x and the remaining elements by sending an Update command message. Next, $\mathcal{Z}^*$ updates the initial witness of x by sending a Wit_Up command message along with the update messages after the addition of x. Observe that $\mathcal{F}_{\text{acc}}$ will return a $\bot$ message with probability at least $\beta'(\lambda)$. On the contrary, in the real setting the $\mathcal{Z}^*$ receives back the updated witness. This completes the first direction of the proof.

($\Leftarrow$). For proving the second direction, assume that $\Pi_{\text{acc}}(\mathbf{P}, \Sigma_{\text{acc}})$ does not UC-realize $\mathcal{F}_{\text{acc}}$. This means that for a dummy adversary $\mathcal{A}_{\text{dummy}}$ and for every simulator $\mathcal{S}$ there exists an environment $\mathcal{Z}$ such that equation 6 holds. We show that Σ_{acc} does not satisfy either Correctness or Soundness.

If Σ_{acc} satisfies Correctness, we construct an adversary $\mathcal{B}$ that wins the soundness game with probability greater than $\beta(\lambda)$, where β is a non-negligible function. Given that Σ_{acc} satisfies Correctness, observe that the probability $\mathcal{F}_{\text{acc}}$ to return a $\perp$ message from the command messages (Update,Wit_Up,Retrieve) given by $\mathcal{Z}$ is negligible. The only way for $\mathcal{F}_{\text{acc}}$ to return a $\perp$ message given that $\mathcal{S}$ provides the algorithms in Σ_{acc}, is when $\mathcal{Z}$ sends the command message $(\text{sid}, \text{Ver\_Status}, \alpha, x, w)$ to an honest party for a value x and the witness of it w such that it has not been accumulated before and the verification algorithm $\mathtt{VerStatus}(\alpha, x, w)$ returns 1.

We define $\mathcal{S}^*$ as follows: Upon receiving the corruption vector from $\mathcal{Z}$ it forwards it to $\mathcal{A}_{\text{dummy}}$ as if it was form $\mathcal{Z}$. Upon receiving it back from $\mathcal{A}_{\text{dummy}}$ it forwards it to $\mathcal{F}_{\text{acc}}$. Upon receiving (sid, Gen) from $\mathcal{F}_{\text{acc}}$, $\mathcal{S}^*$ returns the algorithms in Σ_{acc}. Whatever message receives from $\mathcal{Z}$ or from $\mathcal{F}_{\text{acc}}$ from behalf of a corrupted party, it forwards it to $\mathcal{A}_{\text{dummy}}$ as if it was that party.

From our assumption we know that for such an $\mathcal{S}^*$ there is an environment $\mathcal{Z}^*$ such that equation 6 holds.

Given that $\mathcal{Z}^*$ we construct an adversary $\mathcal{B}$ that internally executes $\mathcal{Z}^*$ to win the Soundness game with non-negligible probability. $\mathcal{B}$ picks at random an honest party P (we know that there exist at least one honest party, else the real from the ideal execution would be indistinguishable). Whatever $(\text{sid}, \text{Update}, \alpha, x)$ command $\mathcal{Z}^*$ sends to $\mathcal{B}$ as if it was the honest party P, $\mathcal{B}$ forwards x to the challenger of the Soundness game and receives back the new accumulated value α_t, the witness w^x, and the update message upmsg_t. Then $\mathcal{B}$ returns $(\text{sid}, \text{Update}, \alpha, x, \alpha_t, w^x_, \text{upmsg}_t)$ to $\mathcal{Z}$ as if it was P. At some point, from our hypothesis, $\mathcal{Z}^*$ sends $(\text{sid}, \text{Ver\_Status}, \alpha, \mathtt{VerStatus}, x, w)$ to $\mathcal{B}$ playing the role of an honest party (not necessary the honest party P), such that x was not queried before by $\mathcal{Z}^*$ and $\mathtt{VerStatus}(\alpha, x, w) = 1$ for a returned accumulated value α received before with probability $\tilde{\beta}(\lambda)$. The probability this party to be P is equal with $1/|\mathbf{P}_{\text{corr}}|$. Given that it was P, $\mathcal{B}$, after checking that $\mathtt{VerStatus}(\alpha, x, w) = 1$ for an unqueried x for an existing accumulated value α, sends the special symbol $\perp$ to the challenger and then sends (x, w). Observe that the probability for $\mathcal{B}$ to win the game is $\tilde{\beta}(\lambda)/|\mathbf{P}_{\text{corr}}|$, thus non-negligible. This completes the proof.

