E-cclesia: Universally Composable Self-Tallying Elections

Myrto Arapinis[1], Nikolaos Labrou[1], Lenka Mareková[2], and Thomas Zacharias[1]

[1] The University of Edinburgh, UK
[2] Royal Holloway, University of London, UK

Abstract. The concept of a self-tallying election (STE) scheme was first introduced by Kiayias and Yung in [22] and captures electronic voting schemes in which the tallying authorities are the voters of the election themselves. This type of electronic voting is particularly compatible with and suitable for (but not only) Blockchain governance, where governance is expected to be maintained in a fully distributed manner.
In this work, we formalize the requirements for secure STE schemes in the Universal Composability (UC) framework. Our model captures the standard voting properties of *eligibility*, *fairness*, *vote-privacy*, and *one voter-one vote*. We present E-CCLESIA, a new family of STE schemes, and prove that it securely UC-realizes the STE functionality. We propose a first concrete instantiation of E-CCLESIA using RSA accumulators in combination with the time-lock puzzles by Rivest *et al.* [30]. To this end, we provide *en passant* the first UC treatment of dynamic accumulators in the public setting as well as time-lock encryption (TLE) schemes, and prove the equivalence of our UC definitions to the standard property-based accumulator definitions.

Keywords: e-voting, accumulators, time-lock encryption

1 Introduction

Voting has been a fundamental component of democratic societies for more than 2500 years. Electronic voting, or e-voting, refers to any voting procedure that involves computer systems to support one (or more) stages, *e.g.* casting and/or tallying. The aspiration being that electronic voting systems could offer elections with higher voter participation and better accuracy, while also providing enhanced security guarantees such as privacy and verifiability, even in the face of dishonest election authorities. With this drive, electronic voting has been of interest to the research community and governments all over the globe for the last three decades. Several cryptographic protocols have been proposed, implemented, and deployed for electronic voting [1,13,21,23,31,29,22,32,20].

The existing electronic voting schemes differ widely, and can be classified per their underlying trust assumptions. On one end, one finds fully centralized schemes which are often vulnerable to large scale attacks on robustness or privacy

due to their centralized nature [4,34,16]; on the other end, schemes that are fully decentralized except maybe for setting up the election, where the (tallying) authorities are the voters of the election themselves. This second type has been termed *self-tallying election schemes* [29,22,32,20]. In between the two ends, and to avoid the difficulties of self-tallying elections (STE) and dangers of centralized elections, a range of e-voting schemes which distribute the trust among a small set of authorities have also been proposed. Such systems achieve security as long as a subset of those authorities is honest. For instance, in mixnet-based schemes at least one authority needs to be honest [1,31], and in threshold schemes at most a fraction ϵ of the authorities can be corrupt and collude [23,21].

Of course, other things being equal, electronic voting schemes at the decentralized end of the spectrum are preferable as they can withstand more powerful adversaries. Self-tallying election schemes are a step forward in this direction, and move us one step closer to direct democracy. With the recent developments of Blockchain technologies and the appeal for on-chain distributed governance, such mechanisms are all the more topical. However, they present major challenges, which explains why full decentralization often needs to be abandoned in favor of achieving all the conflicting requirements that an electronic voting scheme should ensure. The main challenges STEs pose are in guaranteeing that no one (or no coalition of) voter(s) can boycott the election, no intermediate results are being leaked during the casting phase (*fairness*), and no vote can be linked back to the voter that cast it (*vote-privacy*). We refer the reader to [22] for a more in depth discussion on the difficulties STEs present.

Our goal in this paper is to further investigate the self-tallying paradigm, in light of the recent developments in distributed systems that we have been witnessing with the development of Blockchain technologies. Indeed, Nakamoto's Bitcoin protocol [27] for a decentralized banking system has paved a new way for a decentralized future. With this as our credo too, we develop E-CCLESIA, a new family of self-tallying election schemes that satisfy the standard *eligibility*, *fairness*, *vote-privacy*, and *one voter-one vote* requirements for electronic voting. We formally prove that E-CCLESIA satisfies these properties in the Universal Composability paradigm proposed by Canetti in [8], which is a state-of-the-art framework for specifying and analyzing cryptographic protocols, especially when these are run under concurrent sessions. E-CCLESIA further satisfies *individual* and *universal verifiability* as a direct consequence of its decentralized nature, its reliance on a broadcast channel, and *ballot authentication*. Finally, we provide the first provably secure concrete instance of E-CCLESIA. In particular, inspired by Zerocoin [26], we rely on RSA dynamic accumulators for unlinking cast votes from the voters who cast them. For fairness, we rely on a *time-lock encryption* (TLE) scheme, such as the one derived from the time-lock puzzles of Rivest *et al.* [30].

Our contributions and roadmap. In Section 2 we present the necessary background. The subsequent sections are dedicated to the contributions of this work which can be summarized as follows:

Section 3 We provide a UC treatment (i.e. an ideal functionality) of self-tallying elections (STE) and extract its two basic sub-functionalities, namely an *eligibility* and a *vote management* functionality. We then present E-CCLESIA, a family of electronic voting schemes UC-realizing STEs via these two sub-functionalities. The first provably secure concrete instance of E-CCLESIA can be derived from the next two sections.

Section 4 We provide the first UC definition for time-lock encryption (TLE) via the functionality $\mathcal{F}_{\mathsf{TLE}}$, and show that a hybrid protocol that uses $\mathcal{F}_{\mathsf{TLE}}$ as a building block can UC-realize the vote management functionality. Moreover, we demonstrate a UC realization of $\mathcal{F}_{\mathsf{TLE}}$ via Rivest *et al.* time-lock puzzles [30].

Section 5 We provide the first UC definition for dynamic accumulators in the public setting, and prove that any UC-secure dynamic accumulator can be used to realize the eligibility functionality. We further prove that quasi-commutative dynamic accumulators as defined in [7], *e.g.* RSA accumulators are UC-secure dynamic accumulators, and thus provide a concrete realization of the eligibility module of E-CCLESIA.

1.1 Related work

Self-tallying. Two self-tallying protocols established in the literature are the ones of Okamoto *et al.* [29] and Kiayias *et al.* [22]. In the former [29], an election authority uses blind signatures [12] in order to authenticate the eligible voters and their ballots. Next, each voter casts the commitment [5] of their ballot signed by the election authority over an anonymous channel. After the casting phase is over, each voter opens their commitment by casting the de-commitment key over the anonymous channel, so that the tally can be produced. One of the main drawbacks of this proposal is leakage of intermediate results, which violates the fairness property. In the latter proposal [22], the voters agree on a table where the sum of each row equals the voting choice of a voter. The link between the identity of a voter and their vote cannot be inferred from the table. In this proposal, the authors attempt to address the fairness problem by introducing a "dummy" party that casts its ballot last so that no intermediate results are leaked. However, this permits a single voter to be able to cause the protocol to abort. Moreover, the inclusion of such a "dummy" party is still not an optimal solution to the fairness problem as it re-introduces a trusted party in the STE setting. While self-tallying protocols such as [32,20] improve these proposals in terms of efficiency, they still share the limitations of [22]. Our work addresses these limitation, while preserving all the standard security requirements. In addition, in [32] they define an ideal functionality for e-voting but with the limitation that every authority must agree so that the tally can be produced. Moreover, their functionality didn't consider a global coordination between protocol phases(e.g cast, tally phase) via a global clock [2] making it hard to be realised by self tallying protocols without the need of a trusted third party. We address these limitations to our functionality.

Time-lock encryption. TLE is a cryptographic primitive that allows a ciphertext to be decrypted only after a specific time period has elapsed. In some proposals, decryption can further be performed without requiring knowledge of any secret information. Previously proposed constructions are based either on witness encryption [18] or symmetric encryption [24]. The authors of these works provide game-based definitions in order to argue about the security of their constructions. Unfortunately, game-based definitions do not capture the variety of adversarial behaviour the UC framework [8] does. Moreover, the task of transferring these definitions to the UC setting is quite challenging due to some incompatibility between the two settings (concrete vs. asymptotic adversary).
Dynamic accumulators. Accumulators [17,14,35,19,33,7,15] are a well studied cryptographic primitive that take as an input a list of objects and output a representative value instead of the whole list. Despite the maturity of the primitive in terms of years of study, the only security arguments that have been provided are game-based. Recently, a work by Baldimtsi *et al.* [3] provides a UC treatment for accumulators with a trusted accumulator manager. The manager can accumulate and delete values from the list by using some trapdoor information. In our UC treatment, this trusted entity is only responsible for deletion but not for accumulation of the values (accumulation is a publicly available procedure), which is vital for our approach.

2 Preliminaries

Throughout the paper, we use λ as the security parameter and $\cdot$ to denote a wildcard character.

2.1 The Universal Composability model

Overview. The security analysis in this work follows the *Universal Composability (UC)* paradigm introduced by Canetti in [8], 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 by 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* $\mathcal{F}$, which 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 said task should satisfy (not only Π). E.g., 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 $\mathcal{F}$ pass that messages to $\mathcal{S}$ and returns back to P whatever received from $\mathcal{S}$. 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 $\text{EXEC}^{\Pi}_{\mathcal{Z},\mathcal{A}}$ denote an execution of a real-world protocol Π in the presence of the adversary $\mathcal{A}$ scheduled by an environment $\mathcal{Z}$, and $\text{EXEC}^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}$ 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 1 (UC realization [8]). *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* $\text{EXEC}^{\Pi}_{\mathcal{Z},\mathcal{A}}$ *and* $\text{EXEC}^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}$ *are computationally indistinguishable.*

Composition and modularity. Perhaps the most prominent feature of the UC paradigm – which is at the heart of the E-CCLESIA design – 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 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 $\tilde{\Pi}^{\Pi \to \mathcal{F}}$, that enjoys the same security as $\tilde{\Pi}$. Namely, if $\tilde{\Pi}$ UC-realizes some ideal functionality $\tilde{\mathcal{F}}$, then so does $\tilde{\Pi}^{\Pi \to \mathcal{F}}$.

The power of composition facilitates the design of complicated 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.

A major advantage of the modular design in UC is that we can describe the cryptographic scheme in a partially abstract black-box way by simply using the hybrid functionalities instead of the associated real-world subroutines, and then prove security in the corresponding hybrid model. Henceforth, whenever the designer wishes to improve the scheme by implementing a more efficient version of some subroutine, they just need to prove the security of the new subroutine w.r.t. the associated hybrid functionality, which will directly imply the overall security of the improved scheme.

2.2 Setup functionalities

In Subsection 2.1, we summarized the UC model and showed how provably secure design can be done in a modular way by utilizing hybrid functionalities. We stress that in the UC literature, hybrid functionalities do not only play the role of abstracting some UC-secure real-world subroutine (e.g., a secure channel), but also formalize possible setup assumptions that are required to prove security when this is not done (and in many cases even impossible to achieve) in the "standard model". For example, this type of setup functionalities may capture the concept of a trusted source of randomness, a clock, or a Public Key Infrastructure (PKI). Moreover, these setup functionalities can be *global*, i.e. they act as shared states across multiple protocol instances and they can be accessed by other functionalities and even the environment that is external to the current session (recall that standard ideal functionalities do not directly interact with the environment). The extension of the UC framework in the presence of global setups has been introduced by Canetti *et al.* in [10]. Below, we present the setup functionalities that we consider across the E-CCLESIA design.

The global clock functionality $\mathcal{G}_{\mathsf{clock}}$. In Fig. 1, we provide the definition of a *global clock* functionality $\mathcal{G}_{\mathsf{clock}}$ similar to [2]. Time advances only when the environment has allowed all involved parties to advance.

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

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

- Upon receiving $(\mathsf{sid}, \text{ADVANCE\_CLOCK})$ from $P \in \mathbf{P}$, if $P \notin L_{\mathsf{adv}}$, then it adds P to L_{adv}, sends the message $(\mathsf{sid}, \text{ADVANCE\_CLOCK})$ to P and notifies $\mathcal{A}$ by forwarding $(\mathsf{sid}, \text{ADVANCE\_CLOCK}, P)$. If $L_{\mathsf{adv}} = \mathbf{P} \cup \mathbf{F}$, then it updates as $\mathsf{Cl} \leftarrow \mathsf{Cl} + 1$ and resets $L_{\mathsf{adv}} \leftarrow \emptyset$.
- Upon receiving $(\mathsf{sid}, \text{ADVANCE\_CLOCK})$ from $\mathcal{F} \in \mathbf{F}$, if $\mathcal{F} \notin L_{\mathsf{adv}}$, then it adds $\mathcal{F}$ to L_{adv} and sends the message $(\mathsf{sid}, \text{ADVANCE\_CLOCK})$ to $\mathcal{F}$. If $L_{\mathsf{adv}} = \mathbf{P} \cup \mathbf{F}$, then it updates as $\mathsf{Cl} \leftarrow \mathsf{Cl} + 1$ and resets $L_{\mathsf{adv}} \leftarrow \emptyset$.
- Upon receiving $(\mathsf{sid}, \text{READ\_CLOCK})$ from $X \in \mathbf{P} \cup \mathbf{F} \cup \{\mathcal{Z}, \mathcal{A}\}$, then it sends $(\mathsf{sid}, \text{READ\_CLOCK}, \mathsf{Cl})$ to X.

Fig. 1. The global clock functionality $\mathcal{G}_{\mathsf{clock}}(\mathbf{P}, \mathbf{F})$ interacting with the parties in $\mathbf{P}$, the functionalities in $\mathbf{F}$, the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$.

The random oracle functionality $\mathcal{F}_{\mathsf{RO}}$. In Fig. 2, we define a UC *random oracle (RO)* as in [28].

The common reference string functionality $\mathcal{F}_{\mathsf{CRS}}$. Another setup assumption is the *common random string* model, where a single random string is drawn from a uniform distribution over strings. Fig. 3 formally defines $\mathcal{F}_{\mathsf{CRS}}$ as given by

[8]. Note that $\mathcal{F}_{\mathsf{CRS}}$ requests permission from the simulator $\mathcal{S}$ before returning the value, disclosing the identity of the requesting party. This action is often called *(public) delayed output* in the literature. In Section 5, when we refer to e.g. $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$, we mean the output distribution of the function `Gen`.

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

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

■ Upon receiving $(\mathsf{sid}, \textsc{Query}, x)$ from $P \in \mathbf{P}$, 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 $h \xleftarrow{\$} B$, and it inserts the pair to the list $L_{\mathcal{H}} \leftarrow (x, h)$. Then it returns $(\mathsf{sid}, \textsc{Random\_Oracle}, x, h)$ to P.

Fig. 2. The random oracle functionality $\mathcal{F}_{\mathsf{RO}}$ w.r.t. domain A and range B interacting with the parties in $\mathbf{P}$.

The common reference string functionality $\mathcal{F}_{\mathsf{CRS}}(\mathbf{P}, D)$.

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

■ Upon receiving $(\mathsf{sid}, \mathrm{CRS})$ from $P_i \in \mathbf{P}$, if no value r is recorded, it picks $r \xleftarrow{\$} D$, adds P_i to L_{wait} and sends $(\mathsf{sid}, \textsc{Allow}, P_i)$ to $\mathcal{S}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Allowed}, P_i)$ from $\mathcal{S}$, if $P_i \in L_{\mathsf{wait}}$, it sends $(\mathsf{sid}, \mathrm{CRS}, r)$ to P_i and $\mathcal{S}$ and removes P_i from L_{wait}.

Fig. 3. The CRS functionality $\mathcal{F}_{\mathsf{CRS}}$ interacting with the parties in $\mathbf{P}$ and the simulator $\mathcal{S}$, parameterized by distribution D.

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

The functionality initializes a list $L_{\mathsf{pend}} \leftarrow \emptyset$ of messages pending to be broadcast.

■ Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, M)$ from $P_i \in \mathbf{P}$, it adds (M, P_i) to L_{pend} and sends $(\mathsf{sid}, \textsc{Allow}, M)$ to $\mathcal{S}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Allowed}, M)$ from $\mathcal{S}$, if $(M, P_i) \in L_{\mathsf{pend}}$, then it sends $(\mathsf{sid}, \textsc{Broadcast}, M)$ to $P_1, \ldots, P_n$, and $\mathcal{S}$. Then, it removes (M, P_i) from L_{pend}.

Fig. 4. The anonymous broadcast functionality $\mathcal{F}_{\mathsf{an.BC}}$ interacting with the parties in $\mathbf{P} = \{P_1, \ldots, P_n\}$ and the simulator $\mathcal{S}$.

The anonymous broadcast functionality $\mathcal{F}_{\mathsf{an.BC}}$. To guarantee the privacy of our STE scheme we assume that voters communicate via an *anonymous broadcast channel*. This communication interface is formalized via the functionality $\mathcal{F}_{\mathsf{an.BC}}$ described in Fig. 4.

Providing a provably UC-secure realization of $\mathcal{F}_{\mathsf{an.BC}}$ is out of the scope of this work. However, intuitively, one can instantiate an anonymous broadcast channel by deploying a blockchain (or any transaction ledger) where the users access the blockchain via an anonymous communications channel such as Tor or any mix-network routing protocol.

The certification functionality $\mathcal{F}_{\mathsf{cert}}(P, \mathbf{V})$.

- Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{S}$, if $\mathbf{V}_{\mathsf{corr}} \subseteq \mathbf{V} \cup P$, it fixes $\mathbf{V}_{\mathsf{corr}}$ as the set of corrupted parties.
- Upon receiving $(\mathsf{sid}, \textsc{Setup})$ from P, if $\mathsf{sid} = (P, \mathsf{sid}')$ for some sid', it sends $(\mathsf{sid}, \textsc{Setup})$ to $\mathcal{S}$.
- Upon receiving $(\mathsf{sid}, \textsc{Algorithms}, \texttt{Verify}, \texttt{Sign})$ from $\mathcal{S}$, it stores the algorithms and sends $(\mathsf{sid}, \textsc{Setup})$ to P.
- Upon receiving $(\mathsf{sid}, \textsc{Sign}, m)$ from P, it sets $\sigma \leftarrow \texttt{Sign}(m)$. If $\texttt{Verify}(m, \sigma) \neq 1$, it aborts. Otherwise it records (m, σ) and sends $(\mathsf{sid}, \textsc{Signature}, m, \sigma)$ to P.
- Upon receiving $(\mathsf{sid}, \textsc{Verify}, m, \sigma)$ from $V \in \mathbf{V}$, if $\texttt{Verify}(m, \sigma) = 1$, the signer is not corrupted and no entry (m, σ') for any σ' is recorded, it aborts. Otherwise it sends $(\mathsf{sid}, \textsc{Verified}, m, \texttt{Verify}(m, \sigma))$ to V.

Fig. 5. The certification functionality $\mathcal{F}_{\mathsf{cert}}$ interacting with a prover P, a set of verifiers $\mathbf{V}$ and the simulator $\mathcal{S}$.

The certification functionality $\mathcal{F}_{\mathsf{cert}}$. The registration process for voting traditionally makes use of a channel through which the voters can identify themselves to the election authority. It would be natural to utilize a form of public key infrastructure, but modelling it in UC is not straightforward, and turns out to not be necessary for the purposes of the protocol that will be introduced in Section 3. Instead, we will use a certification scheme which provides signatures not bound to keys, but to *identities*. $\mathcal{F}_{\mathsf{cert}}$, shown in Fig. 5 as defined by [8] (2005), provides commands for signature generation and verification, and is tied to a single party (so each party requires a separate instance). It can be realized as in [9] by an EUF-CMA secure signature scheme combined with a party acting as a trusted certificate authority.

3 A framework for STE schemes and the E-cclesia family

In this section, we formalize STE schemes and provide an abstract description of the E-cclesia family along the lines of modular UC design described in

Subsection 2.1. The entities involved in an STE execution include the *voters* $V_1, \ldots, V_n$ and a *setup authority* SA that is active only prior to the voting and tally period and specifies the election parameters (e.g., list of eligible voters and the period of each election phase). In our threat model, the voters are statically corrupted while SA remains honest. Intuitively, the security properties that an STE scheme should satisfy are the following:

1. *Eligibility*: only eligible voters can vote.
2. *Fairness:* before the tally phase begins, no party can learn some partial result.
3. *Voter Privacy*: the voters' identities cannot be linked to their votes.
4. *One voter-one vote*: only one vote per (eligible) voter can be included in the tally.

Our approach to formally describing the E-CCLESIA family is to first capture eligibility, fairness, and voter privacy via two main ideal modules:
(i) A *vote management functionality* $\mathcal{F}_{\mathsf{vm}}$ that handles ballot encryption, anonymous broadcast and opening.
(ii) An *eligibility functionality* $\mathcal{F}_{\mathsf{elig}}$ that is responsible for generating anonymous credentials and authenticating the ballots of the eligible voters. The functionality can also link two ballots originating from the same (corrupted) voter.

We stress that ballot encryption and opening run by $\mathcal{F}_{\mathsf{vm}}$ ensure fairness, while credential generation and ballot authentication run by $\mathcal{F}_{\mathsf{elig}}$ guarantee eligibility. Moreover, both functionalities combined safeguard voter privacy by implicitly featuring anonymous ballot authentication and casting. In addition, one voter-one vote is achieved by using $\mathcal{F}_{\mathsf{elig}}$ for discarding multiple ballots that are linked to the same (corrupted) voter.

Given the aforementioned approach, the major technical challenge in designing E-CCLESIA instantiations is to devise real-world protocols that UC-realize $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{F}_{\mathsf{elig}}$. Upon completion of this task, the step towards full STE security is merely a careful composition in terms of interface by specifying how the STE entities interact with $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{F}_{\mathsf{elig}}$, and check the information they obtain from their engagement in the overall execution, so that the four security properties are preserved.

