Verifiability of E-cclesia

Jingxin Qiao

Master of Science

Cyber Security, Privacy and Trust

School of Informatics

University of Edinburgh

2022

Abstract

The main goal of this project is to prove verifiability properties of E-cclesia, a self-tally scheme (STE) electronic voting (e-voting) protocol. We develop a verification procedure for E-cclesia. To formally capture the four verifiability properties (individual verifiability, universal verifiability, eligibility verifiability and end-to-end verifiability), we exploit universal composition (UC) framework and cast those properties into the STE functionality. Taking advantage of the modular design in the UC framework, we construct a hybrid protocol that formally proved to be verifiable with existing ideal functionalities.

Acknowledgements

First and foremost, I would like to give my sincere gratitude to my supervisor, Myrto Arapinis, for her guidance and patience. Your unfaltering enthusiasm to help, and respond to my incessant questioning during our weekly virtual meetings, is an attitude I could not have completed this project without.

Next, to my parents who sponsor and support my entire MSc study. Their continue investment on my education made me who I am today.

Finally, to my friends. Thank you all for putting up with me and company me through the project submission.

Table of Contents

Chapter 1

Introduction

Electronic voting (e-voting) is fully or partially performing voting in a digital way. In contrast to traditional voting systems, it is faster, cheaper, more reliable, and more mobile. As a voting system, we expect it to be (a) fair: no one is able to learn or partially learn the election result in advance, (b) privacy-preserving: ballots are not traceable back to voters, (c) correct: honest voters should receive the same true result, (d) eligibility-checking: only eligible voters can cast ballots, (e) incoercible: voters cast in their free will, (f) equality: all parties have equal access to the voting process, (f) verifiable: it meets certain verifiability properties, and (g) accountable: every operation in the system is logged and monitored.

In principle, the design of e-voting protocol can be divided into three major categories: (a) centralized systems, but is often vulnerable to privacy or robustness attacks [11][22][23][21], and (b) have a small group of trusted parties, like Helios [1], and (c) no trusted party required where voters finish all tasks by themselves, including ballot collection, tally, result announcement, etc. The third one is a complete distributing trust assumption. In many cases decentralized scheme, which is having no central authority, is desirable. The public may have trust issues on tally authorization and candidates could also worry about the collusion of tally authorization with opponents for vote-rigging. Take the presidential election, for example, over one-third of Americans distrust the 2020 election result, and the defeated Kenyan presidential candidate, Raila Odinga, just declared the result null and void recently. The root of the controversial result is the trust in designated committees, which could have been avoided in decentralized settings.

The self-tally scheme (STE), as its name suggests, has no tally authorization and any

interested party can fulfill the tally duty. It is decentralized for post-casting processes. Since proposed in [13], STE has been a research focus for its born feature of decentralization. A number of voting designs based on blockchain or via broadcasting have been proposed [9][10][16][17][25]. Some challenges are in STE designs, like susceptible to abort, leaking intermediate result or introducing another trusted party to provide fairness. E-cclesia [2] is a STE protocol that successfully tackles those common obstacles. It is provable secure on a list of desirable properties of e-voting (fairness, vote privacy, correctness, eligibility). Of all those properties, E-cclesia left verifiability untouched.

Verifiability guarantees awareness of honest voters when bad things happen. It is still necessary even voters meant to compute results themselves. Honest voters could be under attack unconsciously. There is more trust required in e-voting systems: trust on the system provider, trust on the device provider, trust on information transferring, etc, and verifiability contributes to the trust. There are many types of verifiability for e-voting protocols. Individual verifiability is for each voter to check her ballot is included and counted, whereas universal verifiability focuses on the correctness of the entire tally. Eligibility verifiability is checking if the protocol is diligent to eligibility-checking. End-to-end verifiability guarantees that in cast, record, and tally, every endpoint is verifiable (cast as intended, record as cast, tally as recorded). Each of these verifiability property has been formalized and defined in a game-based format [20][14][15]. Those models or definitions provide standalone security. The other way of modeling is through the universal composability (UC) [6] framework. It preserves security under concurrent execution.

Models and proofs are cornerstones for cryptographic designs. As a variety of real-world attacks, we cannot passively design for known attacks. Many systems, even deployed [11][23], have been exposed to be vulnerable to newly-emerged attacks later on. We need a stronger guarantee that does not merely rely on the inability to find attacks for now, but on the possession of certain properties regardless of real-world situations.

The purpose of this work is to extend E-cclesia, a state-of-art STE e-voting protocol, with verification. For the purpose of continuity and achieving provable security, the presentation and proof for verification are done under the UC framework.

1.1 Contributions

The main contributions of this work are designing a verification for E-cclesia, modeling verifiability properties in the UC framework, and formally proving that E-cclesia satisfies these verifiability properties. In details, we

1. Extend E-cclesia with a verification process.
2. Model five verifiability properties in the UC framework, more specifically, by enriching the current STE functionality ($\mathcal{F}_{STE}$) to simultaneously capture universal verifiability, individual verifiability, eligibility verifiability, and end-to-end verifiability.
3. Re-build the hybrid model of E-cclesia. We construct a setup functionality ($\mathcal{F}_{VSD}$) for voting support devices (VSD) to emulate the interaction between users and VSD. $\mathcal{F}_{VSD}$ along with other existing functionalities ($\mathcal{F}_{NIC}$, $\mathcal{F}_{BC}$, $\mathcal{F}_{elig}$, $\mathcal{F}_{vm}$, $\mathcal{G}_{clock}$) is used as a subroutine in the E-cclesia hybrid protocol.
4. Formally prove that E-cclesia with verification is a UC realization of enriched $\mathcal{F}_{STE}$

1.2 Outline of the report

Chapter 2 prepares the basic knowledge for the cryptographic framework and gives a briefing on protocol E-cclesia.

Chapter 3 describes the verification procedure in an informal language, defines setup assumptions of VSD and extends $\mathcal{F}_{STE}$ to capture verifiability properties.

Chapter 4 design a hybrid model for E-cclesia and formally proves that this hybrid model is a UC realization of $\mathcal{F}_{STE}$.

Chapter 5 concludes what has been done and discusses limitations of the current solution and potential directions for improvement.

Chapter 2

Background

2.1 Universal Composability (UC) framework

One of the important parts of cryptographic protocol designs is to rigorously reason that the protocol has some desirable security properties. Moreover, those security properties should still hold in composition designs, namely the protocol runs as a sub-routine in or along other protocols. The Universal Composability (UC) framework, first proposed in [6], is a framework that formulates a methodology for expressing composable security properties in cryptographic protocols. The general idea is that it compares a real-world protocol to an ideal program that runs locally. The ideal program captures secure properties in syntax and via arguing the execution of protocol and local program is nearly the same, the same security properties apply to the protocol as well.

Real World Scenario

Interactive Turing Machines (ITM) are Turing machines with an extended mechanism that allows multiple machines to communicate internally (through "shared tapes") and each ITM has an identity. In the UC framework, the real-world protocol (Π) is treated as a set of unique ITM. Each party engages with one main machine labeled with session identity (sid) in the protocol. For m-party protocol, it is formalised to Π with m main machines. ITM in the protocol that are not main machines are called internal machines (or seen as sub-routines).

The execution of protocol contains Π and two more ITM, the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$. $\mathcal{Z}$ is the first machine to start and it provides inputs to parties and $\mathcal{A}$. $\mathcal{A}$

talks to parties and $\mathcal{Z}$. Parties in Π can output back to $\mathcal{Z}$, invoke sub-routines, and talk to $\mathcal{A}$. What they cannot do is talking to each other directly.

Ideal World Scenario

There are also three roles in the ideal world execution: an ideal functionality $\mathcal{F}$, the environment $\mathcal{Z}$, and the simulator $\mathcal{S}$. Here, parties in the protocol do not take any other operations rather than forwarding messages or commands from $\mathcal{Z}$ to $\mathcal{F}$. $\mathcal{F}$ is a trusted back-end program that mimics tasks of the protocol in a semantic way. It could emulate the ideal information leakage by talking to $\mathcal{S}$. $\mathcal{S}$ is a conceptual adversary who can talk to $\mathcal{Z}$ at any point through the whole execution lifetime.

One major difference between the real-world execution and the ideal-world execution is that though incidents could happen concurrently in reality, it is always a single-thread execution that only one machine could be active at a time in idealization. The current active machine has the duty of activating the next active machine, so the abstraction of $\mathcal{F}$ should be carefully cautious in capturing concurrent executions. Additionally, because $\mathcal{F}$ is a trusted back-end local program without talking to $\mathcal{Z}$ directly, security properties could be caught in syntax.

UC realization

Before introducing the formal definition of emulations, some premises should be stated. All ITM run in polynomial time (PPT) and steps taken by ITM are bounded by a security parameter λ.

From above, we can see that both real-world and ideal-world execution have a $\mathcal{Z}$ who plays the role of interactive distinguisher in the emulation. $\mathcal{Z}$ runs one of the executions and selects inputs to it. If $\mathcal{Z}$ is unable to tell which execution it is interacting with, we say the real world UC realizes the ideal world. The formal definition is as follows, where $EXEC^{\Pi}_{\mathcal{Z},\mathcal{A}}$ denotes the execution of Π organised by $\mathcal{Z}$ with $\mathcal{A}$ launching attacks and $EXEC^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}$ denotes the execution of $\mathcal{F}$ organised by $\mathcal{Z}$ with $\mathcal{S}$ launching attacks.

> **Definition 2.1.1 (UC realization)** *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 $EXEC^{\Pi}_{\mathcal{Z},\mathcal{A}}$ and $EXEC^{\Pi}_{\mathcal{Z},\mathcal{S}}$ are computationally indistinguishable. More formally:*
>
> $$\left|Pr[EXEC^{\Pi}_{\mathcal{Z},\mathcal{A}}(\lambda) = 1] - Pr[EXEC^{\mathcal{F}}_{\mathcal{Z},\mathcal{S}}(\lambda) = 1]\right| = negl(\lambda)$$

To successfully fool $\mathcal{Z}$, we need to construct a $\mathcal{S}$ that turns every real-world attack into

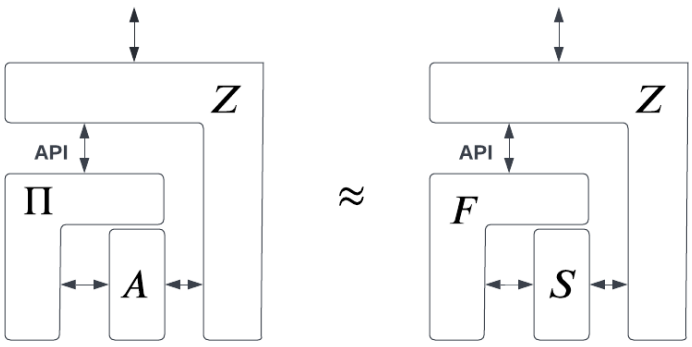

Figure 2.1: Models of UC realization

an attack on $\mathcal{F}$, which is done by the proof for UC realization.

Universal Composition

One of the most outstanding theorems in the UC framework is universal composition. As we illustrated before, Π itself is an ITM and it is composed of a set of ITM. Intuitively, as long as there is another machine that behaves like the current internal machine, the current one is substitutable. Before demonstrating the universal composition theorem, let's give a formalised definition for subroutines and identity-compatible.

Definition 2.1.2 (Subroutine protocols) *Let* ρ *be a protocol, and let* $\Phi \subset \rho$*, namely* Φ *is a subset of the machines in* ρ*. We say that* Φ *is a subroutine protocol of* ρ *if* Φ *is in itself a valid protocol.*

Definition 2.1.3 (Identity-compatible) *We say that protocol* π *is identity-compatible with protocols* ρ *and* ϕ *if no other machine in* π *has the same identity as a machine in* $\rho \backslash \phi$*.*

Theorem 1 (Universal Composition) *Let* ρ*,* ϕ*,* π *be protocols such that* ϕ *is a sub-routine of* ρ*,* π *UC realizes* ϕ*, and* π *is identity-compatible with* ρ *and* ϕ*, then protocol* $\rho^{\pi \to \phi}$ *UC realizes* ρ*.*

This theorem guarantees that security properties of one cryptographic protocol are preserved when it is a sub-routine of a larger protocol $\tilde{\Pi}$. It means a protocol Π can be replaced by a functionality $\mathcal{F}$ that it UC realizes when it is in a composition of another protocol. When comes to a complicated high-level protocol, designers don't have to construct it from the ground but make full use of existing functionalities with desirable

properties. This also simplifies the cryptographic analysis. New protocols just need to prove the UC realization of the "top layer" protocol to ideal functionality without re-doing proof for all internal modular designs. For example, a new protocol Π may be built from a set of functionalities $\{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$ to realize the target functionality $\mathcal{F}_\Pi$. Analysis of Π focuses only on $\Pi \approx \mathcal{F}_\Pi$ without considering the detailed realization of $\{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$.

2.2 Self-tally scheme

Self-tally scheme (STE) is one type of e-voting scheme where no tally authority is involved. Every voter performs a tally herself and obtains the result. It was first introduced in [13] and as the development of decentralized systems, more and more self-tally e-voting models are proposed. There are several desirable properties for an STE e-voting protocol (all properties apply for honest voters only):

1. Voter privacy: voted ballots cannot be traced back to their voters.
2. Correctness: every honest voter's ballot will be included in the tally result. The tally result of each honest voter is not necessarily the same due to adversarial actions, but at least the result of honest voters which is a subset of the tally result is the same for all of them.
3. Eligibility: only ballots of eligible voters can be included in the tally result and no one can cast beyond their legal upper bound.
4. Fairness: not any party can learn partial or all election result before tallying.
5. Verifiability: auditor can check whether the tally goes on as expected.