□

H PROOF OF THEOREM 6.3

Theorem 6.3 *The protocol* $\Pi_{elig}(\mathbf{V}, \text{SA}, \mathcal{F}_{\text{acc}}, \mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{SOK}}, \mathcal{F}_{\text{BC}}, \text{delay\_cast}, \mathtt{Status})$ *described in Figure 3 UC-realizes* $\mathcal{F}_{\text{elig}}(\mathbf{V}, \text{SA}, \text{delay\_cast}, \mathtt{Status})$ *in the* $(\mathcal{F}_{acc}, \mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{SOK}}, \mathcal{F}_{\text{BC}}, \mathcal{G}_{\text{clock}})$*-hybrid model.*

Proof. We show that for every adversary $\mathcal{A}$ there is a simulator $\mathcal{S}$ such that for every environment $\mathcal{Z}$ cannot distinguish the ideal from the real execution except with negligible probability (cf. Definition A.1). More formally:

$$\left|\Pr\left[\mathsf{EXEC}^{\mathcal{F}_{\text{elig}}}_{\mathcal{Z},\mathcal{S}}(\lambda) = 1\right] - \Pr\left[\mathsf{EXEC}^{\Pi_{\text{elig}}}_{\mathcal{Z},\mathcal{A}}(\lambda) = 1\right]\right| = \mathsf{negl}(\lambda). \quad (7)$$

We construct such an $\mathcal{S}$ as follows:

Setup: Upon receiving $(\text{sid}, \text{Setup\_Elig}, \mathbf{V}_{\text{elig}}, t_{\text{cast}}, t_{\text{open}})$ from $\mathcal{F}_{\text{elig}}$, $\mathcal{S}$ sends (sid, Gen) to $\mathcal{A}$ as if it was $\mathcal{F}_{\text{acc}}$. Upon receiving $(\text{sid}, \text{Gen}, \mathtt{Gen}, \mathtt{Update}, \mathtt{WitUp}, \mathtt{VerStatus})$ from $\mathcal{A}$, it computes the initialization triple $(\alpha_0, m_0, v_0) \leftarrow \mathtt{Gen}(1^\lambda)$. If $\mathtt{VerStatus}(\alpha_0, \text{Null}, v_0) = 0$, it returns $\perp$ to $\mathcal{F}_{\text{elig}}$. Else, it sets $St_{\text{gen}} = (\alpha_0, m_0, v_0)$.

Next, $\mathcal{S}$ sends $(\text{sid}, \text{Com\_Setup\_Req}, \text{ssid})$ for a random ssid to $\mathcal{A}$ as if it was $\mathcal{F}_{\text{NIC}}$. Upon receiving $(\text{sid}, \text{Com\_Setup\_Req}, \text{ssid}, m)$ from $\mathcal{A}$, $\mathcal{S}$ parses m as $(\text{cparcom}, \mathtt{COM.TrapCom}, \mathtt{COM.TrapOpen}, \mathtt{COM.Verify}, \text{ctdcom})$ and stores all algorithms.

It sends $(\text{sid}, \text{Setup})$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\text{SOK}}$. Upon receiving $(\text{sid}, \text{Algorithms}, \mathtt{Verify}, \mathtt{Sign}, \mathtt{SimSign}, \mathtt{Extract})$ as if it was $\mathcal{F}_{\text{SOK}}$, it stores all algorithms.