Therefore, in order to avoid repetition and excessive formalization, this section is organized as follows: in Subsection 3.1, we present the general *ideal STE functionality* $\mathcal{F}_{\mathsf{STE}}$ that captures all four aforementioned properties. To provide intuition, we describe $\mathcal{F}_{\mathsf{STE}}$ in a complete yet not strictly formal manner. Next, the functionalities $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{F}_{\mathsf{elig}}$ are defined formally in Subsections 3.2 and 3.3, respectively, with precise references to $\mathcal{F}_{\mathsf{STE}}$'s steps that are essentially handled by $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{F}_{\mathsf{elig}}$. Finally, in Subsection 3.4, we present the E-CCLESIA family as a protocol $\Pi^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}},\mathcal{G}_{\mathsf{clock}}}_{\text{E-CCLESIA}}$ that UC-realizes $\mathcal{F}_{\mathsf{STE}}$ in the $\{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model in a straightforward manner. We note that in the subsequent Sections 4 and 5 we construct two instantiations of $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{F}_{\mathsf{elig}}$, respectively, that specify the first version of E-CCLESIA as a provably secure STE scheme.

3.1 The STE functionality $\mathcal{F}_{\mathsf{STE}}$

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 $\mathcal{S}$. In addition, it forwards all (sid, ADVANCE_CLOCK) and (sid, READ_CLOCK) commands to $\mathcal{G}_{\mathsf{clock}}$ (cf. Fig. 1) on behalf of the (dummy) parties, while it also reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ whenever necessary. In the spirit of [8], we allow $\mathcal{S}$ to provide $\mathcal{F}_{\mathsf{STE}}$ with the election algorithms, noting that this is *only for consistency* of the output format that $\mathcal{F}_{\mathsf{STE}}$ sends to the parties. As it will be clear in the description, the main security properties of eligibility, fairness, voter privacy and one voter-one vote are preserved by $\mathcal{F}_{\mathsf{STE}}$ per se, independently of the security of the algorithms provided by $\mathcal{S}$. The functionality consists of the following phases:

Setup. The functionality initializes as empty the lists of: eligible voters' credentials L_{elig}, ready voters L_{ready}, generated ballots L_{gball}, cast ballots L_{cast}, algorithms L_{vm}, state St_{fin} as 0, and ballots included in the tally set L_{tally}. It also stores the set $\mathbf{V}_{\mathsf{corr}} \subseteq \mathbf{V}$ of corrupted voters provided by $\mathcal{S}$. Then, it operates as follows:

■ Upon receiving (sid, ELECTION_INFO, $\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}$) from SA, where (i) $\mathbf{V}_{\mathsf{elig}}$ is the list of eligible voters and (ii) $t_{\mathsf{cast}} < t_{\mathsf{open}}$ are moments that determine the beginning of the **Cast** and **Tally** phases (see below), it acts as follows: In some cases we need more distinct time points than t_{cast} and t_{open}. However, these points should be able to be derived from t_{cast} and t_{open}, because these are the only times that are provided by $\mathcal{Z}$. For example, a time point can be the t_{wait} where it defines the period in which a voter can cast their ballot (from time t_{cast} until t_{wait}) and the period that everyone must wait before the tally phase (from time t_{wait} until t_{open}). The time point t_{wait} can be essential in some protocols that they need a period of synchronization. For that reason we define the function `define_time` which it takes as input the times t_{cast} and t_{open} and outputs a vector $\boldsymbol{t}$. This vector can be just the pair $t_{\mathsf{cast}}, t_{\mathsf{open}}$ (in that case `define_time` is the identity function) or a vector with more distinct time points(e.g $\boldsymbol{t} = (t_{\mathsf{cast}}, t_{\mathsf{open}}, t_{\mathsf{wait}})$). It depends on the specification of the protocol we want to realize how the function `define_time` is defined. Formally, $\mathcal{F}_{\mathsf{STE}}$ computes $\boldsymbol{t} \leftarrow \mathtt{define\_time}(t_{\mathsf{cast}}, t_{\mathsf{open}})$ and if $\boldsymbol{t} \neq \perp$, it sends ($\mathsf{sid}$, SETUP_OK, $\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}$) to SA after the permission of $\mathcal{S}$ via a public delayed output.

■ Upon receiving (sid, ELIGIBLE) from SA, it informs $\mathcal{S}$, which replies with the *eligibility algorithms* `GenCred`, `AuthBallot`, `VrfyBallot`, `UpState` and an *initial credential state* St_{gen}. Then it provides $\langle V_i \rangle_{i \in [n]}$ and $\mathcal{S}$ with the *election parameters* $\mathsf{elec.par} := (\{\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}\}, St_{\mathsf{gen}})$.

■ Upon receiving (sid, START) from a voter $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$, if $V \notin L_{\mathsf{ready}}$:

1. If $L_{\mathsf{vm}} \neq \emptyset$, it (a) adds V to L_{ready} and (b) returns (sid, START_OK) to V.
2. Else, it sends (sid, START) to $\mathcal{S}$. Upon receiving the *vote management algorithms* `GenBallot`, `OpenBallot` from $\mathcal{S}$, it (a) updates L_{vm} with the algorithms, (b) adds V to L_{ready}, and (c) returns (sid, START_OK) to V.

Credential generation. This phase is active if $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cred}) = \top$. The predicate `Status` shows us in which phase of the protocol we are based on the

current time Cl. It takes as an input the current time Cl, the time vector $\boldsymbol{t}$ and the phase we want to check if it is active at the moment (all the phases are Credential,Cast and Tally and can be checked with the acronym $\mathsf{Cred}, \mathsf{Cast}$ and Tally respectively). If this phase is active $\texttt{Status}$ outputs the special symbol $\perp$, else it outputs $\perp$. The predicate is instantiated according to the protocol we want to realize. An example of what this predicate might be is the check if $t_{\mathsf{cast}} < \mathsf{Cl} < t_{\mathsf{open}}$ in the case we wan to check if we are in the Cast phase.

■ Upon receiving a message $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$ for the first time and after permission from $\mathcal{S}$ via public delayed output, it runs $(\mathsf{cr}, \mathsf{rc}) \leftarrow \texttt{GenCred}(1^\lambda, \mathsf{elec.par})$ and adds $(V, \mathsf{cr}, \mathsf{rc}, 1)$ to L_{elig}. Here, cr, rc play the role of the *private and public part* of the credential, respectively. If there are already tuples $(\cdot, \mathsf{cr}, \cdot, \cdot)$ or $(\cdot, \cdot, \mathsf{rc}, \cdot)$ in L_{elig} or $(\mathsf{cr}, \mathsf{rc}) = \perp$, it sends $(\mathsf{sid}, \textsc{Gen\_Cred}, \perp)$ to V and halts. Else, it adds $(V, \mathsf{cr}, \mathsf{rc}, 1)$ to L_{elig}.

If $\mathcal{F}_{\mathsf{STE}}$ receives some credential pair $(\mathsf{sid}, \textsc{Gen\_Cred}, \mathsf{cr}, \mathsf{rc}, V)$ from $\mathcal{S}$ on behalf of a corrupted yet eligible voter V for the first time, if there are no tuples $(\cdot, \mathsf{cr}, \cdot, 1)$ or $(\cdot, \cdot, \mathsf{rc}, 1)$ in L_{elig}, then $\mathcal{F}_{\mathsf{STE}}$ adds $(V, \mathsf{cr}, \mathsf{rc}, 0)$ to L_{elig}.

In any case of recording $(V, \mathsf{cr}, \mathsf{rc}, \cdot)$, it sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathsf{rc})$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$ after permission of $\mathcal{S}$ via public delayed output.

Cast. This phase is active if $\texttt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \top$ and manages ballot generation, authentication and broadcasting. For ballot authentication, $\mathcal{F}_{\mathsf{STE}}$ once computes the *final credential state* St_{fin} by running $\texttt{UpState}$ on input St_{gen} and the set of public credentials rc included in L_{elig}.

■ Upon receiving a message $\big(\mathsf{sid}, \textsc{Cast}, o\big)$ from an honest and eligible voter V for the first time such that $(V, \mathsf{cr}, \mathsf{rc}, 1) \in L_{\mathsf{elig}}$, it executes the following steps:

1. It generates a *'dummy' ballot* as an encryption of 0s by running $v \leftarrow \texttt{GenBallot}(\mathsf{vote.par}, 0^{|o|}, \lambda)$ and adds the tuple $(V, v, o, 1)$ to the list of the generated ballots L_{gball}. This dummy encryption step captures the semantic security of the $\texttt{GenBallot}$ algorithm.
2. It generates an *authentication receipt* as $\sigma \leftarrow \texttt{AuthBallot}(v, \mathsf{cr}, \mathsf{rc}, St_{\mathsf{fin}})$ for ballot v. If $\texttt{VrfyBallot}(v, \sigma, St_{\mathsf{fin}}, \mathsf{reg.par}) = 0$, it sends $(\mathsf{sid}, \textsc{Auth\_Ballot}, \perp)$ to V and halts.
3. It sends $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Cast\_Allowed})$ from $\mathcal{S}$, it adds $(V, o, v, \mathsf{cr}, \sigma, 1)$ to L_{cast} as the authenticated ballot tuple for V.
4. It sends $\big(\mathsf{sid}, \textsc{Cast}, v, \sigma\big)$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$. Note that $\mathcal{F}_{\mathsf{STE}}$ broadcasts the ballot "anonymously", i.e. without revealing V's identity.

In addition, $\mathcal{F}_{\mathsf{STE}}$ allows $\mathcal{S}$ to cast ballots arbitrarily on behalf of a corrupted voter V: when it receives $\big(\mathsf{sid}, \textsc{Cast}, o, v, \sigma, V\big)$ from $\mathcal{S}$, if there is a tuple $(V, \mathsf{cr}, \mathsf{rc}, 0)$ in L_{elig}, it adds $(V, o, v, \mathsf{cr}, \sigma, 0)$ to L_{cast}, else it adds $(V, o, v, \perp, \sigma, 0)$ to L_{cast}. In any case, it sends $\big(\mathsf{sid}, \textsc{Cast}, v, \sigma\big)$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$.

Tally. This phase is active if $\texttt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Tally}) = \top$ and manages the opening of all the valid ballots that should be included in the tally (thus, capturing fairness).

■ Upon receiving $\big(\mathsf{sid}, \textsc{Tally}\big)$ from a voter V or $\mathcal{S}$, $\mathcal{F}_{\mathsf{STE}}$ executes the following steps:

1. **For** every tuple $(V, o, v, \mathsf{cr}, \sigma, b)$ in L_{cast}, it runs the ballot verification algorithm $x \leftarrow \mathtt{VrfyBallot}(v, \sigma, St_{\mathsf{fin}})$. If $x = 1$, then $\mathcal{F}_{\mathsf{STE}}$ performs the following security checks:
   (i). If there is no tuple $(V, \mathsf{cr}, \mathsf{rc}, \cdot)$ in L_{elig}, then eligibility has been breached and it returns $\big(\mathsf{sid}, \textsc{Tally}, \perp\big)$ to V or $\mathcal{S}$.
   (ii). If there is a tuple $(V, \mathsf{cr}, \mathsf{rc}, 1)$ in L_{elig} and there is a tuple $(\cdot, o', v', \mathsf{cr}', \sigma', 1)$ in L_{cast} such that $(\mathsf{cr}' = \mathsf{cr}) \wedge (v' \neq v)$, then forgery of some honest ballot has happened, so it returns $(\mathsf{sid}, \textsc{Tally}, \perp)$ to V or $\mathcal{S}$. Note that this check implies that the `AuthBallot` algorithm should satisfy unforgeability.
   (iii). Otherwise, it adds (V, o, v, b) to L_{tally}.
   As a result, at the end of the **For** loop in this step, L_{tally} contains successfully verified ballots cast only by the intended eligible voters. Note that multiple valid ballots coming from corrupted eligible voters may still exist at this point, something which is handled at the step below.
2. **For** every tuple $(V, \mathsf{cr}, \mathsf{rc}, \cdot)$ in L_{elig} such that there are multiple tuples $(V, o^1, v^1, \cdot), \ldots, (V, o^n, v^n, \cdot)$ in L_{tally}, it removes all multiple tuples from L_{tally} except the first one it recorded. This step is important to ensure the one voter-one vote property. Thus, at this point L_{tally} contains valid tuples in one-to-one correspondence with the eligible voters that participated in the election.
3. **For** every $(V, o, v, \cdot)$ in L_{tally}, it first allows $\mathcal{S}$ to open the ballot of any corrupted voter V_j to an alternative opening $\tilde{o}_j$ by updating the corresponding tuple in L_{tally} as $(V_j, \tilde{o}_j, v_j, 0)$. Then, it checks the correctness of the honest voters' ballot generation (which implies the correctness of the ballot opening algorithm `OpenBallot`) as follows: if there are two tuples $(V, o, v, 1)$, $(V', o', v', 1)$ in L_{tally} such that $(v' = v) \wedge (o \neq o')$, then it returns $\big(\mathsf{sid}, \textsc{Tally}, \perp\big)$ to V. Otherwise, if no such conflict exists, it returns $\big(\mathsf{sid}, \textsc{Tally}, \{(o, v) | (V, o, v, \cdot) \in L_{\mathsf{tally}}\}\big)$ to V or $\mathcal{S}$, where $\{(o, v) | (V, o, v, \cdot) \in L_{\mathsf{tally}}\}$ is the multiset of eligible voters' tallied options.

■ Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{\mathsf{Clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cred}) = \mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \perp$, then it returns to $\mathcal{S}$ all the triples $(v, o, 1)$ such that $(V, v, o, 1) \in L_{\mathsf{gball}} \wedge (V, v, \sigma, 1) \in L_{\mathsf{cast}}$. This leakage illustrates the fact that in "wait" time period, the period where the protocol is between the **Cast** and **Tally** phase, the adversary might be able to open the ballots. On the other hand, this information at that point is not very useful (the adversary can not break fairness) because the adversary can not change its ballot, as the **Cast** phase is over. In protocols where no such periods exist the condition $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cred}) = \mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \perp$ is never satisfied.

3.2 Vote management funtionality $\mathcal{F}_{\mathsf{vm}}$

The vote management functionality $\mathcal{F}_{\mathsf{vm}}$ takes over the following parts of $\mathcal{F}_{\mathsf{STE}}$ execution: (i) in **Setup** the configuration of the vote management algorithms,

(ii) in **Cast** the secure ballot generation and casting and (iv) in **Tally** the ballot opening step, so that fairness and ballot correctness is preserved (cf. Step 3. in $\mathcal{F}_{\mathsf{STE}}$'s description for $(\mathsf{sid}, \textsc{Tally})$ messages). The functionality is presented in Fig. 6.

The vote management functionality $\mathcal{F}_{\mathsf{vm}}(\mathbf{V}, \mathtt{define\_time}, \mathtt{Status})$.

The functionality initializes the lists of generated ballots L_{gball}, cast ballots L_{cast}, opened ballots L_{open}, ballot algorithms L_{vm}, and ready voters L_{ready} as empty. Then it sets its status to 'exec'.

■ Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{S}$, if $\mathbf{V}_{\mathsf{corr}} \subseteq \mathbf{V}$ and status=exec, it fixes $\mathbf{V}_{\mathsf{corr}}$ as the set of corrupted voters.

■ Upon receiving $(\mathsf{sid}, \textsc{Setup\_Info}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from SA it computes $\boldsymbol{t} \leftarrow \mathtt{define\_time}(t_{\mathsf{cast}}, t_{\mathsf{open}})$. If status=$\mathsf{exec}$ and $\boldsymbol{t} \neq \perp$, it sets $\mathsf{vote.par} := (\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t})$ as voting parameters and sends $(\mathsf{sid}, \textsc{Setup\_OK}, \mathsf{vote.par})$ to SA, after permission from $\mathcal{S}$ via delayed output.

■ Upon receiving $(\mathsf{sid}, \textsc{Start})$ from voter $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$, if $V \notin L_{\mathsf{ready}}$:

1. If $L_{\mathsf{vm}} \neq \emptyset$, it (a) adds V to L_{ready} and (b) returns $(\mathsf{sid}, \textsc{Start\_Ok})$ to V.
2. Else, it sends $(\mathsf{sid}, \textsc{Start})$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Setup\_Info}, \mathtt{GenBallot}, \mathtt{OpenBallot})$ from $\mathcal{S}$, it (a) updates L_{vm} with the algorithms, (b) adds V to L_{ready}, and (c) returns $(\mathsf{sid}, \textsc{Start\_Ok})$ to V.

■ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, o)$ from $V \in L_{\mathsf{ready}}$:

1. If there is no $(V, v', o', 1) \notin L_{\mathsf{gball}}$, it (a) runs $v \leftarrow \mathtt{GenBallot}(\mathsf{vote.par}, 0^{|o|}, \lambda)$, (b) adds $(V, v, o, 1)$ to L_{gball}. Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, v)$ from $\mathcal{S}$, it returns $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, v)$ to V.
2. Else it returns $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, \perp)$ 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 $\mathcal{S}$, if $V \in \mathbf{V}_{\mathsf{corr}}$ it sends $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, v)$ to V.

■ Upon receiving $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ from V, if $V \notin \mathbf{V}_{\mathsf{corr}}$ and status=exec, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \top$, it sets the status to 'open', otherwise:

1. If $(V, v, o', 1) \notin L_{\mathsf{gball}}$, it returns $(\mathsf{sid}, \textsc{Cast}, v, \sigma, \perp)$ to V.
2. If there is no $(V, v', \sigma', 1) \notin L_{\mathsf{cast}}$, it sends $(\mathsf{sid}, \textsc{Allow\_Cast}, v, \sigma)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Cast\_Allowed})$ from $\mathcal{S}$, it adds $(V, v, \sigma, 1)$ to L_{cast} and sends $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$.
3. If there is a tuple $(V, v', \sigma', 1)$ in L_{cast}, it returns $(\mathsf{sid}, \textsc{Cast}, v, \sigma, \perp)$ to V.

■ Upon receiving $(\mathsf{sid}, \textsc{Cast})$ from $V \in \mathbf{V}_{\mathsf{corr}}$, it sends $(\mathsf{sid}, \textsc{Cast}, V)$ to $\mathcal{S}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Cast}, v, \sigma, V)$ from $\mathcal{S}$, if $V \in \mathbf{V}_{\mathsf{corr}}$ and status=exec, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \top$ it sets status to 'open', otherwise it adds $(V, v, \sigma, 0)$ to L_{cast} and sends $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$.

■ 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 $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \top$, it sets the status to 'open' and executes the following steps:

1. If there is a tuple $(V, v, \sigma, \cdot) \in L_{\mathsf{cast}}$, and a *unique* $(V, v, o, 1) \in L_{\mathsf{gball}}$, it sends $(\mathsf{sid}, \textsc{Open}, v, o)$ to P.
2. Else, if there is a tuple $(V, v, \sigma, \cdot) \in L_{\mathsf{cast}}$ and at least two tuples $(V, v, o, 1), (V', v', o', 1) \in L_{\mathsf{gball}}$ such that $(v = v') \wedge (o \neq o')$, it sends $(\mathsf{sid}, \textsc{Open}, v, \perp)$ to P.
3. Else, if there is a tuple $(V, v, \sigma, \cdot) \in L_{\mathsf{cast}}$ but there is no tuple $(V, v, o, 1) \in L_{\mathsf{gball}}$, 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 $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cred}) = \mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \perp$ then it returns to $\mathcal{S}$ all the triples $(v, o, 1)$ such that $(V, v, o, 1) \in L_{\mathsf{gball}} \wedge (V, v, \sigma, 1) \in L_{\mathsf{cast}}$.

Fig. 6. The vote management functionality $\mathcal{F}_{\mathsf{vm}}(\mathbf{V})$ interacting with voters $\mathbf{V}$, SA and the simulator $\mathcal{S}$.

The eligibility functionality $\mathcal{F}_{\mathsf{elig}}(\mathbf{V}, \mathtt{define\_time}, \mathtt{Status})$.

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 $St_{\mathsf{fin}} = 0$, and its status to 'init'.

■ Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ from $\mathcal{S}$, if $\mathbf{V}_{\mathsf{corr}} \subseteq \mathbf{V}$ and status=init, it fixes $\mathbf{V}_{\mathsf{corr}}$ as the set of corrupted voters.

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

1. It computes $\boldsymbol{t} \leftarrow \mathtt{define\_time}(t_{\mathsf{cast}}, t_{\mathsf{open}})$. If $\boldsymbol{t} \neq \perp$, it sets $\mathsf{reg.par} := (\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, St_{\mathsf{gen}})$ as registration parameters.
2. It sets its status to '$\mathsf{credential}$' and sends $(\mathsf{sid}, \textsc{Elig\_Par}, \mathsf{reg.par})$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$, if status=$\mathsf{credential}$, then it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \top$, it sets its status to 'cast'. Otherwise, if $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cred}) = \top$, it executes the following steps:

1. If there is no tuple $(V, \mathsf{cr}', \mathsf{rc}', 1)$ in L_{elig}, it runs $(\mathsf{cr}, \mathsf{rc}) \leftarrow \mathtt{GenCred}(1^\lambda, \mathsf{reg.par})$. If there are tuples $(\cdot, \mathsf{cr}, \cdot, \cdot)$ or $(\cdot, \cdot, \mathsf{rc}, \cdot)$ in L_{elig} or $(\mathsf{cr}, \mathsf{rc}) = \perp$, it sends $(\mathsf{sid}, \textsc{Gen\_Cred}, \perp)$ to V and halts. Else, it adds $(V, \mathsf{cr}, \mathsf{rc}, 1)$ to L_{elig} after permission of $\mathcal{S}$ via public delayed output.
2. It sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathsf{rc})$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$ after permission of $\mathcal{S}$ via public delayed output.

■ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred})$ from $V \in \mathbf{V}_{\mathsf{corr}}$, it forwards the message $(\mathsf{sid}, \textsc{Gen\_Cred}, V)$ to $\mathcal{S}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathsf{cr}, \mathsf{rc})$ from $\mathcal{S}$, if $V \in \mathbf{V}_{\mathsf{corr}}$, then:

1. If $\mathbf{V}_{\mathsf{elig}}$ and there are no tuples $(V, \mathsf{cr}', \mathsf{rc}', 0)$, $(\cdot, \mathsf{cr}, \cdot, 1)$ or $(\cdot, \cdot, \mathsf{rc}, 1)$ in L_{elig}, then it adds $(V, \mathsf{cr}, \mathsf{rc}, 0)$ to L_{elig}.
2. It sends $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathsf{rc})$ to $\langle V_j \rangle_{j \in [n]}$ and $\mathcal{S}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Auth\_Ballot}, v)$ from $V \in \mathbf{V}_{\mathsf{elig}} \setminus \mathbf{V}_{\mathsf{corr}}$, if status=$\mathsf{cast}$, then it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \top$, it sets its status to 'open'. Otherwise, if $\mathtt{Status}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \top$, it executes the following steps:

1. If $St_{\mathsf{fin}} = 0$, then it runs $St_{\mathsf{fin}} \leftarrow \mathtt{UpState}(St_{\mathsf{gen}}, \{\mathsf{rc} | (\cdot, \cdot, \mathsf{rc}, \cdot) \in L_{\mathsf{elig}}\})$.
2. If there is some $(V, \mathsf{cr}, \mathsf{rc}, 1) \in L_{\mathsf{elig}}$ but no $(V, v', \mathsf{cr}, \sigma', 1) \in L_{\mathsf{auth}}$, then it runs $\sigma \leftarrow \mathtt{AuthBallot}(v, \mathsf{cr}, \mathsf{rc}, St_{\mathsf{fin}}, \mathsf{reg.par})$. 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 (a) adds $(V, v, \mathsf{cr}, \sigma, 1)$ to L_{auth}, and (b) returns $(\mathsf{sid}, \textsc{Auth\_Ballot}, v, \sigma)$ to V.

■ Upon receiving $(\mathsf{sid}, \textsc{Auth\_Ballot})$ from $V \in \mathbf{V}_{\mathsf{corr}}$, it forwards the message $(\mathsf{sid}, \textsc{Auth\_Ballot}, V)$ to $\mathcal{S}$.

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

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

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

■ Upon receiving $\big(\mathsf{sid}, \textsc{Link\_Ballots}, (v, \sigma), (v', \sigma')\big)$ from $V \in \mathbf{V}$, if there are tuples $(\cdot, v, \mathsf{cr}, \sigma, \cdot)$, $(\cdot, v', \mathsf{cr}', \sigma', \cdot) \in L_{\mathsf{auth}}$ such that $\mathsf{cr} = \mathsf{cr}'$, then it sets $x = 1$, else $x = 0$. Then, it sends $\big(\mathsf{sid}, \textsc{Link\_Ballots}, (v, \sigma), (v', \sigma'), x\big)$ to V.

Fig. 7. The eligibility functionality $\mathcal{F}_{\mathsf{elig}}(\mathbf{V})$ interacting with voters $\mathbf{V}$, SA, and the simulator $\mathcal{S}$.

3.3 Eligibility functionality $\mathcal{F}_{\mathsf{elig}}$

The eligibility functionality $\mathcal{F}_{\mathsf{elig}}$ takes over the following parts of $\mathcal{F}_{\mathsf{STE}}$ execution: (i) in **Setup** the eligibility algorithms and the initial credential state, (ii) the entire (private and public) **Credential generation**, (iii) in **Cast** the creation of ballot authentication receipts, (iv) in **Tally** the ballot verification step, so that unforgeability is preserved (cf. Step 1. in $\mathcal{F}_{\mathsf{STE}}$'s description for $(\mathsf{sid}, \textsc{Tally})$ messages), and (v) the linkability of two ballots to the same voter, so that one voter - one vote is preserved. The functionality is presented in Fig. 7.

3.4 Description of the E-cclesia family

We provide a description of the E-CCLESIA family $\Pi_{\text{E-CCLESIA}}^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}},\mathcal{G}_{\mathsf{clock}}}$ of STE schemes as a hybrid protocol that makes use of the main modules $\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}$. Any pair of real-world protocols $\Pi_{\mathsf{elig}}, \Pi_{\mathsf{vm}}$ that UC-realize $\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}$ specifies a member of the family. The description of $\Pi_{\text{E-CCLESIA}}^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}},\mathcal{G}_{\mathsf{clock}}}$ follows the phases and command interface of $\mathcal{F}_{\mathsf{STE}}$ in Subsection 3.1 as described below:

Setup.

■ Upon receiving $(\mathsf{sid}, \textsc{Election\_Info}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from $\mathcal{Z}$, if $\mathbf{V}_{\mathsf{elig}} \subseteq \mathbf{V}$ and $t_{\mathsf{cast}} < t_{\mathsf{open}}$, SA sends $(\mathsf{sid}, \textsc{Setup\_Info}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{F}_{\mathsf{vm}}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Eligible})$ from the environment $\mathcal{Z}$, if SA has received $(\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$, it sends $(\mathsf{sid}, \textsc{Eligible}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ to $\mathcal{F}_{\mathsf{elig}}$ which sends $\mathsf{reg.par} := (\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, St_{\mathsf{gen}})$ to $\langle V_i \rangle_{i\in[n]}$. Upon receiving $\mathsf{reg.par}$ from $\mathcal{F}_{\mathsf{elig}}$, each voter $V \in \mathbf{V}$ stores it as the election parameters $\mathsf{elec.par}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Start})$ from $\mathcal{Z}$ for the first time, if voter V is in $\mathbf{V}_{\mathsf{elig}}$, then she sends $(\mathsf{sid}, \textsc{Start})$ to $\mathcal{F}_{\mathsf{vm}}$ which replies with $(\mathsf{sid}, \textsc{Start\_Ok})$.

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 $(\mathsf{sid}, \textsc{Gen\_Cred}, V, \mathsf{rc})$ to $\langle V_j \rangle_{j\in[n]}$ (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 broadcasting 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 with the generated ballot as $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, v)$ (or sends $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, \perp)$ to V and halts).
2. Then, she sends $(\mathsf{sid}, \textsc{Auth\_Ballot}, v)$ to $\mathcal{F}_{\mathsf{elig}}$ which replies with the authentication receipt for v as $(\mathsf{sid}, \textsc{Auth\_Ballot}, v, \sigma)$ (or sends $(\mathsf{sid}, \textsc{Auth\_Ballot}, \perp)$ to V and halts).
3. Finally, she sends $\big(\mathsf{sid}, \textsc{Cast}, v, \sigma\big)$ to $\mathcal{F}_{\mathsf{vm}}$ which broadcasts the message to $\langle V_j \rangle_{j\in[n]}$. In turn, the voters store the received pair (v, σ).

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 $\mathcal{Z}$, V executes the following steps:

1. **For** every tuple $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ she has obtained from $\mathcal{F}_{\mathsf{vm}}$, V sends (sid, VER_BALLOT, v, σ) to $\mathcal{F}_{\mathsf{elig}}$ which replies with (sid, VER_BALLOT, v, σ, 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 sets tally to $\perp$. Otherwise, she includes in her tally set all pairs (v, σ) such that $\mathcal{F}_{\mathsf{elig}}$ replied with $x = 1$.
2. V discards multiple ballots as follows: for every pair (v, σ), (v', σ') in her tally set, she sends $(\mathsf{sid}, \textsc{Link\_Ballots}, (v, \sigma), (v', \sigma'))$ to $\mathcal{F}_{\mathsf{elig}}$. If she gets $(\mathsf{sid}, \textsc{Link\_Ballots}, (v, \sigma), (v', \sigma'), 1)$ 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, σ) in the tally set, V sends (sid, OPEN, v) to $\mathcal{F}_{\mathsf{vm}}$, which replies with the opening (sid, OPEN, v, o). Then, V adds o to the multiset of all opened options (initialized as empty). If at any time $\mathcal{F}_{\mathsf{vm}}$ replies with (sid, OPEN, v, $\perp$), then V sets tally to $\perp$.
4. Finally, she sets the tally result as the multiset of all opened options.

Security. As already mentioned in the introduction of Section 3, proving that $\Pi^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}},\mathcal{G}_{\mathsf{clock}}}_{\text{E-CCLESIA}}$ UC-realizes $\mathcal{F}_{\mathsf{STE}}$ in the $\{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model is straightforward given the description of $\mathcal{F}_{\mathsf{STE}}$, $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$. Below, we provide the theorem statement and proof.

Theorem 1. *The protocol $\Pi^{\mathcal{F}_{\mathsf{elig}},\mathcal{F}_{\mathsf{vm}},\mathcal{G}_{\mathsf{clock}}}_{\text{E-CCLESIA}}$ described in Subsection 3.4 UC-realizes $\mathcal{F}_{\mathsf{STE}}$ in the $\{\mathcal{F}_{\mathsf{elig}}, \mathcal{F}_{\mathsf{vm}}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model.*

Proof. We define a simulator $\mathcal{S}$ as follows: In the cases of public delayed outputs, we assume that $\mathcal{S}$ forwards the message to the adversary $\mathcal{A}$ as if it was from either $\mathcal{F}_{\mathsf{elig}}$ or $\mathcal{F}_{\mathsf{vm}}$ and responds to $\mathcal{F}_{\mathsf{STE}}$ in the same way as $\mathcal{A}$. Moreover, whatever command $\mathcal{F}_{\mathsf{STE}}$ receives on behalf of a corrupted party, we assume that $\mathcal{F}_{\mathsf{STE}}$ forwards that message to $\mathcal{S}$. Then $\mathcal{S}$ forwards that message to $\mathcal{A}$ as if it was from either $\mathcal{F}_{\mathsf{elig}}$ or $\mathcal{F}_{\mathsf{vm}}$ and it returns to $\mathcal{F}_{\mathsf{STE}}$ whatever it receives from $\mathcal{A}$. We describe the details below.

During the **Setup** phase, when $\mathcal{S}$ receives the corruption set $\mathbf{V}_{\mathsf{corr}}$ from $\mathcal{Z}$, it forwards it to $\mathcal{A}$ as if it was from $\mathcal{Z}$. Then $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{elig}}$ and $\mathcal{F}_{\mathsf{vm}}$ and receives back the corruption set from $\mathcal{A}$. Next, $\mathcal{S}$ forwards the corruption set to $\mathcal{F}_{\mathsf{STE}}$. Upon receiving ($\mathsf{sid}$, ELECTION_INFO, $\mathbf{V}_{\mathsf{elig}}$, t_{cast}, t_{open}) from $\mathcal{F}_{\mathsf{STE}}$, $\mathcal{S}$ stores the parameters and it forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{vm}}$. Then, $\mathcal{S}$ returns to $\mathcal{F}_{\mathsf{STE}}$ whatever it receives from $\mathcal{A}$. After $\mathcal{S}$ receives (sid, ELIGIBLE) from $\mathcal{F}_{\mathsf{STE}}$, it plays the role of $\mathcal{F}_{\mathsf{elig}}$ and it sends (sid, SETUP_ELIG) to $\mathcal{A}$. Then, upon receiving the eligibility algorithms (sid, SETUP_ELIG, `GenCred`, `AuthBallot`, `VrfyBallot`, `UpState`, St_{gen}) from $\mathcal{A}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{\mathsf{STE}}$. When $\mathcal{S}$ receives (sid, START) from $\mathcal{F}_{\mathsf{STE}}$, it forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{vm}}$. Upon receiving the vote management algorithms (sid, SETUP_INFO, `GenBallot`, `OpenBallot`) from $\mathcal{A}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{\mathsf{STE}}$.

During the **Credential generation** phase, when $\mathcal{S}$ receives a credential generation request from $\mathcal{F}_{\mathsf{STE}}$ on behalf of a corrupted party V, $\mathcal{S}$ forwards the request to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{elig}}$. Then, upon receiving ($\mathsf{sid}$, GEN_CRED, V, cr, rc) from $\mathcal{A}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{\mathsf{STE}}$.

During the **Cast** phase, when $\mathcal{S}$ receives $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ from $\mathcal{F}_{\mathsf{STE}}$, it forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{vm}}$. Then $\mathcal{S}$ returns whatever it receives from $\mathcal{A}$. Upon receiving a cast ballot request from $\mathcal{F}_{\mathsf{STE}}$ on behalf of a corrupted party V, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{vm}}$. After $\mathcal{S}$ receives $\big(\mathsf{sid}, \textsc{Cast}, \tilde{v}, \tilde{\sigma}, V\big)$ from $\mathcal{A}$, it returns the message $\big(\mathsf{sid}, \textsc{Cast}, o, \tilde{v}, \tilde{\sigma}, V\big)$ to $\mathcal{F}_{\mathsf{STE}}$ for some o. The choice of o is irrelevant because during the **Tally** phase $\mathcal{S}$ will be asked by $\mathcal{F}_{\mathsf{STE}}$ for a new opening of $\tilde{v}$. This happens because every time $\mathcal{F}_{\mathsf{STE}}$ ($\mathcal{F}_{\mathsf{vm}}$ as well) receives a request for opening a malicious ballot, it gives the token to the simulator and it answers according to what $\mathcal{S}$ provides.

During the **Tally** phase, when $\mathcal{S}$ is asked by $\mathcal{F}_{\mathsf{STE}}$ for an alternative ballot opening of the tuple $(V, o, v, \cdot) \in L_{\mathsf{tally}}$ where $V \in \mathbf{V}_{\mathsf{corr}}$, $\mathcal{S}$ sends the message (sid, OPEN, v) to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{vm}}$ and it returns to $\mathcal{F}_{\mathsf{STE}}$ whatever it receives from $\mathcal{A}$. Upon receiving (sid, OPEN, v) from $\mathcal{Z}$ for the first time, $\mathcal{S}$ sends (sid, TALLY) to $\mathcal{F}_{\mathsf{STE}}$. Upon receiving $\big(\mathsf{sid}, \textsc{Tally}, \{(o, v) | (V, o, v, \cdot) \in L_{\mathsf{tally}}\}\big)$ from $\mathcal{F}_{\mathsf{STE}}$, $\mathcal{S}$ records the tally. Next, $\mathcal{S}$ forwards the message $(\mathsf{sid}, \textsc{Open}, v)$ to $\mathcal{A}$ as if it was from $\mathcal{Z}$. Upon receiving $(\mathsf{sid}, \textsc{Open}, v)$, $\mathcal{S}$ playing the role of $\mathcal{F}_{\mathsf{vm}}$ returns the plaintext for that ballot to $\mathcal{A}$. If the message doesn't exist in the list previously provided by $\mathcal{F}_{\mathsf{STE}}$, then $\mathcal{S}$ asks $\mathcal{A}$ for the opening of that message as before.

The distribution of messages is exactly the same in both settings, since the algorithms that are used are the same. As a result the simulation is perfect. □

4 Realizing $\mathcal{F}_{\mathsf{vm}}$ via TLE

We define one of our main building blocks, the $\mathcal{F}_{\mathsf{TLE}}$ functionality that captures the security properties of a time-lock encryption (TLE) scheme. Intuitively speaking, a TLE scheme is a pair of algorithms (e, d) that a party P_i can use in order to encrypt a message m with time label τ_{dec} such that everyone can decrypt that message after the current time exceeds τ_{dec}. The decryption is possible because after the time τ_{dec}, a witness $w_{\tau_{\mathsf{dec}}}$ is available.

In Subsection 4.1, we provide a definition of the *ideal TLE functionality* $\mathcal{F}_{\mathsf{TLE}}$. In Subsection 4.2, we demonstrate a realization of $\mathcal{F}_{\mathsf{TLE}}$ via time-lock puzzles, as the one in [30]. In Subsection 4.3, we construct a real-world protocol that UC-realizes the vote management functionality $\mathcal{F}_{\mathsf{vm}}$ (cf. Subsection 3.2) via $\mathcal{F}_{\mathsf{TLE}}$.

4.1 Definition of $\mathcal{F}_{\mathsf{TLE}}$

We provide the first UC treatment of TLE by defining the functionality $\mathcal{F}_{\mathsf{TLE}}$, following the approach of [6,8]. The functionality is described in Fig. 8, and at a high level operates as follows.

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

The functionality initializes the lists of algorithms L_{algo} and ready parties L_{ready} as empty.

- Upon receiving $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{P}_{\mathsf{corr}})$ from $\mathcal{S}$, it records the corrupted set $\mathbf{P}_{\mathsf{corr}}$.
- Upon receiving $(\mathsf{sid}, \textsc{Start})$ from a party $P \notin L_{\mathsf{ready}}$, it adds $P \rightarrow L_{\mathsf{ready}}$ and:
  1. If $L_{\mathsf{algo}} \equiv \emptyset$, it forwards the message to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, \textsc{Enc}, \textsc{Dec}, e_{\mathcal{S}}, d_{\mathcal{S}})$ from $\mathcal{S}$, it registers the tuple $(e_{\mathcal{S}}, d_{\mathcal{S}}) \rightarrow L_{\mathsf{algo}}$. It returns $(\mathsf{sid}, \textsc{Start\_Ok})$ to P.
  2. Else it returns $(\mathsf{sid}, \textsc{Start\_Ok})$ to P.
- Upon receiving $(\mathsf{sid}, \textsc{Enc}, m, \tau_{\mathsf{dec}})$ from $P \notin \mathbf{P}_{\mathsf{corr}}$:
  1. If $\tau_{\mathsf{dec}} < 0$ or $P \notin L_{\mathsf{ready}}$, it returns $(\mathsf{sid}, \textsc{Enc}, m, \tau_{\mathsf{dec}}, \bot)$ to P.
  2. Else, it adds $(m, c \leftarrow e_{\mathcal{S}}(0^{|m|}, \tau_{\mathsf{dec}}; r), \tau_{\mathsf{dec}}, r) \rightarrow L_{\mathsf{rec}}$, where $r \xleftarrow{\$} \{0,1\}^{|poly(\lambda)|}$ is the randomness of encryption, and returns $(\mathsf{sid}, \textsc{Enc}, m, \tau_{\mathsf{dec}}, c)$ to P after informing the $\mathcal{S}$ for c via public delayed output.
- Upon receiving $(\mathsf{sid}, \textsc{Dec}, c, \tau)$ from $P \notin \mathbf{P}_{\mathsf{corr}}$:
  1. If $\tau < 0$ or $P_i \notin L_{\mathsf{ready}}$, it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau, \bot)$ 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 $(m_1, c, \tau_1, r_1), (m_2, c, \tau_2, r_2)$ in L_{rec} such that $m_1 \neq m_2$ where $\tau \geq \mathsf{max}\{\tau_1, \tau_2\}$, it returns to P $(\mathsf{sid}, \textsc{Dec}, c, \tau, \bot)$.
     – If no tuple $(\cdot, c, \cdot, \cdot)$ 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 $(m, c, \tau_{\mathsf{dec}}, r)$ 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.
- Whatever message it receives from $P \in \mathbf{P}_{\mathsf{corr}}$, it forwards it to $\mathcal{S}$ and vice versa.
- Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and returns $(\mathsf{sid}, \textsc{Leakage}, \{(m, c, \tau_{\mathsf{dec}})\}_{\tau_{\mathsf{dec}}:\tau_{\mathsf{dec}} \leq \mathsf{leak}(\mathsf{Cl})})$ to $\mathcal{S}$.

Fig. 8. Functionality $\mathcal{F}_{\mathsf{TLE}}$ parameterized by λ, a leakage function leak, interacting with simulator $\mathcal{S}$, parties in $\mathbf{P}$, and global clock $\mathcal{G}_{\mathsf{clock}}$.

Initially, the simulator $\mathcal{S}$ provides $\mathcal{F}_{\mathsf{TLE}}$ with the corrupted parties set and the encryption/decryption algorithms. Each time an encryption query issued by an uncorrupted party is given to $\mathcal{F}_{\mathsf{TLE}}$, the functionality records the message and the encryption *not of the message*, but of the all zero string and returns the ciphertext. This illustrates the fact that the ciphertext itself does not contain any information about the message. Next, it handles the decryption queries appropriately, unless it finds two messages recorded with the same ciphertext, in which case it outputs $\bot$. This implies that the encryption/decryption algorithms should satisfy correctness. In the cases of encryption/decryption queries issued

by corrupted parties, $\mathcal{F}_{\mathsf{TLE}}$ responds according to the instructions of $\mathcal{S}$. Moreover, on demand, $\mathcal{F}_{\mathsf{TLE}}$ gives the record of all messages with encryption up to $\mathsf{leak}(\mathsf{Cl})$ to $\mathcal{S}$, where Cl is the current time and leak a leakage function that takes as an input a time and gives as an output a time of at least as high value. This function leak captures the fact that in some cases the adversary can decrypt messages before the time comes. Ideally the "best" leak function with respect to security is the identity one, the one that gives no real advantage to the adversary. Unfortunately, there are some time-lock encryption schemes where the adversary can decrypt a little bit earlier than the honest voters. For example time-lock encryption that is based on bitcoin. This is happening because the adversary can locally compute some witnesses (selfish mining) without announcing them to the rest of the parties. In the context of e-voting, it means that the adversary for these time-lock instantiations might decrypt the ballots before the tally phase, learn the election outcome and change its vote (violation of fairness). Fortunately, even these time-lock encryption schemes are useful in voting if we introduce a "time wait" period, that's it a period where no one can cast a ballot and ballots can not be opened either. In that period, the adversary can open the ballot but it can not change its vote because the casting period is over. This is why we defined the predicates $\texttt{define\_time}, \texttt{Status}$. As we see next, these predicates are instantiated with the leakage function in order our $\mathcal{F}_{\mathsf{TLE}}$ to be able to realise $\mathcal{F}_{\mathsf{vm}}$.

4.2 Realization of $\mathcal{F}_{\mathsf{TLE}}$ via time-lock puzzles

In this subsection we present the realization of $\mathcal{F}_{\mathsf{TLE}}$ via a protocol that uses a pair of encryption/decryption algorithms that satisfy a pair of specific properties (cf. Definition 2). We claim that the construction by Rivest *et al.* proposed in [30] satisfies these properties.

The general idea of a time-lock puzzle scheme is that the parties have restricted access to a specific computation in a period of time in order to solve that puzzle (in [30]'s case that computation is the repeated squaring). Of course, the underlying assumption here is that there is no "better" way to solve that puzzle except for applying the specific computation. Some of the most prominent proposed time-lock constructions are based on such assumption [30,25,2].

