Zerovote: Self-tallying E-voting Protocol in the UC Framework

Lenka Marekova

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2018

Abstract

The main purpose of the project is to define a decentralized e-voting protocol that could be implemented on the blockchain in the universal composition (UC) framework and provide a formal proof that it preserves voter privacy.

The protocol is based on the ideas in Zerocoin and built in a hybrid model utilizing ideal functionalities for digital signatures, commitment schemes and secure accumulators, the last of which is new and with proof of its realizability. As opposed to Zerocoin, the zero-knowledge signature of knowledge is defined in terms of other functionalities to obtain a more versatile construction in the CRS model.

Acknowledgements

I would like to express my gratitude to Myrto Arapinis, my project supervisor, for her guidance, for the interesting discussions as well as for her patience with my questions.

I would also like to thank Nikolaos Lamprou and Thomas Zacharias for helping us understand many confusing aspects of the UC framework.

Lastly, I would like to thank Linas Kondrackis for proofreading an early draft of the report, and my parents for their continued support.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Electronic voting has been of interest to the research community and governments for decades, but it is only the advent of Internet that made truly remote voting feasible. Despite that, adoption so far has not been wide and has met with skepticism. It is as much an issue of technology as it is a social and political one – one only has to compare the optimistic tone of the 2002 Electoral Commission report that promised "Britain's first e-enabled General Election" by 2011 [Com02] with the result of the 2007 evaluations that gave the following recommendation: "There are clearly wider issues associated with the underlying security and transparency of these e-voting solutions and their impact on the electoral process, together with the cost effectiveness of the technology, which need to be addressed." [Jus07]. More than ten years and many pilot studies later, the future of e-voting still appears uncertain [GKTP16].

With this in mind, however, we will concern ourselves with the technical problems involved. Existing e-voting schemes differ widely in the extent to which they make use of technology, ranging from schemes where machines only assist in counting of the votes to full-fledged online voting. We will give a brief overview of both types, but our main focus will be on the latter end of the spectrum, as it is the direction with the most potential but also obstacles that need to be overcome.

There are several apparent benefits to e-voting: a chance to greatly improve accessibility and thereby increase voter turnout, better scalability and lower administrative burden, assuring accuracy, and even the ability to prevent electoral fraud. However, fully electronic voting schemes are often vulnerable to attacks due to their centralized nature and lack of transparency, potentially enabling fraud on a much larger scale, not to mention disruption of the elections by attacks on the network infrastructure (see e.g. [BPR+04], [ED10], [WWIH12] for a few examples of attacks).

Traditional paper-based schemes are by their nature usually distributed and auditable. It is critical that any e-voting system is able to achieve at least this,

otherwise the public trust will be low and as has been shown many times, often for good reason. Blockchains are emerging as a potential solution, offering a new technology that can serve as a (distributed) trusted third party which is fully transparent, therefore making easily auditable schemes possible. However, since everything on the blockchain is public by default, preserving voter privacy becomes a problem.

There have been recent proposals for privacy-preserving e-voting protocols on the blockchain, but so far none have come with formal proofs. As real-world attacks demonstrate, there is a need for stronger security that does not rely just on the designers' inability to find attacks. Provable security provides a means to guarantee protocols will have the properties we want, regardless of the specific attacks (under a given well-defined adversary model), with the caveat that it is not trivial to define the real-world properties in a formal setting. It is an area of active research, and while the definitions may require some tailoring to the specific protocol, we will aim to make them as strong as possible. For this reason we will work in a cryptographic framework of universal composability [Can13] that we will describe in detail in the next chapter.

1.2 Contributions

We build upon the initial sketch of the Zerovote protocol from a report produced by Thomas Ragel during a summer internship with Myrto Arapinis. The protocol was based on the ideas behind Zerocoin [MGGR13], an anonymous e-cash scheme built on top of the blockchain.

To cover both theory and implementation and thereby demonstrate the security and applicability of the protocol, the work was naturally split into two projects. This report concerns the theory part, whereas the implementation part was done by Ivaylo Genev, with whom the initial specification of Zerovote was agreed upon in order to maintain consistency.

The main contributions are:

- *Formalizing the specification of Zerovote.* The original description was a mix of informal smart contract pseudocode and formal definitions. The formal construction is given in Section 3.3.
- *Adapting the protocol for the UC framework.* We considered proving the properties of the protocol via a reduction to Zerocoin, but this would have required a full proof of Zerocoin itself. Due to its non-standard primitives and reliance on rewinding as a proof technique, it was not clear if the proof would have been possible in the UC framework, so we adopted a more modular approach.
  - To this end, we searched for suitable ideal functionalities for the required primitives in the literature (Section 4.1). The ideal functionality

for signatures of knowledge had to be adapted to work with two other functionalities (Section 4.1.5).

  - In the case of dynamic accumulators, no ideal functionality has been published. We gave a definition and proved its equivalence to the original definition of [CL02], thereby proving its realizability (Section 4.2).
  - Finally, the hybrid protocol utilizing these functionalities in place of real primitives was defined (Section 4.3).

- *An ideal functionality for a self-tallying voting protocol.* A definition was given to provide assurance of correctness and privacy (Section 5.1).
- *Proving Zerovote preserves privacy.* The final step consisted of a proof that the hybrid protocol realizes our ideal functionality (Section 5.3).

1.3 Outline of the report

Chapter 2 gives the necessary background for later chapters, describing previous work done in the field as well as providing an overview of e-voting terms and an introduction to universal composability.

Chapter 3 describes the actual protocol, Zerovote, beginning with an informal overview, definitions of the cryptograpic primitives it uses, and a formal specification.

Chapter 4 follows a similar structure as the previous chapter, but now giving ideal functionalities for each of the primitives (with proofs where the definition is new), and a formal specification of the hybrid protocol.

Chapter 5 defines the ideal functionality for a self-tallying e-voting protocol, discusses the properties it aims to provide and contains the proof that it is realizable by our protocol in the hybrid model.

Chapter 6 concludes with a discussion of the work done, limitations discovered and potential ways of addressing them.

Chapter 2

Background

2.1 E-voting

When we talk about e-voting protocols in general, we employ certain terminology that is common to most protocols despite large differences in how they might be constructed, what their trust assumptions are or what networks or devices they make use of.

With this in mind, we consider a set of *voters* (some of whom may be dishonest) and an *authority* or a set of authorities (who may or may not be trusted). The authorities might be responsible for all or some of the following roles: registration of eligible voters, setting up servers or devices for an election, acting as administrators during voting, tallying the votes to compute the final result, supplying information that allows outside parties to audit the election. If no authority is required to tally the votes, such a protocol is called *self-tallying*[1].

We often meet with the notion of a *bulletin board* that represents the information published during the run of the protocol (which can include ballots) and can be defined in terms of various properties. Most often it takes the form of a public *append-only* board on which messages cannot be erased, but there are other things to consider as well - for instance whether parties have to authenticate to post on it, if there is a trusted server administering it or if an attacker can block its communication channel.

[1]Precise definitions of self-tallying differ in aspects such as whether voters are able to obtain partial results (so votes are revealed online, which will be the case of our protocol) or not (so the votes are only released once voting is finished).

2.1.1 Desirable properties

Correctness

First, regardless of the specifics of a voting protocol, it is clear it must be *correct* to be useful at all. This usually means it must satisfy a collection of basic properties about its behaviour: *completeness* (votes cast honestly will be counted), *soundness* (invalid votes will not count), *unreusability* (voters cannot vote more than once) and *eligibility* (only registered voters can vote) to give the most important ones[2].

Privacy

Stated informally, *privacy* simply concerns the requirement that no one can learn how a given voter voted, i.e. the contents of their ballot remain secret. However, achieving this property with real e-voting protocols is far from a simple task, and for this reason there exist many varieties as well as formal definitions. In some settings a stronger property can be achieved called *perfect ballot secrecy* [KY02] that asserts that no information about individual votes is leaked beyond what can be computed from the tally.

Definitions of privacy in the literature are largely distinguished by the underlying formal model they use, some of which we will briefly describe in the next section. (See [DKR09] for a definition in the symbolic model, [BCG+15] for an overview of game-based definitions and [Gro04], [MN10] for functionalities in the simulation-based model.)

Coercion resistance

Privacy by itself is only enough if we can assume that honest voters cannot be coerced to reveal their votes, which is of course a concern in the real world because of vote buying, forced abstention or voting under threat. A natural property called *receipt-freeness*, which says that voters do not obtain proof of how they voted (i.e. receipts as defined in e.g. [MN06]), represents just one type of *coercion resistance* that protocols can try to satisfy (see [JCJ05] for several attacks not captured by receipt-freeness alone). Because coercion resistance is often at odds with satisfying other properties, a weaker property has been proposed in the form of *coercion evidence* [GRBR13] which doesn't aim to prevent coercion from happening, but does provide public evidence that it has taken place.

[2] We use the terms used for instance in [FOO92], but many other formulations exist capturing more or less the same concerns, just usually specific to the given protocol at hand, see e.g. [CGGI13]. Unreusability may commonly be referred to as the impossibility of *double voting*.

Verifiability

Given the real concerns about election manipulation that are only exacerbated in a remote or digital setting, the property of *individual verifiability* gives each voter the ability to check that their vote was received and counted, while *universal verifiability* captures that anyone can verify that the tally was computed correctly. Together, the voting schemes that satisfy them might be called *end-to-end verifiable*, though specific definitions differ.

Intuitively, these properties alone would not be difficult to achieve just by letting all votes be public, but of course this solution would completely violate voter privacy. It is clear then that the major obstacle in formulating e-voting protocols does not lie in satisfying individual properties but rather in the need to achieve all of them simultaneously. (See [CGK+16] for a comprehensive treatment of various definitions of verifiability proposed in the literature.)

Fairness

Another property that can be non-trivial to satisfy by certain types of protocols is *fairness*, i.e. that no partial tally is revealed that could influence the votes of the remaining voters (thereby giving them an unfair advantage). This property may be violated if a protocol releases any information on votes while still accepting new votes. There exist measures to mitigate the issue (e.g. [KY02]), but they may be costly in other ways.

2.1.2 Types of e-voting systems and related work

When talking about electronic voting protocols, large-scale national elections are often assumed as the main application due to their political and societal impact, but voting mechanisms are used on a much more local scale as well by corporations, universities or non-governmental organizations (which is sometimes referred to as *boardroom voting*). While many of the properties are required regardless of scale, there are other reasons that might make voting schemes unsuitable for national elections, such as lack of efficiency or complicated voting (requiring the voter to go through an involved procedure to verify their vote, for instance). These are not such obstacles on a smaller scale, so it makes sense to investigate schemes of both types.

It is also important to clarify the extent to which e-voting systems are actually "electronic" – whether computers are only used to scan paper ballots to compute the tally, if *direct-recording electronic (DRE)* voting machines are deployed to polling stations and ballots transmitted over the network to external servers, or whether voting can be done fully remotely from any Internet-connected device.

A distinction can also be made based on the degree to which the e-voting schemes rely on trusted authorities, for instance based on whether they use a central server

for recording and counting the votes. Such a server is a convenient target of attacks. This can be somewhat mitigated by dividing the computation between several servers instead of one, but it does not change the inherently centralized nature of such schemes[3]. Hence there are ongoing efforts to build fully distributed systems that do not rely on any one authority (see [RG17]).

Many specific e-voting systems have been tried, as reviewed in e.g. [GKTP16]. Perhaps the most studied of them are Helios [Adi08], Prêt à Voter [RS06] and Civitas [CCM08], all aiming to provide E2E verifiability and vote privacy under the assumption of trust in a number of tallying authorities (some of whom can be corrupted without affecting the security of the protocol).

Towards a provably secure self-tallying protocol

Blockchains as an emerging technology have been used not only to implement pseudo-anonymous digital currencies (Bitcoin, [Nak09]) but also as a general platform for distributed computation (Ethereum, [Woo14]). Informally, what makes them suitable for e-voting consists of many of the same reasons: the ability to stop relying on a central authority and a publicly auditable ledger that can serve as a bulletin board. Issues of scaling aside (which might yet have an independent solution), the main challenge of e-voting protocols using the blockchain infrastructure is then preserving ballot privacy in full public view. Several such proposals already exist in the form of Ethereum *smart contracts*, but they do not come with formal proofs and rather focus on implementation (e.g. [MSH17]).

The idea of Thomas Ragel's initial report is to apply the strategy that Zerocoin [MGGR13] uses to build a form of anonymous e-cash on top of the blockchain. We describe the anonymous credentials in Section 3.2, and the construction for the protocol itself (that we will call Zerovote) is given in Section 3.3. Since security of Zerocoin is not established in a formal model that we can build upon, we do not invoke it as a sub-protocol but rather redefine its purpose in our protocol in terms of primitives that we can prove in the UC framework, described in the next section.

We note that development of Zerocoin continued in the form of Zerocash [BCG+14] and Zcash [HBHW16], which are focused on providing a payment scheme with shielded transactions. (A very recent proposal for an e-voting protocol using Zcash was given in [TT17], though it does not come with a formal analysis and relies on the properties provided by Zcash.)

[SP15] gives several definitions for ideal e-voting functionalities, but still works in the model that separates voters and authorities and their self-tallying protocol is not based on anonymous channels.

[3] In practice, they often use *threshold cryptosystems* which require cooperation of a number of authorities in order to decrypt, so a small number of corrupted authorities will not be able to affect the election.

2.2 Universal composition framework

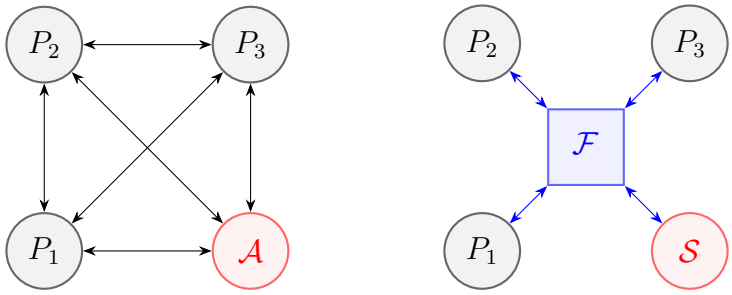
To be able to rigorously analyze the security properties of any protocol, it is necessary to first set it in a formal model, one that is as general as possible to obtain wider applicability. We briefly explain what is often called the standalone model, and then follow onto universal composability and describe how it differs and why it is advantageous, also giving formal definitions and theorems that we will use in later chapters.

2.2.1 Informal overview

Standalone model

The first important notion that underpins the standalone model (also called the *simulation model*) for analyzing the security of cryptographic protocols is the idea of comparing the real world (in which protocols are actually executed) to an *ideal* one in order to prove the validity of certain assertions about the behavior of the protocols.