It sets $\text{reg.par} \leftarrow (\mathbf{V}_{\text{elig}}, \vec{t} := (t_{\text{cast}}, t_{\text{open}}, \text{delay\_cast}), St_{\text{gen}})$ and provides $\mathcal{A}$ with $(\text{sid}, \text{Broadcast}, \text{reg.par})$ as if it was $\mathcal{F}_{\text{BC}}$. Upon receiving the token back from $\mathcal{A}$, $\mathcal{S}$ defines the algorithms $(\mathtt{GenCred}, \mathtt{AuthBallot}, \mathtt{VrfBallot}, \mathtt{UpState})$ as follows:

```
GenCred($1^\lambda$, reg.par)
(1) The parameters cparcom, ctdcom are hard-coded.
(2) It picks cr $\xleftarrow{\$} \{0,1\}^{p_4(\lambda)}$.
(3) It computes $(\hat{\text{cr}}, \text{cinfo}) \leftarrow$ $\mathtt{COM.TrapCom}(\text{sid}, \text{cparcom}, \text{ctdcom})$.
(4) It computes copen $\leftarrow \mathtt{COM.TrapOpen}(\text{sid}, \text{cr}, \text{cinfo})$.
(5) It returns $(\text{cr}, \hat{\text{cr}}, \text{copen})$.
```

```
AuthBallot($v$, cr, $St_{\text{fin}}$, reg.par, copen)
(1) It computes $\sigma \leftarrow \mathtt{SimSign}(v, (\text{cr}, St_{\text{fin}}))$.
(2) If $\mathtt{Verify}(v, (\text{cr}, St_{\text{fin}}), \sigma) = 1$, it returns $\sigma$, else it returns $\perp$.
```

```
VrfBallot($v, \vec{\sigma} = (\text{cr}, \sigma), St_{\text{fin}}$, reg.par)
(1) The relation $M_L$ is hard-coded.
(2) It computes $w \leftarrow \mathtt{Extract}(v, (\text{cr}, St_{\text{fin}}), \sigma)$
(3) If $M_L((\text{cr}, St_{\text{fin}}), w) = 1$, it returns $\mathtt{Verify}(v, (\text{cr}, St_{\text{fin}}), \sigma)$.
(4) If $M_L((\text{cr}, St_{\text{fin}}), w) = 0$ and $\mathtt{Verify}(v, (\text{cr}, St_{\text{fin}}), \sigma) = 1$, it returns $\perp$.
(5) In any other case it returns $\mathtt{Verify}(v, (\text{cr}, St_{\text{fin}}), \sigma)$.
```

```
UpState($St_{\text{gen}}, \{\hat{\text{cr}}_j\}_{j=1}^{p_5(\lambda)}$)
(1) For each $j = 1$ to $p_5(\lambda)$ it computes $(\alpha_j, m_j, w_j, \text{upmsg}_j) \leftarrow \mathtt{Update}(\alpha_{j-1}, m_{j-1}, \hat{\text{cr}}_j)$ and it stores all the resulting values.
(2) It returns $St_{\text{fin}} = \alpha_{p_5(\lambda)}$
```

Then, $\mathcal{S}$ sends $(\text{sid}, \text{Set\_Up}, \mathtt{GenCred}, \mathtt{AuthBallot}, \mathtt{VrfyBallot}, \mathtt{UpState}, St_{\text{gen}})$ to $\mathcal{F}_{\text{elig}}$. Upon receiving $(\text{sid}, \text{Elig\_Par}, \text{reg.par})$ from $\mathcal{F}_{\text{elig}}$ it stores reg.par.

Credential generation: Upon receiving $(\text{sid}, \text{Gen\_Cred}, \hat{cr}, V)$ from $\mathcal{F}_{\text{elig}}$, $\mathcal{S}$ sends $(\text{sid}, \text{Broadcast}, (V, \hat{\text{cr}}))$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\text{BC}}$. Upon receiving the token back from $\mathcal{A}$, it sends $(\text{sid}, \text{Gen\_Cred}, \hat{\text{cr}}, V)$ to $\mathcal{F}_{\text{elig}}$.