In the UC framework, in order to construct a time-lock protocol we need to abstract such computation into an oracle, namely $\mathcal{F}_{\mathsf{eval}}$. The reason behind this is simple. In UC all the parties are allowed to run polynomial time with respect to the protocol's parameter. As a result, it is impossible to restrict a party in a specific period of time to apply a certain number of computations. That is why we need to abstract such computation into a functionality/oracle and wrap the oracle with a *functionality wrapper*, similar to [2], for restricted access.

Next, we present the evaluation oracle $\mathcal{F}_{\mathsf{eval}}$, the functionality wrapper $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$, and we define what is a pair of encryption/decryption algorithms with respect to an oracle. Finally, we present the protocol Π_{TLE} and we provide a security definition that every TLE construction should satisfy such that Π_{TLE} UC realizes $\mathcal{F}_{\mathsf{TLE}}$.

The evaluation functionality $\mathcal{F}_{\mathsf{eval}}(\mathcal{D}, \mathbf{P})$.

Initializes an empty evaluation query list L_{eval}.

- Upon receiving $(\mathsf{sid}, \text{EVALUATE}, x)$ from a party $P \in \mathbf{P}$, it does:

1. It checks if $(x, y) \in L_{\mathsf{eval}}$ for some y. If no such entry exists, it samples y from the distribution $\mathbf{D}_x$ and inserts the pair (x, y) to L_{eval}. Then, it returns $(\mathsf{sid}, \text{EVALUATED}, x, y)$ to P. Else, it returns the recorded pair.

Fig. 9. Functionality $\mathcal{F}_{\mathsf{eval}}$ parameterized by λ,a family of distributions $\mathcal{D} = \{\mathbf{D}_x | x \in \mathbf{X}\}$ and a set of parties $\mathbf{P}$.

The evaluation functionality $\mathcal{F}_{\mathsf{eval}}$. Initially, the functionality $\mathcal{F}_{\mathsf{eval}}$ (cf. Fig 9) creates the list L_{eval} so that it can keep a record of the queries it received so far. After $\mathcal{F}_{\mathsf{eval}}$ receives a query from a party, it checks if this query was issued before. If this is the case, it returns the recorded pair. If not, then for the query x it samples from the distribution $\mathbf{D}_x$ the value y. This distribution in Rivest *et al.* case [30] illustrates the repeated squaring (we define how explicitly later), or in cases such as in [25,2] is a random value over a domain (thus $\mathcal{F}_{\mathsf{eval}}$ is the random oracle). Finally, it returns to that party the pair (x, y).

The functionality wrapper $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$. The functionality wrapper (cf. Fig. 10) when it receives an evaluation query with a ciphertext c from a party P it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If this is the first time that this query with c is issued, then it creates the list $L_{\mathsf{activity}}^{P,c}$ in order to keep track of the queries. In the case that this is the first query for c it checks if the first query is equal with state_0^c and if it is labeled as the first query. The function state_0 takes as an input a ciphertext c and outputs the first query (in Rivest's *et al.* construction this is the base of the repeated squaring). When it receives the answer from the functionality/oracle $\mathcal{F}_{\mathsf{eval}}$, $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$ records it and returns the answer to party P. If this is not the first time a query was issued for the ciphertext c, the functionality wrapper checks if the query is equal with the previous recorded answer of $\mathcal{F}_{\mathsf{eval}}$ (formation of chain) and if the number of queries for that time Cl does not exceed q. This illustrates the fact that the queries should form a chain (e.g., repeated squaring) and parties have limited number of queries per ciphertext per clock round. In [2], the functionality wrapper is defined in the same spirit. The difference is that in [2] the number of queries for each party each round is upper bounded by q, whereas here q is the upper bound for each ciphertext each party each round. The reason behind this is that we care to capture not only the restricted access to the functionality oracle in general but with respect to a ciphertext.

Functionality wrapper $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}}, \mathcal{G}_{\mathsf{clock}}, \mathbf{P}, \mathsf{state}_0)$.

■ Upon receiving $(\mathsf{sid}, \text{EVALUATE}, c, x, \tau_{\mathsf{dec}}, j)$ from $P \in \mathbf{P}$ it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and does:

1. If there is not a list $L^{P,c}_{\mathsf{activity}}$ it creates one and does:
   (a) If $j = 1 \vee \mathsf{state}^c_0 = x$, it inserts the tuple-$(c, \tau_{\mathsf{dec}}, \{\emptyset\}, \mathsf{Cl}, 0)$ to $L^{P,c}_{\mathsf{activity}}$ and sends $(\mathsf{sid}, \text{EVALUATE}, x)$ to $\mathcal{F}_{\mathsf{eval}}$. Upon receiving $(\mathsf{sid}, \text{EVALUATE}, x, y)$ from $\mathcal{F}_{\mathsf{eval}}$, it registers (x, y) as $(\mathsf{state}^c_0, \mathsf{state}^c_1)$ and updates the previous tuple to $(c, \tau_{\mathsf{dec}}, \{(\mathsf{state}^c_0, \mathsf{state}^c_1)\}, \mathsf{Cl}, 1)$ and sends $(\mathsf{sid}, \text{EVALUATE}, c, x, \tau_{\mathsf{dec}}, j, y)$ to P.
2. Else it does:
   (a) If there is a tuple of the form $(c, \tau_{\mathsf{dec}}, \{(\mathsf{state}^c_{k-1}, \mathsf{state}^c_k)\}^{j-1}_{k=1}, \mathsf{Cl}, j-1)$ with $x = \mathsf{state}^c_{j-1} \wedge j-1 \leq q$, it sends $(\mathsf{sid}, \text{EVALUATE}, x)$ to $\mathcal{F}_{\mathsf{eval}}$. Upon receiving $(\mathsf{sid}, \text{EVALUATE}, x, y)$ from $\mathcal{F}_{\mathsf{eval}}$, it registers (x, y) as $(\mathsf{state}^c_{j-1}, \mathsf{state}^c_j)$ and updates the previous tuple to $(c, \tau_{\mathsf{dec}}, \{(\mathsf{state}^c_{k-1}, \mathsf{state}^c_k)\}^{j}_{k=1}, \mathsf{Cl}, j)$. Then it sends $(\mathsf{sid}, \text{EVALUATE}, c, x, \tau_{\mathsf{dec}}, j, y)$ to P.
   (b) Else, if there is a tuple of the form $(c, \tau_{\mathsf{dec}}, \{(\mathsf{state}^c_{k-1}, \mathsf{state}^c_k)\}^{q}_{k=1}, \mathsf{Cl}-1, q)$ with $x = \mathsf{state}^c_q \wedge j = 1$ it sends $(\mathsf{sid}, \text{EVALUATE}, x)$ to $\mathcal{F}_{\mathsf{eval}}$. Upon receiving $(\mathsf{sid}, \text{EVALUATE}, x, y)$ from $\mathcal{F}_{\mathsf{eval}}$, it registers (x, y) as $(\mathsf{state}^c_0, \mathsf{state}^c_1)$ and inserts the tuple-$(c, \tau_{\mathsf{dec}}, \{(\mathsf{state}^c_0, \mathsf{state}^c_1)\}, \mathsf{Cl}, 1)$ to $L^{P,c}_{\mathsf{activity}}$. Then it sends $(\mathsf{sid}, \text{EVALUATE}, c, x, \tau_{\mathsf{dec}}, j, y)$ to P.
3. In any other case it sends $(\mathsf{sid}, \text{EVALUATE}, c, x, \tau_{\mathsf{dec}}, j, \perp)$ to P.

Fig. 10. The Functionality wrapper $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$ parameterized by λ, a number of queries q, an initial state function state_0, functionality $\mathcal{F}_{\mathsf{eval}}$, $\mathcal{G}_{\mathsf{clock}}$ and parties in $\mathbf{P}$.

Algorithms with respect to an oracle. A pair of time-lock algorithms $(e_{\mathcal{O}}, d_{\mathcal{O}})$ with respect to an oracle $\mathcal{O}$[3] are a randomized and a deterministic Turing machine respectively that: The encryption algorithm $e_{\mathcal{O}}$ accepts as an input a message m, the current time Cl, the decryption time τ_{dec} and a randomness r [4]and outputs a ciphertext c. The decryption algorithm $d_{\mathcal{O}}$ accepts as an input the ciphertext c, and the witness w_τ [4]. The function interacts with the oracle so that it can decide what value to output. Specifically, the function checks if the queries that are needed so that the witness w_τ can be constructed are issued through the oracle. This is achieved by an interaction between the algorithm and the oracle itself. Of course, such interaction is compatible with the oracle's interface. For that task, it is necessary that there is a function that "decomposes" the witness $w_{\tau_{\mathsf{dec}}}$ to oracle queries so that $d_{\mathcal{O}}$ can check if these queries are issued through the oracle. As a result, algorithm $d_{\mathcal{O}}$ needs to be parameterized by such a function. For simplicity reasons we keep the same notation. The most "interesting" algorithms from the security perspective are the ones that output $\perp$ in the case that the queries are not recorded to the oracle, and the actual message if they are.

[3] In UC setting, the oracle is the evaluation functionality $\mathcal{F}_{\mathsf{eval}}$.
[4] In UC setting, it also accepts the current sid where the protocol is executed.

Protocol $\Pi_{\mathsf{TLE}}(\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}}), e_{\mathcal{F}_{\mathsf{eval}}}, d_{\mathcal{F}_{\mathsf{eval}}}, \mathcal{G}_{\mathsf{clock}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{an.BC}}, \mathbf{P})$.

■ Upon receiving $(\mathsf{sid}, \textsc{Start})$ from $\mathcal{Z}$, P does the following:

1. If status is '$\mathsf{not\_ready}$', it changes the status to 'ready' and it does:
   (a) It records the pair (e, d), where $e(\cdot_m, \cdot_{\tau_{\mathsf{dec}}}, \cdot_{\mathsf{sid}}, \cdot_{\mathsf{Cl}}; r_1||r_2) \leftarrow (e_{\mathcal{F}_{\mathsf{eval}}}(\cdot_{r_1}, \cdot_{\tau_{\mathsf{dec}}}, \cdot_{\mathsf{sid}}, \cdot_{\mathsf{Cl}}; r_2), \mathcal{H}(\cdot_{r_1}) \oplus \cdot_m, \mathcal{H}(\cdot_{r_1}||\cdot_m))$ and $d(\cdot_c, \cdot_{w_{\tau_{\mathsf{dec}}}}, \cdot_{\mathsf{sid}}) \leftarrow (\cdot_x \leftarrow d_{\mathcal{F}_{\mathsf{eval}}}(\cdot_{c_1}, \cdot_{w_{\tau_{\mathsf{dec}}}}, \cdot_{\mathsf{sid}}), \cdot_m \leftarrow \mathcal{H}(\cdot_x) \oplus \cdot_{c_2}, \cdot_{\mathcal{R}_{\mathsf{equal}}(\mathcal{H}(x||m), c_3)})$, where (i) $\mathcal{R}_{\mathsf{equal}}(a, b)$ is 1, if $a = b$, and 0 otherwise, and (ii) $\mathcal{H}(\cdot)$ is an abbreviation of a call to the random oracle $\mathcal{F}_{\mathsf{RO}}$.
   (b) It outputs $(\mathsf{sid}, \textsc{Start\_Ok})$.

■ Upon receiving $(\mathsf{sid}, \textsc{Enc}, m, \tau_{dec})$ from $\mathcal{Z}$, P does:

1. If $\tau_{dec} < 0$ or status is '$\mathsf{not\_ready}$', it returns $(\mathsf{sid}, \textsc{Enc}, m, \tau_{dec}, \perp)$ to $\mathcal{Z}$.
2. Else it picks $r_1 \xleftarrow{\$} \{0,1\}^{|poly(\lambda)|}$ and sends $(\mathsf{sid}, \textsc{Query}, r_1)$ to $\mathcal{F}_{\mathsf{RO}}$. Upon receiving $(\mathsf{sid}, \textsc{Random\_Oracle}, r_1, h)$ from $\mathcal{F}_{\mathsf{RO}}$, P sends $(\mathsf{sid}, \textsc{Query}, r_1||m)$ to $\mathcal{F}_{\mathsf{RO}}$. Upon receiving $(\mathsf{sid}, \textsc{Random\_Oracle}, r_1||m, c_3)$ from $\mathcal{F}_{\mathsf{RO}}$, P reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and computes $c \leftarrow (e_{\mathcal{F}_{\mathsf{eval}}}(r_1, \tau_{\mathsf{dec}}, \mathsf{sid}, \mathsf{Cl}), h \oplus m, c_3)$. Then, it sends $(\mathsf{sid}, \textsc{Broadcast}, c, \tau_{\mathsf{dec}})$ to $\mathcal{F}_{\mathsf{an.BC}}$. Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, c, \tau_{\mathsf{dec}})$ from $\mathcal{F}_{\mathsf{an.BC}}$, it outputs $(\mathsf{sid}, \textsc{Enc}, m, \tau_{\mathsf{dec}}, c)$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, c := (c_1, c_2, c_3), \tau_{\mathsf{dec}})$ from $\mathcal{F}_{\mathsf{an.BC}}$, P reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and does:

1. It creates an empty list $L^{P,c_1}_{\mathsf{record}}$.
2. It inserts the tuple-$(c_1, \tau_{\mathsf{dec}}, \{\emptyset\}_{\mathsf{queries}}, \mathsf{Cl}, 0) \rightarrow L^{P,c_1}_{\mathsf{record}}$ and computes and stores $\mathsf{state}^{c_1}_0$.
3. For $(j = 1; q; j++)$ It sends $(\mathsf{sid}, \textsc{Evaluate}, c_1, \mathsf{state}^{c_1}_{j-1}, \tau_{\mathsf{dec}}, j)$ to $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$. Upon receiving $(\mathsf{sid}, \textsc{Evaluate}, c_1, \mathsf{state}^{c_1}_{j-1}, \tau_{\mathsf{dec}}, j, y)$ from $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$, it creates $y \rightarrow \mathsf{state}^{c_1}_j$ and updates $(c_1, \tau_{\mathsf{dec}}, \{(\mathsf{state}^{c_1}_{k-1}, \mathsf{state}^{c_1}_k)\}^j_{k=1}, \mathsf{Cl}, j) \rightarrow L^{P,c_1}_{\mathsf{record}}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Advance\_Clock})$ from $\mathcal{Z}$, P reads the time Cl from the $\mathcal{G}_{\mathsf{clock}}$ and does:

1. It collects all tuples of the form $(c_1, \tau_{\mathsf{dec}}, \{(\mathsf{state}^{c_1}_{k-1}, \mathsf{state}^{c_1}_k)\}^q_{k=1}, \mathsf{Cl} - 1, 1) \in L^{P,c_1}_{\mathsf{record}}$ and for each one of them it repeats the previous steps [2,3] with $\mathsf{state}^{c_1}_0 \leftarrow \mathsf{state}^{c_1}_q$.
2. It sends $(\mathsf{sid}, \textsc{Advance\_Clock})$ to $\mathcal{G}_{\mathsf{clock}}$ and returns whatever it receives to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Dec}, c := (c_1, c_2, c_3), \tau_{\mathsf{dec}})$ from $\mathcal{Z}$, P reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and does:

1. If $\tau_{\mathsf{dec}} < 0$ or status is '$\mathsf{not\_ready}$', it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau_{\mathsf{dec}}, \perp)$ to $\mathcal{Z}$.
2. It searches for a tuple of the form $(c_1, \tau_{\mathsf{dec}}, \{(\mathsf{state}^{c_1}_{k-1}, \mathsf{state}^{c_1}_k)\}^q_{k=1}, t, q) \in L^{P,c_1}_{\mathsf{record}}$ with the maximum value t (that can be either $\mathsf{Cl} - 1$ or Cl). If $t = \mathsf{Cl} - 1$, it updates the tuple by running the previous steps [2,3].
3. It collects all the sets $\{(\mathsf{state}^{c_1}_{k-1}, \mathsf{state}^{c_1}_k)\}^q_{k=1}$ for all the values of t in $L^{P,c_1}_{\mathsf{record}}$ and sets them as $w_{\tau_{\mathsf{dec}}}$.
4. It runs $x \leftarrow d_{\mathcal{F}_{\mathsf{eval}}}(c_1, w_{\tau_{\mathsf{dec}}}, \mathsf{sid})$ and it sends $(\mathsf{sid}, \textsc{Query}, x)$ to $\mathcal{F}_{\mathsf{RO}}$. Upon receiving $(\mathsf{sid}, \textsc{Random\_Oracle}, x, h)$ from $\mathcal{F}_{\mathsf{RO}}$, it computes $m \leftarrow h \oplus c_2$. It sends $(\mathsf{sid}, \textsc{Query}, x||m)$ to $\mathcal{F}_{\mathsf{RO}}$. Upon receiving $(\mathsf{sid}, \textsc{Random\_Oracle}, x||m, c^*_3)$ from $\mathcal{F}_{\mathsf{RO}}$: If $c_3 \neq c^*_3$, it returns to $\mathcal{Z}$ $(\mathsf{sid}, \textsc{Dec}, c, \tau_{\mathsf{dec}}, \perp)$. Else, it returns to $\mathcal{Z}$ $(\mathsf{sid}, \textsc{Dec}, c, \tau_{\mathsf{dec}}, m)$.
5. If such tuple doesn't exists then it returns $(\mathsf{sid}, \textsc{Dec}, c, \tau_{\mathsf{dec}}, \perp)$ to $\mathcal{Z}$.

Fig. 11. The Protocol Π_{TLE} in the presence of a functionality wrapper $\mathcal{W}_q$, an evaluation functionality $\mathcal{F}_{\mathsf{eval}}$, a random oracle $\mathcal{F}_{\mathsf{RO}}$, an anonymous broadcast functionality $\mathcal{F}_{\mathsf{an.BC}}$, a global clock $\mathcal{G}_{\mathsf{clock}}$, a pair of algorithms $e_{\mathcal{F}_{\mathsf{eval}}}, d_{\mathcal{F}_{\mathsf{eval}}}$ where is hard-coded in each party in $\mathbf{P}$.

The protocol Π_{TLE}. At the beginning of the protocol (cf. Fig. 11), each party initializes their encryption/decryption algorithms by extending the ones they have already hard-coded. This extension is based on the construction of [28,6] and is important for the realization of $\mathcal{F}_{\mathsf{TLE}}$. Recall that in $\mathcal{F}_{\mathsf{TLE}}$ all the ciphertexts eventually open. Moreover, in order to capture semantic security, the ciphertext contains the encryption of the zero string in contrast to the real protocol that contains the encryption of the actual message. So, in order for the simulator to be able to simulate this difference when the messages are opened, $\mathcal{S}$ must be able to *equivocate* the opening of the ciphertext, else $\mathcal{Z}$ can trivially distinguish the real from the ideal execution of the protocol. When a party receives an encryption request from $\mathcal{Z}$ it follows the description of the encryption algorithm and then broadcasts the ciphertext. The broadcast is necessary because if we want the message to be opened after a specific time for every party by the time of its creation, the receiving parties must start to work on solving the puzzle immediately [30]. In constructions such as in [25] this is not necessary because the puzzle is the same for every ciphertext posted in the blockchain. Next, when a party receives the broadcast ciphertext it reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and it queries q times the $\mathcal{F}_{\mathsf{eval}}$ through the functionality wrapper $\mathcal{W}_q$ for that ciphertext. Similarly, when a party receives an advance clock command from $\mathcal{Z}$ it reads again the time Cl from $\mathcal{G}_{\mathsf{clock}}$ and issues q queries for all stored ciphertexts that it holds. The queries illustrate the "effort" of the party to solve the puzzle so that it can decrypt the message later. Finally, when a party receives a decryption command from $\mathcal{Z}$ for a ciphertext c it uses the last state that it received from the $\mathcal{F}_{\mathsf{eval}}$ through the functionality wrapper $\mathcal{W}_q$ as the decryption key and returns to $\mathcal{Z}$ either a message m if the decryption was successful or $\perp$, otherwise. Note that, as in the construction of [28,6], the third argument in the ciphertext makes the scheme non-malleable.

Definition of a secure TLE scheme. In Fig. 12, we present the experiment $\mathbf{EXP}_{\mathsf{TLE}}$ in the presence of a challenger Ch and an adversary $\mathcal{B}$. This experiment illustrates the security of a TLE scheme in the sense that no adversary can open a message before a certain number of computations have been done. Specifically, we allow in a similar way as previously the adversary to have access to the evaluation oracle $\mathcal{O}_{\mathsf{eval}}$. If the adversary queries the oracle q times for a ciphertext c, the challenger, which maintains a counter for that ciphertext, increases that counter by one. The queries are formed as before with the initial query for ciphertext c the state_0^c. Upon request, the adversary receives a challenging ciphertext from the challenger. If the adversary can guess correctly the underneath plaintext with less than the required computations, then it wins the game. We define a time-lock scheme to be secure if it satisfies two properties: (i) *Correctness* that captures the fact that the decryption of the encryption of a message m leads again to the message m with high probability, and (ii) *q-Security* according to which no adversary can win the experiment $\mathbf{EXP}_{\mathsf{TLE}}$ except with small probability.

We assume that the description of the oracle $\mathcal{O}_{\mathsf{eval}}$ in Fig. 12, is exactly the same as the ideal functionality in Fig. 9 without the UC interface.