2.2.1 Verifiability

Verifiability though could not make a protocol more secure, ensures the awareness of verifiers when bad things happen. Many types of verifiability are proposed and in the field of e-voting, four verifiability notations are most commonly used.

First are individual verifiability and universal verifiability. They are two complementary concepts. Individual verifiability cares about if the verifier's ballot is included in the result, whereas universal verifiability is to check if the announcing result agrees with the outcome directly tallied from the bulletin board. Smyth et al. [20] formally

defined individual verifiability and universal verifiability via two game-based experiments. From an individual level, the game that two honest voters have the same ballot should have a negligible chance of winning. From a universal level, any PPT attacker is unable to get the theoretically incorrect result to pass the verification algorithm. Kremer et al. [15] combined these two verifiability definitions into one with three dimensions: (a) no clashes in ballots, (b) ballots set and the result are at one-to-one correspondence, and (c) the result should agree with voters' true intentions.

Next is eligibility verifiability. It checks if votes in the result are all from eligible voters and no eligible voters have voted beyond her legal casting times. Kremer et al. [15] defined it via credentials. He assumed two types of credentials: public credentials and ballot credentials. Each ballot set uniquely matches one public credential and ballot credential is a permutation of public credential. Eligibility verifiability means only one set of ballot credentials should be accepted by the bulletin board.

Finally is end-to-end verifiability. As its name suggests, it checks the process from one endpoint to another endpoint. More specifically, it covers three procedures: cast, record, and tally and requires cast as expected, record as cast, and tally as recorded. Kiayias et al.[14] provided a game-based definition. His definition needs an algorithm $Extr$ that assigns an option to each active voter trying to deviate from the election result or abort the election. For any PPT attacker, he should win the game with a probability less than a parameter δ, where he wins when algorithm $Extr$ exists.

2.2.2 E-cclesia

E-cclesia[2] is an STE e-voting protocol that captures five properties (Voter privacy, Correctness, Eligibility, One voter-one vote, and Fairness) in a formal way using the UC framework. Overall, it has four phases: Setup, Credential Generation, Cast, and Tally. There is no other third-party engaged in the protocol except for Setup Authority (SA) who is only active prior to voting.

In **Setup** phase, SA generates election parameters and receives a list of eligible voters. In **Credential Generation** phase, each voter generates a pair of credentials (public credentials for registration and private credentials for verification). Its underlying technology is non-interactive commitment (NIC) [5] where the private credential is a secrete selected by voters and the public credential is a commitment of that secrete. Note that NIC is verifiable, namely given a pair of credentials, it can verify whether the public

credential is truly the commitment of the private credential. If the voter is eligible, SA will register voter's identity alongside her public credential by broadcasting them. In E-cclesia, the concept of bulletin board is replaced by a broadcast channel. This is because a centralized bulletin is inappropriate in STE. A broadcast channel, exploiting the decentralization feature of blockchain is an adequate substitution.

In **Cast** phase, voters generate ballots, create a signature for the ballot and cast a bundle of ballot, secrete credential and signature. Ballots are the ciphertext of options using time-lock encryption (TLE) [3]. Only a certain time has passed can the ballot be opened, guaranteeing fairness. The utilization of signature of knowledge (SoK) [7] and the secure accumulator [19] together guard the eligibility of voters without revealing their identities. SoK allows anyone who knows a witness ω under a statement x to generate a signature σ for message m that passes the verification and the secure accumulator can add an item and produce a witness a that verifies correctly without leaking which item it is proving for. Here also casting to the bulletin board is equivalent to broadcasting to other voters.

In **Tally** phase, voters open the time-lock encrypted ballots they received during Cast phase and obtain the tally result.

Thanks to the underlying broadcast channel, E-cclesia naturally satisfies one requirement of end-to-end verifiability: record as cast. The broadcast channel will convey messages to other parties without losing any integrity, therefore what it records is what voter casts.

Chapter 3

E-cclesia with Verification

In this chapter, I will present a real-world verification protocol for E-cclesia and adjust the formalized STE functionality $\mathcal{F}_{STE}$ described in [2] accordingly. The original STE functionality $\mathcal{F}_{STE}$ has four phases, namely Setup, Credential Generation, Cast, and Tally. I extend $\mathcal{F}_{STE}$ with a fifth part: verification, which captures five notations of verifiability: **individual verifiability**, **universal verifiability**, **end-to-end verifiability**, and **eligibility verifiability**.

3.1 Informal overview of protocol $\Pi_{E-cclesia}$ with verification

Currently, E-cclesia is not possible to be end-to-end verifiable, because it ignores the existence of a malicious voting support device (VSD). VSD is used by voters for casting, and it does not always behave honestly. One of the criteria in end-to-end verifiability is cast as intended, which is specifically for verifying VSD. In order to be end-to-end verifiable, E-cclesia has to specify the trust assumption of VSD that each phase runs on. We assume two types of VSD, trust and untrusted.

1. For trusted VSD, it is believed to behave honestly, following rules without breaking any.

2. For untrusted VSD, the general conception is that they could leak some internal secrets to adversaries, take some actions based on adversaries' orders or stay uncompromised.

The formal and detailed assumptions refer to section 3.2.

E-cclesia requires a Setup phase where a setup authority (SA) runs some algorithms to generate election parameters for voting. What voters do in this phase is store parameters received from SA. We hold assumptions that every honest voter receives and stores the same parameters as what SA creates no matter of trust assumptions of VSD. It is not a very strong assumption. Election parameters are public, therefore voters can access correct parameters in the real world in many reliable ways, for example, through physical approaches.

For the rest three phases, Credential Generation, Cast, and Tally, the real-world protocol execution remains the same. However, we assume they run on untrusted VSD that have a risk of running on compromised VSD. The reason for untrusted VSD for these three phases given trusted VSD can be provided is two. On one hand, trusted VSD, like a trustworthy chip on a whole device, could be really expensive and only be used when it is necessary. On the other hand, trusted and untrusted VSD could only be two opposing conceptual devices. In the real world, they could be two independent devices both with a low but not zero risk of being compromised, and the chance that they both are compromised is negligible.

The Verification protocol executes on trusted VSD by assumption, otherwise, the result would have a risk of being tampered and we have no other ways to tell. Before digging into verification, one basic question needs to be answered first: who verifies the result coming from where. One feature of STE is that no tally authorization is required. Voters obtain the tally result themselves. Things get interesting when it comes to verification. It would be voters verify the result they computed, which is not sensible. Our solution is to make some space for verification by receiving a result from the environment. The general idea is that: honest and eligible voters will receive a result as the subject to be verified, and she runs verification to check the received result.

Though verification is a new procedure, there isn't a new phase for verification. Instead, it will share with Tally. That is because Tally is also a part of verifying when voters verify the received result with their own tally outcome. The informal description of verification in E-cclesia is:

■ Upon receiving a message (sid, VERIFY, *rês*) from $\mathcal{Z}$, each voter reads the time Cl from the global clock and checks that the Tally phase is running. If so, she runs verification on trusted VSD by executing the following steps:

1. She verifies the registered commitment of credential $\hat{cr}$ by doing followings:
   (a) She accesses her credential registration tuple $(V, \hat{cr}')$ via the trusted VSD.
   (b) She verifies NIC of $\hat{cr}'$ for cr. (cr is what she remembers by heart). If the verification fails, it means the untrusted VSD has tampered commitment of credential before broadcasting. She returns fail with a reason of dirty credential registration. Otherwise, she proceeds 2.
2. She checks whether there exists a tuple $(o^*, v^*, cr^*, \sigma^*)$ in $\hat{res}$ such that $cr^* = cr$ where cr is the secret only she knows. If such tuple does not exist, she returns fail with a reason of individual verifiability. Otherwise, she proceeds 3.
3. She finds the tuple $(o^*, v^*, cr^*, \sigma^*)$ in $\hat{res}$ such that $cr^* = cr$. If $o^* \neq o$ where o is her intended option, she returns fail with a reason of end-to-end verifiability (cast not intended). Otherwise, she proceeds 4.
4. For every tuple $(o^*, v^*, cr^*, \sigma^*)$ in $\hat{res}$, she verifies the SoK of σ^* for v^* under the statement (cr^*, St_{fin}). If the verification fails, she returns fail with a reason of eligibility. Otherwise, she proceeds 5.
5. For every tuple $(o^*, v^*, cr^*, \sigma^*)$ in $\hat{res}$, she checks if there exists another tuple $(o^{**}, v^{**}, cr^{**}, \sigma^{**})$ such that $cr^* = cr^{**}$. If such tuple exists, she returns fail with a reason of duo-voting. Otherwise, she proceeds 6.
6. She checks universal verifiability by redo tally on trusted VSD and get verification result res_{vrfy}. If $res_{vrfy} \neq \hat{res}$, she returns fail with a reason of universal verifiability. Otherwise, she returns true back to $\mathcal{Z}$.

3.2 Setup functionality for untrusted VSD $\mathcal{F}_{VSD}$

To introduce VSD into the UC framework, I idealize untrusted VSD to a functionality $\mathcal{F}_{VSD}$ which formally depicts trust assumptions. It is a setup functionality, so it is not realized by a protocol but hard-coded on hardware.

3.2.1 On compromised VSD with full abilities

The first kind of assumption is to assume untrusted VSD has full abilities. The general idea is that whatever it learns from voters, it leaks to adversaries, and all the infor-

mation or result it presents to voters is followed by the command from adversaries. It is noticeable that the untrusted VSD is not necessarily compromised, and even the compromised VSD could behave honestly abiding by adversaries' orders. The formal functionality $\mathcal{F}_{VSD}$ is as follows:

■ Upon receiving (sid, CORRUPT, $\mathbf{V}_{corr}$) from $\mathcal{S}$, if $\mathbf{V}_{corr} \subseteq \mathbf{V}$, if fixes $\mathbf{V}_{corr}$ as the set of voters who are corrupted. ■ Upon receiving (sid, COMPROMISE_VSD, $\mathbf{V}_{cVSD}$) from $\mathcal{S}$, if $\mathbf{V}_{cVSD} \subseteq \mathbf{V} \backslash \mathbf{V}_{corr}$, if fixes $\mathbf{V}_{cVSD}$ as the set of honest voters who uses compromised untrusted VSD.	
■ Upon receiving (sid, TAMPER_CRED, cr, $\hat{cr}$) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cred) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends the message (sid, TAMPER_CRED, V, cr) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CRED, V, cr', $\hat{cr}'$) from $\mathcal{S}$, it returns (sid, TAMPER_CRED, cr', $\hat{cr}'$) back to V. Otherwise, it directly returns the same message that it received from V back to V.	Compromised VSD leak the identity and voter's true credential to attackers and presents the voter with what attackers would like her to see. Uncompromised VSD leaks and tampers nothing.
■ Upon receiving (sid, TAMPER_CASTING, V, o, v, cr, σ) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cast) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends the message (sid, TAMPER_CASTING, V, o, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CASTING, V, $\cdot$, v', cr', σ') from $\mathcal{S}$, it returns (sid, TAMPER_CASTING, V, v', cr', σ') back to V. Otherwise, it directly returns original received casting bundle (sid, TAMPER_CASTING, V, v, cr, σ) back to V.	Compromised VSD leaks the real intended option of the voter to adversaries and returns a fake casting bundle (ballot, credential, and signature). Note that it currently does not know about the corresponding option of the fake casting bundle.

■ Upon receiving (sid, OPEN_TAMPERED_CASTING, v, cr, σ) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends the message (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, OPEN_TAMPERED_CASTING, V, o, v, cr, σ) from $\mathcal{S}$, it returns (sid, OPEN_TAMPERED_CASTING, o, v, cr, σ) back to V. Otherwise, it returns (sid, OPEN_TAMPERED_CASTING, $\perp$, v, cr, σ) back to V	Compromised VSD revealed the option of tampered casting bundle based on adversaries' commands when it is time to tally. If the ballot is not from compromised VSD, then it returns $\perp$ as it does not know about the answer.
■ Upon receiving (sid, TALLY, res) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends (sid, TALLY, V) to $\mathcal{S}$. Upon receiving (sid, TALLY, V, res') from $\mathcal{S}$, it sends (sid, TALLY, res') back to V. Otherwise, it directly returns the same message that it received from V back to V.	Compromised VSD blocks the voter from seeing her real tally result, rather shows her the result that adversaries would like her to see.

Table 3.1: $\mathcal{F}_{VSD}$ for full-ability compromised VSD

3.2.2 On compromised VSD with half abilities

The fact is that compromised VSD is not always so powerful to learn full information, block and tamper everything. Therefore, we formed another model for VSD to emulate the functionality of VSD with limited capability. In this assumption, it can only (a) tamper with a credential's commitment without learning the original credential and voter's identity, (b) learn the original casting bundle and change it. The formal functionality $\mathcal{F}_{VSD}$ is as follows:

■ Upon receiving (sid, CORRUPT, $\mathbf{V}_{corr}$) from $\mathcal{S}$, if $\mathbf{V}_{corr} \subseteq \mathbf{V}$, if fixes $\mathbf{V}_{corr}$ as the set of voters who are corrupted.	