Whatever command/message received on behalf of a corrupted party from $\mathcal{F}_{\text{elig}}$, $\mathcal{S}$ forwards it to $\mathcal{A}$ as if it was that party and returns whatever message it receives from $\mathcal{A}$ for that party to $\mathcal{F}_{\text{elig}}$.

As can be seen, the algorithm `AuthBallot` uses the algorithm `SimSign` instead of the `Sign` exactly as $\mathcal{F}_{\text{SOK}}$ does. With this, we can guarantee that the distribution of signed messages between $\mathcal{F}_{\text{elig}}$ and Π_{elig} are the same. Observe that, if a party requests the signature for a message from $\mathcal{F}_{\text{SOK}}$, if the verification returns "false" then $\mathcal{F}_{\text{SOK}}$ returns $\perp$. This behaviour is integrated into `AuthBallot` algorithm and thus the same happens in $\mathcal{F}_{\text{elig}}$ so that the real and ideal executions match. Note that this does not mean that $\mathcal{F}_{\text{elig}}$ does not capture the correctness of a signed ballot and that the property depends on the algorithms; the correctness of an authenticated ballot is captured with the command message AUTH_BALLOT as can be seen in Figure 13.

Similarly, the algorithm `GenCred` integrates a part of what $\mathcal{F}_{\text{NIC}}$ does so that to match both real and ideal execution and specifically the distributions of the credential generation. The same applies for the algorithm `UpState` and $\mathcal{F}_{\text{acc}}$.

Next, we show that the algorithms `GenCred`, `AuthBallot`, `VrfBallot`, `UpState` that $\mathcal{S}$ defines are such that the steps followed in Π_{elig} when a AUTH_BALLOT, VER_BALLOT, or LINK_BALLOTS command is sent by $\mathcal{Z}$, preserve the respective properties captured by $\mathcal{F}_{\text{elig}}$. Recall that $\mathcal{F}_{\text{elig}}$ captures *eligibility* (only the ballots authenticated via the command AUTH_BALLOT pass the verification test via the command VER_BALLOT); *one-voter-one-vote* (multiple ballots from an eligible voter are dropped all except one via the command LINK_BALLOTS). Below we show how each command message in Π_{elig} is related with the algorithms $\mathcal{S}$ provides $\mathcal{F}_{\text{elig}}$ so to match both executions.

- AUTH_BALLOT: In Π_{elig}, the party P computes the final accumulated value by providing $\mathcal{F}_{\text{acc}}$ with the list of all public credentials. Next, for the resulting accumulated value, she requests the witness of her credential from $\mathcal{F}_{\text{acc}}$. Finally, P requests from $\mathcal{F}_{\text{SOK}}$ the signature of knowledge defined for the relation M_L for the requested ballot. Observe that $\mathcal{F}_{\text{SOK}}$ signs the ballot via the algorithm `SimSign` only if P provides a valid witness. A valid witness can be provided if and only if a party is eligible. The only way for a non-eligible party to get access to a valid witness is by extracting it from the signature or the commitment, which is impossible as the `SimSign` and `COM.TrapCom` does not accept the witness or the credential as input.
  In case the witness is not valid or the resulting signature does not pass the verification test of $\mathcal{F}_{\text{SOK}}$, $\mathcal{F}_{\text{SOK}}$ returns $\perp$. This is exactly what $\mathcal{F}_{\text{elig}}$ captures except the case of a valid signature and an invalid verification test as in $\mathcal{F}_{\text{SOK}}$. We include this case in the algorithm `AuthBallot`. Thus, both real and ideal execution behave the same for that command message.
- VER_BALLOT: This command is handled solely by $\mathcal{F}_{\text{SOK}}$. Specifically, $\mathcal{F}_{\text{SOK}}$ returns true if it previously has recorded the input signature (meaning that the signature is issued by an eligible voter, item 2) in $\mathcal{F}_{\text{elig}}$), exactly like $\mathcal{F}_{\text{elig}}$.
  In case the signature is not a legitimate one, meaning that it was not previously recorded in $\mathcal{F}_{\text{SOK}}$, then $\mathcal{F}_{\text{SOK}}$ checks if a valid witness can be extracted by applying the `Extract` function. If yes, then it returns true. We included that check in `VrfBallot` so to capture the case that the verification returns "true" but there is no recorded signature for corrupted parties (item 3) in $\mathcal{F}_{\text{elig}}$). Observe that if a party is corrupted still can be eligible and thus provide a signature under a valid witness.
  In the case that $\mathcal{F}_{\text{SOK}}$ cannot extract any witness, a *forgery* has occurred and it returns $\perp$. This means that either the party is not eligible (item 3) in $\mathcal{F}_{\text{elig}}$) or the adversary managed to make a forgery (item 4) in $\mathcal{F}_{\text{elig}}$). Thus, both real and ideal execution behave the same for this command message.
- LINK_BALLOTS: Finally, in Π_{elig} the voter checks if the input signatures are valid by forwarding them in $\mathcal{F}_{\text{SOK}}$. In turn, $\mathcal{F}_{\text{SOK}}$ checks the validity of the signatures similar to the case that the voter receives the command VER_BALLOT as described before. If this is the case and the providing credentials are the same (meaning that the valid ballots are originated from the same voter) then the voter returns 1, meaning that the ballots are related. Similarly, in $\mathcal{F}_{\text{elig}}$ the functionality checks if there are tuples both from corrupted and uncorrupted, yet eligible, voters with the same credential. If this is the case, $\mathcal{F}_{\text{elig}}$ returns 1. The only way for these commands to not behave the same is $\mathcal{F}_{\text{SOK}}$ to verify as true a signature for a non-eligible party. This case is impossible as we explained in the previous paragraph. Thus, both real and ideal execution behave the same for this command message.