The experiment $\mathbf{EXP}_{\mathsf{TLE}}(\mathcal{B}^{\mathcal{O}_{\mathsf{eval}}}, \mathsf{Ch}^{\mathcal{O}_{\mathsf{eval}}}, q)$

- Ch initialized with $e_{\mathcal{O}_{\mathsf{eval}}}, d_{\mathcal{O}_{\mathsf{eval}}}$ and sends them to $\mathcal{B}$.
- When $\mathcal{B}$ issues the query $(\textsc{Evaluate}, x)$ to $\mathcal{O}_{\mathsf{eval}}$, it gets back $(\textsc{Evaluate}, x, y)$.
- $\mathcal{B}$ can request the encryption of a message $m \in \mathbf{M}_\lambda$ with time label τ_{dec} by sending $(\textsc{Enc}, m, \tau_{\mathsf{dec}})$ to Ch, where $\mathbf{M}_\lambda$ is the domain of the messages.
- When Ch receives a $(\textsc{Enc}, m, \tau_{\mathsf{dec}})$ request from $\mathcal{B}$, it runs the algorithm $e_{\mathcal{O}_{\mathsf{eval}}}(m, \tau_{\mathsf{dec}}) \rightarrow c$ and creates a local counter Cl_c with respect to a ciphertext c. Ch increases Cl_c by 1 (initially is 0) every time $\mathcal{B}$ queries $\mathcal{O}_{\mathsf{eval}}$ q times with respect to c. This is easily checkable from the query path that is recorded to $\mathcal{O}_{\mathsf{eval}}$. The initial query is for the value state_0^c. After this, every query is linked with the previous one. Ch returns c to $\mathcal{B}$.
- $\mathcal{B}$ can request the decryption of a ciphertext c by sending $(\textsc{Dec}, c, w_\tau)$ to Ch. Then, Ch just runs the algorithm $d_{\mathcal{O}_{\mathsf{eval}}}(c, w_{\tau_{\mathsf{dec}}}) \rightarrow y \in \{m, \perp\}$ and returns to $\mathcal{B}$ $(\textsc{Dec}, c, w_\tau, y)$.
- $\mathcal{B}$ can request for a single time a challenge from Ch by sending $(\textsc{Challenge}, \tau)$. Then, Ch picks a value $r \xleftarrow{\$} \mathbf{M}_\lambda$ and sends $(\textsc{Challenge}, \tau, e_{\mathcal{O}_{\mathsf{eval}}}(r, \tau))$ to $\mathcal{B}$.
- B sends as the answer of the challenge $c_r := e_{\mathcal{O}_{\mathsf{eval}}}(r, \tau)$ the value r^* to Ch. Specifically, it sends $(\textsc{Challenge}, \tau, e_{\mathcal{O}_{\mathsf{eval}}}(r, \tau), r^*)$.
- If $(r^* = r) \wedge (\tau > \mathsf{Cl}_{c_r})$ (i.e., $\mathcal{B}$ manages to decrypt c_r before the decryption time comes) then $\mathbf{EXP}_{\mathsf{TLE}}$ outputs 1. Else, $\mathbf{EXP}_{\mathsf{TLE}}$ outputs 0.

Fig. 12. Experiment $\mathbf{EXP}_{\mathsf{TLE}}$ for a number of queries q in the presence of an adversary $\mathcal{B}$, $\mathcal{O}_{\mathsf{eval}}$ and a challenger Ch all parameterized by 1^λ.

We provide our property-based definition of a secure TLE scheme below.

Definition 2. *A secure* time-lock encryption scheme *with respect to a evaluation oracle* $\mathcal{O}_{\mathsf{eval}}$ *for message space* $\mathbf{M}$ *and parameter* λ *is a pair of PPT algorithms* $(e_{\mathcal{O}_{\mathsf{eval}}}, d_{\mathcal{O}_{\mathsf{eval}}})$ *such that:*

- $e_{\mathcal{O}_{\mathsf{eval}}}(m, \tau_{\mathsf{dec}}, 1^\lambda)$*: The encryption algorithm takes as input message a* $m \in \mathbf{M}$*, an integer* $\tau_{\mathsf{dec}} \in \mathbb{N}$*, the security parameter* λ *and outputs a ciphertext* c*.*
- $d_{\mathcal{O}_{\mathsf{eval}}}(c, w_{\tau_{\mathsf{dec}}})$*: The decryption algorithm takes as input* $w_{\tau_{\mathsf{dec}}} \in \{0,1\}^*$ *and a ciphertext* c*, and outputs a message* $m \in \mathbf{M}$ *or* $\perp$*.*

$(e_{\mathcal{O}_{\mathsf{eval}}}, d_{\mathcal{O}_{\mathsf{eval}}})$ *satisfies the following properties:*

1. ***Correctness:*** *For every* $\lambda, \tau_{\mathsf{dec}} \in \mathbb{N}$ *and every* $m \in \mathbf{M}$*, it holds that* $\Pr\left[c \leftarrow e_{\mathcal{O}_{\mathsf{eval}}}(m, \tau_{\mathsf{dec}}, 1^\lambda) : m = d_{\mathcal{O}_{\mathsf{eval}}}(c, w_{\tau_{\mathsf{dec}}})\right] > 1 - negl(\lambda)$ *, where* $w_{\tau_{\mathsf{dec}}}$ *is produced through* $\mathcal{O}_{\mathsf{eval}}$*.*
2. q***-Security:*** *For every PPT adversary* $\mathcal{B}$ *with access to oracle* $\mathcal{O}_{\mathsf{eval}}$*, the probability to win the experiment* $\mathbf{EXP}_{\mathsf{TLE}}(\mathcal{B}^{\mathcal{O}_{\mathsf{eval}}}, \mathsf{Ch}^{\mathcal{O}_{\mathsf{eval}}}, q)$ *in Fig.12 is at most* $negl(\lambda)$*.*

Next, we show that if the TLE scheme used in protocol Π_{TLE} in Fig. 11 is a secure time-lock encryption scheme according to Definition 2 then the protocol Π_{TLE} UC realizes $\mathcal{F}_{\mathsf{TLE}}$.

Theorem 2. *Let* $(e_{\mathcal{O}_{\mathsf{eval}}}, d_{\mathcal{O}_{\mathsf{eval}}})$ *be a pair of* `encryption`/`decryption` *algorithms that satisfies Definition 2. Then, the protocol* $\Pi_{\mathsf{TLE}}(\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}}), e_{\mathcal{F}_{\mathsf{eval}}}, d_{\mathcal{F}_{\mathsf{eval}}}, \mathcal{G}_{\mathsf{clock}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{an.BC}}, \mathbf{P})$ *UC-realizes functionality* $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ *with leakage function* leak *to be the identity function.*

Proof (Sketch). Let us suppose that protocol Π_{TLE} does not UC-realize $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$. Then, by Definition 1, there is an adversary $\mathcal{A}$ s.t. for every simulator $\mathcal{S}$ there is an environment $\mathcal{Z}$ s.t.: $|\Pr[\text{EXEC}_{\mathcal{Z},\mathcal{A}}^{\Pi_{\mathsf{TLE}}} = 0] - \Pr[\text{EXEC}_{\mathcal{Z},\mathcal{S}}^{\mathcal{F}_{\mathsf{TLE}}^{\mathsf{TLE}}} = 0]| > \alpha(\lambda)(1)$, where $\alpha()$ is a non negligible function.

Now consider the specific simulator $\mathcal{S}$ below: At the beginning, $\mathcal{S}$ receives the corruption vector from $\mathcal{Z}$ and informs $\mathcal{A}$. When $\mathcal{S}$ gets the token back from $\mathcal{A}$, it sends the corruption vector to $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$. If $\mathcal{S}$ receives Start from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$, it generates a pair of algorithms $(e_{\mathcal{S}}, d_{\mathcal{S}})$ with the same description as the pair (e, d) in the protocol Π_{TLE} except that the created cipher texts c_2, c_3 are equal to a random value which is part of the random coin the algorithm $e_{\mathcal{S}}$ uses. When $\mathcal{S}$ receives an encryption request from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ on behalf of a corrupted player, it forwards the message to $\mathcal{A}$ the request as if it was from that party. Then, $\mathcal{S}$ returns whatever receives from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$. Observe that there is no interaction between $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ and $\mathcal{S}$ in an encryption request on behalf of an honest party. Similarly, in protocol Π_{TLE} there is not interaction between the encryption algorithm and the $\mathcal{F}_{\mathsf{eval}}$. In case $\mathcal{S}$ receives a decryption request from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ with ciphertext c and time label τ_{dec} on behalf of an honest party, it does:$\mathcal{S}$ generates the witness $w_{\tau_{\mathsf{dec}}}$ and updates its list $L_{\mathsf{eval}}^{\mathcal{S}}$(initially empty) exactly as $\mathcal{F}_{\mathsf{eval}}$ in protocol Π_{TLE} for consistency between the witness and the oracle queries. Then, $\mathcal{S}$ returns to $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ the message $m \leftarrow d_{\mathcal{S}}(c, w_{\tau_{\mathsf{dec}}})$.

If $\mathcal{S}$ receives a decryption request for a ciphertext c with time label τ_{dec} from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ on behalf of a corrupted party, it forwards the message to $\mathcal{A}$ as if it was from that party. Then, if $\mathcal{S}$ is activated via the algorithm $d_{\mathcal{S}}$ for the ciphertext c with witness $w_{\tau_{\mathsf{dec}}}$(see definition 4.2), it returns $m \leftarrow d_{\mathcal{S}}(c, w_{\tau_{\mathsf{dec}}})$ to $\mathcal{A}$. $\mathcal{S}$ returns whatever receives from $\mathcal{A}$ as if it was the corrupted party back to $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$. In case $\mathcal{S}$ receives a random oracle query request from $\mathcal{F}_{\mathsf{TLE}}$ on behalf of a corrupted party, it forwards the message to $\mathcal{A}$ as if it was from that party. When $\mathcal{S}$ receives this request from $\mathcal{A}$ playing the role of $\mathcal{F}_{\mathsf{RO}}$, it sends the command LEAKAGE to $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$. Then $\mathcal{S}$ checks if the received record from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ contains any relation between a message m and the random oracle query that $\mathcal{S}$ received initially from the corrupted party. If $\mathcal{S}$ finds such relation, it programs the oracle so that ciphertext can be opened to message m. Then, it responds to $\mathcal{A}$ as if it was the $\mathcal{F}_{\mathsf{RO}}$. In the case $\mathcal{S}$ founds the oracle query but the list does not contain the message, it outputs "$\perp$"(meaning that the adversary was lucky enough to guess a plaintext before the time comes, or the adversary "broke" the security of the encryption scheme). In any other case it behaves just like a random oracle. Finally, when $\mathcal{S}$ receives the command $\mathsf{Evaluate}$ from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ on behalf of a corrupted party, it

forwards the message to $\mathcal{A}$ as if it was that party. When $\mathcal{S}$ receives the $\mathsf{Evaluate}$ command from $\mathcal{A}$ on behalf of the corrupted party as if it was $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$, it behaves exactly as $\mathcal{W}_q(\mathcal{F}_{\mathsf{eval}})$ in protocol Π_{TLE}. Therefore, for $\mathcal{S}$ defined above there is an $\mathcal{Z}_{\mathcal{S}}$ such that Eq. (1) holds. Moreover lets us suppose that the pair $(e_{\mathcal{O}_{\mathsf{eval}}}, d_{\mathcal{O}_{\mathsf{eval}}})$ satisfies the **Correctness** property. We construct an adversary $\mathcal{B}$ that can break the $q-$**Security** with probability at least $\tilde{\alpha}(\lambda)$, where $\tilde{\alpha}()$ a non negligible function. Observe that the only way $\mathcal{Z}_{\mathcal{S}}$ to distinguish the real from the ideal execution of our protocol with non-negligible probability is to decrypt the first argument of a ciphertext, namely c_1, generated by an honest party before the time comes. This is possible if $\mathcal{Z}_{\mathcal{S}}$ is able to construct a witness $w_{\tau_{\mathsf{dec}}}$ for a honest generated ciphertext c_1 via the queries issued by a corrupted party to $\mathcal{W}_{\mathcal{F}_{\mathsf{eval}}}$ in the real execution of the protocol or in $\mathcal{S}$ in the ideal execution given that the global time Cl provided by $\mathcal{G}_{\mathsf{clock}}$ is strictly smaller than τ_{dec}. Next, $\mathcal{Z}_{\mathcal{S}}$ will request a random oracle query by a corrupted party with the query value to be the plaintext of the ciphertext c_1. Next, $\mathcal{S}$ in order to equivocate correctly, it needs the corresponding message. But if the time of that message has not come yet (e.g $\mathsf{Cl} < \tau_{\mathsf{dec}}$), the recorded table that $\mathcal{S}$ will request from $\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}$ via the LEAKAGE command, it will not contain that message. As a result, $\mathcal{S}$ will fail to equivocate correctly and $\mathcal{Z}_{\mathcal{S}}$ can distinguish the two executions. Now $\mathcal{B}$ takes advantage of that environment, and uses it in order to win the experiment $\mathbf{EXP}_{\mathsf{TLE}}$ with non negligible probability in the following way: $\mathcal{B}$ simulates the interface to the environment as in the ideal execution of the protocol in the presence of the global clock, except when receiving $(\mathsf{sid}, \mathsf{Evaluate}, c, x, \tau\mathsf{dec}, j)$ from $\mathcal{Z}$ where it forwards the message $(\text{EVALUATE}, x)$ to the oracle $\mathcal{O}_{\mathsf{eval}}$ in $\mathbf{EXP}_{\mathsf{TLE}}$. When $\mathcal{Z}_{\mathcal{S}}$ advances the clock, $\mathcal{B}$ does nothing more than allowing $\mathcal{Z}_{\mathcal{S}}$ to make more evaluation queries to $\mathcal{F}_{\mathsf{eval}}$ for each ciphertext in the way described before. Any other procedure is run locally by $\mathcal{B}$. Now, $\mathcal{B}$ knows that the environment will make at most $p_{\mathcal{H}}(\lambda), p_{\mathsf{enc}}(\lambda)$ random oracle and encryption queries respectively, where $p_{\mathcal{H},\mathsf{enc}}()$ are polynomial functions. At least one of these random oracle queries made by $\mathcal{Z}_{\mathcal{S}}$, from our hypothesis, will contain the plainttext (namely the value r_1 as described in Fig. 11) of one of the $p_{\mathsf{enc}}(\lambda)$ ciphertexts that has been decrypted by $\mathcal{Z}_{\mathcal{S}}$ before its time with non negligible probability $\alpha(\lambda)$. Therefore, $\mathcal{B}$ picks $j_1 \xleftarrow{\$} \{1, \ldots, p_{\mathsf{enc}}(\lambda)\}$. When $\mathcal{Z}_{\mathcal{S}}$ issues the j_1-th encryption query $(\mathsf{sid}, \text{ENC}, m, \tau_{\mathsf{dec}})$ to an honest party simulated by $\mathcal{B}$, $\mathcal{B}$ proceeds as follows: If $\tau_{\mathsf{dec}} > \mathsf{Cl}$, then it sends $(\text{CHALLENGE}, \tau_{\mathsf{dec}} - \mathsf{Cl})$ to Ch. When $\mathcal{B}$ receives $(\text{CHALLENGE}, \tau_{\mathsf{dec}} - \mathsf{Cl}, c_1)$ from Ch, $\mathcal{B}$ picks c_2, c_3 exactly as $\mathcal{F}_{\mathsf{TLE}}$ and returns $(\mathsf{sid}, \text{ENC}, m, \tau, c \leftarrow (c_1, c_2, c_3))$ to $\mathcal{Z}_{\mathcal{S}}$. Then, $\mathcal{B}$ picks $j_2 \xleftarrow{\$} \{1, \ldots, p_{\mathcal{H}}(\lambda)\}$. When $\mathcal{Z}_{\mathcal{S}}$ issues the j_2-th random oracle query $(\mathsf{sid}, \text{QUERY}, x)$ to a corrupted party, $\mathcal{B}$ sends x to Ch as the answer to the challenge. It can be seen that the probability x to be the answer of the challenge is at least $1/(p_{\mathsf{enc}}(\lambda)p_{\mathcal{H}}(\lambda)) \cdot \tilde{\alpha}(\lambda)$. □

Finally, we claim that the TLE construction of Rivest *et al.* [30] is a secure time-lock encryption scheme with:

1. The oracle queries are of the form (EVALUATE, $(2^{2^j}, 2)$) with oracle response $h = 2^{2^{j+1}} \mod n$ where n is a safe composite number [30]. As a result, the distribution $\mathbf{D}_{x=(2^{2^j},2)}$ is the constant distribution.
2. The encryption and decryption algorithms $(e_{\mathcal{F}_{\mathsf{eval}}}, d_{\mathcal{F}_{\mathsf{eval}}})$ are described in [30] with puzzle time complexity $\tau_{\mathsf{dec}} - \mathsf{Cl}$ (in the UC description). The time τ_{dec} gives us the essence of absolute time that a ciphertext should be opened. On the other hand, $e_{\mathcal{F}_{\mathsf{eval}}}$ in [30] functions in relativistic time. To compute relativistic time, both values Cl and τ_{dec} are provided to $e_{\mathcal{F}_{\mathsf{eval}}}$. Specifically, by considering the notation in Fig. 11 the algorithm $e_{\mathcal{F}_{\mathsf{eval}}}$ works as follows: 1) The algorithm picks at random a safe composite number n by using part of the randomness r_2. 2) It uses a standard symmetric key encryption scheme in order to encrypt message r_1 with a random key r_2^*, which can be derived from randomness r_2. The resulting ciphertext is c_{r_1,r_2^*}. 3) It picks a random $\alpha \mod n$ and computes $c_{r_2^*} = r_2^* + \alpha^{2^{\tau_{\mathsf{dec}}-\mathsf{Cl}}} \mod n$. 4) It outputs the ciphertext $c_1 = (n, \alpha, \tau_{\mathsf{dec}} - \mathsf{Cl}, c_{r_2^*}, c_{r_1,r_2^*})$.
   Similarly the decryption algorithm $d_{\mathcal{F}_{\mathsf{eval}}}$ uses the witness $w_{\tau_{\mathsf{dec}}} = r_2^*$ to retrieve the message r_1 from the ciphertext c_{r_1,r_2^*} .

4.3 A protocol $\Pi_{\mathsf{vm}}^{\mathcal{F}_{\mathsf{TLE}},\{\mathcal{F}_{\mathsf{cert}}^P\},\mathcal{F}_{\mathsf{an.BC}},\mathcal{G}_{\mathsf{clock}}}$ that realizes $\mathcal{F}_{\mathsf{vm}}$

The description of $\Pi_{\mathsf{vm}}^{\mathcal{F}_{\mathsf{TLE}},\{\mathcal{F}_{\mathsf{cert}}^P\},\mathcal{F}_{\mathsf{an.BC}},\mathcal{G}_{\mathsf{clock}}}$ shown in Fig. 13 follows the phases and the command interface of $\mathcal{F}_{\mathsf{vm}}$ in Subsection 3.2.

Definition 3. *Let* leak *be a leakage function and* $t_{\mathsf{cast}}, t_{\mathsf{open}}$ *be time points. We say that the pair of functions* $\mathtt{Status}_{\mathsf{leak}}, \mathtt{define\_time}_{\mathsf{leak}}$ *with respect to a leakage function* leak *is* ***phase preserving*** *if there is no time s.t. the* **Cast** *and* **Tally** *phases are simultaneously active. Formally, the following property holds:*

$$\forall \mathsf{Cl} \text{ such that } \mathtt{Status}_{\mathsf{leak}}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \top \Rightarrow \mathtt{Status}_{\mathsf{leak}}(\mathsf{leak}(\mathsf{Cl}), \boldsymbol{t}, \mathsf{Open}) = \bot \,,$$

where $\boldsymbol{t} \leftarrow \mathtt{define\_time}_{\mathsf{leak}}(t_{\mathsf{cast}}, t_{\mathsf{open}})$

Theorem 3. *Π_{vm} UC-realizes $\mathcal{F}_{\mathsf{vm}}$ in the $\{\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}, \mathcal{F}_{\mathsf{an.BC}}, \{\mathcal{F}_{\mathsf{cert}}^P\}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model given that the pair of functions $\mathtt{Status}_{\mathsf{leak}}, \mathtt{define\_time}_{\mathsf{leak}}$ is* ***phase preserving****.*

Proof. In cases where a corrupted party receives an input and we do not describe her behaviour, we assume that the message is sent to $\mathcal{S}$ from $\mathcal{F}_{\mathsf{vm}}$ and $\mathcal{S}$ forwards that message to $\mathcal{A}$ as if it was from that party. Then $\mathcal{S}$ returns to $\mathcal{F}_{\mathsf{vm}}$ whatever 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 $\mathcal{F}_{\mathsf{TLE}}, \mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$, $\mathcal{S}$ forwards it to $\mathcal{F}_{\mathsf{vm}}$. When $\mathcal{S}$ receives the setup information $(\mathsf{sid}, \text{SETUP\_INFO}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathbf{V}_{\mathsf{elig}})$ from $\mathcal{F}_{\mathsf{vm}}$, $\mathcal{S}$ sends $((\mathsf{SA}, \mathsf{sid}), \text{SETUP})$ to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$. Upon receiving $((\mathsf{SA}, \mathsf{sid}), \text{ALGORITHMS}, \mathtt{Verify}, \mathtt{Sign})$ from $\mathcal{A}$ playing the role of $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$, $\mathcal{S}$ stores

the algorithms $(\texttt{Verify}, \texttt{Sign})$ and produces the signature $\sigma \leftarrow \texttt{Sign}(t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathbf{V}_{\mathsf{elig}})$.

The vote management protocol $\Pi_{\mathsf{vm}}^{\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}, \mathcal{F}_{\mathsf{an.BC}}, \{\mathcal{F}_{\mathsf{cert}}^{i}\}, \mathcal{G}_{\mathsf{clock}}}$.

Each voter V maintains a list of the cast ballots L_{cast}^{V} initially as empty. Each voter initializes its status to 'exec'.