■ Upon receiving (sid, COMPROMISE_VSD, $\mathbf{V}_{cVSD}$) from $\mathcal{S}$, if $\mathbf{V}_{cVSD} \subseteq \mathbf{V} \backslash \mathbf{V}_{corr}$, if fixes $\mathbf{V}_{cVSD}$ as the set of honest voters who uses compromised untrusted VSD.	
■ Upon receiving (sid, TAMPER_CRED, cr, $\hat{cr}$) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cred) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends the message (sid, TAMPER_CRED) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CRED, cr', $\hat{cr}'$) from $\mathcal{S}$, it returns the same message back to V. Otherwise, it directly returns the same message that it received from V back to V.	Compromised VSD only tampers the commitment of credential without leaking any information on the identity and the original credential.
■ Upon receiving (sid, TAMPER_CASTING, V, o, v, cr, σ) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cast) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends the message (sid, TAMPER_CASTING, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CASTING, v', cr', σ') from $\mathcal{S}$, it returns (sid, TAMPER_CASTING, V, v', cr', σ') back to V. Otherwise, it directly returns original received casting bundle (sid, TAMPER_CASTING, V, v, cr, σ) back to V.	No additional information is provided to adversaries. It casts as what adversaries want.
■ Upon receiving (sid, OPEN_TAMPERED_CASTING, v, cr, σ) from V, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\top$ and $V \in \mathbf{V}_{cVSD}$, it sends the message (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, OPEN_TAMPERED_CASTING, V, o, v, cr, σ) from $\mathcal{S}$, it returns (sid, OPEN_TAMPERED_CASTING, o, v, cr, σ) back to V. Otherwise, it returns (sid, OPEN_TAMPERED_CASTING, $\perp$, v, cr, σ) back to V	Compromised VSD revealed the option of tampered casting bundle when it is time to tally based on adversaries' command. If the ballot is not from compromised VSD, then it returns $\perp$ as it does not know about the answer.

Table 3.2: $\mathcal{F}_{VSD}$ for half-ability compromised VSD

3.3 The STE functionality $\mathcal{F}_{STE}$

To make a formal proof for E-cclesia with such newly introduced VSD, we first rewrite four phases (setup, credential generation, cast, and tally) in $\mathcal{F}_{STE}$ to add the impact of compromised VSD. Next, we demonstrate the verification phase in $\mathcal{F}_{STE}$ which captures four verifiable properties (individual verifiability, universal verifiability, eligibility verifiability, and end-to-end verifiability).

3.3.1 $\mathcal{F}_{STE}$ under full-ability compromised VSD

The functionality $\mathcal{F}_{STE}$ interacts with SA, a set of voters $\mathbf{V} = \{V_1, \ldots, V_n\}$ and a simulator $\mathcal{S}$. $\mathbf{V}_{elig}$ is the set of eligible voters. $\mathbf{V}_{corr}$ is the set of corrupted voters who are completely manipulated by $\mathcal{S}$. $\mathbf{V}_{cVSD}$ is the set of honest voters who unfortunately use corrupted VSD. $\mathbf{O}$ is the set of valid election options. t_{cast} and t_{open} are two time points that define when to cast ballot and when to tally (open ballots). The functionality initializes as empty the following lists:

• L_{cr}: Eligible and honest voters' real credentials and its corresponding credential commitments.

• L_{elig}: The registered commitments and its corresponding credentials for eligible voters. Credentials and commitments do not necessarily match, and they do not necessarily agrees with L_{cr} (voters' real credential) either, due to the potential effect of compromised VSD.

• L_{gball}: Voters' intentional options and generated ballots.

• L_{cast}: Cast ballots, credentials and ballots' corresponding signatures. For each voter, ballot voted is not necessarily ballot generated.

• L_{tally}: Local ballot tally set.

• L_{vrfy}: Local ballot set for universal verifying.

Function define_time is to derive internal trivial other time points given t_{cast} and t_{open} as input. Algorithm **GenCred** generates credential and its corresponding commitment using algorithms from the functionality of NIC $\mathcal{F}_{NIC}$ [5]. Algorithm **UpState** is to update the casting state for the functionality of secure accumulator $\mathcal{F}_{acc}$ [19]. Algorithm **AuthBallot** is to make a signature of ballot under the credential and **VrfyBallot** is to verify the signature of ballot. They both use algorithms from $\mathcal{F}_{SoK}$ [7]. Function Status returns $\top$ or $\bot$, where $\top$ means it is legal to run the given phase at this time point and $\bot$ means it is not legal.

Setup: It needs to additionally register the set of voters whose VSD is compromised.

■ Upon receiving (sid, CORRUPT, $\mathbf{V}_{corr}$) from $\mathcal{S}$, if $\mathbf{V}_{corr} \subseteq \mathbf{V}$, it fixes $\mathbf{V}_{corr}$ as corrupted voters.

■ Upon receiving (sid, COMPROMISE_VSD, $\mathbf{V}_{cVSD}$) from $\mathcal{S}$, if $\mathbf{V}_{cVSD} \subseteq \mathbf{V}\backslash\mathbf{V}_{corr}$, if fixes $\mathbf{V}_{cVSD}$ as the set of voters who uses compromised malicious VSD.

Register for voters with compromised VSD.

■ Upon receiving (sid, SETUP_INFO, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) from SA the first time, if $\mathbf{V}_{elig} \subseteq \mathbf{V}$ and $t_{cast} < t_{open}$, it forwards message (sid, SETUP_INFO, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) to $\mathcal{S}$. Upon receiving (sid, SETUP_OK, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) from $\mathcal{S}$, it sets $\overrightarrow{t} \leftarrow$ define_time(t_{cast}, t_{open}) and vote.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$).

■ Upon receiving (sid, ELIGIBLE) from SA, it informs $\mathcal{S}$, which replies with eligibility algorithms **GenCred**, **VrfyBallot**, **AuthBallot**, **UpState** and initial accumulator state St_{gen}.It sets reg.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$, St_{gen}).

Credential Generation: It generates credentials and commitments for honest voters, and presents voters with true commitments. However, for users with compromised VSD, it leaks credential information and registers another credential followed by adversaries' orders. For corrupted voters, $\mathcal{S}$ takes the full control.

■ Upon receiving (sid, GEN_CRED) from $V \in \mathbf{V}_{elig}$for the first time, it reads the time Cl from $\mathcal{G}_{clock}$.

1. If Status(Cl, $\overrightarrow{t}$, Cred) = $\top$ and $V \in \mathbf{V}\backslash\mathbf{V}_{corr}$, it does the following:

   (a) If there is no tuple (V, cr, $\hat{cr}$) in L_{cr}, it runs (cr, $\hat{cr}$) $\rightarrow$ **GenCred** (1^{λ}, reg.par). If there are tuples ($\cdot$, cr, $\cdot$) or ($\cdot$, $\cdot$, $\hat{cr}$) in L_{cr} or (cr, $\hat{cr}$) = $\perp$, it sends (sid, GEN_CRED, $\perp$) to V hand halts. Else, it adds (V, cr, $\hat{cr}$) to L_{cr}.

   (b) If tuple (V, cr, $\hat{cr}$) is successfully added to L_{cr} and $V \in \mathbf{V}_{mVSD}$, it sends (sid, TAMPER_CRED, V, cr) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CRED, V, cr', $\hat{cr}'$) from $\mathcal{S}$, it adds (V, cr', $\hat{cr}'$, 1) to L_{elig} after permission of $\mathcal{S}$ via public delay output with (V, $\hat{cr}'$) as information leakage.

Compromised VSD leaks voter's identity and real credential. It could tamper the commitment.

Else if $V \in \mathbf{V}\backslash\mathbf{V}_{mVSD}$, it directly add $(V, cr, \hat{cr}, 1)$ to L_{elig} after public delay output.

(c) It sends (sid, GEN_CRED, V, $\hat{cr}$, sender) to V and (sid, GEN_CRED, V, $\hat{cr}'$) to all other voters in $\mathbf{V}\backslash\{V\}$ and $\mathcal{S}$.

Present real commitments but broadcast tampered credentials received obtained adversaries for registration.

2. Else if Status(Cl, $\overrightarrow{t}$, Cred) = $\top$ but $V \in \mathbf{V}_{corr}$, it forwards the message (sid, GEN_CRED, V) to $\mathcal{S}$. Upon receiving (sid, GEN_CRED, V, cr, $\hat{cr}$) from $\mathcal{S}$, if there are no tuples $(V, cr^*, \hat{cr}^*, 0)$, $(\cdot, \cdot, \hat{cr}, 1)$ or $(\cdot, cr, \cdot, 1)$, it adds $(V, cr, \hat{cr}, 0)$ to L_{elig}. Then it sends (sid, GEN_CRED, V, $\hat{cr}$) to all voters in $\mathbf{V}\backslash V$ and $\mathcal{S}$.

Cast: It generates ballots and authenticates ballots for honest voters. For honest voters with compromised VSD, every casting information is leaked. Also, compromised VSD would cast what adversary would like to vote.

■ Upon receiving (sid, CAST, o) from $V \in \mathbf{V}_{elig}\backslash\mathbf{V}_{corr}$ for the first time such that (V, cr, $\hat{cr}$, 1) $\in L_{elig}$ and $o \in \mathbf{O}$, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cast) = $\top$, it does:

1. It picks $tag \xleftarrow{\$} TAG$ and it inserts the tuple $(V, \text{NULL}, o, tag, 1) \rightarrow L_{gball}$.
2. It sends (sid, GEN_BALLOT, tag, $0^{|o|}$) to $\mathcal{S}$. Upon receiving (sid, GEN_BALLOT, tag, $0^{|o|}$, v) from $\mathcal{S}$, it updates (V, NULL, o, tag, 1) in L_{gball} to (V, v, o, tag, 1).

   Ballots are always generated by adversaries. However, it won't learn anything about the real option in this step because it is replaced by dummy information ($0^{|o|}$).

3. It computes the final accumulator state $St_{fin} \leftarrow \mathbf{UpState}(St_{gen}, \hat{cr})$ for the purpose of signature generation. Then it generates the signature σ of ballot v: $\sigma \leftarrow \mathbf{AuthBallot}$ (v, cr, $\hat{cr}$, St_{fin}). If $\mathbf{VrfyBallot}$ (v, σ, reg.par) = 0, it sends (sid, AUTH_BALLOT, $\perp$) to V and halts.

4. If $V \in \mathbf{V}_{cVSD}$, it sends (sid, TAMPER_CASTING, V, o, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CASTING, V, $\cdot$, $v^{'}$, $cr^{'}$, $\sigma^{'}$) from $\mathcal{S}$, it adds (V, $v^{'}$, $\cdot$, $cr^{'}$, $\sigma^{'}$, 2) to L_{cast} after public delay output.

   Compromised VSD leaks all the information it knows to adversaries, include identity, intended option, ballots, credential and signature of ballots. The bundle it casts is fabricated by adversaries.

   Else, it directly adds (V, v, o, cr, σ, 1) to L_{cast} after public delay output.

■ Upon receiving (sid, CAST) from $V \in \mathbf{V}_{corr}$, it informs $\mathcal{S}$. Upon (sid, CAST, v, σ, V) from $\mathcal{S}$ for the first time, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cast) = $\top$ and there is a tuple (V, cr, $\hat{cr}$, 0) $\in L_{elig}$, it adds (V, v, $\cdot$, cr, σ, 0) to L_{cast}.

Tally: It performs tally operation by first filtering cast ballots by eligibility and duo-voting and then opening filtered ballots. For voters with compromised VSD, it does not present her with the real tally result but rather a result that adversaries would like her to see.

■ Upon receiving (sid, TALLY) from a voter $V \in \mathbf{V}$, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\top$, it does:

1. If $res = \phi$, it does the following:

   (a) For every (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) in L_{cast}, it runs the ballot verification algorithm $x \leftarrow$ **VrfBallot** (v^*, σ^*, St_{fin}). If $x = 1$, then $\mathcal{F}_{STE}$ performs the following security checks:

      i. If there is no tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig}, it sets res to $\perp$.

      ii. If there is a tuple (V^*, cr^*, $\hat{cr}^*$, 1) in L_{elig} and there is a tuple ($\cdot$, v^{**}, o^{**}, cr^{**}, σ^{**}, 1) in L_{cast} such that $(cr^* = cr^{**}) \wedge (v^* \neq v^{**})$, it sets res to $\perp$.

      iii. Otherwise, it adds (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) to L_{tally}.

   (b) For every tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig} such that there are multiple tuples (V, v^1, o^1, cr^1, σ^1, $\cdot$),...,(V, v^n, o^n, cr^n, σ^n, $\cdot$) in L_{tally}, it removes all multiple tuples from L_{tally} except the first one it recorded.

(c) For every tuple (V^*, v^*, $\cdot$, cr^*, σ^*, 2) in L_{tally}, it sends (sid, OPEN_TAMPERED_BALLOT, V^*, v^*, cr^*, σ^*) to $\mathcal{S}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V^*, o^*, v^*, cr^*, σ^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple to (V^*, v^*, o^*, cr^*, σ^*, 2) in L_{tally}.

Compromised VSD reveals the corresponding option of tampered casting bundle.

(d) For every tuple (V^*, v^*, $\cdot$, cr^*, σ^*, 0) in L_{tally}, it sends (sid, OPENING, V^*, v^*) to $\mathcal{S}$. Upon receiving (sid, OPENING, V^*, v^*, o^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple to (V^*, v^*, o^*, cr^*, σ^*, 0) in L_{tally}.