The *ideal protocol* (or *functionality*) represents a trusted third party which securely computes the function of some real protocol and outputs the result, so in the ideal world it is impossible for any adversary to perform any attacks. If we can then show that any attack against the real protocol can also be mounted in the ideal scenario (we say that outputs of the adversaries are *indistinguishable*), we can prove our protocol to be secure. The adversary in the ideal world is often called the *simulator*, since it has to internally run the real adversary to show that any attack of the real adversary can be simulated.



(a) Real world (b) Ideal world

Figure 2.1: Protocol execution in the standalone model

Figure 2.1 illustrates the idea of comparing the real with the ideal, with *honest* parties P_i and adversaries $\mathcal{A}$ and $\mathcal{S}$ that act on behalf of the *corrupted* parties (we will define corruption of parties precisely in later sections, but for now it suffices to say that they are under complete control of the attacker). In the real protocol, the parties compute the function by mutual interaction, whereas in the ideal world they merely give their inputs to the functionality $\mathcal{F}$ and receive their outputs.

Assuming that the functionality is defined in such a way that it correctly captures the security properties required, this notion may seem as a sufficient proof of security, and indeed in many settings it represents a reasonable result. However, as we will see there are ways to make the model stronger. As the name *standalone* implies, only one instance of the protocol is considered here, running in isolation, which does not correspond to what happens when most protocols are actually implemented.

Hence the *composition* that can be obtained in this model is limited to the so-called *sequential composition*, that can guarantee security under composition if protocols are run one after another, one at a time. As a consequence, *hybrid* protocols can be built, in which a real sub-protocol can be securely replaced by an ideal protocol that we know is indistinguishable from the real one as long as no other parties are allowed to communicate while the ideal protocol executes. If many copies of the ideal protocol are used, they are all used independently. (See [Lin17] for more detail.)

Composing protocols in a more realistic setting is not straightforward, though, and it is usually not feasible to consider everything that may happen during an execution for every protocol that we analyze. Despite this, we would like to model protocols executing concurrently, in a complex and potentially unknown environment, which leads us to the notion of *universal composition*.

Universal composability

Observe that protocol execution in the original simulation model can be redefined in terms of (restricted) interaction with a new entity called the *environment*: this entity will provide inputs to all parties at the beginning of the protocol and collect their outputs at the end, and so instead of directly comparing the outputs of the real adversary and the simulator to prove indistinguishability of the protocols, it is the aim of the environment to distinguish between the executions based on the outputs it sees[4]. The adversary is not allowed to interact with the environment, and so this definition is equivalent to the original one.

The UC framework of [Can13] builds on the standalone model by strengthening two of the components of the new definition.

First, consider adding the notion of *interactivity* to the environment, granting it power to observe and schedule the execution of the parties besides just supplying input, and also letting it freely interact with the adversary (which is why it is also called an "interactive distinguisher").

Second, consider making the ideal functionality (still the trusted party that computes our protocol) interactive as well and granting it the ability to maintain state between calls. Then we will say that the real protocol *realizes* the ideal protocol if any real adversary can be simulated in the ideal world such that no

[4]It is allowed to do some computation on the outputs before outputting its decision.

environment is able to distinguish between them (we give the formal definition in the next section).

Universal composition is then defined again in terms of hybrid protocols, but without the limitations of sequential composition. Once we know that a certain protocol realizes a functionality, we can use it as a building block in other protocols while preserving the security properties, which enables intuitive modular design and analysis of protocols. Hence we obtain stronger guarantees without having to explicitly model what happens in the environment or whether multiple instances are running concurrently.

2.2.2 Formal model

Here we summarize the main elements of the UC framework [Can13] at a level relevant to this work[5].

Preliminaries

A *probability distribution ensemble* $X = \{X(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}*}$ is a collection of probability distributions parametrized by input λ, z. We will restrict our attention to binary distributions, i.e. over $\{0, 1\}$. There are three major definitions that capture how "close" the distributions are: *perfect*, *statistically close* and *computationally indistinguishable*.

For computational indistinguishability, a definition is given below (which could have been equivalently defined in terms of a *negligible function* of λ). Perfect closeness is in contrast achieved only if the ensembles are equal and statistical closeness requires their *statistical distance* to be a negligible function.

Definition 2.2.1 (Definition 4 of [Can13])
Let X, Y be binary probability distribution ensembles. X and Y are indistinguishable (written as $X \approx Y$) if for all $c, d \in \mathbb{N}$ there is $\lambda_0 \in \mathbb{N}$ such that for all $\lambda > \lambda_0$ and $z \in \cup_{\kappa\leq\lambda^d}\{0, 1\}^\kappa$ we have

$$|\Pr[X(\lambda, z) = 1] - \Pr[Y(\lambda, z) = 1]| < \frac{1}{\lambda^c}.$$

Model of computation

All parties participating in a protocol (including the adversary) are modeled as *interactive Turing machines* (ITMs), which are Turing machines extended with a number of special tapes and new instructions. We will not go into the details of the syntax, only note that each ITM has a read-only *identity tape* containing its

[5]We treat the low-level definitions more informally as we don't need to refer to the full details, but state the theorems relating to emulation and composition as they were given.

code and a unique identity string, as well as 3 types of "shared" tapes intended for different kinds of interaction with other ITMs: *input*, *subroutine output* and *incoming communication*. The first two are meant to capture a trusted computing environment in which the parties can verify each other's code and identity, while the last one represents communication over an untrusted medium without any assurances about the writing party. Unless specified otherwise, we will almost always consider *probabilistic polynomial-time* (PPT) machines.

An *ITM instance* (ITI) is defined to be a specific instance of the ITM running on some data. A *system of ITMs* consists of an initial ITM and a *control function* that governs which instructions will be carried out and in what order. The *execution* of the system (with some input) is then a sequence of activations of its ITIs. It is standard to express the bound on runtime of such an execution by parameterizing the system with a *security parameter* 1^λ.

The following convention is used for naming interaction between ITIs μ, μ':

μ *sends* m to μ'	μ writes m to the incoming communication tape of μ'
μ *passes input* x to μ'	μ writes x to the input tape of μ'
μ' *outputs* x to μ	μ' writes x to the subroutine output tape of μ

A *protocol* is simply defined as a single ITM that describes the programs to be run by each party. The identity of each ITI is represented by a pair (sid, pid) for a session identifier *sid* and a party identifier *pid*, so that an *instance of a protocol* is a set of ITIs with the same code and *sid*. Every ITI in a protocol instance is a *party* of that protocol instance.

Defining security

We can now define the actual *model for executing a protocol* π *with environment* $\mathcal{E}$ *and adversary* $\mathcal{A}$ as a parametrized system of PPT ITMs π, $\mathcal{E}$, $\mathcal{A}$ (where $\mathcal{E}$ is the initial ITM) with a control function that enforces the following:

- The input of $\mathcal{E}$ represents some initial state of the environment (including the inputs of all parties). $\mathcal{E}$ may only pass inputs to parties of π.
- The first ITI invoked by $\mathcal{E}$ is $\mathcal{A}$, which can write to any tape of any ITI without restrictions. To explicitly model the role of $\mathcal{A}$ in "controlling" the network, ITIs of π are only allowed to send messages to $\mathcal{A}$ (we say that $\mathcal{A}$ *delivers* m if $\mathcal{A}$ sends m to some ITI, which may not be the same message that $\mathcal{A}$ received) and pass inputs/output to ITIs other than $\mathcal{E}$ or $\mathcal{A}$.
- Whenever a party of π outputs to a target with *sid* other than that of π, the control function outputs it to $\mathcal{E}$ with the code of the writing party removed.

In essence, the model "hides" the existence of the environment from the parties of the protocol while letting it observe the execution from the outside, without access to the local computation of the parties.

The output of the execution will be represented by a binary probability distribution ensemble of the form

$$\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{E}} := \{\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}*}$$

where the input z is polynomial in λ, the security parameter.

We can now state the definition of protocol emulation.

Definition 2.2.2 (Definition 5 of [Can13])
Let π, ϕ be PPT protocols. π UC-emulates ϕ if for all PPT adversaries $\mathcal{A}$ there is a PPT adversary $\mathcal{S}$ such that for all balanced[6] *PPT environments $\mathcal{E}$ we have*

$$\mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{E}}.$$

Note that UC-emulation is transitive (i.e. if π_1 emulates π_2 and π_2 emulates π_3, then π_1 emulates π_3).

There is also an alternative way to work with UC-emulation in practice that relies on the idea that if the environment has access to communication in the execution, it is capable of running any adversary by itself. This leads to the definition of a *dummy adversary*, that simply forwards all information to the environment and follows its instructions. The equivalence to the original notion of emulation is contained in the following theorem.

Theorem 2.2.1 (Claim 10 of [Can13])**.** *Let π, ϕ be PPT protocols. Then π UC-emulates ϕ if and only if it UC-emulates ϕ with respect to the dummy adversary $\mathcal{D}$, i.e. there is a PPT adversary $\mathcal{S}$ such that for all balanced PPT environments $\mathcal{E}$ we have*

$$\mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \mathrm{EXEC}_{\pi,\mathcal{D},\mathcal{E}}.$$

To capture execution in an ideal world, we define an *ideal functionality* as an ITM, and the *ideal protocol* $\mathcal{I}_\mathcal{F}$ for an ideal functionality $\mathcal{F}$ as a "wrapper" for the functionality so that an instance of $\mathcal{F}$ syntactically looks like and behaves like an instance of a multi-party protocol. This is achieved by the notion of *dummy parties* that act as placeholders for the parties of the calling protocol instance and simply relay all inputs/outputs. In this setting, messages delivered by $\mathcal{A}$ are ignored and corruption requests have to go through $\mathcal{F}$. A protocol that makes subroutine calls to $\mathcal{I}_\mathcal{F}$ is called an *$\mathcal{F}$-hybrid* protocol.

Definition 2.2.3 (Definition 9 of [Can13])
Let π be a protocol and $\mathcal{F}$ an ideal functionality. π UC-realizes $\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$.

A note on session ids: An *sid* of $\mathcal{F}$ has to be unique, and by convention its *pid* is set to $\perp$ while the identity of each of its dummy parties is of the form (sid, pid')

[6] The environment has to be *balanced* as per the definition in Section 4.2 of [Can13].

for pid' corresponding to the parties of the calling protocol. There are several ways for parties to agree on a *sid*, three of which are discussed in [Can13].

It is also worth noting that the basic model of execution does not capture concepts such as corruption or more involved network types or channels of communication. All of these have to be defined either in terms of special protocol instructions or ideal functionalities. In case of corruption of parties, we will define it when it is relevant to the protocol.

Composition

The operation of UC composition is described with respect to the model of protocol execution[7]. For protocols ρ, ϕ, π such that ρ makes calls to ϕ, the execution with $\rho^{\phi \to \pi}$ is defined as the same as with ρ except the control function invokes instances of π instead of ϕ. When ϕ is the ideal protocol $\mathcal{I}_{\mathcal{F}}$ for some $\mathcal{F}$, we write $\rho^{\pi/\mathcal{F}}$ for the composed protocol.

The general statement of the universal composition theorem is followed by corollaries that conclude the results needed for building composable protocols in the framework.

Theorem 2.2.2 (Theorem 13 of [Can13])**.** *Let ρ, ϕ, π be PPT protocols such that π UC-emulates ϕ and both ϕ and π are subroutine respecting. Then $\rho^{\phi \to \pi}$ UC-emulates ρ.*

Corollary 2.2.1 (Corollary 14 of [Can13])**.** *Let ρ, π be PPT protocols such that π UC-realizes a PPT functionality $\mathcal{F}$ and both $\mathcal{I}_{\mathcal{F}}$ and π are subroutine respecting. Then $\rho^{\pi/\mathcal{F}}$ UC-emulates ρ.*

Corollary 2.2.2 (Corollary 15 of [Can13])**.** *Let $\mathcal{F}$, $\mathcal{G}$ be ideal functionalities such that $\mathcal{F}$ is PPT. Let ρ, π be protocols such that ρ UC-realizes $\mathcal{G}$, π UC-realizes $\mathcal{F}$ and both ρ and π are subroutine respecting. Then $\rho^{\pi/\mathcal{F}}$ UC-realizes $\mathcal{G}$.*

[7] See Chapter 5 of [Can13] for a precise description in terms of the required subroutine structure of composed protocols.

Chapter 3

Zerovote protocol in the real model

We are now ready to define a protocol which aims to address some of the issues described in Section 2.1, namely to fulfill the apparently contradictory goal of decentralized voting that would happen in public view while preserving voter privacy. The resolution of this paradox lies in several key ideas which we will first explain informally, and then give a formal definition of the full protocol and the cryptographic primitives it will use.

However, we have to point out that the term "real model" as we use it here is not entirely correct with respect to the UC framework. This is because already at this level we will work with certain constructs that we won't give an implementation for - the main one being the reliance on an ability to use blockchains anonymously, and hence abstracting the network setup simply as an anonymous broadcast channel.

That is a very broad and simplifying assumption, of course, but it allows us to separate the concerns of the voting protocol from the actual details of computing on blockchains in e.g. Ethereum [Woo14], since the protocol could function independently of the platform as long it runs on an anonymous broadcast channel. We also note that there is ongoing research in formal models of blockchains that could provide the tools needed to obtain a full "real model" (see [BMTZ17] for UC treatment of Bitcoin and [KMS+16] for a UC model of smart contracts).

3.1 Informal overview

We assume that there is a single trusted election authority responsible for registering eligible voters and setting up the election by creating a bulletin board and embedding it with public parameters[1]. Unless specified otherwise, the voters use

[1]Note that while our broad aim is to decentralize voting, in most applications an authority of some sort cannot be entirely removed. The goal is then to minimize the amount of things it is trusted with to only the necessary things (e.g. in national elections the officials still have to confirm the eligible voters).

anonymous means to communicate with the bulletin board. As justified earlier, we abstract this into the notion of an *anonymous broadcast channel* (which is unblockable).

The protocol is scheduled in a sequence of non-overlapping phases, the starts and ends of which are announced on the board by the election authority:

1. **Setup**
   The election authority submits the list of eligible voters to the bulletin board. Anyone can read the list, the election schedule or parameters.

2. **Credential generation**
   Before the actual voting starts, voters locally generate voting credentials (unique tokens) and publish them signed on the bulletin board. The purpose of this step is to allow eligible voters to obtain a credential that will let them vote anonymously later. Anyone will be able to verify the signatures and compile a list of valid credentials.