■ Upon receiving $(\mathsf{sid}, \textsc{Setup\_Info}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathbf{V}_{\mathsf{elig}})$ for the first time from $\mathcal{Z}$, SA sends $((\mathsf{SA}, \mathsf{sid}), \textsc{Setup})$ to $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$, and waits to receive $((\mathsf{SA}, \mathsf{sid}), \textsc{Setup})$ from $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$. Then SA computes $\boldsymbol{t} \leftarrow \texttt{define\_time}_{\mathsf{leak}}(t_{\mathsf{cast}}, t_{\mathsf{open}})$ and if $\boldsymbol{t} \neq \perp$ it sends $((\mathsf{SA}, \mathsf{sid}), \textsc{Sign}, (\mathsf{SA}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, \mathbf{V}_{\mathsf{elig}}))$ to $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$. Upon receiving $((\mathsf{SA}, \mathsf{sid}), \textsc{Signature}, (\mathsf{SA}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, \mathbf{V}_{\mathsf{elig}}), s)$ from $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$, it sends $(\mathsf{sid}, \textsc{Broadcast}, (\mathsf{SA}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, \mathbf{V}_{\mathsf{elig}}, s))$ to $\mathcal{F}_{\mathsf{an.BC}}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Broadcast}, (\mathsf{SA}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, \mathbf{V}_{\mathsf{elig}}, s))$ from $\mathcal{F}_{\mathsf{an.BC}}$, V sends $((\mathsf{SA}, \mathsf{sid}), \textsc{Verify}, (\mathsf{SA}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, \mathbf{V}_{\mathsf{elig}}), s)$ to $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$. If it returns 1, she stores t_{cast}, $t_{\mathsf{open}}, \boldsymbol{t}, \mathbf{V}_{\mathsf{elig}}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Start})$ from $\mathcal{Z}$, V forwards the message to $\mathcal{F}_{\mathsf{TLE}}$. Upon receiving $(\mathsf{sid}, \textsc{Start\_Ok})$ from $\mathcal{F}_{\mathsf{TLE}}$, V outputs the received message.

■ Upon receiving $(\mathsf{sid}, \textsc{Gen\_Ballot}, o)$ from $\mathcal{Z}$, V does:

1. If this is the first time receiving this command-message and $V \in \mathbf{V}_{\mathsf{elig}}$, V sends to $\mathcal{F}_{\mathsf{TLE}}$ $(\mathsf{sid}, \textsc{Enc}, o, t_{\mathsf{open}})$. Upon receiving $(\mathsf{sid}, \textsc{Enc}, o, t_{\mathsf{open}}, v)$ from $\mathcal{F}_{\mathsf{TLE}}$, V stores the pair $(V, v, o, 1)$ and returns the message $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, v)$ to $\mathcal{Z}$.
2. Else, V returns to $\mathcal{Z}$ $(\mathsf{sid}, \textsc{Gen\_Ballot}, o, \perp)$.

■ Upon receiving $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ from $\mathcal{Z}$, if her status is 'exec', V reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. If $\texttt{Status}_{\mathsf{leak}}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \top$, V sets the status to 'open'. Otherwise V does:

1. 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 $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ to $\mathcal{F}_{\mathsf{an.BC}}$. Upon receiving $(\mathsf{sid}, \textsc{Cast}, v, \sigma)$ from $\mathcal{F}_{\mathsf{an.BC}}$, V^* stores the tuple (v, σ) to $L_{\mathsf{cast}}^{V^*}$.
2. Else, V returns $(\mathsf{sid}, \textsc{Cast}, v, \sigma, \perp)$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \textsc{Open}, v^*)$ from $\mathcal{Z}$, if there is a tuple $(v^*, \sigma^*) \in L_{\mathsf{cast}}^{V}$, V sends $(\mathsf{sid}, \textsc{Dec}, v^*, t_{\mathsf{open}})$ to $\mathcal{F}_{\mathsf{TLE}}$.

1. Upon receiving $(\mathsf{sid}, \textsc{Dec}, v^*, t_{\mathsf{open}}, o^*)$ from $\mathcal{F}_{\mathsf{TLE}}$, V returns the message $(\mathsf{sid}, \textsc{Open}, v^*, o^*)$ to $\mathcal{Z}$.
2. Upon receiving $(\mathsf{sid}, \textsc{Dec}, v^*, t_{\mathsf{open}}, \perp)$ from $\mathcal{F}_{\mathsf{TLE}}$, V returns the message $(\mathsf{sid}, \textsc{Open}, v^*, \perp)$ to $\mathcal{Z}$.

Fig. 13. Definition of Π_{vm} in the $\{\mathcal{F}_{\mathsf{TLE}}^{\mathsf{leak}}, \mathcal{F}_{\mathsf{an.BC}}, \{\mathcal{F}_{\mathsf{cert}}^{P}\}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model in the presence of $\mathbf{V}$ and SA.

Then $\mathcal{S}$ asks $\mathcal{A}$ if it allows the cast of the message $(t_{\mathsf{cast}}, t_{\mathsf{open}}, \mathbf{V}_{\mathsf{elig}}, \sigma)$ as if it was from $\mathcal{F}_{\mathsf{an.BC}}$.

Then, $\mathcal{S}$ responds to $\mathcal{F}_{\mathsf{vm}}$ according to the answer of $\mathcal{A}$.

Next, if $\mathcal{S}$ receives $(\mathsf{sid}, \textsc{Start})$ from $\mathcal{F}_{\mathsf{vm}}$, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{TLE}}$. Upon receiving the time-lock algorithms (sid, Enc, Dec, $e_{\mathcal{A}}$, $d_{\mathcal{A}}$) from $\mathcal{A}$, $\mathcal{S}$ returns the message (sid, Setup_Info, $\texttt{GenBallot} = e_{\mathcal{A}}$, $\texttt{OpenBallot} = d_{\mathcal{A}}$) to $\mathcal{F}_{\mathsf{vm}}$.

Upon receiving a Gen_Ballot request from $\mathcal{F}_{\mathsf{vm}}$ on behalf of a corrupted party V_i, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{TLE}}$ and it returns the response of $\mathcal{A}$ to $\mathcal{F}_{\mathsf{vm}}$. Upon receiving $(\mathsf{sid}, \textsc{Allow\_Cast}, v, \sigma)$ from $\mathcal{F}_{\mathsf{vm}}$, $\mathcal{S}$ asks $\mathcal{A}$ if it allows the broadcast of (v, σ) as if it was from $\mathcal{F}_{\mathsf{an.BC}}$. If the broadcast is allowed, $\mathcal{S}$ sends $(\mathsf{sid}, \textsc{Cast\_Allowed})$ to $\mathcal{F}_{\mathsf{vm}}$. When $\mathcal{S}$ receives a Cast request from $\mathcal{F}_{\mathsf{vm}}$ on behalf of a corrupted party V_i, it forwards the message to $\mathcal{A}$ as if it was from $\mathcal{F}_{\mathsf{an.BC}}$ and it returns the message it received from $\mathcal{A}$ back to $\mathcal{F}_{\mathsf{vm}}$.

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

Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{Z}$, $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was from $\mathcal{Z}$. Upon receiving $(\mathsf{sid}, \textsc{Leakage})$ from $\mathcal{A}$, $\mathcal{S}$ reads the time Cl from $\mathcal{G}_{\mathsf{clock}}$. Then $\mathcal{S}$ playing the role of $\mathcal{F}_{\mathsf{TLE}}$ returns to $\mathcal{A}$ all the maliciously generated cipher texts with time labeling until time $\mathsf{leak}(\mathsf{Cl})$. If $\texttt{Status}_{\mathsf{leak}}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Open}) = \texttt{Status}_{\mathsf{leak}}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \texttt{Status}_{\mathsf{leak}}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cred}) = \perp$, then $\mathcal{S}$ can request and give also the honest generated cipher texts from $\mathcal{F}_{\mathsf{vm}}$ and returns them to $\mathcal{A}$ as if it was from from $\mathcal{F}_{\mathsf{TLE}}$. Note that all honest parties will use $\mathcal{F}_{\mathsf{TLE}}$ with the same time labeling for encryption requests and at most once. The only way simulation fails is when the $\texttt{Status}_{\mathsf{leak}}(\mathsf{Cl}, \boldsymbol{t}, \mathsf{Cast}) = \top$ and $\texttt{Status}_{\mathsf{leak}}(\mathsf{leak}(\mathsf{Cl}), \boldsymbol{t}, \mathsf{Open}) = \top$ because in that case $\mathcal{S}$ can not retrieve the honestly generated plain texts from $\mathcal{F}_{\mathsf{vm}}$ so it can't give a response on behalf of $\mathcal{F}_{\mathsf{TLE}}$ to $\mathcal{A}$ on a $(\mathsf{sid}, \textsc{Leakage})$ command. By the definition of $\texttt{Status}_{\mathsf{leak}}$ and $\texttt{define\_time}_{\mathsf{leak}}$ this is impossible to happen.

The distribution of messages is exactly the same in both the ideal and the hybrid setting, as the algorithms $\texttt{GenBallot} \equiv e_{\mathcal{A}}$ and $\texttt{OpenBallot} \equiv d_{\mathcal{A}}$ that are used are the same. As a result the simulation is perfect. □

5 Realizing $\mathcal{F}_{\mathsf{elig}}$ via accumulators

This section describes the primitives needed to build a real protocol that realizes the eligibility functionality. Since cryptographic accumulators do not have a suitable UC treatment in the literature that would fit our purpose , Subsections 5.1 and 5.2 provide an ideal accumulator functionality and link it to a standard definition. Note that recently, [3] provided a UC functionality for accumulators, however it is not suitable for our protocol since it requires the accumulation operation to be managed centrally by some authority. Subsection 5.3 presents the Π_{elig} protocol.

5.1 Definition of $\mathcal{F}_{\mathsf{acc}}$

The purpose of secure accumulators is to provide an object representing a set and create witnesses for specific items being in the set. Our starting point for

this functionality is the property-based definition by [7] provided below. In this model, it is assumed that a trusted accumulator manager runs $\texttt{Gen}$ to generate both the public parameters and the secret trapdoor. Note that the deletion command in the interface of $\mathcal{F}_{\mathsf{acc}}$ is optional and is included for completeness. Specifically, in the context of E-CCLESIA, deletion is totally omitted making the accumulator manager to participate only in the $\texttt{Setup}$ phase.

Definition 4 (Accumulator). $\mathsf{AC} = (\mathcal{X}_\lambda, \texttt{Gen})$ *is an accumulator scheme for a family of inputs* $\{\mathcal{X}_\lambda\}$ *if it has the following properties:*

1. ***Efficient generation***
   $\texttt{Gen}$ *is an efficient probabilistic algorithm such that* $\texttt{Gen}(1^\lambda) \to (f, aux_f)$ *for random* $f \in F_\lambda$*, where* F_λ *is a family of functions.*
2. ***Efficient evaluation***
   $f \in F_\lambda$ *is a polynomial-size circuit such that* $f(w, x) \to a$ *for* $(w, x) \in \mathcal{U}_f \times \mathcal{X}_\lambda$ *and* $a \in \mathcal{U}_f$ *where* $\mathcal{U}_f$ *is an efficiently samplable domain for* f*.*
3. ***Correctness***
   $w \in \mathcal{U}_f$ *is a* witness *for* $x \in \mathcal{X}_\lambda$ *in the accumulator* $a \in \mathcal{U}_f$ *under* f *if* $\texttt{Verify}_f(w, x, a) \to \top$ *where* $\texttt{Verify}_f(w, x, a) := f(w, x) \stackrel{?}{=} a$*.*
4. ***Quasi-commutativity***
   For all $f \in F_\lambda$*,* $u \in \mathcal{U}_f$*,* $x_1, x_2 \in \mathcal{X}_\lambda$*:* $f(f(u, x_1), x_2) = f(f(u, x_2), x_1)$*. So for* $X = \{x_1, \ldots, x_m\} \subset \mathcal{X}_\lambda$*,* $f(u, X) := f(f(\ldots(u, x_1), \ldots), x_m)$*.*
5. ***Witness unforgeability***
   Let $\mathcal{U}'_f \times \mathcal{X}'_\lambda$ *denote the domains for which the computational procedure for function* $f \in F_\lambda$ *is defined (so* $\mathcal{U}_f \subseteq \mathcal{U}'_f, \mathcal{X}_\lambda \subseteq \mathcal{X}'_\lambda$*).*
   AC *is* secure *if for all PPT adversaries* $\mathcal{A}_\lambda$*:*
   $$Pr[f \leftarrow \texttt{Gen}(1^\lambda); u \leftarrow \mathcal{U}_f; (x, w, X) \leftarrow \mathcal{A}_\lambda(f, \mathcal{U}_f, u) :$$
   $$X \subset \mathcal{X}_\lambda; w, a \in \mathcal{U}'_f; x \in \mathcal{X}'_\lambda; x \notin X;$$
   $$\texttt{Verify}_f(u, X, a) = \texttt{Verify}_f(w, x, a) = \top] = \mathsf{negl}(\lambda)\,.$$
6. ***Efficient deletion***
   AC *is* dynamic *if there exist efficient algorithms* $\texttt{Delete}$*,* $\texttt{Update}$ *such that if* $x, x' \in X$ *and* $\texttt{Verify}_f(u, X, a) = \texttt{Verify}_f(w, x, a) = \top$*, then:*
   $\texttt{Delete}(aux_f, a, x') \to a'$ *such that* $\texttt{Verify}_f(u, X \backslash \{x'\}, a') = \top$*,*
   and $\texttt{Update}(f, a, a', x, x') \to w'$ *such that* $\texttt{Verify}_f(w', x, a') = \top$*.*

Remark 1. Definition 4 is well known, but it is not the only definition of accumulators in the standard model [15]. The reason for matching our functionality to this particular definition is that it suits the public setting that we wanted to model the best. As opposed to the more common definitions which involve a central authority responding to requests for accumulation, here the accumulator is a public function which any party can use.

The definition of the ideal functionality $\mathcal{F}_{\mathsf{acc}}$ is shown in Figures 14 and 15. $\mathcal{F}_{\mathsf{acc}}$, parameterized by the input set $\mathcal{X}_\lambda$, stores a list L_{acc} of (a, X, w) entries where a is the accumulator value, X is the set of accumulated items and w is either another accumulator or the basis u, a fixed value representing the empty

Setup.

The lists L_{acc}, L_{sets} and L_{ref} are initialized as empty.

■ Upon receiving (sid, SETUP) from the manager M for the first time, it sends (sid, SETUP, $\mathcal{X}_\lambda$) to $\mathcal{S}$. Upon receiving (sid, ALGS, $(\mathcal{U}_f, u)$, f, `Verify`, `Delete`, `Update`) from $\mathcal{S}$, it sets params $= (\mathcal{X}_\lambda, \mathcal{U}_f, u)$ and sends (sid, ALGS, params, f, `Verify`, `Delete`, `Update`) to M.

■ Upon receiving (sid, PARAMS) from any party P, it returns (sid, PARAMS, params, f, `Verify`, `Update`) to P.

Accumulation.

■ Upon receiving (sid, ACCUM, w, X) from party P:

1. If $X \not\subseteq \mathcal{X}'_\lambda$ or $w \notin \mathcal{U}'_f$, it aborts.
2. If there is $(a, X, w) \in L_{\mathsf{acc}}$ for some a, it returns (sid, ACCUM, a) to P.
3. It computes $a \leftarrow f(w, X)$.
4. If $a \notin \mathcal{U}_f$ or $\texttt{Verify}(w, X, a) \neq \top$, it aborts.
5. If $w = u$, it creates $L_a \leftarrow [X]$.
   Else if there is $(w, L_w) \in L_{\mathsf{sets}}$ for some L_w, it creates $L_a \leftarrow [W \cup X : W \in L_w]$.
   Else it adds (w, W_r) to L_{ref}, and creates $L_a \leftarrow [W_r \cup X]$.
6. If there is $(a, L'_a) \in L_{\mathsf{sets}}$ for some L'_a, it replaces the pair with $(a, L'_a \,||\, L_a)$ (i.e. it appends L_a to L'_a) unless $L'_a = L_a$.
   Else if for any $A \in L_a$, there is $(a', L'_a) \in L_{\mathsf{sets}}$ for some $a' \neq a$ and $A \in L'_a$, it aborts.
   Else if $(a, A_r) \in L_{\mathsf{ref}}$ for some A_r:
   (a) For every $(a', L'_a) \in L_{\mathsf{sets}}$, it replaces each occurrence of A_r in L'_a with items from L_a (so each $A' = A_r \cup \ldots$ can expand into multiple $E = A \cup \ldots$).
   (b) For any E, if there is $(a'', L''_a) \in L_{\mathsf{sets}}$ for some $a'' \neq a'$ such that $E \in L''_a$, it aborts.
   (c) It removes (a, A_r) from L_{ref} and it adds (a, L_a) to L_{sets}.
   Else, it adds (a, L_a) to L_{sets}.
7. It adds (a, X, w) to L_{acc} and returns (sid, ACCUM, a) to P.

Deletion.

■ Upon receiving (sid, DELETE, a, x, A) from manager M:

1. If $A \not\subseteq \mathcal{X}'_\lambda$, $a \notin \mathcal{U}_f$ or $x \notin A$, it aborts.
2. If there is no $(a, \{x\}, w) \in L_{\mathsf{acc}}$ for some w, and no $(a, L_a) \in L_{\mathsf{sets}}$ such that $A \in L_a$, it aborts.
3. It computes $a' \leftarrow \texttt{Delete}(a, x)$.
4. If $a' = u$ and $A = \{x\}$, it skips to Step 7.
5. If $\texttt{Verify}(u, A\backslash\{x\}, a') \neq \top$, it aborts.
6. It runs Step 6 of ACCUM for a' and $L_a = [A\backslash\{x\}]$.
7. It returns (sid, DELETED, a') to M.

■ Upon receiving (sid, UPDATE, a, a', x, x', A) from P:

1. If $A \not\subseteq \mathcal{X}'_\lambda$, a, $a' \notin \mathcal{U}_f$ or x, $x' \notin A$, it aborts.
2. It runs Step 2 of DELETE.
3. It computes $w' \leftarrow \texttt{Update}(a, a', x, x')$. If $\texttt{Verify}(w', \{x\}, a') \neq \top$, it aborts.
4. It runs Steps 5 to 7 of ACCUM for w' and a'.
5. It returns (sid, UPDATED, w') to P.

Fig. 14. Setup, accumulation and deletion commands of $\mathcal{F}_{\mathsf{acc}}$.

set. Since $\mathcal{F}_{\mathsf{acc}}$ is not producing the witnesses itself, to ensure correctness it needs to keep some auxiliary information. L_{sets} is a list of $(a, [S_1, S_2, \dots])$ entries for each a in L_{acc}, where $[S_1, S_2, \dots]$ is a list of all known representations of the contents of the set accumulated in a. L_{ref} is a list of (a, A_r) entries which represents references to accumulators that have not been expanded yet. We call an accumulator *expanded* if it has an entry in L_{sets}, i.e., there exists some representation of its contents. If it is not expanded, there is no such entry, but A_r may appear (syntactically) as part of some list S_i for another entry. We denote all such unexpanded references in capitals and with the $\_r$ suffix. We call a set *fully-expanded* if it does not contain references to any unexpanded accumulators.

To illustrate the use of these lists, we give an example execution for the accumulation of items x_1, x_2, x_3:

1. *Accumulate* $\{x_2, x_3\}$ *with witness* a*:* $\mathcal{F}_{\mathsf{acc}}$ computes $f(a, \{x_2, x_3\}) = b$, adds $(b, \{x_2, x_3\}, a)$ to L_{acc}, $(b, [A_r \cup \{x_2, x_3\}])$ to L_{sets} and (a, A_r) to L_{ref}.
2. *Accumulate* x_1 *with witness* u*:* $\mathcal{F}_{\mathsf{acc}}$ computes $f(u, x_1) = a$ and adds $(a, \{x_1\}, u)$ to L_{acc} and $(a, [\{x_1\}])$ to L_{sets}. Since (a, A_r) is in L_{ref}, the first entry in L_{sets} expands to $(b, [\{x_1, x_2, x_3\}])$ and (a, A_r) is removed from L_{ref}.
3. *Accumulate* $\{x_1, x_2, x_3\}$ *with witness* u*:* $\mathcal{F}_{\mathsf{acc}}$ computes $f(u, \{x_1, x_2, x_3\}) = c$. If $c = b$, it adds $(c, \{x_1, x_2, x_3\}, u)$ to L_{acc}, otherwise it aborts.

We can now describe the individual commands.

Setup. Upon initialization, $\mathcal{F}_{\mathsf{acc}}$ asks the adversary for the parameters and for the accumulation and verification algorithms. It distributes these to any party that asks. The party M that first calls the functionality is the designated accumulator manager. The `Delete` algorithm implicitly includes the trapdoor (aux_f in Definition 4).

Accumulation. In an ACCUM call, $\mathcal{F}_{\mathsf{acc}}$ first checks if the given input has an associated accumulator value in L_{acc} and returns it if that's the case. If not, it computes the value and checks that it verifies. Step 5 ensures the propagation of known sets throughout the L_{sets} list.

For example, suppose $\mathcal{F}_{\mathsf{acc}}$ is given witness w and input $\{z\}$. If $(w, [X_r \cup \{y\}, \{x, y\}]) \in L_{\mathsf{sets}}$ and $(x, X_r) \in L_{\mathsf{ref}}$, then the L_a corresponding to the newly calculated accumulator a will have to be of the form $[X_r \cup \{y, z\}, \{x, y, z\}]$. Step 6 ensures that the output is consistent and does not invalidate previous values, and that references to incomplete sets are updated correctly. Continuing the example, if $(a, [Y_r \cup \{x, z\}]) \in L_{\mathsf{sets}}$, $\mathcal{F}_{\mathsf{acc}}$ appends the newly calculated sets to the existing list, replacing the old entry with $(a, [Y_r \cup \{x, z\}, X_r \cup \{y, z\}, \{x, y, z\}])$. If there is no such entry, but there is $a' \neq a$ such that $(a', [\{x, y, z\}]) \in L_{\mathsf{sets}}$, then correctness is violated and $\mathcal{F}_{\mathsf{acc}}$ aborts. The third condition deals with the case when the computed a has a reference $(a, A_r) \in L_{\mathsf{ref}}$. The inputs w and $\{z\}$ reveal something about the contents of A_r, which lets $\mathcal{F}_{\mathsf{acc}}$ update all the entries in L_{sets} which contain A_r as long as there are no conflicts. A conflict could arise if there were $(a', [A_r \cup \{s\}])$, $(a'', [\{x, y, z, s\}]) \in L_{\mathsf{sets}}$ with $a' \neq a''$, because $A_r \cup \{s\}$ expands to $\{x, y, z, s\}$.