Corrupted voters reveals the corresponding option of voted ballot.

(e) If there are two tuples (V^*, v^*, o^*, cr^*, σ^*, 1), ($V^{*'}$, $v^{*'}$, $o^{*'}$, $cr^{*'}$, $\sigma^{*'}$, 1) such that $v^* = v^{*'} \wedge o^* \neq o^{*'}$, it sets *res* to $\perp$.

Honest voters with uncompromised VSD could also encounter a failure of opening a ballot.

(f) If $V \in \mathbf{V}_{cVSD}$, it sends (sid, TALLY, V) to $\mathcal{S}$. Upon receiving (sid, TALLY, V, res') from $\mathcal{S}$, it sets $res = res'$. Otherwise, it sets tally result *res* as the multiset $\{ (o^*, v^*, cr^*, \sigma^*) \mid (V, v^*, o^*, cr^*, \sigma^*, \cdot) \in L_{tally} \}$.

Compromised VSD shows the voter with a result from adversaries.

2. It returns (sid, TALLY, *res*) to V.

■ Upon receiving (sid,LEAKAGE) from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = Status(Cl, $\overrightarrow{t}$, Cred) = Status(Cl, $\overrightarrow{t}$, Cast) = $\perp$, it returns to $\mathcal{S}$ all the tuples (v, o, 1) such that (V^*, v, o, tag^*, 1) $\in L_{gball} \wedge$ (V^*, v, o, cr^*, σ^*, 1) $\in L_{cast}$.

Verification: The Verification is run on a trusted VSD which means there is no effect of malicious VSD. The captured four verifiability properties will be specifically pointed out in this part of $\mathcal{F}_{STE}$. As stated in the overview of verification protocol, not only a single binary result but also a reason for failing the verification are returned. The

verification returning format is (sid, VERIFY, *rês*, true/false, code of reason). Table 3.3 lists possible number for code of reason (CoR) and meanings.

Code	**Meaning**
0	Verification passes.
1	Dirty credential commitment.
2	Individual verifiability is violated.
3	End-to-End verifiablity: cast is not as intended.
4	Eligibility verifiablity: not eligible voters.
5	Eligibility verifiablity: duo-voting.
6	Universal verifiability & End-to-end verifiability: tally is not as recorded

Table 3.3: CoR and its corresponding meaning

■ Upon receiving (sid, VERIFY, *rês*) from a voter $V \in \mathbf{V} \backslash \mathbf{V}_{corr}$, it reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\top$, it does:

1. It finds the tuple (V, cr, $\hat{cr}$) in L_{cr} and (V, cr', $\hat{cr}'$, 1) in L_{elig}. If $\hat{cr} \neq \hat{cr}'$ or $cr \neq cr'$, it returns (sid, VERIFY, *rês*, 0, 1).

   It checks if the registered credential agrees with the real credential. This step lays a groundwork for individual and end-to-end verifiability.

2. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, it checks if there exists one tuple such that $cr^* = cr$ where cr is from the tuple (V, cr, $\hat{cr}$, 1) in L_{elig}. If such tuple does not exist, it returns (sid, VERIFY, *rês*, 0, 2).

   It checks if there is one ballot that is cast using voter's credential. If such ballot does not exist, it means the result set does not contain V's ballot. Individual verifiability is violated.

   Else, it finds the tuple (o^*, v^*, cr, σ^*) in *rês*. If $o^* \neq o$ where o is from the tuple (V, v, o, tag, 1) in L_{gball}, it returns (sid, VERIFY, *rês*, 0, 3).

   The option in the ballot of V does not match with her real option. It means ballot has been tampered before casting. This is one aspect of end-to-end verifiablity: cast as intended.

Note that both individual and end-to-end verifiability needs to pin point the verifier's ballot in the set using credential. It is built on the condition that the credential is not tampered which is the first step used for.

3. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, it runs the ballot verification algorithm $x \rightarrow VrfyBallot(v^*, \sigma^*, St_{fin})$. If $x = 0$, $x = \perp$ or there is no tuple ($\cdot$, cr^*, $\cdot$, $\cdot$) in L_{elig}, it returns (sid, VERIFY, *rês*, 0, 4). Otherwise,

   It verifies the signature of each ballot to check if they are all from eligible voters. This step ensures eligibility.

4. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, it checks if there exists another tuple (o^{**}, v^{**}, cr^{**}, σ^{**}) such that $cr^* = cr^{**}$. If exists, it returns (sid, VERIFY, *rês*, 0, 5).

   It checks if there are two ballots casting with the same credential. This could happen because of either double voting from corrupted voters or forgeries of honest voters.

5. It does the following to check universal verifiability and end-to-end verifiability (tally as recorded):

   (a) For every (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) in L_{cast}, it runs the ballot verification algorithm $x \leftarrow$ **VrfBallot** (v^*, σ^*, St_{fin}). If $x = 1$, then $\mathcal{F}_{STE}$ performs the following security checks:

      i. If there is no tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig}, it sets res_{vrfy} to $\perp$.

      ii. If there is a tuple (V^*, cr^*, $\hat{cr}^*$, 1) in L_{elig} and there is a tuple ($\cdot$, v^{**}, o^{**}, cr^{**}, σ^{**}, 1) in L_{cast} such that $(cr^* = cr^{**}) \wedge (v^* \neq v^{**})$, it sets res_{vrfy} to $\perp$.

      iii. Otherwise, it adds (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) to L_{vrfy}.

   (b) For every tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig} such that there are multiple tuples (V, v^1, o^1, cr^1, σ^1, $\cdot$),...,(V, v^n, o^n, cr^n, σ^n, $\cdot$) in L_{vrfy}, it removes all multiple tuples from L_{vrfy} except the first one it recorded.

   (c) For every tuple (V^*, v^*, $\cdot$, cr^*, σ^*, 2) in L_{vrfy}, it sends (sid, OPEN_TAMPERED_BALLOT, V, v^*, cr^*, σ^*) to $\mathcal{S}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, o^*, v^*, cr^*, σ^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple to (V^*, v^*, o^*, cr^*, σ^*, 2) in L_{vrfy}.

(d) For every tuple $(V^*, v^*, \cdot, cr^*, \sigma^*, 0)$ in L_{vrfy}, it sends (sid, OPENING, V^*, v^*) to $\mathcal{S}$. Upon receiving (sid, OPENING, V^*, v^*, o^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple to $(V^*, v^*, o^*, cr^*, \sigma^*, 0)$ in L_{vrfy}.

(e) It checks the correctness of ballot generated from honest voters with uncompromised VSD as followings: if there are two tuples $(V^*, v^*, o^*, cr^*, \sigma^*, 1)$, $(V^{*'}, v^{*'}, o^{*'}, cr^{*'}, \sigma^{*'}, 1)$ in L_{vrfy} such that $(v^* = v^{*'}) \wedge (o^* \neq o^{*'})$, then it sets res_{vrfy} to $\perp$. Else, it sets verify result res_{vrfy} as the multiset $\{ (o^*, v^*, cr^*, \sigma^*) \mid (V, v^*, o^*, cr^*, \sigma^*, \cdot) \in L_{vrfy} \}$.

(f) If $\hat{res} = res_{vrfy}$, it returns (sid, VERIFY, $\hat{res}$, 1, 0) to V. Else, it returns (sid, VERIFY, $\hat{res}$, 0, 6) to V.

Perform tally on L_{cast} which is the cast set recorded via the broadcast channel. Compare this result with the verification result. This step captures universal verifiability and the other aspect of end-to-end verifiability which is tally as recorded.

The verification in this functionality captures two aspects of end-to-end verifiability: cast as intended and record as tally. Plus the utilization of the broadcast channel, record-as-cast is a born feature. Three aspects of end-to-end verifiability are all satisfied, therefore end-to-end verifiable.

■ Upon receiving (sid, VERIFY, $\hat{res}$) from a voter $V \in \mathbf{V}_{corr}$, it sends (sid, VERIFY, $\hat{res}$) to $\mathcal{S}$, and replies to V whatever it receives from $\mathcal{S}$.

The verification result of a corrupted voter is out of the concern, therefore fully decided by $\mathcal{S}$.

3.3.2 $\mathcal{F}_{STE}$ under half-ability compromised VSD

From above $\mathcal{F}_{STE}$, we can see that all the security properties (correctness, eligibility, fairness, voter privacy, one voter-one vote) that E-cclesia used to possess are gone due to the existence of compromised VSD.

We argue that with the half-ability VSD assumption (formally presented in Table 3.2), the original five security properties, namely correctness, eligibility, fairness, voter pri-

vacy, and one voter-one vote, are preserved. Additionally, four verifiability properties guaranteed in the verification still holds.

To reduce redundancy, I won't repeat the unchanged part of $\mathcal{F}_{STE}$, but rather illustrate the different part that makes influences to the preservation of security properties. The complete description of $\mathcal{F}_{STE}$ under setup functionality showed in Table 3.2 is in Appendix A.1.

In Credential Generation phase, when $\mathcal{F}_{STE}$ receives a command from an eligible and honest voter, after generating a credential pair, it allows $\mathcal{S}$ to tamper credential and its corresponding commitment without revealing the identity of voter and the original credential. This means adversaries do not know about links among voter identities, credentials and credential commitments any more.

Half-ability VSD:	**Full-ability VSD:**
If tuple (V, cr, $\hat{cr}$) is successfully added to L_{cr} and $V \in \mathbf{V}_{mVSD}$, it sends (sid, TAMPER_CRED) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CRED, $cr^{'}$, $\hat{cr}^{'}$) from $\mathcal{S}$, it adds (V, $cr^{'}$, $\hat{cr}^{'}$, 1) to L_{elig} after permission of $\mathcal{S}$ via public delay output with (V, $\hat{cr}^{'}$) as information leakage.	If tuple (V, cr, $\hat{cr}$) is successfully added to L_{cr} and $V \in \mathbf{V}_{mVSD}$, it sends (sid, TAMPER_CRED, V, cr) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CRED, V, $cr^{'}$, $\hat{cr}^{'}$) from $\mathcal{S}$, it adds (V, $cr^{'}$, $\hat{cr}^{'}$, 1) to L_{elig} after permission of $\mathcal{S}$ via public delay output with (V, $\hat{cr}^{'}$) as information leakage.

In Cast phase, $\mathcal{F}_{STE}$ only sends ballots, credentials, and corresponding signatures to adversaries. That information is going to be broadcasted later anyway. Therefore, adversaries learn nothing more and blindly alter them, whereas $\mathcal{F}_{STE}$ for full-ability VSD tells everything. This used to be where fairness and vote privacy broke.

Half-ability VSD:	**Full-ability VSD:**
If $V \in \mathbf{V}_{cVSD}$, it sends (sid, TAMPER_CASTING, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CASTING, $v^{'}$, $cr^{'}$, $\sigma^{'}$) from $\mathcal{S}$, it adds (V, $v^{'}$, $\cdot$, $cr^{'}$, $\sigma^{'}$, 2) to L_{cast} after public delay output.	If $V \in \mathbf{V}_{cVSD}$, it sends (sid, TAMPER_CASTING, V, o, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CASTING, V, $\cdot$, $v^{'}$, $cr^{'}$, $\sigma^{'}$) from $\mathcal{S}$, it adds (V, $v^{'}$, $\cdot$, $cr^{'}$, $\sigma^{'}$, 2) to L_{cast} after public delay output.

In Tally phase, $\mathcal{F}_{STE}$ does not allow adversaries to present tally result. Because of pre-

vious steps ”filtering” tally set, correctness, eligibility and, one voter-one vote properties are guaranteed.

Half-ability VSD:	**Full-ability VSD:**
It sets tally result *res* as the multiset $\{ (o^*, v^*, cr^*, \sigma^*) \mid (V, v^*, o^*, cr^*, \sigma^*, \cdot) \in L_{tally} \}$.	If $V \in \mathbf{V}_{cVSD}$, it sends (sid, TALLY, V) to $\mathcal{S}$. Upon receiving (sid, TALLY, V, res') from $\mathcal{S}$, it sets $res = res'$. Otherwise, it sets tally result *res* as the multiset $\{ (o^*, v^*, cr^*, \sigma^*) \mid (V, v^*, o^*, cr^*, \sigma^*, \cdot) \in L_{tally} \}$.

Chapter 4

Design the Hybrid Protocol

In chapter 3, we have given an overview of protocol E-cclesia and formally depicted the ideal functionality $\mathcal{F}_{STE}$ that it aims to UC realize. The composition theorem of the UC framework guarantees that standalone security properties of a protocol remain the same as subroutine protocols, so in this chapter, we will construct a hybrid protocol for E-cclesia using existing ideal functionalities. Additionally, we will formally prove that the hybrid model UC realizes the $\mathcal{F}_{STE}$. Due to limited space, the hybrid protocol and its corresponding proof are both under full-ability compromised VSD. For a half-ability compromised VSD scenario, please refer to Appendix A.2.1 and A.2.2.

4.1 Primitive ideal functionalities

In line along the original design of E-cclesia, we will re-build the hybrid protocol with four ideal functionalities $\mathcal{F}_{NIC}$, $\mathcal{F}_{BC}$, $\mathcal{F}_{elig}$, $\mathcal{F}_{vm}$, $\mathcal{F}_{VSD}$ and $\mathcal{G}_{clock}$. This could help to, on one hand, specify the protocol from general to details, on the other hand, maintain security guarantees in the case underlying cryptographic primitives change.