3. **Vote commitment**
   When a voter decides on a choice he would like to give his vote to, he publicly commits to it without disclosing the vote and uses his credential to produce a proof of his eligibility without disclosing his identity or the specific credential. This step makes crucial use of two constructs: commitment schemes, which allow one to bind to a value while the value remains hidden until revealed at a later time, and secure accumulators, which make it possible to prove ownership of an item in a given set without disclosing which item it is.

4. **Ballot opening**
   When new ballots (i.e. commitments to votes) are no longer accepted, voters can anonymously reveal the votes they committed to earlier for them to be counted. Anyone can compute the result (partial or final) from the bulletin board by tallying the votes corresponding to valid commitments and proofs of eligibility.

To give a formal treatment of the protocol, it is necessary to first describe the cryptographic primitives it is composed of, which we do in the next section.

3.2 Cryptographic primitives

We follow the paper that introduces Zerocoin [MGGR13] by making use of their construction for a decentralized e-cash scheme, where minting a coin will correspond to generating a voting credential and spending the coin will represent committing to a vote.

Their construction depends on strong RSA accumulators [CL02], and coins (which we will call credentials from now on) are realized as commitments to unique serial numbers using Pedersen's commitment scheme [Ped91].

Their zero-knowledge signature of knowledge scheme proves the knowledge of two objects: an accumulated credential and its corresponding serial number. The first proof is based on the protocol described in [CL02] and converted to a non-interactive form via the Fiat-Shamir heuristic. The latter is a new signature of knowledge (referring to the definition in [CL06]), the construction of which is described in [MGGR13] but without an actual proof given.

Since proofs in the random oracle model which require rewinding are not possible in the UC framework, and [CL06] already provides a UC-realizable general definition of a signature of knowledge for any language, we opt for their construction in place of the one from Zerocoin.

We also make use of a commitment scheme in constructing ballots. A digital signature scheme is required in order to restrict participation in the protocol to eligible voters.

3.2.1 Digital signatures

The registration process for voting traditionally makes use of a channel through which the voters can identify themselves to the election authority (achieved by various means). We want to capture as wide a setting as possible, so in principle it does not matter how the initial registration happens as long as at the end of it the election authority knows the identities of eligible voters and the voters know the identity of the election authority.

It would be natural to utilize a form of public-key infrastructure, but modeling it in UC is not straightforward, and turns out to not be necessary for the purposes of this protocol. What we require is either signatures (corresponding to signing and verification key-pairs, not necessarily tied to identities) and an authenticated channel through which the verification keys can be distributed, or signatures combined with certificates confirming the identity of the signer, i.e. a certification scheme. (We will give more detail on this in Section 4.1.3.)

A signature scheme is composed of polynomial-time algorithms for key generation, signing and verification. We refer to the standard definition of *existential unforgeability against chosen message attacks* as first given in [GMR88] and summarized in [Can13] (2005 version[2]):

Definition 3.2.1
A signature scheme $\Sigma = (\mathtt{Gen}, \mathtt{Sig}, \mathtt{Ver})$ *is called* EU-CMA *if the following properties hold for any negligible function* $\mathsf{negl}(\lambda)$*, and all large enough values of the security parameter* 1^λ*:*

1. ***Completeness***
   For any message m*,*
   $\Pr[(\bar{s}, \bar{v}) \leftarrow \mathtt{Gen}(1^\lambda);\ \sigma \leftarrow \mathtt{Sig}(\bar{s}, m);\ 0 \leftarrow \mathtt{Ver}(m, \sigma, \bar{v})] < \mathsf{negl}(\lambda)$.

[2]The chapter on defining functionalities is not present in the latest version of the UC paper, so we sometimes refer to the 2005 version.

2. ***Consistency (Non-repudiation)***
   For any m,
   $\Pr[(\bar{s},\bar{v}) \leftarrow \mathtt{Gen}(1^\lambda);\ \sigma \leftarrow \mathtt{Sig}(\bar{s},m);$
   $b \leftarrow \mathtt{Ver}(m,\sigma,\bar{v})$ *and* $b' \leftarrow \mathtt{Ver}(m,\sigma,\bar{v})$ *such that* $b \neq b'] < \mathsf{negl}(\lambda)$.

3. ***Unforgeability***
   For any PPT forger F,
   $\Pr[(\bar{s},\bar{v}) \leftarrow \mathtt{Gen}(1^\lambda);\ (m,\sigma) \leftarrow F^{\mathtt{Sig}(\bar{s},\cdot)}(\bar{v});\ 1 \leftarrow \mathtt{Ver}(m,\sigma,\bar{v})$
   and F never asked $\mathtt{Sig}$ *to sign* $m] < \mathsf{negl}(\lambda)$.

Informally, these 3 properties assure the following: that a signature generated with the correct signing key will verify under the corresponding (public) verification key, that verifying a given signature will give the same result every time, and that an attacker will not be able to construct a message and a forged signature that would verify without knowledge of the signing key.

We do not give a specific implementation as there are many in use and it does not impact the analysis of the protocol.

3.2.2 Commitments

A non-interactive commitment scheme can be defined by algorithms for setup, commitment generation and verification (a *non-interactive protocol* presumes the algorithms are executed locally and not in collaboration with other parties, though global parameters may need to be shared at the start).

The security of the scheme ($\mathtt{CSetup}, \mathtt{Com}, \mathtt{VfCom}$) (we follow the format of [CDR16] used in later chapters) is analyzed in terms of two properties:

- *Binding.* There are no messages $m \neq m'$ such that $\mathtt{Com}(m) = \mathtt{Com}(m')$, i.e. each commitment can only be opened to one committed value.
- *Hiding.* No information about m is leaked by $\mathtt{Com}(m)$.

  Trapdoor. If a secret trapdoor associated with the scheme's parameters is known, a commitment can be created such that it opens to any message. This implies the hiding property [Gro09].

The commitment algorithm also generates a random nonce which is often called the *opening*[3], as it has to be revealed for the commitment to be verified.

Both properties can achieve different levels of security: *perfect*, *statistical* and *computational*, in the usual meanings. We will use Pedersen's commitment scheme [Ped91] which is perfectly hiding with a trapdoor and computationally binding assuming the hardness of the discrete log problem. (Note that a scheme that is perfectly hiding and perfectly binding is impossible.)

[3] In some commitment schemes the nonce is taken as input into the algorithm rather than being generated by it, but this does not influence the properties of the scheme.

The pseudocode for an implementation of this scheme can be found in Appendix A.

3.2.3 Secure accumulators

As briefly mentioned in the informal overview of the protocol, the purpose of secure accumulators is to provide an object that can represent a set and provide witnesses for specific items being in the set (the primitive itself cannot give us the anonymous ownership proof that we need, but we will combine it with zero-knowledge proofs of knowledge in the following section).

We first give the general definition of a secure accumulator (separated into 2 parts for clarity) from [CL02].

Definition 3.2.2 (Accumulator)
Let 1^λ *be the security parameter.*

SA $= (\mathcal{X}_\lambda, \mathtt{Gen})$ *is an accumulator scheme for a family of inputs* $\{\mathcal{X}_\lambda\}$ *if it has the following properties:*

1. ***Efficient generation***
   $\mathtt{Gen}$ *is an efficient probabilistic algorithm such that* $\mathtt{Gen}(1^\lambda) \to (f, aux_f)$ *for random* $f \in F_\lambda$.
2. ***Efficient evaluation***
   $f \in F_\lambda$ *is a polynomial-size circuit such that* $f(u, x) \to v$ *for* $(u, x) \in \mathcal{U}_f \times \mathcal{X}_\lambda$ *and* $v \in \mathcal{U}_f$ *where* $\mathcal{U}_f$ *is efficiently samplable.*
3. ***Quasi-commutative***
   For all $f \in F_\lambda, u \in \mathcal{U}_f$ *and* $x_1, x_2 \in \mathcal{X}_\lambda$*:* $f(f(u, x_1), x_2) = f(f(u, x_2), x_1)$.
   Hence $f(u, X)$ *for* $X = \{x_1, \ldots, x_m\} \subset \mathcal{X}_\lambda$ *denotes* $f(f(\ldots(u, x_1), \ldots), x_m)$.
4. ***Witnesses***
   $w \in \mathcal{U}_f$ *is a witness for* $x \in \mathcal{X}_\lambda$ *in the accumulator* $v \in \mathcal{U}_f$ *under* f *if* $v = f(w, x)$.

Definition 3.2.3 (Secure accumulator)
SA $= (\mathcal{X}_\lambda, \mathtt{Gen})$ *is a secure accumulator scheme if it also satisfies:*

5. ***Witness unforgeability***
   Let $\mathcal{U}'_f \times \mathcal{X}'_\lambda$ *denote the domains for which the computational procedure for function* $f \in F_\lambda$ *is defined (so* $\mathcal{U}_f \subseteq \mathcal{U}'_f, \mathcal{X}_\lambda \subseteq \mathcal{X}'_\lambda$*).*
   For all PPT adversaries $\mathcal{A}_\lambda$*:*

$$\Pr[f \leftarrow \mathtt{Gen}(1^\lambda); u \leftarrow \mathcal{U}_f; (x, w, X) \leftarrow \mathcal{A}_\lambda(f, \mathcal{U}_f, u) : X \subset \mathcal{X}_\lambda; w \in \mathcal{U}'_f; x \in \mathcal{X}'_\lambda; x \notin X; f(w, x) = f(u, X)] = \mathsf{negl}(\lambda)\,.$$

In the Zerovote protocol we will make use of a specific scheme, called the strong RSA accumulator, that implements the above definition (pseudocode is given in

Figure A.2 in Appendix A). It was shown in [CL02] that SA_{RSA} is a secure accumulator under the strong RSA assumption.

Definition 3.2.4 (Strong RSA accumulator)
Let $\text{SA}_{\text{RSA}} = (\mathcal{X}_{A,B}, \texttt{GenRSA})$ *where:*

- $\texttt{GenRSA}(1^\lambda)$ *generates a random* $n = pq$ *of length* λ *where* p, q *are safe primes, i.e.* $p = 2p' + 1$, $q = 2q' + 1$ *for* p', q' *prime.*
  Then the function $f = f_n$ *can be defined as* $f(u, x) = u^x \mod n$ *with* $\mathcal{U}_f = \{u \in \text{QR}_n : u \neq 1\} \subseteq \mathcal{U}'_f = \mathbb{Z}_n^*$.
- $\mathcal{X}_{A,B} = \{e \text{ prime } | e \neq p', q' \wedge A \leq e \leq B\}$ *for* A, B *with arbitrary polynomial dependence on* λ *as long as* $2 < A$ *and* $B < A^2$, *and* $\mathcal{X}'_{A,B}$ *is any subset of* $\{x \in \mathbb{Z} : 2 \leq x \leq A^2 - 1\}$ *such that* $\mathcal{X}_{A,B} \subseteq \mathcal{X}'_{A,B}$.

As in the commitment scheme, with multiple parties in the protocol sharing the parameters we need a setup algorithm that will generate them in a secure way (in our case this will be the role of the trusted election authority).

3.2.4 Signatures of knowledge

The final component of the protocol will bring together several related concepts in cryptography, so we will describe them in order.

First, a *proof of knowledge* is an interactive protocol that enables one party (called the prover) to convince another party (the verifier) that he knows a certain value x. This is normally encoded in two properties: *completeness*, meaning that a prover who does have knowledge of x will always convince the verifier, and conversely *soundness (validity)*, meaning that a prover without the knowledge of x cannot convince the verifier[4].

A *zero-knowledge proof* considers proving some statement in general, but without revealing anything other than the fact that the prover knows the statement in question. Thus a zero-knowledge proof of knowledge has a third property called *zero-knowledge*, which is normally defined with respect to a simulator (and in terms of probability distributions can be perfect, statistical or computational).

We have already defined the notion of a *non-interactive* protocol. It can be shown that non-interactive zero-knowledge proofs are impossible in the so-called standard model, and so they require additional assumptions such as the random oracle model (ROM) or the common reference string (CRS).

Finally, a *signature of knowledge* (as defined by [CL06]) is a specialized version of a non-interactive zero-knowledge proof of knowledge. It functions much like a normal signature, except that instead of binding signatures to public keys, it allows anyone with knowledge of a given value to sign. We say that the signer holds a secret *witness* to a given statement from a given language.

[4]Certain computational assumptions are often used in the formal definitions, but we do not require the details in this case.

Pseudocode for a generic construction (also given by [CL06]) can be found in Figure A.3 in Appendix A, which will be used in ZEROVOTE to provide a signature of knowledge on a ballot that proves knowledge of a valid credential.

3.3 Formal specification

We first formally describe the basic setting of the protocol.

Parties: Trusted election authority **EA**, public bulletin board **BB**[5], n voters $\mathbf{V_i}$ out of which at most t are corrupted by the adversary $\mathcal{A}$.

Inputs: Common security parameter 1^λ. Each $\mathbf{V_i}$ is given a choice o_i corresponding to the candidate they wish to vote for[6] and **EA** is given the list of eligible voters E.

Outputs: The tally $T = (o_i)$ computed by **BB**.

Registration: We assume that each eligible voter $\mathbf{V_j}$ is given (id_j, sk_j) representing their public identity and a secret key, **EA** is given their own pair (id_{EA}, sk_{EA}) and the list (id_j) corresponding to eligible voters in E, and all voters receive id_{EA}. This happens in an authenticated manner, e.g. using a physical channel (such as when voters physically go to register at a predetermined place).

Communication channel: Assuming an untrusted network controlled by $\mathcal{A}$, we require two additional properties from our communication channel: *anonymity* (the receiving party does not learn the identity of the sending party) and a *broadcast* (the messages are sent to all parties)[7].

Authentication: If a message m is signed, we indicate this by placing it within $\{m\}_P$ where P is the signer. **EA** signs all its messages to **BB**.

Scheduling: The protocol is executed in well-defined stages with no time overlap between them, which is enforced via begin/end commands given by **EA** and published on **BB** (we will not list them explicitly in the protocol).

Revote policy: Only the first valid vote counts.

Next, we go over the stages of the protocol in order, giving pseudocode for the relevant functions in the stage where they are used or referring to schemes described in Section 3.2 and Appendix A.

[5]The bulletin board is in this case more of a useful abstraction rather than a necessary entity. So when we refer to the actions of **BB**, what we really mean is that they can be performed by any party, since its purpose is only to collect all broadcasted messages and verify them using public data.

[6]We impose no initial restrictions on the type or format of o_i (i.e. it could be invalid).

[7]This means that the public view of the protocol is essentially an ordered list of messages without identities of the senders.

1. Setup

In the first stage, **EA** initializes **BB** with public parameters via an `Init` function that we omit here, since we are not concerned with the medium on which the protocol is implemented as long as it provides an anonymous broadcast channel.