Deletion. (Optional) To match the model of [7], only the manager M can delete, but any party P can update their witness after a deletion. We stress that in E-

CCLESIA deletion commands never occur, so the manager is not required after the **Setup** phase.

Verification. There are two ways a forgery could occur. The adversary could try to verify an accumulator either with an item that has not been accumulated in it (e.g. the tuple $(u, \{x\}, a)$ when there is $(a, [\{y, z\}]) \in L_{\mathsf{sets}}$ such that $x \notin \{y, z\}$), or with an invalid witness (e.g. $(w, \{x\}, a)$ when there are $(w, [\{x, y\}])$, $(a, [\{x, y, z\}]) \in L_{\mathsf{sets}}$ such that $z \neq x$).

Verification.

■ Upon receiving (sid, VERIFY, w, X, a) from party P:

1. If $X \not\subseteq \mathcal{X}'_\lambda$, $a \notin \mathcal{U}_f$ or $w \notin \mathcal{U}'_f$, it aborts.
2. If $(a, X, w) \in L_{\mathsf{acc}}$, it returns (sid, VERIFIED, $\top$) to P.
3. If $\mathtt{Verify}(w, X, a) = \top$ and (a) or (b) holds, it aborts:
   (a) There is $(a, L_a) \in L_{\mathsf{sets}}$ such that for some fully-expanded $A \in L_a$ we have $X \not\subseteq A$.
   (b) There are (w, L_w), $(a, L_a) \in L_{\mathsf{sets}}$ such that for some $W \in L_w$ and $A \in L_a$ we have $A = W \cup V$ where V is a fully-expanded set with $V \not\subseteq X$.
4. It returns (sid, VERIFIED, $\mathtt{Verify}(w, X, a)$) to P.

Fig. 15. Verification commands of $\mathcal{F}_{\mathsf{acc}}$.

The accumulation protocol Π_{AC} for $\mathsf{AC} = (\mathcal{X}_\lambda, \mathtt{Gen})$.

■ Upon receiving (sid, SETUP) for the first time, M sends (sid, CRS) to $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$ to receive (sid, CRS, $(f, \mathcal{U}_f, u, aux_f)$). It sets $\mathsf{params} = (\mathcal{X}_\lambda, \mathcal{U}_f, u)$ and returns (sid, ALGS, params, f, $\mathtt{Verify}$, $\mathtt{Delete}(aux_f, \cdot, \cdot)$, $\mathtt{Update}(f, \cdot, \cdot, \cdot, \cdot)$).

■ Upon receiving (sid, PARAMS), P sends (sid, CRS) to $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$ to receive (CRS, sid, $(f, \mathcal{U}_f, u)$). It sets $\mathsf{params} = (\mathcal{X}_\lambda, \mathcal{U}_f, u)$ and returns (sid, PARAMS, params, f, $\mathtt{Verify}$, $\mathtt{Update}$).

■ Upon receiving (sid, ACCUM, w, X), P aborts if $X \not\subseteq \mathcal{X}'_\lambda$ or $w \notin \mathcal{U}'_f$. Otherwise it returns (sid, ACCUM, $f(w, X)$).

■ Upon receiving (sid, DELETE, a, x', A), M aborts if $A \not\subseteq \mathcal{X}'_\lambda$, $a \notin \mathcal{U}'_f$, $x' \notin A$ or if $\mathtt{Verify}(u, A, a) \neq \top$. Otherwise it returns (sid, DELETED, $\mathtt{Delete}(aux_f, a, x')$).

■ Upon receiving (sid, UPDATE, a, a', x, x', A), P aborts if $A \not\subseteq \mathcal{X}'_\lambda$, a, $a' \notin \mathcal{U}'_f$, x, $x' \notin A$ or if $\mathtt{Verify}(u, A, a) \neq \top$. Otherwise it returns (sid, UPDATED, $\mathtt{Update}(a, a', x, x')$).

■ Upon receiving (sid, VERIFY, w, X, a), P aborts if $X \not\subseteq \mathcal{X}'_\lambda$, $w \notin \mathcal{U}'_f$. Else it returns (sid, VERIFIED, $\mathtt{Verify}(w, X, a)$).

Fig. 16. Definition of Π_{AC} in the $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$-hybrid model.

5.2 A protocol that realizes $\mathcal{F}_{\mathsf{acc}}$

In Fig. 16, we define a real protocol Π_{AC} that uses an accumulator scheme in the CRS model. The following Theorems 4 and 5 capture the equivalence between our functionality and secure accumulator schemes, and in particular imply that $\mathcal{F}_{\mathsf{acc}}$ can be realized by a strong RSA accumulator construction [7].

The presented proofs for realizing $\mathcal{F}_{\mathsf{acc}}$ follow the general structure of the realizability proofs for the signature functionality in the 2005 version of [8]. We first prove a lemma that will be used to prove one direction of the equivalence.

Lemma 1. *If* AC *is an accumulator scheme and* $\mathcal{F}_{\mathsf{acc}}$ *is given parameters and algorithms according to* AC*, then* $\mathcal{F}_{\mathsf{acc}}$ *never aborts on a valid* ACCUM *query.*

Proof. Suppose we have a query $(\mathsf{sid}, \text{ACCUM}, w, X)$ for $w \in \mathcal{U}'_f$, $X \subset \mathcal{X}'_\lambda$. f is deterministic and $\mathcal{F}_{\mathsf{acc}}$ only adds properly calculated values to L_{acc}, so $\mathcal{F}_{\mathsf{acc}}$ either returns $a = f(w, X)$ or aborts. Aborts can only occur in Step 6, so we show that they are never satisfied:

1. Suppose there is $A \in L_a$ *and* $(a', L'_a) \in L_{\mathsf{sets}}$ *such that* $A \in L'_a$ *and* $a \neq a'$. After Step 5, A is one of three forms: $A = W \cup X$ where W is empty, or there is $(w, W) \in L_{\mathsf{ref}}$ for some w, or there is $(w, [W_1, W_2, \dots]) \in L_{\mathsf{sets}}$ with $W = W_i$ for some i. Further, we can write $W = W' \cup Y$ for some (possibly empty) fully-expanded set $Y = \{y_1, y_2, \dots\}$. Due to the way that items are added or modified within L_a, there can be at most one unexpanded witness reference in each, and in all cases X is added to each set. Since $A \in L'_a$, there must have been a sequence of queries which accumulated all elements of Y as well as X into a', so that $f(w', Y \cup X) = a'$. From the current query we know that $f(w, X) = a$ and $f(w', Y) = w$, which implies that $a = f(w, X) = f(f(w', Y), X) = f(w', Y \cup X) = a'$, clearly a contradiction.

2. Suppose $(a, W) \in L_{\mathsf{ref}}$ *and there is* $A \in L_a$, $(a', L'_a) \in L_{\mathsf{sets}}$ *such that* $W \cup B \in L'_a$ *for some* $B = \{b_1, b_2, \dots\}$ *and* $(a'', L''_a) \in L_{\mathsf{sets}}$ *such that* $A \cup B \in L''_a$ *and* $a' \neq a''$. $A \cup B \in L''_a$ implies that $f(w', Y \cup X \cup B) = a''$. The current query implies that $f(w, X)$ is the accumulator for W, and $W \cup B \in L'_a$ gives us $f(a, B) = a'$. Combining the last two we get $a' = f(f(w, X), B) = f(f(w', Y), X \cup B) = f(w', Y \cup X \cup B) = a''$, which is a contradiction.

No abort can occur, so (a, X, w) is recorded in L_{acc} and $(\mathsf{sid}, \text{ACCUM}, a)$ is returned. □

Theorem 4. *If* $\mathsf{AC} = (\mathcal{X}_\lambda, \mathtt{Gen})$ *is a secure dynamic accumulator scheme, then* Π_{AC} *UC-realizes* $\mathcal{F}_{\mathsf{acc}}$ *in the* $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$*-hybrid model.*

Proof. We prove the contrapositive, i.e. if Π_{AC} does not UC-realize $\mathcal{F}_{\mathsf{acc}}$, then AC is not a secure dynamic accumulator scheme, so at least one property is not satisfied. Let Π stand for Π_{AC} and $\mathcal{F}$ for the ideal protocol for $\mathcal{F}_{\mathsf{acc}}$. By Definition 1 we know that there exist an adversary $\mathcal{A}$ such that for every simulator $\mathcal{S}$ there exists an environment $\mathcal{Z}$ which can distinguish between $\mathcal{F}$ and Π with non-negligible probability. So, for a specific simulator $\mathcal{S}$, as we define below, we use such environment $\mathcal{Z}$ to construct an adversary $\mathcal{G}$ against witness unforgeability

because $\mathcal{Z}$ must trigger a VERIFY query during its run if it is to succeed. We consider $\mathcal{S}$ which supplies the correct algorithms from the AC scheme to $\mathcal{F}$.

We construct $\mathcal{G}$ by simulating the ideal execution with $\mathcal{F}$ and $\mathcal{S}$ for $\mathcal{Z}$. Like $\mathcal{S}$, $\mathcal{G}$ also runs a simulated $\mathcal{A}$. When $\mathcal{Z}$ activates M or P with SETUP, $\mathcal{G}$ returns algorithms as given by $\mathcal{S}$. $\mathcal{G}$ answers all $(\mathsf{sid}, \text{ACCUM}, w', X')$ calls with $(\mathsf{sid}, \text{ACCUM}, f(w', X'))$. When $\mathcal{Z}$ sends $(\mathsf{sid}, \text{VERIFY}, v, Y, a)$, $\mathcal{G}$ checks if $\mathtt{Verify}_f(v, Y, a) = \top$ and one of the following holds:

1. *There is* $(a, L_a) \in L_{\mathsf{sets}}$ *such that* $Y \not\subseteq A$ *for some fully expanded* $A \in L_a$*, so that* $x \in Y$ *but* $x \notin A$.
   Let $w = f(v, Y \backslash \{x\})$ so $f(w, x) = f(v, Y) = a = f(u, A)$.
2. *There are* (v, L_w) *and* $(a, L_a) \in L_{\mathsf{sets}}$ *such that* $A = W \cup V$ *and* $V \not\subseteq Y$ *for some* $A \in L_a$*,* $W \in L_w$ *and some fully-expanded* V*, so that* $x \in V$ *but* $x \notin Y$.
   Similarly, $w = f(v, V \backslash \{x\})$ so $f(w, x) = f(v, V) = a = f(v, Y) = f(u, Y \backslash W)$.

If it does, $\mathcal{G}$ returns the forgery (x, w, X) for $X = A$ or $X = Y \backslash W$ and halts. If not, it continues the simulation. Let E be the event that $\mathcal{Z}$ runs SETUP and a $(\mathsf{sid}, \text{VERIFY}, v, Y, a)$ query such that $\mathtt{Verify}(v, Y, a) = \top$ and a forgery conditions holds. By Lemma 1, $\mathcal{F}$ never aborts a well-formed ACCUM query and its outputs are consistent with Π. VERIFY is identical between $\mathcal{F}$ and Π unless a forgery condition is triggered. Hence the view of $\mathcal{Z}$ of the real execution is indistinguishable from the ideal one as long as E doesn't occur. We know that $\mathcal{Z}$ can distinguish with non-negligible probability, E must occur with the same probability and so $\mathcal{G}$ obtains a forgery. □

Theorem 5. *If* Π_{AC} *UC-realizes* $\mathcal{F}_{\mathsf{acc}}$ *in the* $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$*-hybrid model, then* $\mathsf{AC} = (\mathcal{X}_\lambda, \mathtt{Gen})$ *is a secure dynamic accumulator scheme.*

Proof. We show the contrapositive. Lack of efficient generation or evaluation would fail SETUP, so we only consider commutativity, witness unforgeability and the dynamic procedures. In each case, we construct $\mathcal{Z}$ that distinguishes for all $\mathcal{S}$, with an inactive $\mathcal{A}$. $\mathcal{Z}$ always first obtains algorithms on behalf of M and P.

Suppose AC does not satisfy commutativity, so there are $f \in F_\lambda$, $u \in \mathcal{U}_f$ and $x_1, x_2 \in \mathcal{X}_\lambda$ such that $f(f(u, x_1), x_2) \neq f(f(u, x_2), x_1)$. $\mathcal{Z}$ activates P with ACCUM for the following pairs of inputs and outputs: $(u, x_1) \to a$, $(u, x_2) \to a'$, $(a, x_2) \to b$ and $(a', x_1) \to b'$. In the execution with $\mathcal{F}$, $\mathcal{Z}$ has no control over what algorithms $\mathcal{S}$ supplies, but regardless of that $\mathcal{F}$ can only abort or return values such that $b = b'$. The parties executing Π use $g \leftarrow \mathtt{Gen}(1^\lambda)$ obtained from $\mathcal{F}_{\mathsf{CRS}}^{\mathtt{Gen}}$. Since $\mathtt{Gen}$ returns random functions from F_λ and we are only guaranteed the existence of one f that fails commutativity for our u, x_1, x_2, the probability of $\mathtt{Gen}$ returning f has a lower bound of $\frac{1}{|F_\lambda|}$. The size of F_λ is polynomially constrained by 1^λ, so the probability cannot be negligible. Then Π returns $b = f(a, x_2) = f(f(u, x_1), x_2) \neq f(f(u, x_2), x_1) = f(a', x_1) = b'$. If the protocol returns $b = b'$ or aborts, $\mathcal{Z}$ outputs 0, otherwise it outputs 1, succeeding at distinguishing between $\mathcal{F}$ and Π with non-negligible probability, depending on if Π uses a non-commutative accumulation function.

Suppose AC does not satisfy witness-unforgeability. $\mathcal{Z}$ runs an instance of $\mathcal{G}(f,\mathcal{U}_f,u)$ to get a forgery (x,w,X) with non-negligible probability. $\mathcal{Z}$ activates P with (sid, ACCUM, u, X) to get (sid, ACCUM, a), and then with (sid, VERIFY, w, $\{x\}$, a). In the execution with $\mathcal{F}$, there is no entry for $(a,\{x\},w)$ in L_{acc}. If $\texttt{Verify}(w,\{x\},a) = \perp$ for $\texttt{Verify}$ supplied by $\mathcal{S}$, $\mathcal{F}$ outputs $\perp$. Otherwise, it finds there is $(a,[X])$ in L_{sets} such that $x \notin X$ and aborts. Π computes $\texttt{Verify}(w,\{x\},a) = \top$ since $a = f(u,X)$ and returns $\top$. Hence if the protocol returns $\perp$ or aborts, $\mathcal{Z}$ outputs 0, otherwise it outputs 1, succeeding in distinguishing with non-negligible probability depending only on whether $\mathcal{G}$ supplies the forgery.

Lastly, if AC has algorithms $\texttt{Delete}$, $\texttt{Update}$ such that their outputs do not always verify, then as in the previous two cases we use this to construct a distinguishing environment $\mathcal{Z}$. It activates parties to accumulate the inputs for which $\texttt{Delete}$ or $\texttt{Update}$ produce invalid outputs, then activates M to delete those items and observes if the protocol aborts (as $\mathcal{F}_{\mathsf{acc}}$ will if $\texttt{Verify}$ outputs $\perp$) or returns the expected output. □

Commitment functionality $\mathcal{F}_{\mathsf{NIC}}$.

(sid, SETUP) → (sid, PARAMS, params, $\texttt{Verify}$) delayed output.
(sid, COMM, m) → (sid, COMM, c, o).
(sid, VERIFY, c, m, o) → (sid, VERIFIED, b).

Signature of knowledge functionality $\mathcal{F}_{\mathsf{SoK}}(\mathcal{F}_{\mathsf{acc}},\mathcal{F}_{\mathsf{NIC}})$.

(sid, SETUP) → (sid, ALGS, $\texttt{Sign}$, $\texttt{Verify}$).
(sid, SIGN, m, (a,S), (c,w,r)) → (sid, SIGN, m, (a,S), σ).
(sid, VERIFY, m, (a,S), σ) → (sid, VERIFIED, b).

Fig. 17. Interfaces provided by $\mathcal{F}_{\mathsf{NIC}}$ and $\mathcal{F}_{\mathsf{SoK}}(\mathcal{F}_{\mathsf{acc}},\mathcal{F}_{\mathsf{NIC}})$.

5.3 A protocol that realizes $\mathcal{F}_{\mathsf{elig}}$

To realize $\mathcal{F}_{\mathsf{elig}}$, we need two other primitives besides accumulators, both of which have UC formulations in the literature: *non-interactive commitments* [5] and (non-interactive zero-knowledge) *signatures of knowledge* [11].

We note the command interfaces that they provide in Fig. 17. $\mathcal{F}_{\mathsf{NIC}}$ allows a party to commit to a message m via a commitment c that can be verified using the opening o, and $\mathcal{F}_{\mathsf{SoK}}$ allows a party to produce a signature σ on message m if they know the opening r to some commitment c (which commits to some value S) which has been accumulated in a with witness w. $\mathcal{F}_{\mathsf{SoK}}$ can internally use any functionality that represents a language of statements on which signatures of knowledge can be made. In our case, it uses both $\mathcal{F}_{\mathsf{acc}}$ and $\mathcal{F}_{\mathsf{NIC}}$. The model of [11] is easily extended to the case of two "internal" functionalities, since we

The eligibility protocol $\Pi_{\mathsf{elig}}^{\mathcal{F}_{\mathsf{SoK}}(\mathcal{F}_{\mathsf{acc}},\mathcal{F}_{\mathsf{NIC}}),\mathcal{F}_{\mathsf{an.BC}},\{\mathcal{F}_{\mathsf{cert}}^{P}\},\mathcal{G}_{\mathsf{clock}}}$.

All parties initialize the set $C \leftarrow \emptyset$. Moreover, all parties have hard coded the predicates ($\texttt{define\_time}, \texttt{Status}$) If at any point a hybrid functionality aborts, the party returns $\perp$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \text{ELIGIBLE}, \mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}})$ from $\mathcal{Z}$, if $\mathbf{V}_{\mathsf{elig}} \subseteq \mathbf{V}$ and SA's status=init, SA runs:

1. It sets its status to 'credential'.
2. It computes $\boldsymbol{t} \leftarrow \texttt{define\_time}(t_{\mathsf{cast}}, t_{\mathsf{open}})$.
3. It sends $(\mathsf{sid}, \text{SETUP})$ to $\mathcal{F}_{\mathsf{acc}}$ to get (sid, ALGS, $\mathsf{params}_{\mathsf{acc}}$, f, $\texttt{Verify}_{\mathsf{acc}}$, $\texttt{Delete}$, $\texttt{Update}$). It parses $\mathsf{params}_{\mathsf{acc}}$ as $(\mathcal{X}_\lambda, \mathcal{U}_f, u)$. It sends $(\mathsf{sid}, \text{SETUP})$ to $\mathcal{F}_{\mathsf{NIC}}$ to get (sid, PARAMS, $\mathsf{params}_{\mathsf{NIC}}$, $\texttt{Verify}_{\mathsf{NIC}}$). It parses $\mathsf{params}_{\mathsf{NIC}}$ as $(\mathcal{M}, \mathcal{R})$. It sends $(\mathsf{sid}, \text{SETUP})$ to $\mathcal{F}_{\mathsf{SoK}}$ to get (sid, ALGS, $\texttt{Sign}$, $\texttt{Verify}_{\mathsf{SoK}}$). It saves all algorithms and parameters except $\texttt{Delete}$ in St_{gen}.
4. It sets $\mathsf{reg.par} := (\mathbf{V}_{\mathsf{elig}}, t_{\mathsf{cast}}, t_{\mathsf{open}}, \boldsymbol{t}, St_{\mathsf{gen}})$.
5. It sends $((\mathsf{SA}, \mathsf{sid}), \text{SETUP})$ to $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$, waits for confirmation and then sends $((\mathsf{SA}, \mathsf{sid}), \text{SIGN}, (\mathsf{SA}, \mathsf{reg.par}))$ back to receive $((\mathsf{SA}, \mathsf{sid}), \text{SIGNATURE}, (\mathsf{SA}, \mathsf{reg.par}), s)$. Then it sends ($\mathsf{sid}$, BROADCAST, ($\mathsf{SA}$, $\mathsf{reg.par}$, s)) to $\mathcal{F}_{\mathsf{an.BC}}$.
6. It returns $(\mathsf{sid}, \text{ELIG\_PAR}, \mathsf{reg.par})$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \text{BROADCAST}, (\mathsf{SA}, \mathsf{reg.par}, s))$ from $\mathcal{F}_{\mathsf{an.BC}}$, V sends $((\mathsf{SA}, \mathsf{sid}), \text{VERIFY}, (\mathsf{SA}, \mathsf{reg.par}), s)$ to $\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}$. If it returns 1, she stores $\mathsf{reg.par}$ and sets her status to 'credential'.

■ Upon receiving $(\mathsf{sid}, \text{GEN\_CRED})$ from $\mathcal{Z}$, if V's status=credential, she reads CI from $\mathcal{G}_{\mathsf{clock}}$. If $\texttt{Status}(\mathsf{CI}, \boldsymbol{t}, \mathsf{Cast}) = \top$, she sets her status to 'cast'. Otherwise, if $\texttt{Status}(\mathsf{CI}, \boldsymbol{t}, \mathsf{Cred}) = \top$, then V acts as follows:

1. She computes $S \xleftarrow{\$} \mathcal{M}$ and sends $(\mathsf{sid}, \text{COMM}, S)$ to $\mathcal{F}_{\mathsf{NIC}}$ to get (sid, COMM, c, r). If $c \notin \mathcal{X}_\lambda$, it repeats this step until it does.
2. She sends $((V, \mathsf{sid}), \text{SETUP})$ to $\mathcal{F}_{\mathsf{cert}}^{V}$, waits for confirmation and then sends $((V, \mathsf{sid}), \text{SIGN}, (V, c))$ back to receive $((V, \mathsf{sid}), \text{SIGNATURE}, (V, c), s)$. Then it sends $(\mathsf{sid}, \text{BROADCAST}, (V, (c, s)))$ to $\mathcal{F}_{\mathsf{an.BC}}$.