The voting support device functionality $\mathcal{F}_{VSD}$ defined in 3.2 mimics the action of real-world devices.

The global clock functionality [4] $\mathcal{G}_{clock}$ allows any party to read synchronized time at any moment.

The non-interactive commitment functionality [5] $\mathcal{F}_{NIC}$ handles the credential commitment generation and the verification of the commitment. The related commands and messages are:

• (sid, COM_COMMIT_INI, cr): It creates a commitment $\hat{cr}$ for cr and returns $\hat{cr}$ in a form of message (sid, COM_COMMIT_END, $\hat{cr}$).
• (sid, COM_VERIFY_INI, $\hat{cr}$, cr): It verifies that if $\hat{cr}$ is a commitment of cr. It returns in a form of message (sid, COM_VERIFY_END, y) where y could be 0, 1 or $\perp$. 0 means the verification fails. 1 means the verification passes and $\perp$ is for illegal or dirty input.

The broadcast channel functionality [8] $\mathcal{F}_{BC}$ is to broadcast the received message to all parties. The related commands and messages are:

• (sid_{all}, BROADCAST, $\hat{cr}'$): It sends message (sid_{all}, BROADCAST, (V, $\hat{cr}'$)) to all parties where V indicates the source of broadcasting.

The eligibility functionality [2] $\mathcal{F}_{elig}$ is for ballot authentication, ballot verification, and ballot linking. The related commands and messages are:

• (sid, AUTH_BALLOT, v): It creates a signature for ballot and returns (sid, AUTH_BALLOT, v, σ) to eligible voters and (sid, AUTH_BALLOT, v, $\perp$) to non-eligible voters.
• (sid, VER_BALLOT, v, σ): It verifies the signature of ballot and checks if it is a valid signature signed from an eligible voter. It return the message in a form of (sid, VER_BALLOT, v, σ, x) where x could be 0, 1 or $\perp$. $\perp$ means the signature is invalid. 0 means the signature is signed by an non-eligible voter. 1 stands for passing the verification.
• (sid, LINK_BALLOTS, (v, σ), (v', σ')): It checks if the given two ballot pairs are voted using the same credential. It returns (sid, LINK_BALLOTS, (v, σ), (v', σ'), 1) for reusing and (sid, LINK_BALLOTS, (v, σ), (v', σ'), 0) for not.

The vote management functionality [2] $\mathcal{F}_{vm}$ is for ballot generation and ballot opening. The related commands and messages are:

• (sid, GEN_BALLOT, o): It allows adversaries to generate a ballot using dummy information as options. In this way, adversaries could learn nothing about voters' true options. It returns (sid, GEN_BALLOT, o, v) if it is the first time the voter request for. Otherwise, it returns (sid, GEN_BALLOT, o, $\perp$).
• (sid, OPEN, v): If the command comes from corrupted voter, it let adversaries open the ballot and returns the result in a form of (sid, OPEN, v, o). Otherwise, it checks whether the ballot is valid. If it is not, it returns (sid, OPEN, v, $\perp$). Else, it returns the voter's option that she sent before (via GEN_BALLOT command) in a form of (sid, OPEN, v, o).

4.2 Hybrid Protocol $\Pi_{E-cclesia}^{\mathcal{F}_{NIC},\mathcal{F}_{BC},\mathcal{F}_{elig},\mathcal{F}_{vm},\mathcal{F}_{VSD},\mathcal{G}_{clock}}$

Taking the advantage of composition and modularity of the UC framework, if a protocol UC realizes a functionality, the protocol as a subroutine of another protocol can be replace by the functionality. We design the hybrid protocol $\Pi_{E-cclesia}^{\mathcal{F}_{NIC},\mathcal{F}_{BC},\mathcal{F}_{elig},\mathcal{F}_{vm},\mathcal{F}_{VSD},\mathcal{G}_{clock}}$ as follows:

Setup:

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

■ Upon receiving (sid, ELIGIBLE) from $\mathcal{Z}$, if SA has received ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$) from $\mathcal{F}_{vm}$, it sends (sid, ELIGIBLE, $\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$) to $\mathcal{F}_{elig}$ which sends reg.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$, St_{gen}) to $< V_i >_{i\in[n]}$. Upon receiving reg.par from $\mathcal{F}_{elig}$, each $V \in \mathbf{V}$ stores it as the election parameters reg.par.

Credential Generation:

■ Upon receiving (sid, GEN_CRED) from $\mathcal{Z}$, for the first time, V reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cred) = $\top$, she does:

1. She picks a random cr from the message space $\mathcal{M}$ and sends (sid, COM_COMMIT_INI, cr) to $\mathcal{F}_{NIC}$. Upon receiving (sid, COM_COMMIT_END, $\hat{cr}$) from $\mathcal{F}_{NIC}$, if $\hat{cr} \notin D$, she repeats this step until it does.

2. She sends (sid, TAMPER_CRED, cr, $\hat{cr}$) to $\mathcal{F}_{VSD}$. Upon receiving (sid, TAMPER_CRED, $cr^{'}$, $\hat{cr}^{'}$) from $\mathcal{F}_{VSD}$, she stores (cr, $\hat{cr}^{'}$).

3. She sends (sid_{all}, BROADCAST, $\hat{cr}^{'}$) to $\mathcal{F}_{BC}$. Upon receiving (sid_{all}, BROAD-

CAST, (V ,$\hat{cr}^{'}$)) from $\mathcal{F}_{BC}$, she appends the pair (V ,$\hat{cr}^{'}$) to L_{cred}.

Cast:

■ Upon receiving (sid, CAST, o) from $\mathcal{Z}$, V executes the following steps:

1. She sends (sid, GEN_BALLOT, o) to F_{vm} which replies with the generated ballot as (sid, GEN_BALLOT, o, v) (or sends (sid, GEN_BALLOT, o, $\perp$) and halts).

2. She sends (sid, AUTH_BALLOT, v) to $\mathcal{F}_{elig}$ which replies with the authentication receipt for v (sid, AUTH_BALLOT, v, σ) (or sends (sid, AUTH_BALLOT, v, $\perp$) and halts).

3. She sends (sid, TAMPER_CASTING, V, o, v, cr, σ) to $\mathcal{F}_{VSD}$. Upon receiving (sid, TAMPER_CASTING, V, $v^{'}$, $cr^{'}$, $\sigma^{'}$) from $\mathcal{F}_{VSD}$, she sends (sid, CAST, $v^{'}$, $cr^{'}$, $\sigma^{'}$) to $\mathcal{F}_{vm}$ which broadcasts the message to $< V_j >_{j\in[n]}$. In turn, voters store the received bundle (v, cr, σ).

Tally

■ Upon receiving a message (sid, TALLY) from $\mathcal{Z}$, V reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\perp$, she ignores the message. Otherwise, if $res = \phi$, she executes the following steps:

1. For every tuple (sid, Cast, v, cr, σ) she has obtained from $\mathcal{F}_{vm}$, she sends (sid, VER_BALLOT, v, σ) to $\mathcal{F}_{elig}$ which replies with (sid, VER_BALLOT, v, σ, x), where $x \in \{0, 1, \perp\}$. If there is any ballot verification request such that $\mathcal{F}_{elig}$ replied with $x = \perp$, then she discards the ballot. Otherwise, she includes in her tally set all tuples (v, cr, σ) such that $\mathcal{F}_{elig}$ replied with $x = 1$.

2. She discards multiple ballots by sending (sid, LINK_BALLOTS, (v, σ), (v', σ')) to $\mathcal{F}_{elig}$ for every pair (v, σ), (v', σ') in her tally set. If she gets a (sid, LINK_BALLOTS, (v, σ), (v', σ'), 1) respond, then she discards the ballot she received the last out of those two.

3. For every tuple (v, cr, σ) in the tally set, she sends (sid, OPEN_TAMPERED_CASTING, v) to $\mathcal{F}_{VSD}$, which could reply with (sid, OPEN_TAMPERED_CASTING, v, o) or (sid, OPEN_TAMPERED_CASTING, v, $\perp$). If the opening is $\perp$, then she sends (sid, OPEN, v) to $\mathcal{F}_{vm}$ to try to open again. It replies with the opening (sid, OPEN, v, o). If at any time $\mathcal{F}_{vm}$ replies with (sid, OPEN, v, $\perp$), then she sets res to $\perp$.

4. V sends (sid, TALLY, *res*) to $\mathcal{F}_{VSD}$. Upon receiving (sid, TALLY res') from $\mathcal{F}_{VSD}$, she updates *res* to res'.

Verification:

■ Upon receiving (sid, VERIFY, *res*) from $\mathcal{Z}$, V reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\top$, she does:

1. She accesses her registered credential (V, $\hat{cr}'$) on a trusted VSD. Then she verifies if her credential has been tampered by sending (sid, COM_VERIFY_INI, $\hat{cr}'$, *cr*) to $\mathcal{F}_{NIC}$ where *cr* is what she remembers. Upon receiving (sid, COM_VERIFY_END, *y*) from $\mathcal{F}_{NIC}$, if $y = 0$ or $y = \perp$, she returns false with a reason of dirty credential registration. Otherwise, she proceeds 2.

2. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, she checks if there exists a tuple such that $cr^* = cr$ where *cr* is the credential she memorise by heart. If such tuple does not exist, she returns false with a reason of individual verifiability. Otherwise, she proceeds 3.

3. She finds the tuple (o^*, v^*, cr^*, σ^*) in *rês* such that $cr^* = cr$. If $o^* \neq o$ where o is her intended option, she returns fail with a reason of end-to-end verifiability: cast is not as intended. Otherwise, she proceeds 4.

4. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, she sends (sid, VER_BALLOT, v^*, σ^*) to $\mathcal{F}_{elig}$ which replies with (sid, VER_BALLOT, v^*, σ^*, x), where $x \in \{0, 1, \perp\}$. If there is any ballot verification request such that $\mathcal{F}_{elig}$ replied with $x = \perp$ or 0, then she returns fail with a reason of eligibility verifiability: not eligible voters. Otherwise, she proceeds 5.

5. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, she sends (sid, LINK_BALLOTS, (v^*, σ^*), ($v^{*\prime}$, $\sigma^{*\prime}$)) to $\mathcal{F}_{elig}$ for every pair (v^*, σ^*), ($v^{*\prime}$, $\sigma^{*\prime}$) in her tally set. If she gets a (sid, LINK_BALLOTS, (v^*, σ^*), ($v^{*\prime}$, $\sigma^{*\prime}$), 1) respond, she returns fail with a reason of eligibility verifiability: duo-voting. Otherwise, she proceeds 6.

6. She re-do tally on a trusted VSD to verify universal verifiability and end-to-end verifiablity (tally as recorded). Specifically,

   (a) For every tuple (sid, Cast, v, cr, σ) she has obtained from $\mathcal{F}_{vm}$ accessing via a trusted VSD, she sends (sid, VER_BALLOT, v, σ) to $\mathcal{F}_{elig}$ which replies with (sid, VER_BALLOT, v, σ, x), where $x \in \{0, 1, \perp\}$. If there is any ballot

verification request such that $\mathcal{F}_{elig}$ replied with $x = \perp$, then she discards the ballot. Otherwise, she includes in her verification set all tuples (v, cr, σ) where $\mathcal{F}_{elig}$ replied with $x = 1$.

(b) She discards multiple ballots by sending (sid, LINK_BALLOTS, (v, σ), (v', σ')) to $\mathcal{F}_{elig}$ for every pair (v, σ), ($v^{'}$, $\sigma^{'}$) in her verification set. If she gets a (sid, LINK_BALLOTS, (v, σ), ($v^{'}$, $\sigma^{'}$), 1) respond, then she discards the ballot she received the last out of those two.

(c) For every tuple (v, cr, σ) in the verification set, she sends (sid, OPEN_TAMPERED_CASTING, v) to $\mathcal{F}_{VSD}$, which could reply with (sid, OPEN_TAMPERED_CASTING, v, o) where she would add (o, v, cr, σ) to res_{vrfy} or (sid, OPEN_TAMPERED_CASTING, v, $\perp$). If the opening is $\perp$, she sends (sid, OPEN, v) to $\mathcal{F}_{vm}$ to try to open again. Upon receiving the opening (sid, OPEN, v, o) from $\mathcal{F}_{vm}$, V adds (o, v, cr, σ) to res_{vrfy}. If at any time $\mathcal{F}_{vm}$ replies with (sid, OPEN, v, $\perp$), then she sets res_{vrfy} to $\perp$.

(d) She compares res_{vrfy} with $\hat{res}$. If $res_{vrfy} \neq \hat{res}$, she returns fail with reason of universal verifiability. Otherwise, she returns true back to $\mathcal{Z}$

We also design a hybrid protocol in Appendix A.2.1 that UC realizes $\mathcal{F}_{STE}$ in Appendix A.1 and provide a formal proof in Appendix A.2.2.

4.3 Proof for UC realization

Theorem 2 *The hybrid protocol* $\Pi_{E-cclesia}^{\mathcal{F}_{NIC},\mathcal{F}_{BC},\mathcal{F}_{elig},\mathcal{F}_{vm},\mathcal{F}_{VSD},\mathcal{G}_{clock}}$ *in section 4.2 UC-realizes the* $\mathcal{F}_{STE}$ *functionality in 3.3.1.*

PROOF. A simulator $\mathcal{S}$ who executes as follows to be indistinguishable to any adversary $\mathcal{A}$ from any PPT environment.