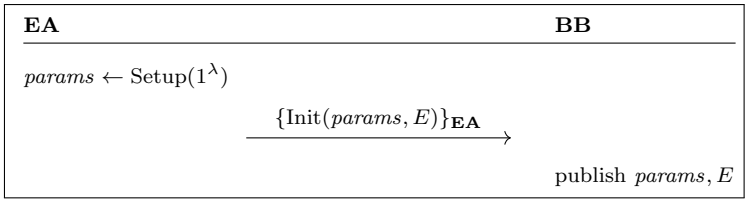


Figure 3.1: Setup stage

`Setup` generates parameters for the accumulator, commitment and signature of knowledge schemes. The parameters provided by `CSetup` will be used by $\mathbf{V_i}$ to construct both the voting credential and the ballot in the next stages.

Setup(1^λ)

1 : $(N, u, \mathcal{X}_{A,B}) \leftarrow \text{ASetup}(1^\lambda)$
2 : $(p, q, g, h) \leftarrow \text{CSetup}(1^\lambda)$
3 : $\rho \xleftarrow{R} \{0,1\}^\lambda$
4 : **return** $((N, u, \mathcal{X}_{A,B}), (p, q, g, h), \rho)$

2. Credential generation

The second stage proceeds as follows:

1. $\mathbf{V_i}$ locally computes a *voting credential* c and its *secret* $sk_c = (S, r)$ using the public parameters from **BB**.
2. $\mathbf{V_i}$ sends c (signed with their private key sk_i) to **BB**, which will add c to the *set of credentials* C and id_i to the *set of voters with credentials* E_c if the signature verifies and the (eligible) voter with $id_i \in E$ had not already sent a credential. Note that anyone can perform the same checks as **BB**.

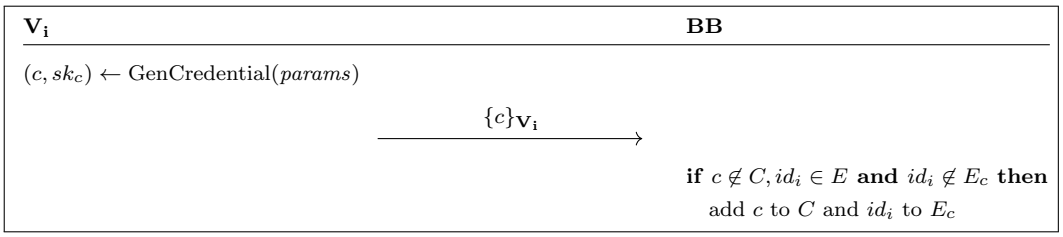


Figure 3.2: Credential generation stage

`GenCredential` generates a credential c as a commitment to a serial number S with a random opening r, which are kept secret until the voting stage. The commitment functions `Com`, `VfCom` are given according to Figure A.1. Any EU-CMA secure digital signature scheme (`Gen`, `Sig`, `Ver`) can be used for signing as long as the verification keys can be tied to the voters' identities via some certification authority.

GenCredential(*params*)

1 : $\mathcal{X}_{A,B}, (p, q, g, h) \leftarrow params$
2 : **repeat**
3 : $\quad S \xleftarrow{R} \mathbb{Z}_q$
4 : $\quad (c, r) \leftarrow \text{Com}(params, S)$
5 : **until** $c \in \mathcal{X}_{A,B}$
6 : **return** $(c, (S, r))$

Note that this is the only stage where voters identify themselves to **BB**.

3. Vote commitment

The third stage continues as follows:

1. All $\mathbf{V_i}$ can retrieve the set of credentials C from **BB**.
2. $\mathbf{V_i}$ computes a *ballot* x_i representing a commitment to a *choice* o_i[8].
3. $\mathbf{V_i}$ constructs a zero-knowledge *signature of knowledge* σ on x_i on behalf of knowing the secret sk_c to one of the credentials in C.
4. $\mathbf{V_i}$ sends x_i and (σ, S) (for the credential *serial number* S in $sk_c = (S, r)$) to **BB**. Then **BB** verifies σ, adds S to the *set of used serial numbers* N and adds x_i to the *set of committed ballots* B_c.

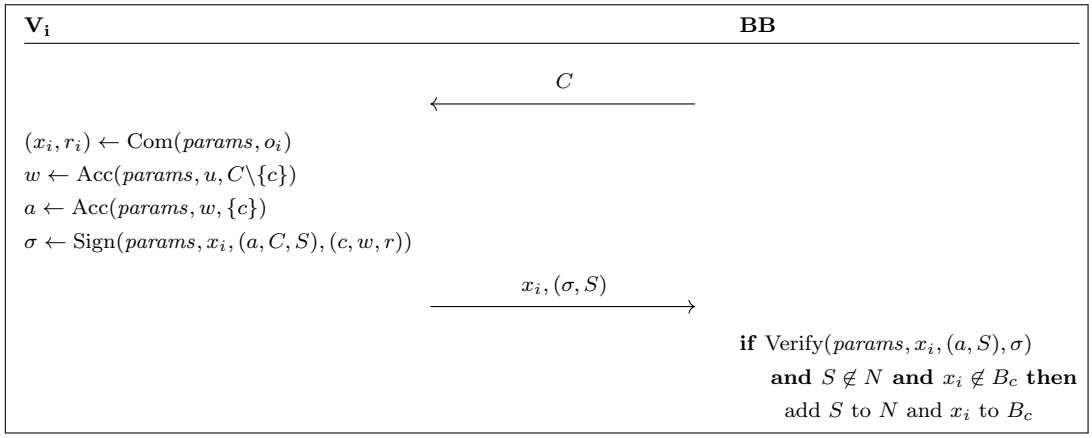


Figure 3.3: Vote commitment stage

The functions `Acc`, `VfAcc` are given according to the strong RSA accumulator scheme of [CL02] with pseudocode in Figure A.2, and the functions `Sign` and `Verify` come from the signature of knowledge of [CL06] given in Figure A.3 in Appendix A.

4. Ballot opening

The final stage proceeds as follows:

1. $\mathbf{V_i}$ reveals their vote o_i by sending x_i and the committed values to **BB**, which verifies the commitment, adds x_i to the *set of open ballots* B_o and updates the *tally* T with o_i.
2. Anyone can call **BB** to output the tally T. If instructed so, **BB** can at this point perform validity checks on the individual o_i contained in T and compute a result function on the set.

[8] For honest voters, we assume a suitable encoding so that $o_i \in \mathbb{Z}_q$.

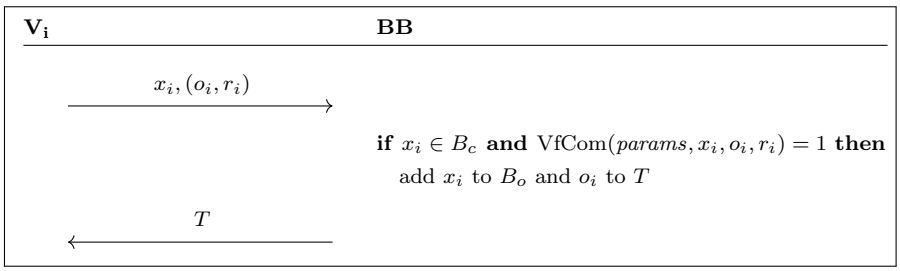
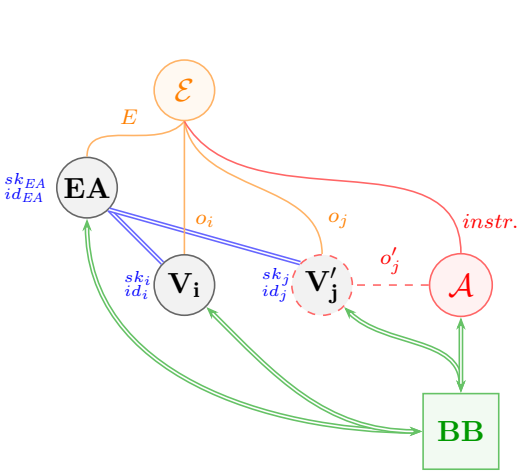


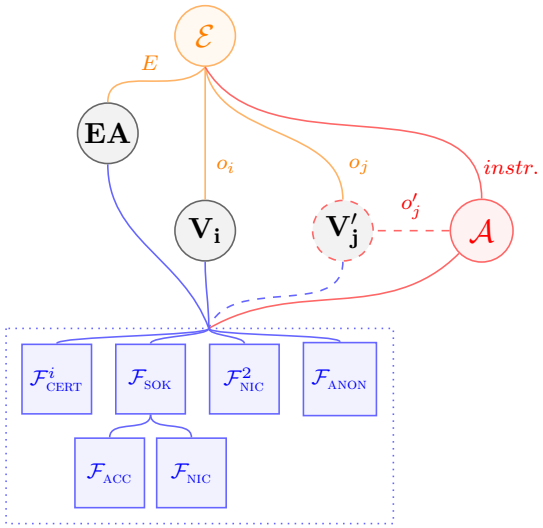
Figure 3.4: Ballot opening stage

3.4 Next steps

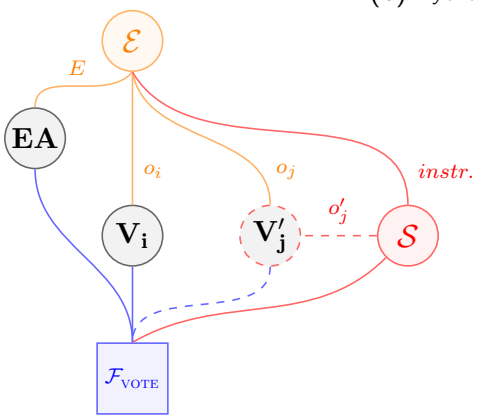
Now we could define an ideal functionality for privacy of the protocol and attempt to prove it, but instead we will take a more roundabout route to make use of the modularity that the UC framework provides in order to obtain a simpler proof. We will proceed in two parts, first defining a hybrid protocol that is indistinguishable from ZEROVOTE and only then giving the ideal functionality.



(a) *Real execution with* ZEROVOTE



(b) *Hybrid execution with* Π_{ZV}



(c) *Ideal execution with* $\mathcal{F}_{\text{VOTE}}$

Figure 3.5: The real vs. hybrid vs. ideal world.

As shown in Figure 3.5, all three protocols will have to share a certain structure (i.e. the format of the inputs and outputs of the parties will have to match) so that from the point of view of the environment, the interaction is the same.

Internally, however, while the voters in ZEROVOTE will interact using a real anonymous broadcast channel (represented by the bulletin board), in Π_{ZV} they will use an ideal functionality in its place. The composition theorem will allow us to make similar substitutions for all the cryptographic schemes that ZEROVOTE uses. Indistinguishability of this hybrid protocol Π_{ZV} from $\mathcal{F}_{\text{VOTE}}$ will then guarantee that privacy is preserved.

Chapter 4

Building the hybrid model

In chapters 1 and 2 we have given a brief outline of our strategy for proving that Zerovote preserves voter privacy, which will consist of an intermediate step in the form of a hybrid protocol indistinguishable from the real one due to the UC composition theorem. In this chapter, we define all the necessary ideal functionalities for the cryptographic primitives (described in Section 3.2) that the protocol uses and compose them to build the hybrid model.

We will use the notation for parties of the protocol established in Chapter 3 and assume they are represented by ITMs as in the real execution, except with certain local subroutines replaced by calls to ideal functionalities.

We will need the following for the basic setting of the protocol:

- anonymous broadcast channel
- common reference string

We will also need functionalities for the following schemes:

- digital signatures
- commitments
- secure accumulators
- signatures of knowledge

4.1 Known ideal functionalities for primitives

In this section, we briefly describe and recount relevant results with respect to realizability of ideal functionalities for the primitives that Zerovote uses and that have been previously defined in the literature. (This comprises all in the list above with the exception of secure accumulators.)

4.1.1 Anonymous broadcast channel ($\mathcal{F}_{\text{anon}}$)

As noted at the beginning of Chapter 3, the real protocol is not defined with respect to a specific implementation of an anonymous broadcast channel, hence already being "hybrid" in nature. Nevertheless, we make its properties explicit by defining a simple functionality that fulfills its purpose, in line with the UC design norm which separates the bare computational model from assumptions about the network and codifies all the desired properties of a given channel in the form of an ideal functionality (see the discussion at the beginning of Chapter 6 of [Can13]).

Hence we define $\mathcal{F}_{\text{ANON}}$ as the medium through which all the messages meant for the bulletin board must go in order to be stripped of the sender's identity and broadcasted to all parties, including the adversary $\mathcal{A}$. It requires an initial `Setup` command so that parties can register to send and receive messages.

Functionality $\mathcal{F}_{\text{anon}}$:

- **Setup**
  - On input $(\texttt{Setup}, sid)$ from party P:
    1. Add P to the set $\mathbb{P}$.
    2. Output $(\texttt{Setup}, sid)$ to P and $\mathcal{A}$.
- **Communication channel**
  - On input $(\texttt{Post}, sid, m)$ from party P:
    1. If $P \notin \mathbb{P}$, abort.
    2. Output $(\texttt{Post}, sid, m)$ to all parties in $\mathbb{P}$ and to $\mathcal{A}$.

There are many existing definitions of functionalities for bulletin boards, but we choose to define ours purely in terms of the properties of the channel for simplicity.

4.1.2 Common reference string model ($\mathcal{F}_{\text{crs}}$)

Realizing some functionalities in the UC framework requires additional setup assumptions (as without them there are some impossibility results). One common type of setup is achieved by the *common random string* model, which captures the idea of a shared source of randomness and is represented by a single random draw from a uniform distribution over strings that is saved and can be then requested by any participating party. The *common reference string* is then a generalization to arbitrary distributions which can be over parameters that need to be shared in some multi-party protocol.

Note that the so-called CRS model is different from the notion of a *random oracle*, a construction that is most often used in the standalone model and represents an

ideal hash function.

A functionality for the common reference string is formally defined below (with a small change from the 2005 version of [Can13][1]). Here a *public delayed output* means that the functionality first hands the output and the identity of the party to the adversary and only sends the output to the party after a confirmation from the adversary.

Functionality $\mathcal{F}_{\text{crs}}$ parametrized by distribution D:

- On input $(\texttt{CRS}, sid)$ from party P:
  1. If no value r is recorded, choose random $r \xleftarrow{R} D$ and record it.
  2. Send a public delayed output $(\texttt{CRS}, sid, r)$ to P.