By the above, the distributions of real and ideal setting are exactly the same thus the proof is complete. □

I PROOF OF THEOREM 6.4

THEOREM 6.4. *The protocol* $\Pi_{\text{vm}}(\mathbf{V}, \text{SA}, \mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{an.BC}}^{\ell,1,p}, \texttt{Status})$ *described in Figure 4 UC-realizes* $\mathcal{F}_{\text{vm}}(\mathbf{V}, \text{SA}, \text{delay\_gen}, \text{delay\_cast}, \texttt{Status})$ *in the* $(\mathcal{F}_{\text{TLE}}^{\text{leak,delay\_gen}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{an.BC}}^{l,1,p}, \mathcal{G}_{\text{clock}})$*-hybrid model, where* $\text{leak}(\text{Cl}) = \text{Cl} + 1$, $\text{delay\_cast} = l + 1$, *and* $p(\lambda)$ *is the length of a pair of a ballot* v *and authentication data* $\vec{\sigma}$.

PROOF. In cases where a corrupted party receives input and we do not describe her behaviour, we assume that the message is sent to $\mathcal{S}$ from $\mathcal{F}_{\text{vm}}$ and $\mathcal{S}$ forwards that message to $\mathcal{A}$ as if it was from that party. Then $\mathcal{S}$ returns to $\mathcal{F}_{\text{vm}}$ whatever it receives from $\mathcal{A}$.

We describe the ideal adversary $\mathcal{S}$. When $\mathcal{S}$ receives the corruption vector from $\mathcal{Z}$, $\mathcal{S}$ forwards it to $\mathcal{A}$ as if it was from $\mathcal{Z}$. When $\mathcal{S}$ receives back the corruption vector from $\mathcal{A}$ playing the role of both of $\mathcal{F}_{\text{TLE}}, \mathcal{F}_{\text{BC}}$, $\mathcal{S}$ forwards it to $\mathcal{F}_{\text{vm}}$. When $\mathcal{S}$ receives the election information (sid, ELECTION_INFO, vote.par) from $\mathcal{F}_{\text{vm}}$, $\mathcal{S}$ sends (sid, BROADCAST, (SA, vote.par)) to $\mathcal{A}$ as if it was $\mathcal{F}_{\text{BC}}$.