To make the proof more concise, I will only present in details what $\mathcal{S}$ would operates when receiving messages or commands from un-corrupted parties. For messages from corrupted parties, $\mathcal{S}$ will forward whatever it receives from $\mathcal{F}_{STE}$ to $\mathcal{A}$ as if it was that party and returns whatever message it receives from $\mathcal{A}$ to $\mathcal{F}_{STE}$. Also, as for public delay output, what $\mathcal{S}$ does is merely conveying messages to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$ or $\mathcal{F}_{elig}$ and returning the permission back to $\mathcal{F}_{STE}$. The detailed execution is as follows:

Setup:

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

■ Upon receiving (sid, COMPROMISE_VSD, $\mathbf{V}_{cVSD}$), $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was $\mathcal{Z}$. Upon receiving (sid, COMPROMISE_VSD, $\mathbf{V}_{cVSD}$) from $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$, it forwards the message to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, SETUP_INFO, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) from $\mathcal{F}_{STE}$, it sets $\overrightarrow{t} \leftarrow$ define_time(t_{cast},t_{open}) and vote.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$). Then it sends (sid, SETUP_OK, vote.par) to $\mathcal{A}$ as if it was from $\mathcal{F}_{vm}$. Upon receiving the permission from $\mathcal{A}$, $\mathcal{S}$ sends (sid, SETUP_OK, V_{elig}, $\mathbf{O}$, t_{cast}, t_{open}) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, ELIGIBLE) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, SETUP_ELIG) to $\mathcal{A}$ as if it was from $\mathcal{F}_{elig}$. Upon receiving (sid, SETUP_ELIG, **GenCred**, **VrfyBallot**, **AuthBallot**, **UpState**, St_{gen}) from $\mathcal{A}$, $\mathcal{S}$ sends (sid, ELIGIBLE, **GenCred**, **VrfyBallot**, **AuthBallot**, **UpState**, St_{gen}) back to $\mathcal{F}_{STE}$.

Credential Generation:

■ Upon receiving (sid, TAMPER_CRED, V, cr) from $\mathcal{F}_{STE}$, $\mathcal{S}$ forwards the same message to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, TAMPER_CRED, V, $cr^{'}$, $\hat{cr}^{'}$) from $\mathcal{A}$, $\mathcal{S}$ sends (sid, TAMPER_CRED, V, $cr^{'}$, $\hat{cr}^{'}$) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, GEN_CRED, V, $\hat{cr}$) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, BRAODCAST, (V, $\hat{cr}$)) to $\mathcal{A}$ as if it was $\mathcal{F}_{BC}$. Upon receiving the token back from $\mathcal{A}$, $\mathcal{S}$ sends (sid, GEN_CRED, V, $\hat{cr}$) to $\mathcal{F}_{STE}$.

Cast:

■ Upon receiving (sid, GEN_BALLOT, tag, $0^{|o|}$) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, GEN_BALLOT, tag, $0^{|o|}$) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, GEN_BALLOT, tag, $0^{|o|}$, v) from $\mathcal{A}$, $\mathcal{S}$ forwards the same message (sid, GEN_BALLOT, tag, $0^{|o|}$, v) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, TAMPER_CASTING, V, o, v, cr, σ) from $\mathcal{F}_{STE}$, it forwards the message to $\mathcal{S}$ as if it was $\mathcal{F}_{VSD}$ and returns whatever it receives from $\mathcal{S}$ back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, CAST, v, cr, σ) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, ALLOW_CAST, v, σ) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, CAST_ALLOWED) from $\mathcal{A}$, $\mathcal{S}$ returns the same message back to $\mathcal{F}_{STE}$.

Tally:

■ Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ).

■ Upon receiving (sid, OPENING, V, v) from $\mathcal{F}_{STE}$ for an alternative ballot opening for a corrupted voter V, $\mathcal{S}$ sends (sid, OPEN, v) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, OPEN, v, o) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPENING, V, v, o) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, TALLY, V) from $\mathcal{F}_{STE}$, $\mathcal{S}$ forwards (sid, TALLY, V) to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, TALLY, V, res') from $\mathcal{A}$, $\mathcal{S}$ returns the same message (sid, TALLY, V, res') back to $\mathcal{F}_{STE}$.

Verification:

■ Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ).

■ Upon receiving (sid, OPENING, V, v) from $\mathcal{F}_{STE}$ for an alternative ballot opening for a corrupted voter V, $\mathcal{S}$ sends (sid, OPEN, v) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, OPEN, v, o) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPENING, V, v, o) back to $\mathcal{F}_{STE}$.

Chapter 5

Conclusions and Future Work

5.1 Summary and Discussion

The main objective of this work is to prove that E-cclesia satisfies verifiability properties. We begin by giving a high-level description of E-cclesia with verification. Next, we construct a new setup functionality $\mathcal{F}_{VSD}$ to introduce the concept of VSD into the UC framework and formally depict trust assumptions. Based on the capability of compromised VSD, we propose two different $\mathcal{F}_{VSD}$: full-ability where it leaks information out and receives adversarial commands, and half-ability where it only follows orders from adversaries. The next contribution of this work is to model five verifiability properties, which are individual verifiability, universal verifiability, eligibility verifiability, and end-to-end verifiability in the UC framework. We model them in the ideal STE functionally $\mathcal{F}_{STE}$. Then, we formally design a hybrid protocol for E-cclesia with verification using existing ideal functionalities and $\mathcal{F}_{VSD}$ we built. Finally, we prove that the hybrid protocol is a UC realization of $\mathcal{F}_{STE}$. Since $\mathcal{F}_{STE}$ captures five verifiability properties, we argue that E-cclesia holds the same properties.

However, there are limitations of the current solution:

1. Efficiency. One of the verifiability properties is universal variability. Voters need to redo the tally on received ballots set to verify it which is very computationally expensive. That is because the underlying technology that supports fairness is time-lock encryption (TLE). It allows you to decrypt ciphertext only after a certain time has passed. The high-level logic behind it is to hide the decryption key into a puzzle and that puzzle requires a lot of computation. Due to the fact

that access to computation sources is limited, time-lock is guaranteed. If voters perform the tally again for verifying, they all have to do computations and solve the puzzle.

2. Coercion. Coercion is when adversaries force voters to leak secrecy. In our solution, we do not consider the existence of a coercer, so the verification phase cannot give an alert when it happens.

5.2 Future work

Regarding previous limitations, here are some directions for future work.

The first one is solving the efficiency problem through zero-knowledge proof. Informally, zero-knowledge proof is to convince the verifier that the prover knows some information without leaking any extra (zero-knowledge). Voters receive a result as well as evidence. With the evidence, voters can verify whether this result comes from solving the puzzle without personally solving the puzzle. There have been some studies on verifiable time-lock encryption [18]. The next step could begin with the optimization of current TLE functionality. We can model a verifiable TLE so that it can provide an API for more efficient universal verifiability.

The other one is capturing coercion in verification. Some work has designed a coercion-resistant system, like [12] by generating information that is indistinguishable from adversaries but distinguishable from tally authority and [24] by employing a receiver-deniable encryption scheme. Future work could start by realizing coercion-resistance, and then achieve coercion verifiable.

Bibliography

[1] Ben Adida. Helios: Web-based open-audit voting. In *USENIX security symposium*, volume 17, pages 335–348, 2008.

[2] Myrto Arapinis, Nikolaos Lamprou, Lenka Mareková, Thomas Zacharias, Léo Ackermann, and Pavlos Georgiou. E-cclesia: Universally composable self-tallying elections. Cryptology ePrint Archive, Paper 2020/513, 2020. https://eprint.iacr.org/2020/513.

[3] Myrto Arapinis, Nikolaos Lamprou, and Thomas Zacharias. Astrolabous: A universally composable time-lock encryption scheme. Cryptology ePrint Archive, Paper 2021/1246, 2021. https://eprint.iacr.org/2021/1246.

[4] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Annual international cryptology conference*, pages 324–356. Springer, 2017.

[5] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. Uc commitments for modular protocol design and applications to revocation and attribute tokens. In *Annual International Cryptology Conference*, pages 208–239. Springer, 2016.

[6] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[7] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *Annual International Cryptology Conference*, pages 78–96. Springer, 2006.

[8] Juan A Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively secure broadcast, revisited. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 179–186, 2011.

[9] Jens Groth. Efficient maximal privacy in boardroom voting and anonymous broadcast. In *International Conference on Financial Cryptography*, pages 90–104. Springer, 2004.

[10] Gang Han, Yannan Li, Yong Yu, Kim-Kwang Raymond Choo, and Nadra Guizani. Blockchain-based self-tallying voting system with software updates in decentralized iot. *IEEE Network*, 34(4):166–172, 2020.

[11] David Jefferson, Aviel D Rubin, Barbara Simons, and David Wagner. Analyzing internet voting security. *Communications of the ACM*, 47(10):59–64, 2004.

[12] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Towards Trustworthy Elections*, pages 37–63. Springer, 2010.

[13] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In *International Workshop on Public Key Cryptography*, pages 141–158. Springer, 2002.

[14] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. End-to-end verifiable elections in the standard model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 468–498. Springer, 2015.

[15] Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In *European Symposium on Research in Computer Security*, pages 389–404. Springer, 2010.

[16] Yannan Li, Willy Susilo, Guomin Yang, Yong Yu, Dongxi Liu, Xiaojiang Du, and Mohsen Guizani. A blockchain-based self-tallying voting protocol in decentralized iot. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[17] Somnath Panja, Samiran Bag, Feng Hao, and Bimal Roy. A smart contract system for decentralized borda count voting. *IEEE Transactions on Engineering Management*, 67(4):1323–1339, 2020.

[18] Krzysztof Pietrzak. Simple verifiable delay functions. In *10th innovations in theoretical computer science conference (itcs 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[19] Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed pki. In *International Conference on Security and Cryptography for Networks*, pages 292–309. Springer, 2016.

[20] Ben Smyth, Steven Frink, and Michael R Clarkson. Computational election verifiability: Definitions and an analysis of helios and jcj. 2015.

[21] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J Alex Halderman. Security analysis of the estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 703–715, 2014.

[22] Scott Wolchok, Eric Wustrow, J Alex Halderman, Hari K Prasad, Arun Kankipati, Sai Krishna Sakhamuri, Vasavya Yagati, and Rop Gonggrijp. Security analysis of india's electronic voting machines. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 1–14, 2010.

[23] Scott Wolchok, Eric Wustrow, Dawn Isabel, and J Alex Halderman. Attacking the washington, dc internet voting system. In *International Conference on Financial Cryptography and Data Security*, pages 114–128. Springer, 2012.

[24] Kaili Ye, Dong Zheng, Rui Guo, Jiayu He, Yushuang Chen, and Xiaoling Tao. A coercion-resistant e-voting system based on blockchain technology. *Int. J. Netw. Secur*, 23:791–806, 2021.

[25] Gongxian Zeng, Meiqi He, Siu Ming Yiu, and Zhengan Huang. A self-tallying electronic voting based on blockchain. *The Computer Journal*, 2021.

Appendix A

Under Half-ability Compromised VSD

A.1 $\mathcal{F}_{STE}$ under half-ability compromised VSD

Setup

■ Upon receiving (sid, CORRUPT, $\mathbf{V}_{corr}$) from $\mathcal{S}$, if $\mathbf{V}_{corr} \subseteq \mathbf{V}$, it fixes $\mathbf{V}_{corr}$ as corrupted voters who are manipulated by $\mathcal{S}$.

■ Upon receiving (sid, COMPROMISE_VSD, $\mathbf{V}_{cVSD}$) from $\mathcal{S}$, if $\mathbf{V}_{cVSD} \subseteq \mathbf{V}$, if fixes $\mathbf{V}_{cVSD}$ as the set of voters who uses compromised malicious VSD.

■ Upon receiving (sid, SETUP_INFO, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) from SA the first time, if $\mathbf{V}_{elig} \subseteq \mathbf{V}$ and $t_{cast} < t_{open}$, it forwards message (sid, SETUP_INFO, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) to $\mathcal{S}$. Upon receiving (sid, SETUP_OK, $\mathbf{V}_{elig}$, $\mathbf{O}$, t_{cast}, t_{open}) from $\mathcal{S}$, it sets $\overrightarrow{t} \leftarrow$ define_time(t_{cast}, t_{open}) and vote.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$).

■ Upon receiving (sid, ELIGIBLE) from SA, it informs $\mathcal{S}$, which replies with eligibility algorithms **GenCred**, **VrfyBallot**, **AuthBallot**, **UpState** and initial accumulator state St_{gen}.It sets reg.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$, St_{gen}).

Credential Generation

■ Upon receiving (sid, GEN_CRED) from $V \in \mathbf{V}_{elig}$for the first time, it reads the time Cl from $\mathcal{G}_{clock}$.

1. If Status(Cl, $\overrightarrow{t}$, Cred) = $\top$ and $V \in \mathbf{V} \backslash \mathbf{V}_{corr}$, it does the following:

   (a) If there is no tuple (V, cr, $\hat{cr}$) in L_{cr}, it runs (cr, $\hat{cr}$) $\rightarrow$ **GenCred** (1^{λ}, reg.par). If there are tuples ($\cdot$, cr, $\cdot$) or ($\cdot$, $\cdot$, $\hat{cr}$) in L_{cr} or (cr, $\hat{cr}$) = $\perp$,

it sends (sid, GEN_CRED, $\perp$) to V hand halts. Else, it adds (V, cr, $\hat{cr}$) to L_{cr}.