The functionalities we will use for commitments and accumulators are UC-realizable in the $\mathcal{F}_{\text{CRS}}$-hybrid model (it is necessary for any commitment functionality, as shown in [CF01]), which means that the protocols that realize them can request the common reference string from the given distributions over the schemes' parameters generated by $\mathcal{F}_{\text{CRS}}$ upon first activation.

Realizing $\mathcal{F}_{\text{CRS}}$ is by itself not a trivial task, so we do not choose a specific protocol. However, we note the efforts to relax the definition to allow for more practical implementations, e.g. [CPS07].

4.1.3 Digital certificates ($\mathcal{F}_{\text{cert}}$)

In the description of the real protocol, we have taken the liberty of using a signature scheme without precisely clarifying how the public verification keys would be distributed. Intuitively, it seems that we would require an authenticated channel between all the parties and the election authority to achieve this, but there is actually a simpler way: to use a *certification scheme* which provides signatures not bound to keys, but to *identities*.

$\mathcal{F}_{\text{CERT}}$ as defined by [Can13] (2005) provides commands for signature generation and verification, and is tied to a single party (so that in our voting protocol we will need a separate instance for each voter). To capture allowed adversarial influence, it obtains the signing algorithm and the verification algorithm from the adversary, but it makes sure that the algorithms are *consistent* and *complete* by keeping records of generated message-signature pairs and aborting if it detects a violation. Crucially, it does not allow the adversary to forge a valid signature for an honest party that never generated it.

[1] The difference from [Can13] is in dropping their restriction to a fixed set of participating parties. In our setting this is neither necessary (the functionalities we need it for don't require the stronger version) nor desirable (we want to model parties on an unrestricted public network), even though it means we are technically working with a weaker version.

Functionality $\mathcal{F}_{\text{cert}}$:

- **Setup**
  - On input $(\texttt{Setup}, sid)$ from party P:

    If $sid = (P, sid')$ for some sid', hand $(\texttt{Setup}, sid)$ to $\mathcal{A}$, else ignore.
  - On input $(\texttt{Algorithms}, sid, \texttt{Verify}, \texttt{Sign})$ from $\mathcal{A}$ for DPT ITM `Verify` and PPT ITM `Sign`:

    Store the algorithms and output $(\texttt{Setup}, sid)$ to P.
- **Signature generation**
  - On input $(\texttt{Sign}, sid, m)$ from party P:
    1. Set $\sigma \leftarrow \texttt{Sign}(m)$.
    2. If $\texttt{Verify}(m, \sigma) \neq 1$, abort.
    3. Record (m, σ) and output $(\texttt{Signature}, sid, m, \sigma)$ to P.
- **Signature verification**
  - On input $(\texttt{Verify}, sid, m, \sigma)$ from party V:
    1. If $\texttt{Verify}(m, \sigma) = 1$, the signer is not corrupted and no entry (m, σ') for any σ' is recorded, abort.
    2. Output $(\texttt{Verified}, sid, m, \texttt{Verify}(m, \sigma))$ to V.

Without going into the full details, we simply state the result of [Can04] and note that it can be adapted for the functionality we are using from [Can13][2].

Theorem 4.1.1. *$\mathcal{F}_{\text{CERT}}$ can be UC-realized in the ($\mathcal{F}_{\text{SIG}}$,$\mathcal{F}_{\text{CA}}$)-hybrid model.*

Note that the hybrid protocol of [Can04] realizing $\mathcal{F}_{\text{CERT}}$ can be fully realized with an EU-CMA secure signature scheme as defined in Section 3.2.1 and with **EA** or other trusted party acting as the certificate authority. See Appendix B for a discussion about the choice of $\mathcal{F}_{\text{CERT}}$ as opposed to an authenticated channel functionality $\mathcal{F}_{\text{AUTH}}$, as well as for definitions of $\mathcal{F}_{\text{SIG}}$ and $\mathcal{F}_{\text{CA}}$.

4.1.4 Commitments ($\mathcal{F}_{\text{nic}}$)

We will use the functionality $\mathcal{F}_{\text{NIC}}$ defined in [CDR16] for non-interactive commitments, since it seems to be the only definition in the literature that is non-

[2] [Can13] (2005) does not redefine $\mathcal{F}_{\text{CERT}}$ either, only noting that it is possible. While a fully rigorous treatment is always preferable, the issues presented here do not impact the voting protocol much and we therefore opt to give more focus to the problems that do not relate to registration (since voter registration is not an issue specific to Zerovote). An interesting potential solution that tries to realistically model a global PKI is explored in [CSV16].

interactive, multi-party and meant for modular use. We adapt the syntax to be compatible with our other definitions: we change the names of interfaces and certain arguments, and replace the `Validate` command with a request for the verification algorithm (so it does not need to be provided alongside the commitment, since the verification in the original formulation aborted if given a different algorithm).

The functionality keeps a record of parameters, a set of participants $\mathbb{P}$ and a table of the format $[commitment, message, opening, v]$ for validity bit v, all initialized to empty upon first activation. It requests algorithms and parameters from the adversary, which it uses to compute the commitment and opening values – however the commitment generation algorithm `TrapCom` does not take the message *msg* as input and so cannot leak anything about it, and `Verify` is required to be consistent with what the other algorithms generate. $\mathcal{F}_{\text{NIC}}$ then ensures the binding of messages to commitments via its records.

Note that it also makes use of *public delayed output*, which as we mentioned previously means that the functionality first hands a note with the output to $\mathcal{A}$, and only sends the output to P once it receives confirmation from $\mathcal{A}$ to proceed.

Functionality $\mathcal{F}_{\text{nic}}$ parametrized by system parameters *sp*:

- **Setup**
  - On input (`Setup`, *sid*) from party P:

    If (*sid*, *params*, *trapdoor*, *algs*) already stored:

    1. Include P in the set $\mathbb{P}$.
    2. Send a delayed output (`Parameters`, *sid*, *params*)[a] to P.

    Else:

    1. Generate random *ssid* and store (*ssid*, P).
    2. Send (`RequestAlgorithms`, *sid*, *ssid*) to $\mathcal{A}$.
  - On input (`Algorithms`, *sid*, *ssid*, *params*, *trapdoor*, `TrapCom`, `TrapOpen`, `Verify`[b]) from $\mathcal{A}$:
    1. If no (*ssid*, P) stored for some P, abort.
    2. Delete (*ssid*, P).
    3. If (*sid*, *params*, *trapdoor*, *algs*) not already stored, store the tuple.
    4. Include P in the set $\mathbb{P}$.
    5. Send a delayed output (`Parameters`, *sid*, *params*) to P.
  - On input (`VerificationAlgorithm`, *sid*) from party P:

Output (`VerificationAlgorithm`, *sid*, `Verify`(*sid*, *params*, ·)) to P.

- **Commitment generation**
  - On input (`Commit`, *sid*, *msg*) from party P:
    1. If $P \notin \mathbb{P}$ or $msg \notin M$ (for M defined in *params*), abort.
    2. Compute $(comm, info) \leftarrow$ `TrapCom`$(sid, params, trapdoor)$.
    3. If there is an entry $[comm, msg', open', 1]$ with $msg \neq msg'$, abort.
    4. Compute $open \leftarrow$ `TrapOpen`$(sid, msg, info)$.
    5. If `Verify`$(sid, params, comm, msg, open) \neq 1$, abort.
    6. Record the entry $[comm, msg, open, 1]$.
    7. Send (`Commitment`, *sid*, *comm*, *open*) to P.
- **Commitment verification**
  - On input (`Verify`, *sid*, *comm*, *msg*, *open*) from party P:
    1. If $P \notin \mathbb{P}$, $msg \notin M$ or $open \notin R$ (for M, R defined in *params*), abort.
    2. If there is an entry $[comm, msg, open, u]$, set $v \leftarrow u$.

       Else:

       If there is $[comm, msg', open', 1]$ with $msg \neq msg'$:
       Set $v \leftarrow 0$.

       Else:
       Set $v \leftarrow$ `Verify`$(sid, params, comm, msg, open)$.
    3. Send (`Verified`, *sid*, v) to P.

[a] Note that this output was (`Setup`, *sid*) in the original formulation, but explicitly sending the (public) parameters to P does not impact the security properties of $\mathcal{F}_{\text{NIC}}$.

[b] In the original definition, `Verify` is a probabilistic algorithm. For compatibility with our signature of knowledge, we restrict it to be deterministic, and further simplify the commitment verification function since $\mathcal{F}_{\text{NIC}}$ no longer needs to record adversarially-generated commitments to ensure consistency at the end of Step 2.

This functionality can be realized by Pedersen's commitment scheme, as shown more generally in [CDR16] (we give Theorem 2 from Section 3.4), using a protocol Π_{NIC} we do not repeat here.

Theorem 4.1.2. *Π_{NIC} UC-realizes $\mathcal{F}_{\text{NIC}}$ in the $\mathcal{F}_{\text{CRS}}^{\texttt{CSetup}}$-hybrid model if the underlying commitment scheme (`CSetup`, `Com`, `VfCom`) is binding and trapdoor.*

As before, note that each copy of the protocol which realizes it will require its own copy of $\mathcal{F}_{\text{CRS}}$.

4.1.5 Signatures of knowledge ($\mathcal{F}_{\text{sok}}$)

We will now skip ahead and assume that we have a functionality $\mathcal{F}_{\text{ACC}}$ for secure accumulators, treating it as a black box for the purposes of this part and deferring our definition and proof of its realizability to the next section.

Referring back to the definitions given in Section 3.2.4, we are interested in (non-interactive zero-knowledge) signatures of knowledge of a specific function of accepting inputs to previously defined $\mathcal{F}_{\text{NIC}}$ and $\mathcal{F}_{\text{ACC}}$, since we wish to construct signatures proving knowledge of an accumulated credential that is a commitment to a certain value. To this end we extend the functionality described in [CL06] (Section 4.2), which was defined for only one functionality but is easily generalized.

Both $\mathcal{F}_{\text{NIC}}$ and $\mathcal{F}_{\text{ACC}}$ have to satisfy the definition of an *explicit verification functionality* [CL06], since:

- They have to be initialized by a `Setup` query, during which they obtain a deterministic polynomial-time verification algorithm `Verify` from $\mathcal{A}$.
- To (`Verify`, *sid*, *input*, *witness*) queries they either respond with the output of `Verify`(*input*, *witness*) or halt with an error.
- An additional query (`VerificationAlgorithm`, *sid*) was easily added to them, in which they answer any party P by returning the verification algorithm `Verify`.

We do not go over the details of $\mathcal{F}_{\text{SOK}}$ but note the resemblance to the certification functionality defined earlier (with a similar reasoning for how consistency and completeness are achieved), and the *zero-knowledge* property that is assured by the signing algorithm not taking the secret values (witnesses) as input.

Functionality $\mathcal{F}_{\text{sok}}$ ($\mathcal{F}_{\text{nic}}$, $\mathcal{F}_{\text{acc}}$):

- **Setup**
  - On first input (`Setup`, *sid*) from party P:

    If $sid = (\mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{ACC}}, sid_{\text{NIC}}, sid_{\text{ACC}}, sid')$ for some sid_{NIC}, sid_{ACC}, sid':

    i. Set $\mathcal{M}_{\text{NIC}} \leftarrow$ `GetLanguage`$(\mathcal{F}_{\text{NIC}}, sid_{\text{NIC}})$.

    ii. Set $\mathcal{M}_{\text{ACC}} \leftarrow$ `GetLanguage`$(\mathcal{F}_{\text{ACC}}, sid_{\text{ACC}})$.

    iii. Define $\mathcal{M}_L((a, S), (c, w, r)) = \mathcal{M}_{\text{ACC}}((a, \{c\}), w) \wedge \mathcal{M}_{\text{NIC}}((c, S), r)$.

    iv. Hand (`Setup`, *sid*) to $\mathcal{A}$.

> $\texttt{GetLanguage}(\mathcal{F}_0, sid_0)$:
>
> * Create an instance of $\mathcal{F}_0^w$ ith session id sid_0.
> * Send $(\texttt{Setup}, sid_0)$ and $(\texttt{VerificationAlgorithm}, sid_0)$ to $\mathcal{F}_0$ on behalf of P.
> * Receive $(\texttt{VerificationAlgorithm}, sid_0, \mathcal{M})$ and output $\mathcal{M}$.

– On input $(\texttt{Algorithms}, sid, \texttt{Verify}, \texttt{Sign}, \texttt{SimSign}, \texttt{Extract})$ from $\mathcal{A}$ for DPT ITM `Verify` and the rest PPT ITMs:

Store the algorithms, output $(\texttt{Algorithms}, sid, \texttt{Sign}(\mathcal{M}_L, \cdot, \cdot, \cdot), \texttt{Verify}(\mathcal{M}_L, \cdot, \cdot, \cdot))$ to P.

- **Signature generation**
  - On input $(\texttt{Sign}, sid, m, (a, S), (c, w, r))$ from party P:
    1. If $\mathcal{F}_{\text{ACC}}$ accepts $(\texttt{Verify}, sid_{\text{ACC}}, a, \{c\}, w)$ and $\mathcal{F}_{\text{NIC}}$ accepts $(\texttt{Verify}, sid_{\text{NIC}}, c, S, r)$ when queried by P:
       i. If $\mathcal{M}_L((a, S), (c, w, r)) \neq 1$, abort.
       ii. Compute $\sigma \leftarrow \texttt{SimSign}(\mathcal{M}_L, m, (a, S))$.
       iii. If $\texttt{Verify}(\mathcal{M}_L, m, (a, S), \sigma) \neq 1$, abort.
       iv. Record $[m, (a, S), \sigma]$ and output $(\texttt{Signature}, sid, m, (a, S), \sigma)$ to P.

> $\mathcal{F}_{\text{SOK}}$ forwards all queries between $\mathcal{F}_{\text{ACC}}$, $\mathcal{F}_{\text{NIC}}$ and P or $\mathcal{A}$ (the formal details of this can be found in [CL06], but we will describe all such queries explicitly when they are used in the voting protocol).

- **Signature verification**
  - On input $(\texttt{Verify}, sid, m, (a, S), \sigma)$ from party V:
    1. If $[m, (a, S), \sigma']$ is recorded for some σ':

       Output $(\texttt{Verified}, sid, \texttt{Verify}(\mathcal{M}_L, m, (a, S), \sigma))$ to V.

       Else:
       i. Set $(c, w, r) \leftarrow \texttt{Extract}(\mathcal{M}_L, m, (a, S), \sigma)$.
       ii. If $\mathcal{M}_L((a, S), (c, w, r)) = 1$:
          a. If $\mathcal{F}_{\text{ACC}}$ doesn't accept $(\texttt{Verify}, sid_{\text{ACC}}, a, \{c\}, w)$ or $\mathcal{F}_{\text{NIC}}$ doesn't accept $(\texttt{Verify}, sid_{\text{NIC}}, c, S, r)$, abort.
          b. Output $(\texttt{Verified}, sid, \texttt{Verify}(\mathcal{M}_L, m, (a, S), \sigma))$ to V.