Upon receiving a GEN_BALLOT request from $\mathcal{F}_{\text{vm}}$ on behalf of a voter V, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\text{TLE}}$ and it returns the response of $\mathcal{A}$ to $\mathcal{F}_{\text{vm}}$. Upon receiving an ADVANCE_CLOCK command from $\mathcal{G}_{\text{clock}}$ on behalf of a voter V, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was $\mathcal{G}_{\text{clock}}$. Upon receiving an UPDATE command as if it was $\mathcal{F}_{\text{TLE}}$ from $\mathcal{A}$, it forwards the message to $\mathcal{F}_{\text{vm}}$.

Upon receiving $(\text{sid}, \text{CAST\_BALLOT}, (v, \vec{\sigma}))$ from $\mathcal{F}_{\text{vm}}$ it sends $(\text{sid}, \text{BROADCAST}, (v, \vec{\sigma}))$ to $\mathcal{A}$ as if it was $\mathcal{F}_{\text{an.BC}}^{\text{delay\_cast},1,p}$.

When $\mathcal{S}$ receives a CAST request from $\mathcal{F}_{\text{vm}}$ on behalf of a corrupted voter V, it forwards the message as a BROADCAST request to $\mathcal{A}$ as if it was from $\mathcal{F}_{\text{an.BC}}^{\text{delay\_cast},1,p}$ and it returns the message it received from $\mathcal{A}$ back to $\mathcal{F}_{\text{vm}}$.

Upon receiving $(\text{sid}, \textsc{Open}, v)$ from $\mathcal{F}_{\text{vm}}$ (where v is a ballot not generated by $\mathcal{F}_{\text{vm}}$), $\mathcal{S}$ sends $(\text{sid}, \textsc{Dec}, v, t_{\text{open}})$ to $\mathcal{A}$ as if it was from $\mathcal{F}_{\text{TLE}}$. When $\mathcal{S}$ receives $(\text{sid}, \textsc{Dec}, v, t_{\text{open}}, o)$ from $\mathcal{A}$, it returns the message $(\text{sid}, \textsc{Open}, v, o)$ to $\mathcal{F}_{\text{TLE}}$.

Upon receiving $(\text{sid}, \textsc{Leakage})$ from $\mathcal{Z}$, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was from $\mathcal{Z}$. Upon receiving $(\text{sid}, \textsc{Leakage})$ from $\mathcal{A}$ as if it was $\mathcal{F}_{\text{TLE}}$ it forwards the message to $\mathcal{F}_{\text{vm}}$ and returns to $\mathcal{A}$ whatever receives. Observe that the simulation fails only in case that the provided tuples from $\mathcal{F}_{\text{vm}}$ are not the same as the provided one from $\mathcal{F}_{\text{TLE}}$. Due to the fact that all honest parties encrypt with time t_{open} and $\text{delay\_cast} \geq 1$ this never occurs. Specifically, $\mathcal{A}$ expects to receive all pairs of plaintexts/ciphertexts from $\mathcal{F}_{\text{TLE}}$ at time $\text{Cl} \geq t_{\text{open}} - 1$ (because $\text{leak} = 1$). By that time it holds that $\texttt{Status}(\text{Cl}, \vec{t}, \text{Cred}) = \texttt{Status}(\text{Cl}, \vec{t}, \text{Cast}) = \texttt{Status}(\text{Cl}, \vec{t}, \text{Tally}) = \perp$ or $\texttt{Status}(\text{Cl}, \vec{t}, \text{Tally}) = \top$. Thus, $\mathcal{S}$ can retrieve also all the pairs of plaintexts/ciphertexts from $\mathcal{F}_{\text{vm}}$ at time $\text{Cl} = t_{\text{open}} - 1$. $\mathcal{S}$ reads the time Cl from $\mathcal{G}_{\text{clock}}$. Then $\mathcal{S}$ playing the role of $\mathcal{F}_{\text{TLE}}$ returns to $\mathcal{A}$ all the maliciously generated ciphertexts with time labelling until time $\text{leak}(\text{Cl})$.

The distribution of messages is the same in both the ideal and the hybrid setting. As a result, the simulation is perfect. □