(b) If tuple (V, cr, $\hat{cr}$) is successfully added to L_{cr} and $V \in \mathbf{V}_{mVSD}$, it sends (sid, TAMPER_CRED) to $\mathcal{S}$. (It allows $\mathcal{S}$ to tamper credential and its corresponding commitment without revealing the identity of voter and the original credential.) Upon receiving (sid, TAMPER_CRED, cr', $\hat{cr}'$) from $\mathcal{S}$, it adds (V, cr', $\hat{cr}'$, 1) to L_{elig} after permission of $\mathcal{S}$ via public delay output with (V, $\hat{cr}$) as information leakage.
Else if $V \in V \backslash \mathbf{V}_{mVSD}$, it directly add ($V$, cr, $\hat{cr}$, 1) to L_{elig} after public delay output.

(c) It sends (sid, GEN_CRED, V, $\hat{cr}$, sender) to V and (sid, GEN_CRED, V, $\hat{cr}$) to all other voters in $\mathbf{V} \backslash V$ and $\mathcal{S}$.

2. Else if Status(Cl, $\overrightarrow{t}$, Cred) = $\top$ but $V \in \mathbf{V}_{corr}$, it forwards the message (sid, GEN_CRED, V) to $\mathcal{S}$. Upon receiving (sid, GEN_CRED, V, cr, $\hat{cr}$) from $\mathcal{S}$, if there are no tuples (V, cr^*, $\hat{cr}^*$, 0), ($\cdot$, $\cdot$, $\hat{cr}$, 1) or ($\cdot$, cr, $\cdot$, 1), it adds (V, cr, $\hat{cr}$, 0) to L_{elig}. Then it sends (sid, GEN_CRED, V, $\hat{cr}$) to all voters in $\mathbf{V} \backslash V$ and $\mathcal{S}$.

Cast

■ Upon receiving (sid, CAST, o) from $V \in \mathbf{V}_{elig} \backslash \mathbf{V}_{corr}$ for the first time such that (V, cr, $\hat{cr}$, 1) $\in L_{elig}$ and $o \in \mathbf{O}$, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cast) = $\top$, it does:

1. It picks $tag \xleftarrow{\$} TAG$ and it inserts the tuple (V, NULL, o, tag, 1) $\rightarrow L_{gball}$.

2. It sends (sid, GEN_BALLOT, tag, $0^{|o|}$) to $\mathcal{S}$. Upon receiving (sid, GEN_BALLOT, tag, $0^{|o|}$, v) from $\mathcal{S}$, it updates (V, NULL, o, tag, 1) in L_{gball} to (V, v, o, tag, 1).

3. It computes the final accumulator state $St_{fin} \leftarrow$ **UpState**(St_{gen}, $\hat{cr}$) for the purpose of signature generation. Then it generates the signature σ of ballot v: $\sigma \leftarrow$ **AuthBallot** (v, cr, $\hat{cr}$, St_{fin}). If **VrfyBallot** (v, σ, reg.par) = 0, it sends (sid, AUTH_BALLOT, $\perp$) to V and halts.

4. If $V \in \mathbf{V}_{cVSD}$, it sends (sid, TAMPER_CASTING, v, cr, σ) to $\mathcal{S}$. Upon receiving (sid, TAMPER_CASTING, v', cr', σ') from $\mathcal{S}$, it adds (V, v', $\cdot$, cr', σ', 2) to L_{cast} after public delay output.

■ Upon receiving receiving (sid, CAST) from $V \in \mathbf{V}_{corr}$, it informs $\mathcal{S}$. Upon (sid, CAST, v, σ, V) from $\mathcal{S}$ for the first time, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cast) = $\top$ and there is a tuple (V, cr, $\hat{cr}$, 0) $\in L_{elig}$, it adds (V, v, $\cdot$, cr, σ, 0) to L_{cast}.

Tally

■ Upon receiving (sid, TALLY) from a voter $V \in \mathbf{V}$, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\perp$, it does:

1. If $res = \phi$, it does the following:

   (a) For every (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) in L_{cast}, it runs the ballot verification algorithm $x \leftarrow$ **VrfBallot** (v^*, σ^*, St_{fin}). If $x = 1$, then $\mathcal{F}_{STE}$ performs the following security checks:

      i. If there is no tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig}, it sets res to $\perp$.

      ii. If there is a tuple (V^*, cr^*, $\hat{cr}^*$, 1) in L_{elig} and there is a tuple ($\cdot$, v^{**}, o^{**}, cr^{**}, σ^{**}, 1) in L_{cast} such that $(cr^* = cr^{**}) \wedge (v^* \neq v^{**})$, it sets res to $\perp$.

      iii. Otherwise, it adds (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) to L_{tally}.

   (b) For every tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig} such that there are multiple tuples (V, v^1, o^1, cr^1, σ^1, $\cdot$),...,(V, v^n, o^n, cr^n, σ^n, $\cdot$) in L_{tally}, it removes all multiple tuples from L_{tally} except the first one it recorded.

   (c) For every tuple (V^*, v^*, $\cdot$, cr^*, σ^*, 2) in L_{tally}, it sends (sid, OPEN_TAMPERED_BALLOT, V^*, v^*, cr^*, σ^*) to $\mathcal{S}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V^*, o^*, v^*, cr^*, σ^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple (V^*, v^*, o^*, cr^*, σ^*, 2) in L_{tally}.

   (d) For every tuple (V^*, v^*, $\cdot$, cr^*, σ^*, 0) in L_{tally}, it sends (sid, OPENING, V^*, v^*) to $\mathcal{S}$. Upon receiving (sid, OPENING, V^*, v^*, o^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple (V^*, v^*, o^*, cr^*, σ^*, 0) in L_{tally}.

   (e) If there are two tuples (V^*, v^*, o^*, cr^*, σ^*, 1), ($V^{*\prime}$, $v^{*\prime}$, $o^{*\prime}$, $cr^{*\prime}$, $\sigma^{*\prime}$, 1) such that $v^* = v^{*\prime} \wedge o^* \neq o^{*\prime}$, it sets res to $\perp$

(f) It sets tally result *res* as the multiset $\{ (o^*, v^*, cr^*, \sigma^*) \mid (V, v^*, o^*, cr^*, \sigma^*, \cdot) \in L_{tally} \}$.

2. It returns (sid,TALLY, *res*) to V.

■ Upon receiving (sid, LEAKAGE) from $\mathcal{S}$, it reads the time Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\perp$ or Status(Cl, $\overrightarrow{t}$, Cred) = Status(Cl, $\overrightarrow{t}$, Cast) = Status(Cl, $\overrightarrow{t}$, Tally) = $\perp$, it returns to $\mathcal{S}$ all the tuples (v, o, 1) such that (V^*, v, o, tag^*, 1) $\in L_{gball} \wedge$ (V^*, v, o, cr^*, σ^*, 1) $\in L_{cast}$.

Verification

■ Upon receiving (sid, VERIFY, *rês*) from a voter $V \in \mathbf{V} \backslash \mathbf{V}_{corr}$, it reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Verification) = $\perp$, it does:

1. It finds the tuple (V, cr, $\hat{cr}$) in L_{cr} and (V, cr', $\hat{cr}'$, 1) in L_{elig}. If $\hat{cr} \neq \hat{cr}'$ or $cr \neq cr'$, it returns (sid, VERIFY, *rês*, 0, 1).

2. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, it checks if there exists one tuple such that $cr^* = cr$ where cr is from the tuple (V, cr, $\hat{cr}$, 1) in L_{elig}.
   If such tuple does not exist, it returns (sid, VERIFY, *rês*, 0, 2).
   Else, it finds the tuple (o^*, v^*, cr, σ^*) in *rês*. If $o^* \neq o$ where o is from the tuple (V, v, o, tag, 1) in L_{gball}, it returns (sid, VERIFY, *rês*, 0, 3).

3. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, it runs the ballot verification algorithm $x \rightarrow VrfyBallot(v^*, \sigma^*, St_{fin})$. If $x = 0$, $x = \perp$ or there is no tuple ($\cdot$, cr^*, $\cdot$, $\cdot$) in L_{elig}, it returns (sid, VERIFY, *rês*, 0, 4). Otherwise,

4. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, it checks if there exists another tuple (o^{**}, v^{**}, cr^{**}, σ^{**}) such that $cr^* = cr^{**}$. If exists, it returns (sid, VERIFY, *rês*, 0, 5).

5. It does the following to check universal verifiability and ene-to-end verifiability (tally as recorded):

   (a) For every (V^*, v^*, o^*, cr^*, σ^*, $\cdot$) in L_{cast}, it runs the ballot verification algorithm $x \leftarrow$ **VrfBallot** (v^*, σ^*, St_{fin}). If $x = 1$, then $\mathcal{F}_{STE}$ performs the following security checks:

      i. If there is no tuple (V^*, cr^*, $\hat{cr}^*$, $\cdot$) in L_{elig}, it sets res_{vrfy} to $\perp$.

ii. If there is a tuple $(V^*, cr^*, \hat{cr}^*, 1)$ in L_{elig} and there is a tuple $(\cdot, v^{**}, o^{**}, \hat{cr}^{**}, \sigma^{**}, 1)$ in L_{cast} such that $(\hat{cr}^* = \hat{cr}^{**}) \wedge (v^* \neq v^{**})$, it sets res_{vrfy} to $\perp$.

iii. Otherwise, it adds $(V^*, v^*, o^*, \hat{cr}^*, \sigma^*, \cdot)$ to L_{vrfy}.

(b) For every tuple $(V^*, cr^*, \hat{cr}^*, \cdot)$ in L_{elig} such that there are multiple tuples $(V, v^1, o^1, cr^1, \sigma^1, \cdot)$,...,$(V, v^n, o^n, cr^n, \sigma^n, \cdot)$ in L_{vrfy}, it removes all multiple tuples from L_{vrfy} except the first one it recorded.

(c) For every tuple $(V^*, v^*, \cdot, cr^*, \sigma^*, 2)$ in L_{vrfy}, it sends (sid, OPEN_TAMPERED_BALLOT, V, v^*, cr^*, σ^*) to $\mathcal{S}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, o^*, v^*, cr^*, σ^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple to $(V^*, v^*, o^*, cr^*, \sigma^*, 2)$ in L_{vrfy}.

(d) For every tuple $(V^*, v^*, \cdot, cr^*, \sigma^*, 0)$ in L_{vrfy}, it sends (sid, OPENING, V^*, v^*) to $\mathcal{S}$. Upon receiving (sid, OPENING, V^*, v^*, o^*) from $\mathcal{S}$, if $o^* \in \mathbf{O}$ then it updates the tuple $(V^*, v^*, o^*, cr^*, \sigma^*, 0)$ in L_{vrfy}.

(e) It checks the correctness of ballot generated from honest voters with uncompromised VSD as followings: if there are two tuples $(V^*, v^*, o^*, cr^*, \sigma^*, 1)$, $(V^{*'}, v^{*'}, o^{*'}, cr^{*'}, \sigma^{*'}, 1)$ in L_{vrfy} such that $(v^* = v^{*'}) \wedge (o^* \neq o^{*'})$, then it sets res_{vrfy} to $\perp$. Else, it sets verify result res_{vrfy} as the multiset $\{ (o^*, v^*, cr^*, \sigma^*) \mid (V, v^*, o^*, cr^*, \sigma^*, \cdot) \in L_{vrfy} \}$.

(f) If $\hat{res} = res_{vrfy}$, it returns (sid, VERIFY, $\hat{res}$, 1, 0) to V. Else, it returns (sid, VERIFY, $\hat{res}$, 0, 6) to V.

■ Upon receiving (sid, VERIFY, $\hat{res}$) from a voter $V \in \mathbf{V}_{corr}$, it sends (sid, VERIFY, $\hat{res}$) to $\mathcal{S}$, and replies to V whatever it receives from $\mathcal{S}$.

A.2 Realizing $\mathcal{F}_{STE}$ under half-ability compromised VSD

A.2.1 Hybrid protocol $\Pi'^{\mathcal{F}_{NIC},\mathcal{F}_{BC},\mathcal{F}_{elig},\mathcal{F}_{vm},\mathcal{F}_{VSD},\mathcal{G}_{clock}}_{E-cclesia}$

Setup:

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

■ Upon receiving (sid, ELIGIBLE) from $\mathcal{Z}$, if SA has received ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$) from $\mathcal{F}_{vm}$, it sends (sid, ELIGIBLE, $\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$) to $\mathcal{F}_{elig}$ which sends reg.par := ($\mathbf{V}_{elig}$, $\mathbf{O}$, $\overrightarrow{t}$, St_{gen}) to $< V_i >_{i\in[n]}$. Upon receiving reg.par from $\mathcal{F}_{elig}$, each $V \in \mathbf{V}$ stores it as the election parameters reg.par.

Credential Generation:

■ Upon receiving (sid, GEN_CRED) from $\mathcal{Z}$, for the first time, V reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Cred) = $\top$, she does:

1. She picks a random cr from the message space $\mathcal{M}$ and sends (sid, COM_COMMIT_INT, cr) to $\mathcal{F}_{NIC}$. Upon receiving (sid, COM_COMMIT_END, $\hat{cr}$) from $\mathcal{F}_{NIC}$, if $\hat{cr} \notin D$, she repeats this step until it does.