Else if $\texttt{Verify}(\mathcal{M}_L, m, (a, S), \sigma) \neq 1$:

Output $(\texttt{Verified}, sid, 0)$ to V.

Else abort.

The machine which accepts for the language we are interested in is defined by

$$\mathcal{M}_L((a, S), (c, w, r)) = \mathcal{M}_{\text{ACC}}((a, \{c\}), w) \wedge \mathcal{M}_{\text{NIC}}((c, S), r),$$

where the split into the (*input*, *witness*) pairs is there to emphasize the distinction in what we are treating as secret in the overall language versus the languages defined by the functionalities (though it only matters in the voting protocol itself[3], since the verification algorithms in our case behave the same regardless of the particular input/witness split).

We have the following general result with respect to UC-realizability of $\mathcal{F}_{\text{SOK}}$ due to [CL06] (Theorem 4.1 and the discussion on extensions in Section 4.3 of the cited paper), and a simple corollary for the functionality we defined above.

Theorem 4.1.3. *Let $\mathcal{F}_0$, $\mathcal{F}_1$ be explicit verification functionalities. Assuming SimExt-secure[4] signatures of knowledge, $\mathcal{F}_{\text{SOK}}$ ($\mathcal{F}_0$,$\mathcal{F}_1$) is non-trivially UC-realizable in the $\mathcal{F}_{\text{CRS}}$-hybrid model if and only if $\mathcal{F}_0$, $\mathcal{F}_1$ are non-trivially UC-realizable in the $\mathcal{F}_{\text{CRS}}$-hybrid model, where we consider a realization to be nontrivial if it never halts with an error message.*

Corollary 4.1.1. *$\mathcal{F}_{\text{SOK}}$ ($\mathcal{F}_{\text{NIC}}$,$\mathcal{F}_{\text{ACC}}$) is non-trivially realizable in the $\mathcal{F}_{\text{CRS}}$-hybrid model.*

4.2 Ideal accumulator functionality

To the best of our knowledge, no prior formulations of functionalities for accumulators in the UC framework exist in the literature, so in this section we will define our own. As a reference point we will use the secure accumulator definition from [CL02] given in Section 3.2.

4.2.1 Definition ($\mathcal{F}_{\text{acc}}$)

First, since we do not require the *delete* function which defines *dynamic* accumulators, we ignore the aux_f output of `Gen` and assume all trapdoors are destroyed if a scheme generates them. This limitation could be overcome by extending the proposed functionality to take trapdoors into account, but since it is not the main focus of this work, we do not consider this option here.

[3] Observe that while c is part of the witness for $\mathcal{M}_L$, it appears in the input statements of both $\mathcal{M}_{\text{ACC}}$ and $\mathcal{M}_{\text{NIC}}$.

[4] [CL06] used a game-based definition which they then proved to be equivalent to the functionality, so we do not give it here.

Now we can define a new functionality $\mathcal{F}_{\text{ACC}}$ to fulfill the properties provided by secure accumulators, modeling it after $\mathcal{F}_{\text{NIC}}$ and verification functionalities defined in [CL06].

Recall that the purpose of the accumulator functionality is to take a set of objects and output another object that represents the set as a whole, so that we can obtain witnesses for each member of the accumulated set, and that at the same time it is difficult to forge witnesses for objects which are not in the set. It is natural then to let $\mathcal{F}_{\text{ACC}}$ store a table of records of the form [*accumulator*, *input*, *witness*] where *input* is a set and *witness* is either another accumulator or some fixed element denoted as the *basis*, which represents the empty set (often written as u).

However, this alone is not enough to ensure consistency of the output – the quasi-commutativity property states that the order of accumulation does not matter, and since $f(w, x)$ is simply a function in the original definition, we want to be able to construct our accumulators using values for w which may not have been generated by a call to the functionality yet. Hence $\mathcal{F}_{\text{ACC}}$ will need to keep track of some auxiliary information about the items accumulated so far. This can be achieved via the following data structures:

- *support*, a list of entries of the form $(accumulator, [set_1, set_2, \dots])$ for each *accumulator* in the main table, where $[set_1, set_2, \dots]$ is a list of all the known representations of the contents of the set corresponding to the accumulator,
- *refs*, a list of references to accumulators that have not been expanded yet (i.e. that do not have an entry in *support* but occur in an expansion of other entries).

To illustrate the functionality and the use of these lists, we first give a simple example execution. Suppose we want to accumulate 3 items x_1, x_2, x_3 and we have a basis accumulator u and the accumulation function f that $\mathcal{F}_{\text{ACC}}$ should compute. One valid way the queries could go is the following:

1. *Accumulate* $\{x_2, x_3\}$ *with witness* a*:* $\mathcal{F}_{\text{ACC}}$ computes $f(a, \{x_2, x_3\}) = b$, adds $[b, \{x_2, x_3\}, a]$ to the main table, $(b, [\{x_2, x_3\} \cup A])$ to *support* and A to *refs*.
2. *Accumulate* x_1 *with witness* u*:* $\mathcal{F}_{\text{ACC}}$ computes $f(u, x_1) = a$ and adds $[a, \{x_1\}, u]$ to the main table and $(a, [\{x_1\}])$ to *support*. Since a is associated with the reference A, the first entry in *support* expands to $(b, [\{x_1, x_2, x_3\}])$ and A is removed from *refs*.
3. *Accumulate* $\{x_1, x_2, x_3\}$ *with witness* u*:* $\mathcal{F}_{\text{ACC}}$ computes $f(u, \{x_1, x_2, x_3\}) = c$. If $c = b$, it adds $[c, \{x_1, x_2, x_3\}, u]$ to the main table, otherwise it aborts.

We now give the formal definition of $\mathcal{F}_{\text{ACC}}$ split according to the commands it accepts so that we can give an informal explanation of each. It is parametrized by $\mathcal{X}_\lambda$, the set from which objects to be accumulated can be drawn.

Functionality $\mathcal{F}_{\mathsf{acc}}$ parametrized by $\mathcal{X}_\lambda$:

$\mathcal{F}_{\text{ACC}}$ is initialized when some party calls `Setup`. It asks the adversary for the parameters and for the accumulation and verification algorithms. It distributes these to any party that asks (we want to make the public nature of the scheme explicit).

- **Setup**
  - On input (`Setup`, *sid*) from party P:

    If this is the first time the functionality is called, hand (`Setup`, $sid, \mathcal{X}_\lambda$) to $\mathcal{A}$.
  - On input (`Algorithms`, *sid*, *params′*, `Acc`, `Verify`) from $\mathcal{A}$ for DPT ITM `Verify` and a PPT ITM `Acc`:
    1. For $params' = (\mathcal{U}_{\texttt{Acc}}, u)$, store $(\mathcal{X}_\lambda, params')$ as *params*, and record the algorithms as well.
    2. Output (`Algorithms`, *sid*, *params*, `Acc`, `Verify`) to P.
  - On input (`Parameters`, *sid*) from any party P:

    Output (`Parameters`, *sid*, *params*, `Acc`) to P.
  - On input (`VerificationAlgorithm`, *sid*) from any party P:

    Output (`VerificationAlgorithm`, *sid*, `Verify`) to P.

When a party asks $\mathcal{F}_{\text{ACC}}$ to accumulate some *input* with *witness*, several things need to happen. First, $\mathcal{F}_{\text{ACC}}$ needs to check if the given pair already has an associated accumulator value in the table and return it if that's the case. If not, then it can compute the value and check that it verifies.

Then, Step 5 in the definition below will assure the propagation of known set expansions throughout the *support* list, so that all associated sets are accounted for, including yet-unexpanded references. For example, suppose we are given $witness = w$ and $input = \{z\}$. If we have $(w, [\{x\} \cup Y, \{x, y\}])$ in *support* and Y in *refs*, then the *accsets* corresponding to the newly calculated $accumulator = a$ will have to be of the form $[\{x, z\} \cup Y, \{x, y, z\}]$.

Finally, the purpose of the complicated conditions in Step 6 is to make sure that `Acc` satisfies the quasi-commutativity property, so that regardless of the order of accumulation, the output remains consistent:

1. Following on from the example above, if we find that we already have some entry $(a, [\{y, z\} \cup X])$ in *support*, we can simply append the newly calculated sets to the existing list, replacing the entry with $(a, [\{y, z\} \cup X, \{x, z\} \cup Y, \{x, y, z\}])$.
2. If there is no such entry and it happens that there is $a' \neq a$ such that we have $(a', [\{x, y, z\}])$ in *support*, then consistency would be violated (since

we can't have the same set have two different representations) and $\mathcal{F}_{\text{ACC}}$ would abort.

3. The third condition deals with the case when the computed a references some A from *refs*. Our inputs w and $\{z\}$ now tell us something about the contents of a, which lets us update all the entries in *support* which contain A as long as there are no conflicts. An example of a conflict in this case would be if we found $(a', [\{s\} \cup A])$ and $(a'', [\{s, x, y, z\}])$ in *support* with $a' \neq a''$, because $\{s\} \cup A$ has expanded to $\{s, x, y, z\}$ (it's essentially the same issue as in the previous case).

If all checks pass, we can simply add the new entries to *support* and the main table.

- **Accumulation**
  - On input (`Accumulate`, *sid*, *witness*, *input*) from any party P:
    1. If *params*, `Acc` not stored, *input* $\not\subseteq \mathcal{X}'_\lambda$ or *witness* $\notin \mathcal{U}'_{\texttt{Acc}}$, abort.
    2. If there is an entry $[accum', input, witness]$:
       Output (`Accumulator`, *sid*, $accum'$) to P.
    3. Compute $accum \leftarrow \texttt{Acc}(witness, input)$.[a]
    4. If $accum \notin \mathcal{U}_{\texttt{Acc}}$ or $\texttt{Verify}(accum, input, witness) \neq 1$, abort.
    5. If $witness = u$:

       Set $accsets \leftarrow [input]$ (a list with one element).

       Else if there is $(witness, witsets) \in support$ for some *witsets*:

       Set $accsets \leftarrow [input \cup witset$ for each *witset* in *witsets*$]$.

       Else:

       i. Define $witset_r$ as the unexpanded reference for *witness* and add it to *refs*.

       ii. Set $accsets \leftarrow [input \cup witset_r]$.
    6. If there is $(accum, accsets') \in support$ for some $accsets'$:

       Replace the pair with $(accum, accsets' \,||\, accsets)$ (i.e. append *accsets* to $accsets'$) unless $accsets' = accsets$.

       Else if for any $accset \in accsets$, there is $(accum', accsets') \in support$ for some $accum' \neq accum$ and $accset \in accsets'$, abort.

       Else if *accum* has an associated reference $witset_r \in refs$:

       i. Scan the full *support* list and for every $(accum', accsets')$, replace each occurrence of $witset_r$ in $accsets'$ with items

from *accsets* (hence each $accset' = \ldots \cup witset_r$ expands into multiple $expset = \ldots \cup accset$).

ii. For any expansion, if there is $(accum'', accsets'')$ for some $accum'' \neq accum'$ and $expset \in accsets''$, abort.

iii. Remove $witset_r$ from the list of references.

iv. Add $(accum, accsets)$ to the list.

Else:

Add $(accum, accsets)$ to the list.

7. Record the entry $[accum, input, witness]$ in the main table.
8. Output $(\texttt{Accumulator}, sid, accum)$ to P.

[a] We are using the same shorthand as in f to mean accumulation of all elements in some order, which in itself says nothing about quasi-commutativity of `Acc`.

In a verification query, the two abort conditions attempt to capture the unforgeability property for adversarial values since there are two ways a forgery could occur. The adversary could attempt to verify an accumulator either with an item that has not been accumulated in it (e.g. the tuple $(a, \{x\}, u)$ when the functionality has $(a, [\{y, z\}])$ in *support* such that $x \notin \{y, z\}$), or with an invalid witness (e.g. $(a, \{x\}, w)$ when there are $(w, [\{x, y\}])$ and $(a, [\{x, y, z\}])$ such that $\{x, y, z\} = \{x, y\} \cup \{z\}$ but $z \neq x$).

- **Verification**
  - On input $(\texttt{Verify}, sid, accum, input, witness)$ from party P:
    1. If *params* or `Acc` are not stored, $input \not\subseteq \mathcal{X}'_\lambda$, $accum \notin \mathcal{U}_{Acc}$ or $witness \notin \mathcal{U}'_{Acc}$, abort.
    2. If there is an entry $[accum, input, witness]$:
       Output $(\texttt{Verified}, sid, 1)$ to P.
    3. If $\texttt{Verify}(accum, input, witness) = 1$ and either of the following holds, abort:
       (a) There is $(accum, accsets) \in support$ such that *accsets* contains a fully-expanded (i.e. without any elements in *refs*) set A with $input \not\subseteq A$.
       (b) There are $(witness, witsets)$, $(accum, accsets) \in support$ such that for some $W \in witsets$ and $A \in accsets$ we have $A = W \cup V$ where V is a fully-expanded set with $V \not\subseteq input$.
    4. Output $(\texttt{Verified}, sid, \texttt{Verify}(accum, input, witness))$ to P.

Observe that in $\mathcal{F}_{\text{ACC}}$ we define the accumulation function f (called `Acc` here) as

a polynomial-time Turing machine, which is not equivalent to a polynomial-size circuit that is used in Definition 3.2.2. However, we claim that in this case the choice of the computation model is not of particular concern as the constructions we are interested in satisfy our restriction to Turing machines, and the authors of the definition did not emphasize this distinction either [CL02].

4.2.2 Proof of realizability

Recall that in order to prove that the functionality we just defined indeed captures the ideal secure accumulator, we will prove it is equivalent to the standard definition of [CL02] that we gave in Section 3.2.3. To this end, we can now define a real protocol that will make use of this accumulator scheme[5].