■ Upon receiving (sid, BROADCAST, $(V', (c', s'))$) from $\mathcal{F}_{\mathsf{an.BC}}$, V sends $((V', \mathsf{sid}), \text{VERIFY}, (V', c'), s')$ to $\mathcal{F}_{\mathsf{cert}}^{V'}$. If it returns 1, it adds c' to C and returns $(\mathsf{sid}, \text{GEN\_CRED}, V', c')$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \text{AUTH\_BALLOT}, v)$ from $\mathcal{Z}$, if V's status=cast, it reads CI from $\mathcal{G}_{\mathsf{clock}}$. If $\texttt{Status}(\mathsf{CI}, \boldsymbol{t}, \mathsf{Open}) = \top$, it sets its status to 'open'. Otherwise, if $\texttt{Status}(\mathsf{CI}, \boldsymbol{t}, \mathsf{Cast}) = \top$, then V:

1. She sends $(\mathsf{sid}, \text{ACCUM}, u, C \backslash \{c\})$ to $\mathcal{F}_{\mathsf{acc}}$ to get (sid, ACCUM, w), and (sid, ACCUM, w, $\{c\}$) to get (sid, ACCUM, a).
2. She sends $(\mathsf{sid}, \text{SIGN}, v, (a, S), (c, w, r))$ to $\mathcal{F}_{\mathsf{SoK}}$ to get (sid, SIGN, v, (a, S), ϕ). She sets $\sigma := (\phi, S)$.
3. She returns $(\mathsf{sid}, \text{AUTH\_BALLOT}, v, \sigma)$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \text{VER\_BALLOT}, v, \sigma)$ from $\mathcal{Z}$, V:

1. She parses σ as (ϕ, S) and sends $(\mathsf{sid}, \text{ACCUM}, u, C)$ to $\mathcal{F}_{\mathsf{acc}}$ to get (sid, ACCUM, a) and $(\mathsf{sid}, \text{VERIFY}, v, (a, S), \phi)$ to $\mathcal{F}_{\mathsf{SoK}}$ to get (sid, VERIFIED, x).
2. She returns $(\mathsf{sid}, \text{VER\_BALLOT}, v, \sigma, x)$ to $\mathcal{Z}$.

■ Upon receiving $(\mathsf{sid}, \text{LINK\_BALLOTS}, (v, \sigma), (v', \sigma'))$ from $\mathcal{Z}$, V:

1. She parses σ as (ϕ, S) and σ' as (ϕ', S'). If $S = S'$, she sets $x = 1$, otherwise $x = 0$.
2. She returns $(\mathsf{sid}, \text{LINK\_BALLOTS}, (v, \sigma), (v', \sigma'), x)$ to $\mathcal{Z}$.

Fig. 18. Definition of Π_{elig} in the $\{\mathcal{F}_{\mathsf{SoK}}(\mathcal{F}_{\mathsf{acc}}, \mathcal{F}_{\mathsf{NIC}}), \mathcal{F}_{\mathsf{an.BC}}, \{\mathcal{F}_{\mathsf{cert}}^{P}\}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model in the presence of $\mathbf{V}$ and SA.

can define the language that our $\mathcal{F}_{\mathsf{SoK}}$ accepts as $\mathcal{L} = \{(a, S, c, w, r) : (w, \{c\}, a) \in \mathcal{L}_{\mathsf{acc}} \wedge (c, S, r) \in \mathcal{L}_{\mathsf{NIC}}\}$, where $\mathcal{L}_{\mathsf{acc}}$, $\mathcal{L}_{\mathsf{NIC}}$ are the languages accepted by $\mathcal{F}_{\mathsf{acc}}$ and $\mathcal{F}_{\mathsf{NIC}}$, respectively.

Fig. 18 defines an eligibility protocol that realizes $\mathcal{F}_{\mathsf{elig}}$. For ease of notation, when we say P calls $\mathcal{F}_{\mathsf{acc}}$ or $\mathcal{F}_{\mathsf{NIC}}$, we mean that the communication goes through $\mathcal{F}_{\mathsf{SoK}}$ since the functionalities are embedded within. [11] describes formally how this is achieved.

Π_{elig} also uses the functionalities for anonymous broadcast, the global clock and the certification functionality instantiated for SA and all V which we denote by $\{\mathcal{F}_{\mathsf{cert}}^{P}\}$, where $P \in \{V, \mathsf{SA}\}$. Note that when we write $\mathcal{F}_{\mathsf{cert}}^{P}$ we mean $\mathcal{F}_{\mathsf{cert}}(P, \mathbf{V} \cup \mathsf{SA})$.

Remark 2. Note that in the definitions of $\mathcal{F}_{\mathsf{elig}}$ and Π_{elig}, we implicitly assume that the commands with which the environment can activate the corrupted parties are the same as for the honest parties. However, for corrupted parties, $\mathcal{F}_{\mathsf{elig}}$ will first reach out to the simulator to request their internal state, which the simulator will provide (this part is captured explicitly), and then $\mathcal{F}_{\mathsf{elig}}$ will output as usual based on what $\mathcal{S}$ provides.

Theorem 6. *Π_{elig} UC-realizes $\mathcal{F}_{\mathsf{elig}}$ in the $\{\mathcal{F}_{\mathsf{SoK}}(\mathcal{F}_{\mathsf{acc}}, \mathcal{F}_{\mathsf{NIC}}), \mathcal{F}_{\mathsf{an.BC}}, \{\mathcal{F}_{\mathsf{cert}}^{P}\}, \mathcal{G}_{\mathsf{clock}}\}$-hybrid model.*

Proof. In a direct proof of UC-realizability, given a real adversary $\mathcal{A}$ we have to construct a simulator $\mathcal{S}$ that can make the ideal execution indistinguishable from the real one from the point of view of the environment $\mathcal{Z}$. Since we are in a hybrid model, $\mathcal{S}$ will internally simulate the real execution for $\mathcal{A}$ by playing the part of voters and authorities as well as the part of the hybrid functionalities. In case of corrupted voters, it will pass any requests from $\mathcal{A}$ onto $\mathcal{F}_{\mathsf{elig}}$.

Since both $\mathcal{F}_{\mathsf{elig}}$ and all the functionalities we are working with expect the adversary to provide algorithms and parameters during setup, $\mathcal{S}$ will have to ask $\mathcal{A}$ for the algorithms for $\mathcal{F}_{\mathsf{acc}}$, $\mathcal{F}_{\mathsf{NIC}}$ and $\mathcal{F}_{\mathsf{SoK}}$ and use them to define the algorithms that $\mathcal{F}_{\mathsf{elig}}$ expects, as shown in Fig. 19.

We do not describe the reasoning behind the hybrid functionalities' algorithms here, as we are referring to the definitions established in [5] and [11] respectively for $\mathcal{F}_{\mathsf{NIC}}$ and $\mathcal{F}_{\mathsf{SoK}}$. $\mathtt{TrapCom}$, $\mathtt{TrapOpen}$ and $\mathtt{Verify}_{\mathsf{NIC}}$ belong to $\mathcal{F}_{\mathsf{NIC}}$ and $\mathtt{SimSign}$, $\mathtt{Verify}_{\mathsf{SoK}}$ and $\mathtt{Extract}$ are from $\mathcal{F}_{\mathsf{SoK}}$, and $\mathtt{Verify}_{\mathsf{cert}}$ and $\mathtt{Sign}_{\mathsf{cert}}$ are given to all $\mathcal{F}_{\mathsf{cert}}^{P}$ (which maybe different for each party). The rest have been explained in the context of $\mathcal{F}_{\mathsf{acc}}$ in Subsection 5.1.

Then $\mathcal{S}$ continues observing the ideal execution as allowed by $\mathcal{F}_{\mathsf{elig}}$ and its task is to use this to play the part of real voters in the simulated execution with $\mathcal{A}$. Using the commitment algorithms obtained earlier by $\mathcal{S}$, $\mathcal{F}_{\mathsf{elig}}$ can generate valid credentials for all eligible voters who have been activated in the ideal execution.

On the other hand, in the ideal execution when $\mathcal{S}$ is asked by $\mathcal{F}_{\mathsf{elig}}$ if it allows the credential to be generated via public delayed output, $\mathcal{S}$ signs the credential and checks the signature with the algorithms provided by $\mathcal{A}$. Note that in the ideal execution there are not any signatures involved with each voter's credential because we assume that $\mathcal{F}_{\mathsf{elig}}$ broadcasts and authenticates the public part of

the credential at the same time. If the verification fails, $\mathcal{S}$ does not allow the generation of the credential. Similarly, when $\mathcal{S}$ is asked by $\mathcal{F}_{\mathsf{elig}}$ if it allows to send the credential via public delay output to the other parties, $\mathcal{S}$ asks $\mathcal{A}$ if it allows the broadcast as if it was from $\mathcal{F}_{\mathsf{an.BC}}$ and responds appropriately to $\mathcal{F}_{\mathsf{elig}}$.

$\underline{\texttt{GenCred}(1^\lambda, \mathsf{reg.par})}$:
1. $(\mathsf{comm}, \mathsf{info}) \leftarrow \texttt{TrapCom}(\mathsf{sid}, \mathsf{params}_{\mathsf{NIC}}, \mathsf{trapdoor})$.
2. Let $(\mathcal{M}, \mathcal{R}) \leftarrow \mathsf{params}_{\mathsf{NIC}}$ and compute $S \xleftarrow{\$} \mathcal{M}$.
3. $\mathsf{open} \leftarrow \texttt{TrapOpen}(\mathsf{sid}, S, \mathsf{info})$.
4. If $\texttt{Verify}_{\mathsf{NIC}}(\mathsf{params}_{\mathsf{NIC}}, \mathsf{comm}, S, \mathsf{open}) \neq \top$, return $\bot$.
5. Return $(\mathsf{cr}, \mathsf{rc}) := ((S, \mathsf{open}), (\mathsf{comm}))$.

$\underline{\texttt{UpState}(St_{\mathsf{gen}}, C)}$:
1. Return $St_{\mathsf{fin}} := C$.

$\underline{\texttt{AuthBallot}(v, \mathsf{cr}, \mathsf{rc}, St_{\mathsf{fin}}, \mathsf{reg.par})}$:
1. Let $(S, r) := \mathsf{cr}$, $(c, s) := \mathsf{rc}$ and $C := St_{\mathsf{fin}}$.
2. Compute $w \leftarrow f(u, C \backslash \{c\})$ and $a \leftarrow f(w, \{c\})$.
3. Compute $a' \leftarrow f(u, C)$. If $a' \neq a$, return $\bot$.
4. If $\texttt{Verify}_{\mathsf{acc}}(u, C \backslash \{c\}, w) \neq \top$, $\texttt{Verify}_{\mathsf{acc}}(w, \{c\}, a) \neq \top$ or $\texttt{Verify}_{\mathsf{acc}}(u, C, a) \neq \top$, return $\bot$.
5. Else compute $\phi \leftarrow \texttt{SimSign}(\mathcal{L}, v, (a, S))$.
6. If $\texttt{Verify}_{\mathsf{SoK}}(\mathcal{L}, v, (a, S), \phi) \neq \top$, return $\bot$.
7. Else return $\sigma := (\phi, S)$.

$\underline{\texttt{VrfyBallot}(v, \sigma, St_{\mathsf{fin}}, \mathsf{reg.par})}$:
1. Let $C := St_{\mathsf{fin}}$ and $(\phi, S) := \sigma$. Compute $a \leftarrow f(u, C)$.
2. If $\texttt{Verify}_{\mathsf{acc}}(u, C, a) \neq \top$, return $\bot$.
3. Else compute $(c, w, r) \leftarrow \texttt{Extract}(\mathcal{L}, v, (a, S), \phi)$.
4. If $\texttt{Verify}_{\mathsf{acc}}(w, \{c\}, a) \neq \top$ or $\texttt{Verify}_{\mathsf{NIC}}(\mathsf{params}_{\mathsf{NIC}}, c, S, r) \neq \top$, return $\bot$.
5. Else return $x \leftarrow \texttt{Verify}_{\mathsf{SoK}}(\mathcal{L}, v, (a, S), \phi)$.

Fig. 19. Algorithms supplied by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{elig}}$ during Setup, using algorithms supplied by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{acc}}$, $\mathcal{F}_{\mathsf{NIC}}$ and $\mathcal{F}_{\mathsf{SoK}}$.

Note that $\mathcal{S}$ has no visibility into the interaction between voters and $\mathcal{F}_{\mathsf{elig}}$ with respect to Auth_Ballot and Ver_Ballot commands, just like $\mathcal{A}$ would not in the real protocol. That is not the case for $\mathcal{Z}$, as it can activate parties at will and supplies their inputs, so indistinguishability of the outputs of those commands is ensured by the matching of the algorithms described earlier. In this way, properties guaranteed by $\mathcal{F}_{\mathsf{elig}}$ directly translate to (one or several) properties provided by the hybrid functionalities, e.g. eligibility relies on binding commitments and ballot unforgeability depends on unforgeability of signatures of knowledge.

A detailed construction of $\mathcal{S}$ is provided below.

1. Send $(\mathsf{sid}, \textsc{Corrupt}, \mathbf{V}_{\mathsf{corr}})$ to $\mathcal{F}_{\mathsf{elig}}$ where $\mathbf{V}_{\mathsf{corr}}$ is the set of voters that $\mathcal{A}$ has decided to corrupt at the beginning of the real execution.
2. Wait to receive $(\mathsf{sid}, \textsc{Setup\_Elig})$ from $\mathcal{F}_{\mathsf{elig}}$.
3. Simulate the 'init' stage of Π_{elig}:
   - Simulate setup of the hybrid functionalities $\mathcal{F}_{\mathsf{acc}}$, $\mathcal{F}_{\mathsf{NIC}}$, $\{\mathcal{F}_{\mathsf{cert}}^{\mathsf{SA}}\}$ and $\mathcal{F}_{\mathsf{SoK}}$ by requesting algorithms and parameters from $\mathcal{A}$ for each of them. Store all except the accumulator `Delete` algorithm in St_{gen}.
   - Use the received algorithms to define the new algorithms as in Fig. 19 and send $(\mathsf{sid}$, $\textsc{Setup\_Elig}$, `GenCred`, `AuthBallot`, `VrfyBallot`, `UpState`, $St_{\mathsf{gen}})$ to $\mathcal{F}_{\mathsf{elig}}$.
   - Wait to receive $(\mathsf{sid}, \textsc{Elig\_Par}, \mathsf{reg.par})$ from $\mathcal{F}_{\mathsf{elig}}$.
4. Begin receiving $(\mathsf{sid}, \textsc{Gen\_Cred}, V_i, \mathsf{rc}_i)$ from $\mathcal{F}_{\mathsf{elig}}$ for honest voters V_i. Sign and verify the credential with the algorithms provided upon request from $\mathcal{A}$ by playing the role of $\mathcal{F}_{\mathsf{cert}}^{V_i}$. If verification succeeds, then collate their public credentials in the set C.
5. Send $(\mathsf{sid}, \textsc{Gen\_Cred}, V_i', \mathsf{cr}_i', \mathsf{rc}_i')$ to $\mathcal{F}_{\mathsf{elig}}$ for each corrupted voter V_i', where rc_i' is the public credential generated by $\mathcal{A}$ and cr_i' is the private credential generated by $\mathcal{A}$ or a random unique value, if $\mathcal{A}$ has generated the public part in another way.
6. $\mathcal{S}$ has no visibility into $\textsc{Auth\_Ballot}$ queries between $\mathcal{F}_{\mathsf{elig}}$ and the dummy honest voters.
7. Simulate the 'cast' stage of Π_{elig}:
   - Respond to $\mathcal{A}$'s requests to hybrid functionalities involved in authentication.
   - Verification and linking of ballots is performed by the voters themselves, so $\mathcal{S}$ can again only respond to $\mathcal{A}$'s requests to hybrid functionalities. The answers should be consistent with those of $\mathcal{F}_{\mathsf{elig}}$, since they are computed using the same algorithms.
8. $\mathcal{S}$ can provide the authentication receipts on behalf of corrupted voters by sending $(\mathsf{sid}$, $\textsc{Auth\_Ballot}$, V_i', v_i', $\sigma_i')$ to $\mathcal{F}_{\mathsf{elig}}$, using the outputs given by $\mathcal{A}$ in the simulation.

Hence the ideal and the real distribution will be the same.

□

6 Conclusion

In this work we have proposed a formal treatment in the UC framework for self tallying elections. We presented it in a modular way such that future designs of self tallying protocols can be adapted and analysed easier. That is, the substitution of one crypto primitive does not lead to reproof the whole security of our protocol, instead it is only nessecary to prove the realization of the functionality that this primitive is related to (e.g., if we replace the time-lock algorithm of Rivest *et al.* [30] with the one of [25] only the realization of $\mathcal{F}_{\mathsf{TLE}}$ will be affected, and not of $\mathcal{F}_{\mathsf{STE}}$). Moreover, we have presented E-CCLESIA, a (family

of) e-voting scheme(s) that realizes our ideal functionality $\mathcal{F}_{\mathsf{STE}}$. E-CCLESIA is the first self tallying scheme that solves the fairness problem without the presence of a trusted third party. We hope our work will revive the interest for self tallying elections from researchers in the field of e-voting, in the context where decentralized solutions are necessary (e.g blockchain governance).

In order to define $\mathcal{F}_{\mathsf{STE}}$ and prove the security of E-CCLESIA in a modular way, we had to define from scratch functionalities such as $\mathcal{F}_{\mathsf{acc}}$ and $\mathcal{F}_{\mathsf{TLE}}$. Specifically, with $\mathcal{F}_{\mathsf{TLE}}$ functionality we solved the fairness problem that every previous self tallying election protocol was vulnerable. These functionalities are of independent interest. Moreover, in order to realize our $\mathcal{F}_{\mathsf{TLE}}$ we extend Rivest *et al.* [30] time-lock construction so that the realization is feasible.

Despite all these there is room for future work. For example, there should be an implementation of our protocol in order to see how efficient and scalable it is in real world numbers. Another thing is the usability of our protocol. The interaction of a voter with the protocol should be as simple as possible while keeping the voter's participation time as low as it can be (ideally the voter should vote and then go). There is also room for improvements in the privacy aspect of E-CCLESIA. At the moment E-CCLESIA does not provide protection against a coercer (either active or passive). There are ideas in the literature on how we can make a protocol to satisfy that property but for the moment we do not know if these solutions are compatible with our setting. Last, we could explore alternative realizations of our $\mathcal{F}_{\mathsf{TLE}}$ functionality. For instance, by using a time-lock protocol based on [25,2] which in turn is based on Bitcoin. One of the advantages of such a construction would be that the computational effort in order to solve a puzzle and decrypt a message is not part of the client (in our case of a voter). Instead, the miners in Bitcoin network are responsible for the solution of the puzzle and thus the decryption of the message. What makes this approach more interesting is the fact that the miners do not try to decrypt any message at all rather than extent the Bitcoin's blockchain.

References

1. Ben Adida. Helios: Web-based open-audit voting. In *USENIX security symposium*, pages 335–348, 2008.
2. Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, pages 324–356, 2017.
3. Foteini Baldimtsi, Ran Canetti, and Sophia Yakoubov. Universally composable accumulators. *IACR Cryptology ePrint Archive*, 2018:1241, 2018.
4. Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach. Hack-a-vote: Security issues with electronic voting systems. *IEEE Security & Privacy*, 2(1):32–37, 2004.
5. Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. UC commitments for modular protocol design and applications to revocation and attribute tokens. In *CRYPTO*, 2016.
6. Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Uc-secure non-interactive public-key encryption. In *CSF 2017*, pages 217–233. IEEE Computer Society, 2017.

7. Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, pages 61–76, 2002.
8. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
9. Ran Canetti. Universally composable signatures, certification and authentication. *IACR Cryptology ePrint Archive*, 2003:239, 2003.
10. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*, pages 61–85, 2007.
11. Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *CRYPTO*, pages 78–96, 2006.
12. David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.
13. Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. Sok: Verifiability notions for e-voting protocols. In *IEEE S&P*, 2016.
14. David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *CT-RSA*, 2015.
15. David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *CT-RSA*, 2015.
16. Saghar Estehghari and Yvo Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking helios 2.0 as an example. In *EVT/WOTE*, 2010.
17. Nelly Fazio and Antonio Nicolosi. Cryptographic accumulators: Definitions, constructions and applications. *Paper written for course at New York University: www. cs. nyu. edu/nicolosi/papers/accumulators. pdf*, 2002.
18. Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *STOC*, 2013.
19. Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasic. An efficient dynamic and distributed cryptographic accumulator. In *ISC*, 2002.
20. Jens Groth. Efficient maximal privacy in boardroom voting and anonymous broadcast. In *FC*, 2004.
21. Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Proc. of the ACM workshop on privacy in the electronic society*, pages 61–70, 2005.
22. Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In *PKC*, 2002.
23. Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. End-to-end verifiable elections in the standard model. In *EUROCRYPT*, 2015.
24. Czesław Kościelny, Mirosław Kurkowski, and Marian Srebrny. *Foundations of Symmetric Cryptography*, pages 77–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
25. Jia Liu, Tibor Jager, Saqib A. Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, Feb 2018.
26. Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE S&P*, 2013.
27. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf.
28. Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *CRYPTO*, 2002.

29. Tatsuaki Okamoto. Receipt-free electronic voting schemes for large scale elections. In *Security Protocols*, 1998.
30. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.
31. Peter Y. A. Ryan and Steve A. Schneider. Prêt-à-voter with re-encryption mixes. In *ESORICS*, 2006.
32. Alan Szepieniec and Bart Preneel. New techniques for electronic voting. Washington, D.C., 2015. USENIX Association.
33. Edward Tremel. Real-world performance of cryptographic accumulators. *Undergraduate Honors Thesis, Brown University*, 2013.
34. Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. Attacking the washington, D.C. internet voting system. In *FC*, 2012.
35. Y. Zhang, J. Katz, and C. Papamanthou. An expressive (zero-knowledge) set accumulator. In *Euro S&P*, 2017.