2. She sends (sid, TAMPER_CRED, cr, $\hat{cr}$) to $\mathcal{F}_{VSD}$. Upon receiving (sid, TAMPER_CRED, cr', $\hat{cr}'$) from $\mathcal{F}_{VSD}$, she stores (cr, $\hat{cr}'$).

3. She sends (sid_{all}, BROADCAST, $\hat{cr'}$) to $\mathcal{F}_{BC}$. Upon receiving (sid_{all}, BROADCAST, (V ,$\hat{cr'}$)) from $\mathcal{F}_{BC}$, she appends the pair (V ,$\hat{cr'}$) to L_{cred}.

Cast:

■ Upon receiving (sid, CAST, o) from $\mathcal{Z}$, V executes the following steps:

1. She sends (sid, GEN_BALLOT, o) to F_{vm} which replies with the generated ballot as (sid, GEN_BALLOT, o, v) (or sends (sid, GEN_BALLOT, o, $\perp$) and halts).

2. She sends (sid, AUTH_BALLOT, v) to $\mathcal{F}_{elig}$ which replies with the authentication receipt for v (sid, AUTH_BALLOT, v, σ) (or sends (sid, AUTH_BALLOT, v, $\perp$) and halts).

3. She sends (sid, TAMPER_CASTING, V, o, v, cr, σ) to $\mathcal{F}_{VSD}$. Upon receiving (sid, TAMPER_CASTING, V, v', cr', σ') from $\mathcal{F}_{VSD}$, she sends (sid, CAST, v', cr', σ') to $\mathcal{F}_{vm}$ which broadcasts the message to $< V_j >_{j\in[n]}$. In turn, voters store the received pair (v, cr, σ).

Tally

■ Upon receiving a message (sid, TALLY) from $\mathcal{Z}$, V reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Tally) = $\perp$, she ignores the message. Otherwise, if $res = \phi$, she executes the following steps:

1. For every tuple (sid, Cast, v, cr, σ) she has obtained from $\mathcal{F}_{vm}$, she sends (sid, VER_BALLOT, v, σ) to $\mathcal{F}_{elig}$ which replies with (sid, VER_BALLOT, v, σ, x), where $x \in \{0, 1, \perp\}$. If there is any ballot verification request such that $\mathcal{F}_{elig}$ replied with $x = \perp$, then she discards the ballot. Otherwise, she includes in her tally set all tuples (v, cr, σ) such that $\mathcal{F}_{elig}$ replied with $x = 1$.

2. She discards multiple ballots by sending (sid, LINK_BALLOTS, (v, σ), (v', σ')) to $\mathcal{F}_{elig}$ for every pair (v, σ), (v', σ') in her tally set. If she gets a (sid, LINK_BALLOTS, (v, σ), (v', σ'), 1) respond, then she discards the ballot she received the last out of those two.

3. For every tuple (v, cr, σ)in the tally set, she sends (sid, OPEN_TAMPERED_CASTING, v) to $\mathcal{F}_{VSD}$, which could reply with (sid, OPEN_TAMPERED_CASTING, v, o) or (sid, OPEN_TAMPERED_CASTING, v, $\perp$). If the opening is $\perp$, then she sends (sid, OPEN, v) to $\mathcal{F}_{vm}$ to try to open again. Upon receiving the opening (sid, OPEN, v, o), she adds (o, v, cr, σ) to *res*. If at any time $\mathcal{F}_{vm}$ replies with (sid, OPEN, v, $\perp$), she sets *res* to $\perp$.

Verification:

■ Upon receiving (sid, VERIFY, *res*) from $\mathcal{Z}$, V reads Cl from $\mathcal{G}_{clock}$. If Status(Cl, $\overrightarrow{t}$, Verification) = $\perp$, she does:

1. She accesses her registered credential (V, $\hat{cr}'$) on a trusted VSD. Then she verifies if her credential has been tampered by sending (sid, COM_VERIFY_INI, $\hat{cr}'$, cr) to $\mathcal{F}_{NIC}$. Upon receiving (sid, COM_VERIFY_END, y) from $\mathcal{F}_{NIC}$, if $y = 0$ or $y = \perp$, she returns false with a reason of dirty credential registration. Otherwise, she proceeds 2.

2. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, she checks if there exists a tuple such that $cr* = cr$ where cr is the credential she memorise by heart. If such tuple does not exist, she returns false with a reason of individual verifiability. Otherwise, she proceeds 3.

3. She finds the tuple (o^*, v^*, cr^*, σ^*) in *rês* such that $cr^* = cr$. If $o^* \neq o$ where o is her intended option, she returns fail with reason of end-to-end verifiability: cast is not as intended. Otherwise, she proceeds 4.

4. For every tuple (o^*, v^*, cr^*, σ^*) in *rês*, she sends (sid, VER_BALLOT, v^*, σ^*) to $\mathcal{F}_{elig}$ which replies with (sid, VER_BALLOT, v^*, σ^*, x), where $x \in \{0, 1, \perp\}$.

If there is any ballot verification request such that $\mathcal{F}_{elig}$ replied with $x = \perp$ or 0, then she returns fail with reason of eligibility verifiablity: not eligible voters. Otherwise, she proceeds 5.

5. For every tuple (o^*, v^*, cr^*, σ^*) in $\hat{res}$, she sends (sid, LINK_BALLOTS, (v^*, σ^*), ($v^{*'}$, $\sigma^{*'}$)) to $\mathcal{F}_{elig}$ for every pair (v^*, σ^*), ($v^{*'}$, $\sigma^{*'}$) in her tally set. If she gets a (sid, LINK_BALLOTS, (v^*, σ^*), ($v^{*'}$, $\sigma^{*'}$), 1) respond, she returns fail with a reason of eligibility verifiability: duo-voting. Otherwise, she proceeds 6.

6. She re-do tally on a trusted VSD to verify universal verifiability and end-to-end verifiability (tally as recorded). Specifically,

   (a) For every tuple (sid, Cast, v, cr, σ) she has obtained from $\mathcal{F}_{vm}$ accessing via a trusted VSD, she sends (sid, VER_BALLOT, v, σ) to $\mathcal{F}_{elig}$ which replies with (sid, VER_BALLOT, v, σ, x), where $x \in \{0, 1, \perp\}$. If there is any ballot verification request such that $\mathcal{F}_{elig}$ replied with $x = \perp$, then she discards the ballot. Otherwise, she includes in her verification set all tuples (v, cr, σ) such that $\mathcal{F}_{elig}$ replied with $x = 1$.

   (b) She discards multiple ballots by sending (sid, LINK_BALLOTS, (v, σ), (v', σ')) to $\mathcal{F}_{elig}$ for every pair (v, σ), (v', σ') in her verification set. If she gets a (sid, LINK_BALLOTS, (v, σ), (v', σ'), 1) respond, then she discards the ballot she received the last out of those two.

   (c) For every tuple (v, cr, σ)in the verification set, she sends (sid, OPEN_TAMPERED_CASTING, v) to $\mathcal{F}_{VSD}$, which could reply with (sid, OPEN_TAMPERED_CASTING, v, o) or (sid, OPEN_TAMPERED_CASTING, v, $\perp$). If the opening is $\perp$, then she sends (sid, OPEN, v) to $\mathcal{F}_{vm}$ to try to open again. Upon receiving the opening (sid, OPEN, v, o), V adds (o, v, cr, σ) to res_{vrfy}. If at any time $\mathcal{F}_{vm}$ replies with (sid, OPEN, v, $\perp$), she sets res_{vrfy} to $\perp$.

   (d) She compares res_{vrfy} with $\hat{res}$. If $res_{vrfy} \neq \hat{res}$, she returns fail with reason of universal verifiability. Otherwise, she returns true back to $\mathcal{Z}$

A.2.2 Formal proof

Theorem 3 *The hybrid protocol* $\Pi'^{\mathcal{F}_{NIC},\mathcal{F}_{BC},\mathcal{F}_{elig},\mathcal{F}_{vm},\mathcal{F}_{VSD},\mathcal{G}_{clock}}_{E-cclesia}$ *in Appendix A.2.1 UC-realizes the* $\mathcal{F}_{STE}$ *functionality in Appendix A.1.*

PROOF. A simulator $\mathcal{S}$ who executes as following to be indistinguishable to any adversary $\mathcal{A}$ from any PPT time environment.

To make the proof more concise, I will only present in details what $\mathcal{S}$ would operates when receiving messages or commands from un-corrupted party. For messages from corrupted party, $\mathcal{S}$ will forward whatever it receives from $\mathcal{F}_{STE}$ to $\mathcal{A}$ as if it was that party and returns whatever message it receives from $\mathcal{A}$ to $\mathcal{F}_{STE}$. Also, as for public delay output, what $\mathcal{S}$ does is merely conveying messages to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$ or $\mathcal{F}_{elig}$ and returning the permission back to $\mathcal{F}_{STE}$. The detailed execution is as following:

Setup:

■ Upon receiving (sid, CORRUPT, V_{corr}), $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was $\mathcal{Z}$. Upon receiving (sid, CORRUPT, V_{corr}) from $\mathcal{A}$ as if it was $\mathcal{F}_{elig}$, $\mathcal{S}$ forwards the same message as if it was from $\mathcal{Z}$ to $\mathcal{A}$. Upon receiving (sid, CORRUPT, V_{corr}) from $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, COMPROMISE_VSD, V_{cVSD}), $\mathcal{S}$ forwards the message to $\mathcal{A}$ as if it was $\mathcal{Z}$. Upon receiving (sid, COMPROMISE_VSD, V_{cVSD}) from $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$, it forwards the message to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, SETUP_INFO, V_{elig}, **O**, t_{cast}, t_{open}) from $\mathcal{F}_{STE}$, it sets $\overrightarrow{t} \leftarrow$ define_time(t_{cast},t_{open}) and vote.par := (V_{elig}, **O**, $\overrightarrow{t}$). Then it sends (sid, SETUP_OK, vote.par) to $\mathcal{A}$ as if it was from $\mathcal{F}_{vm}$. Upon receiving the permission from $\mathcal{A}$, $\mathcal{S}$ sends (sid, SETUP_OK, V_{elig}, **O**, t_{cast}, t_{open}) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, ELIGIBLE) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, SETUP_ELIG) to $\mathcal{A}$ as if it was from $\mathcal{F}_{elig}$. Upon receiving (sid, SETUP_ELIG, **GenCred**, **VrfyBallot**, **AuthBallot**, **UpState**, St_{gen}) from $\mathcal{A}$, $\mathcal{S}$ sends (sid, ELIGIBLE, **GenCred**, **VrfyBallot**, **AuthBallot**, **UpState**, St_{gen}) back to $\mathcal{F}_{STE}$.

Credential Generation:

■ Upon receiving (sid, TAMPER_CRED) from $\mathcal{F}_{STE}$, $\mathcal{S}$ forwards the same message to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, TAMPER_CRED, $cr^{'}$, $\hat{cr}^{'}$) from $\mathcal{A}$, $\mathcal{S}$ sends (sid, TAMPER_CRED, $cr^{'}$, $\hat{cr}^{'}$) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, GEN_CRED, V, $\hat{cr}$) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, BRAODCAST, (V, $\hat{cr}$)) to $\mathcal{A}$ as if it was $\mathcal{F}_{BC}$. Upon receiving the token back from $\mathcal{A}$, $\mathcal{S}$ sends (sid, GEN_CRED, V, $\hat{cr}$) to $\mathcal{F}_{STE}$.

Cast:

■ Upon receiving (sid, GEN_BALLOT, tag, $0^{|o|}$) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, GEN_BALLOT,

tag, $0^{|o|}$) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, GEN_BALLOT, tag, $0^{|o|}$, v) from $\mathcal{A}$, $\mathcal{S}$ forwards the same message (sid, GEN_BALLOT, tag, $0^{|o|}$, v) back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, TAMPER_CASTING, v, cr, σ) from $\mathcal{F}_{STE}$, it forwards the message to $\mathcal{S}$ as if it was $\mathcal{F}_{VSD}$ and returns whatever it receives from $\mathcal{S}$ back to $\mathcal{F}_{STE}$.

■ Upon receiving (sid, CAST, v, cr, σ) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, ALLOW_CAST, v, σ) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, CAST_ALLOWED) from $\mathcal{A}$, $\mathcal{S}$ returns the same message back to $\mathcal{F}_{STE}$.

Tally:

■ Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ).

■ Upon receiving (sid, OPENING, V, v) from $\mathcal{F}_{STE}$ for an alternative ballot opening for a corrupted voter V, $\mathcal{S}$ sends (sid, OPEN, v) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, OPEN, v, o) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPENING, V, v, o) back to $\mathcal{F}_{STE}$.

Verification:

■ Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) from $\mathcal{F}_{STE}$, $\mathcal{S}$ sends (sid, OPEN_TAMPERED_BALLOT, V, v, cr, σ) to $\mathcal{A}$ as if it was $\mathcal{F}_{VSD}$. Upon receiving (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPEN_TAMPERED_BALLOT, V, o, v, cr, σ).

■ Upon receiving (sid, OPENING, V, v) from $\mathcal{F}_{STE}$ for an alternative ballot opening for a corrupted voter V, $\mathcal{S}$ sends (sid, OPEN, v) to $\mathcal{A}$ as if it was $\mathcal{F}_{vm}$. Upon receiving (sid, OPEN, v, o) from $\mathcal{A}$, $\mathcal{S}$ returns (sid, OPENING, V, v, o) back to $\mathcal{F}_{STE}$.