Protocol $\Pi_{\mathbf{sa}}$ for $\mathbf{sa} = (\mathcal{X}_\lambda, \mathtt{Gen})$ in the $\mathcal{F}_{\mathbf{crs}}^{\mathtt{Gen}}$-hybrid model:

- **Setup**
  - On first input $(\mathtt{Setup}, sid)$, party P runs the following program:
    1. Send $(\mathtt{CRS}, sid)$ to $\mathcal{F}_{\mathrm{CRS}}^{\mathtt{Gen}}$ to receive $(\mathtt{CRS}, sid, (f, \mathcal{U}_f, u))$.
    2. Store $params = (\mathcal{X}_\lambda, \mathcal{U}_{\mathtt{f}}, u)$ and f.
    3. Define $\mathtt{Verify}(a, X, w) := a \stackrel{?}{=} f(w, X)$.
    4. Output $(\mathtt{Algorithms}, sid, params, f, \mathtt{Verify})$.
  - On input $(\mathtt{Params}, sid)$, P outputs $(\mathtt{Params}, sid, params, f)$.
  - On input $(\mathtt{VerificationAlgorithm}, sid)$, P outputs $(\mathtt{VerificationAlgorithm}, sid, \mathtt{Verify})$.
- **Accumulation**
  - On input $(\mathtt{Accumulate}, sid, w, X)$, party P runs:
    1. If $params, f$ not stored, $X \not\subseteq \mathcal{X}'_\lambda$ or $w \notin \mathcal{U}'_{\mathtt{f}}$, abort.
    2. Compute $a \leftarrow f(w, X)$.
    3. Output $(\mathtt{Accumulator}, sid, a)$.
- **Verification**
  - On input $(\mathtt{Verify}, sid, a, X, w)$, party P runs:
    1. If $params, f$ not stored, $X \not\subseteq \mathcal{X}'_\lambda$ or $w \notin \mathcal{U}'_f$, abort.
    2. Output $(\mathtt{Verified}, sid, \mathtt{Verify}(a, X, w))$.

The general structure of the proofs of realizability that we will give is be mod-

[5]For generality, the shared parameters are obtained in the CRS model, but in our voting protocol the election authority is trusted with generating the parameters.

eled after the proofs for Canetti's signature functionality in the 2005 version of [Can13].

However, first we prove a partial result about the behavior of $\mathcal{F}_{\text{ACC}}$.

Lemma 4.2.1
If SA *is an accumulator scheme satisfying the quasi-commutativity property and* $\mathcal{F}_{\text{ACC}}$ *is given parameters and algorithms according to* SA*, then* $\mathcal{F}_{\text{ACC}}$ *never aborts on a valid* `Accumulate` *query.*

Proof. We will work with a query of the form $(\texttt{Accumulate}, sid, w, \{x\})$ with single-value input for simplicity, but the result can be generalized for arbitrary sets. We also assume that the query is well-formed, i.e. $w \in \mathcal{U}'_f$ and $x \in \mathcal{X}'_\lambda$.

It is clear that since f is deterministic and $\mathcal{F}_{\text{ACC}}$ only adds properly calculated accumulator values to its table, consistency is assured and $\mathcal{F}_{\text{ACC}}$ either returns $a = f(w, x)$ or aborts.

Aborts can only occur in the second and third condition of Step 6, so we will look at them in turn and show that they are never satisfied:

- *There is* $A \in$ *accsets and* $(a', [A'_1, A'_2, \dots]) \in$ *support such that* $A = A'_j$ *for some* j *and* $a \neq a'$.
  We will show the impossibility of this by contradiction. Suppose it is true. First, note that after Step 5, A is one of three forms, which we can summarize as $A = w' \cup Y \cup \{x\}$ for $Y = \{y_1, y_2, \dots\}$ (possibly empty), and w' is either equal to w, to $f(w', Y)$ or to u (in which case it is only a placeholder for the empty set). This is correct since due to the way that items are added or modified within *accsets*, there can be at most one witness reference in each, and in all cases $\{x\}$ is added to each set.
  Since we have $A'_j = A$, there must have been a sequence of queries which accumulated all elements of Y as well as x into a', so that $f(w', Y \cup \{x\}) = a'$. From the current query we know that $f(w, x) = a$ and $f(w', Y) = w$, which implies that $a = f(w, x) = f(f(w', Y), x) = f(w', Y \cup \{x\}) \neq a'$, clearly a contradiction.

- *We have a associated with some unexpanded reference* $w_r \in$ *refs. There is* $A \in$ *accsets,* $(a', [A'_1, A'_2, \dots]) \in$ *support such that* $A'_i = w_r \cup B$ *where* $B = \{b_1, b_2, \dots\}$ *for some* i *and* $(a'', [A''_1, A''_2, \dots]) \in$ *support such that* $A''_j = A \cup B$ *for some* j *and* $a' \neq a''$.
  We proceed as with the previous claim. $A''_j = A \cup B$ implies that $f(w', Y \cup \{x\} \cup B) = a''$. The current query implies that $f(w, x)$ is the accumulator for w_r, and $A'_i = w_r \cup B$ gives us $f(a, B) = a'$. Combining the last two we get $a' = f(f(w, x), B) = f(f(w', Y), \{x\} \cup B) = f(w', Y \cup \{x\} \cup B) = a''$, which is again a contradiction.

Since no abort can occur, $[a, x, w]$ is recorded in the table and $(\texttt{Accumulator}, sid, a)$ is returned. □

In the following two theorems, we capture the equivalence between our functionality and secure accumulator schemes.

Theorem 4.2.1. *If* $\text{SA} = (\mathcal{X}_\lambda, \mathtt{Gen})$ *is a secure accumulator scheme, then* Π_{SA} *UC-realizes* $\mathcal{F}_{\text{ACC}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\mathtt{Gen}}$*-hybrid model.*

Proof. We will prove the contrapositive, i.e. the claim that if Π_{SA} does not UC-realize $\mathcal{F}_{\text{ACC}}$, then SA is not a secure accumulator scheme, so at least one property is not satisfied. We can therefore assume that SA is an accumulator scheme, since otherwise the result would already be true.

For ease of notation let Π stand for Π_{SA} and $\mathcal{F}$ for the ideal protocol for $\mathcal{F}_{\text{ACC}}$.

By the definition of UC-emulation we have:

There exists a PPT adversary $\mathcal{A}$ *such that for all PPT simulators* $\mathcal{S}$ *there is a balanced PPT environment* $\mathcal{E}$ *with* $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}} \not\approx \text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$.

This means that we have an environment which can distinguish between the ideal and the real protocol with non-negligible probability. We will show that we can use it to construct an adversary $\mathcal{G}$ against witness unforgeability because the environment must trigger a `Verify` query during its run if it is to succeed.

By the definition of witness unforgeability we need a forger $\mathcal{G}$ such that:

Given $(f, \mathcal{U}_f, u)$ *for* $f \leftarrow \mathtt{Gen}(1^\lambda)$, $\mathcal{U}_f$ *and* $u \leftarrow \mathcal{U}_f$, *it returns* (x, w, X) *for* $x \in \mathcal{X}_\lambda$, $X \subset \mathcal{X}'_\lambda$, $w \in \mathcal{U}'_f$ *such that* $x \notin X$ *and* $f(w, x) = f(u, X)$ *with non-negligible probability.*

Since our environment distinguishes for all simulators, take $\mathcal{S}$ as a generic simulator that supplies correct algoritms from the SA scheme to $\mathcal{F}$, i.e. for $f \leftarrow \mathtt{Gen}(1^\lambda)$, $u \leftarrow \mathcal{U}_f$, defines $\mathtt{Verify}(a, X, w) := a \stackrel{?}{=} f(w, X)$ and sends $(\mathtt{Algorithms}, sid, (\mathcal{X}_\lambda, \mathcal{U}_f, u), f, \mathtt{Verify})$ during setup.

Now we can construct $\mathcal{G}$ by simulating the ideal execution with $\mathcal{F}$ and $\mathcal{S}$ for $\mathcal{E}$.

Like $\mathcal{S}$, $\mathcal{G}$ also runs a simulated $\mathcal{A}$. When $\mathcal{E}$ activates the party P with $(\mathtt{Setup}, sid)$ for some sid, $\mathcal{G}$ returns $(\mathtt{Algorithms}, sid, (\mathcal{X}_\lambda, \mathcal{U}_f, u), f, \mathtt{Verify})$ from its input values and `Verify` defined as $\mathcal{S}$ would define it.

$\mathcal{G}$ answers all $(\mathtt{Accumulate}, sid, w', X')$ queries with $(\mathtt{Accumulator}, sid, f(w', X'))$. This is consistent with behavior of $\mathcal{F}$ assuming f is quasi-commutative, as shown in Lemma 4.2.1. $\mathcal{G}$ also keeps track of all partial records of sets as $\mathcal{F}$ would.

Now we can turn our attention to when $\mathcal{E}$ sends a query $(\mathtt{Verify}, sid, a, Y, v)$. $\mathcal{G}$ checks if $f(v, Y) = a$ and one of the following conditions for a forgery holds:

1. *There is* $(a, [A_1, A_2, \dots])$ *such that* $Y \not\subseteq A_i$ *for some fully expanded* $A_i = \{a_1, a_2, \dots\}$, *so that we have* $x \in Y$ *but* $x \notin A_i$. *Let* $w = f(v, Y \backslash \{x\})$ *and hence* $f(w, x) = f(v, Y) = a = f(u, A_i)$.

2. *There are* $(v, [W_1, W_2, \dots])$[6] *and* $(a, [A_1, A_2, \dots])$ *such that* $A_k = W_j \cup V_i$ *and* $V_i \not\subseteq Y$ *for some fully expanded* V_i *and* W_j*, so that there is* $x \in V_i$ *but* $x \notin Y$*. Let* $w = f(v, V_i \backslash \{x\})$ *and hence* $f(w, x) = f(v, V_i) = a = f(v, Y) = f(u, Y \backslash W_j)$.

If it does hold, $\mathcal{G}$ outputs the forgery (x, w, X) with $X = A_i$ or $X = Y \backslash W_j$ and halts. If not, it continues the simulation.

To compute the success probability of $\mathcal{G}$, we define the event E that during the protocol execution, $\mathcal{E}$ runs `Setup` and a $(\texttt{Verify}, sid, a, Y, v)$ query such that $\texttt{Verify}(a, Y, v) = 1$ and one of the previously described forgery conditions holds. We have shown in Lemma 4.2.1 that assuming quasi-commutativity of the scheme, $\mathcal{F}$ never aborts a well-formed `Accumulate` query and returns results consistent with Π. In addition, it is clear that the behavior of the `Verify` query is identical between $\mathcal{F}$ and Π unless a forgery condition is triggered. Hence the view of $\mathcal{E}$ of the real execution is indistinguishable from the ideal one as long as E doesn't occur, but since we know that $\mathcal{E}$ is in fact able to distinguish with non-negligible probability, E must occur with the same probability and therefore $\mathcal{G}$ can obtain a forgery as desired. □

Theorem 4.2.2. *If* Π_{SA} *UC-realizes* $\mathcal{F}_{\text{ACC}}$ *in the* $\mathcal{F}_{\text{CRS}}^{\texttt{Gen}}$*-hybrid model, then* $\text{SA} = (\mathcal{X}_\lambda, \texttt{Gen})$ *is a secure accumulator scheme.*

Proof. We adopt the same notation as in the previous proof, and prove the contrapositive. To show that if SA is not a secure accumulator scheme then Π does not UC-realize $\mathcal{F}$, we must focus on each property of the scheme in turn. Not having efficient generation or efficient evaluation would make the execution fail at the `Setup` phase, so we only need to consider quasi-commutativity and witness unforgeability.

In each case, we need to show:

There is a PPT adversary $\mathcal{A}$ *such that for all PPT simulators* $\mathcal{S}$ *there is a balanced PPT environment* $\mathcal{E}$ *with* $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}} \not\approx \text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$.

Suppose SA does not satisfy quasi-commutativity:

There are $f \in F_\lambda$, $u \in \mathcal{U}_f$ *and* $x_1, x_2 \in \mathcal{X}_\lambda$ *such that* $f(f(u, x_1), x_2) \neq f(f(u, x_2), x_1)$.

We construct $\mathcal{A}$ and $\mathcal{E}$ in the following way:

1. No inputs are passed to $\mathcal{A}$.
2. $\mathcal{E}$ sets sid and activates party P with $(\texttt{Setup}, sid)$ to obtain $(\texttt{Algorithms}, sid, (\mathcal{X}_\lambda, \mathcal{U}_f, u), f, \texttt{Verify})$ on their behalf.
3. $\mathcal{E}$ activates P with $(\texttt{Accumulate}, sid, w, X)$ for the following pairs of (w, X) with the outputs obtained: $(u, x_1) \to a$, $(u, x_2) \to a'$, $(a, x_2) \to b$ and $(a', x_1) \to b'$.

[6]Note that we are still using the convention that if $v = u$, all $W_j = \emptyset$.

(a) In the execution with $\mathcal{F}$, $\mathcal{E}$ has no control over what algorithms $\mathcal{S}$ supplies (and specifically whether f is quasi-commutative for given u, x_1, x_2), but this does not matter since $\mathcal{F}$ can only abort or return values such that $b = b'$. (This is easy to see for this particular set of values, but can be shown to be true in general.)

(b) Parties executing Π use $g \leftarrow \texttt{Gen}(1^\lambda)$ obtained from $\mathcal{F}_{\text{CRS}}^{\texttt{Gen}}$. Since `Gen` returns random functions from F_λ and we are only guaranteed the existence of one f that fails commutativity for our u, x_1, x_2, the probability of `Gen` returning f has a lower bound of $\frac{1}{|F_\lambda|}$, assuming `Gen` is not biased. The size of F_λ is in turn polynomially constrained by the security parameter 1^λ, so the probability cannot be negligible.
In such a case then Π returns

$$b = f(a, x_2) = f(f(u, x_1), x_2) \neq f(f(u, x_2), x_1) = f(a', x_1) = b'.$$

So if the protocol returns $b = b'$ or aborts, $\mathcal{E}$ outputs 0, otherwise it outputs 1, succeeding at distinguishing between $\mathcal{F}$ and Π with non-negligible probability, which depends on whether Π uses a non-quasi-commutative accumulation function.

Now, suppose SA does not satisfy witness-unforgeability:

There exists PPT forger $\mathcal{G}$ which given $(f, \mathcal{U}_f, u)$ for $f \leftarrow \texttt{Gen}(1^\lambda)$ and $u \leftarrow \mathcal{U}_f$, returns (x, w, X) for $x \in \mathcal{X}_\lambda$, $X \subset \mathcal{X}'_\lambda$, $w \in \mathcal{U}'_f$ such that $x \notin X$ and $f(w, x) = f(u, X)$ with non-negligible probability.

We construct $\mathcal{A}$ and $\mathcal{E}$ in the following way:

1. No inputs are passed to $\mathcal{A}$.
2. $\mathcal{E}$ sets *sid* and activates party P with $(\texttt{Setup}, sid)$ to obtain $(\texttt{Algorithms}, sid, (\mathcal{X}_\lambda, \mathcal{U}_f, u), f, \texttt{Verify})$ on their behalf.
3. Then $\mathcal{E}$ internally runs an instance of the forger $\mathcal{G}(f, \mathcal{U}_f, u)$ to get a forgery tuple (x, w, X) with non-negligible probability.
4. $\mathcal{E}$ activates P with $(\texttt{Accumulate}, sid, u, X)$ to obtain $(\texttt{Accumulator}, sid, a)$.
5. Lastly, $\mathcal{E}$ activates P with $(\texttt{Verify}, sid, a, \{x\}, w)$:
   (a) In the execution with $\mathcal{F}$, there is no entry for $[a, \{x\}, w]$. If $\texttt{Verify}(a, \{x\}, w) = 0$ for `Verify` supplied by $\mathcal{S}$, $\mathcal{F}$ outputs $(\texttt{Verified}, sid, 0)$. Otherwise, it finds there is $(a, [X])$ such that $x \notin X$ and aborts.
   (b) Π computes $\texttt{Verify}(a, \{x\}, w) = a \stackrel{?}{=} f(w, x) = 1$ since $a = f(u, X)$ and outputs $(\texttt{Verified}, sid, 1)$.

Hence if the protocol outputs 0 or aborts, $\mathcal{E}$ outputs 0, otherwise it outputs 1, succeeding in distinguishing with non-negligible probability depending only on whether $\mathcal{G}$ supplies the forgery. □

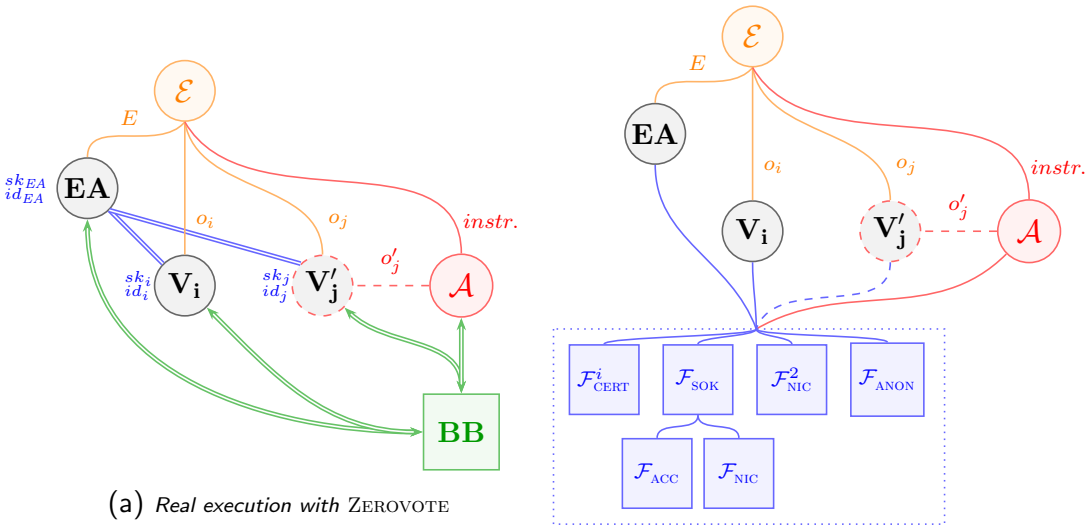
We can now state the following easy corollary of Theorem 4.2.1 for a strong RSA accumulator construction [CL02], which we explicitly make use of in the real protocol as described in Section 3.2.3.

Corollary 4.2.1. $\Pi_{\text{RSA-SA}}$ *UC-realizes* $\mathcal{F}_{\text{ACC}}$ *in the* $\mathcal{F}_{\text{CRS}}^{Gen}$*-hybrid model.*

A note on the construction. The definition of $\mathcal{F}_{\text{ACC}}$ is not straightforward and it is possible a different formulation could capture the required properties in a simpler or more elegant way. An early attempt to express it purely in terms of sets without unexpanded references which resulted in a correspondence to a strictly weaker (but possibly still useful) notion of unforgeability is documented in Appendix C.

4.3 Hybrid protocol Π_{zv}

Having defined ideal functionalities for all of the primitives of Zerovote, we can now construct a hybrid protocol from the real one (defined in Section 3.3) by replacing all calls to real functions with the ideal calls. We describe the voting protocol in the ($\mathcal{F}_{\text{ANON}}$, $\{\mathcal{F}_{\text{CERT}}^{i}\}$, $\mathcal{F}_{\text{SOK}}$ ($\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ACC}}$), $\mathcal{F}_{\text{NIC}}^{2}$)-hybrid model[7]. The composition theorem 2.2.2 in conjunction with proofs of realizability of each of the ideal functionalities will guarantee that this hybrid protocol is indistinguishable from the real one.



(a) *Real execution with* ZEROVOTE

(b) *Hybrid execution with* Π_{ZV}

Figure 4.1: Execution in the real vs. hybrid model.

[7] A separate commitment functionality is used to emphasize that it serves a different purpose than the commitments within the signature of knowledge, but in principle $\mathcal{F}_{\text{NIC}}^{2}$ could be merged with $\mathcal{F}_{\text{NIC}}$ which is "inside" $\mathcal{F}_{\text{SOK}}$. Also note that there are in fact several "layers" of hybrid protocols – the given one, the lower $\mathcal{F}_{\text{CRS}}$-hybrid model which is required to realize the $\mathcal{F}_{\text{NIC}}$ and $\mathcal{F}_{\text{ACC}}$ functionalities as well as the ($\mathcal{F}_{\text{SIG}}$,$\mathcal{F}_{\text{CA}}$)-hybrid model needed for each $\{\mathcal{F}_{\text{CERT}}^{i}\}$ where i is taken over all voters $\mathbf{V_i}$ plus **EA**.

The general assumptions of the real protocol described at the beginning of Section 3.3 apply here, so we do not repeat them, only note that the anonymous broadcast channel is now encoded explicitly as an ideal functionality. Since all messages are sent to $\mathcal{F}_{\text{ANON}}$, which forwards them anonymously to all parties (including the adversary), **BB** is implicit in this setting as each party can perform the same verifications.

The setup of the real protocol also makes certain assumptions about how registration of eligible voters is achieved. To make them explicit, we model the registration process using ideal functionalities for signatures tied to "certificates" (one for each voter), as explained in more detail in Section 4.1.3. Hence we assume that at the beginning of the protocol **EA** is given as input a list of eligible voters (which it makes public). We write $\{m\}_P$ to denote a message m signed by party P. We say a party *posts* something when it is anonymously broadcast to all parties (i.e. recorded on **BB**).

Remember that it is crucial that the individual stages of the protocol follow in order and do not overlap, which is enforced by **EA** via posting signed messages of the `Begin stage` n and `End stage` n form. We assume this (and that honest voters comply) implicitly in the definition that follows.

Lastly, the *standard corruption* model is assumed, i.e. voters react to `Corrupt` messages that signify the party is now under control of the adversary, and the environment is notified.

Zerovote protocol $\Pi_{\mathbf{zv}}$:

1. Setup

- On $(\texttt{Setup}, sid, E)$ the election authority **EA** runs the following:
  i. Invoke $\mathcal{F}_{\text{ANON}}$ with $(\texttt{Setup}, sid_a)$ for some sid_a and $\mathcal{F}_{\text{CERT}}^{EA}$ with $(\texttt{Setup}, \mathbf{EA})$.
  ii. Initialize $\mathcal{F}_{\text{SOK}}$ with $(\texttt{Setup}, sid_s)$ where $sid_s = (\mathcal{F}_{\text{NIC}}, \mathcal{F}_{\text{ACC}}, sid_{\text{NIC}}, sid_{\text{ACC}}, sid')$ for some sid_{NIC}, sid_{ACC}, sid', which runs the setup for $\mathcal{F}_{\text{NIC}}$ and $\mathcal{F}_{\text{ACC}}$ from within the signature of knowledge functionality.
  iii. Sign the list of eligible voter identities E using $\mathcal{F}_{\text{CERT}}^{EA}$ with (`Sign`, **EA**, E) to get $(\texttt{Signature}, \mathbf{EA}, \{E\}_{\mathbf{EA}})$.
  iv. Post signed E using $\mathcal{F}_{\text{ANON}}$ with $(\texttt{Post}, sid_a, \{E\}_{\mathbf{EA}})$.
- On $(\texttt{Setup}, sid)$ each voter $\mathbf{V_i}$ runs the following:
  i. Invoke $\mathcal{F}_{\text{ANON}}$ with $(\texttt{Setup}, sid_a)$ and $\mathcal{F}_{\text{CERT}}^{i}$ with $(\texttt{Setup}, \mathbf{V_i})$.
  ii. Invoke $\mathcal{F}_{\text{NIC}}$ via ${\mathcal{F}_{\text{SOK}}}^a$ with ($\mathcal{F}_{\text{NIC}}$-`Setup`, sid_{NIC}, sid_s) to get ($\mathcal{F}_{\text{NIC}}$-`Parameters`, $sid_{\text{NIC}}, sid_s, params_{\text{NIC}}$).

iii. Invoke $\mathcal{F}_{\text{ACC}}$ via $\mathcal{F}_{\text{SOK}}$ with ($\mathcal{F}_{\text{ACC}}$-`Parameters`, sid_{ACC}, sid_s) to receive ($\mathcal{F}_{\text{ACC}}$-`Parameters`, $sid_{\text{ACC}}, sid_s, params_{\text{ACC}}, f$) where $params_{\text{ACC}} = (\mathcal{X}_\lambda, \mathcal{U}_f, u)$.

iv. Output (`Setup`, sid, E).

Any party can verify ***EA****'s signature via a* `Verify` *query to* $\mathcal{F}_{\text{CERT}}^{EA}$.

[a]Note that $\mathcal{F}_{\text{SOK}}$ will only respond once it has been initialized by **EA**.

2. Credential generation

- On (`GenerateCredential`, sid) each voter $\mathbf{V_i}$ runs the following:

  i. Generate a random serial number $S \in M$ for M defined in $params_{\text{NIC}}$.

  ii. Pass ($\mathcal{F}_{\text{NIC}}$-`Commit`, sid_{NIC}, sid, S) to $\mathcal{F}_{\text{SOK}}$, which forwards it to $\mathcal{F}_{\text{NIC}}$ and sends back its response ($\mathcal{F}_{\text{NIC}}$-`Commitment`, $sid_{\text{NIC}}, sid, c, r$).

  iii. If the received credential $c \notin \mathcal{X}_\lambda$, generate a new S and repeat the $\mathcal{F}_{\text{NIC}}$-`Commit` query until the condition is satisfied.

  iv. Sign c using $\mathcal{F}_{\text{CERT}}^{i}$ with (`Sign`, $\mathbf{V_i}, c$) to get (`Signature`, $\mathbf{V_i}, \{c\}_{\mathbf{V_i}}$).

  v. Post the signed c via $\mathcal{F}_{\text{ANON}}$ with (`Post`, $sid_a, \{c\}_{\mathbf{V_i}}$).

  vi. Output (`Credential`, $sid, \mathbf{V_i}, c$).

  Any party can check that each c is unique and each $\boldsymbol{V_i}$ *sent at most one by verifying the signature via a* `Verify` *query to* $\mathcal{F}_{\text{CERT}}^{i}$.

  After the end of the stage is announced, each party can collate the valid credentials into the set C.

3. Ballot casting

- On (`CastBallot`, sid, o_i) each voter $\mathbf{V_i}$ runs the following:

  i. Invoke $\mathcal{F}_{\text{NIC}}^{2}$ with (`Setup`, sid_n) to get ($\mathcal{F}_{\text{NIC}}^{2}$-`Parameters`, sid_{NIC}^{2}, $sid_s, params_{\text{NIC}}^{2}$).

  ii. Send a query (`Commit`, sid_n, o_i) to $\mathcal{F}_{\text{NIC}}^{2}$ to receive (`Commitment`, sid_n, x_i, r_i), where x_i represents the ballot bound to the vote o_i.

  iii. Send two queries to $\mathcal{F}_{\text{SOK}}$ to be forwarded to $\mathcal{F}_{\text{ACC}}$ in order to compute the accumulator a and witness w: ($\mathcal{F}_{\text{ACC}}$-`Accumulate`, $sid_{\text{ACC}}, sid, u, C\backslash\{c\}$) to obtain ($\mathcal{F}_{\text{ACC}}$-`Accumulator`, sid_{ACC}, sid, w) and similarly $(w, \{c\})$ to get a.

  iv. Send a query (`Sign`, $sid, x_i, (a, S), (c, w, r)$) to $\mathcal{F}_{\text{SOK}}$ to receive (`Signature`, $sid, x_i, (a, S), \sigma$) containing the signature σ on x_i.

v. Post the unsigned message $(x_i, (\sigma, S))$ via $\mathcal{F}_{\text{ANON}}$ with $(\texttt{Post}, sid_a, (x_i, (\sigma, S)))$.

vi. Output $(\texttt{Ballot}, sid, x_i)$.

Any party can check that x_i, S are unique, and that σ is valid by a `Verify` *query to $\mathcal{F}_{\text{SOK}}$.*

Ballot opening (tally)

- On $(\texttt{OpenBallot}, sid)$ each voter $\mathbf{V_i}$ runs the following:
  i. Post the opening (o_i, r_i) via $\mathcal{F}_{\text{ANON}}$ with $(\texttt{Post}, sid_a, (x_i, (o_i, r_i)))$ to reveal the vote.
  ii. Output $(\texttt{Vote}, sid, o_i)$.
- On $(\texttt{Tally}, sid)$ each voter $\mathbf{V_i}$ runs the following:
  i. Set $T = \{o_j\}$ for votes o_j received from $\mathcal{F}_{\text{ANON}}$ which are valid (i.e. their corresponding ballots and proofs verify).
  ii. Output $(\texttt{Tally}, sid, T)$.

Any party can verify the validity of the ballots by querying $\mathcal{F}^2_{\text{NIC}}$.

It is not stated explicitly in the run of the protocol, but the definitions of the functionalities that we are using imply that $\mathcal{A}$ provides them with algorithms and parameters during their `Setup` phases.

We do note what verification requests the parties can use in each stage of the protocol (or rather, can use from that stage onwards), but since public verification is a feature necessary to compute the true tally, we briefly summarize the path that a party must take to verify a given vote at the end of the protocol. After the second stage, the (honest) eligible voters will not include the invalid credentials in the set C from which they compute the accumulator a, so the signatures of knowledge made of invalid c' will not be valid either (because $c' \notin C$ or they were computed with a different accumulator $a' \neq a$), hence invalidating the ballots they sign in the third stage (even if the actual commitments to votes are computed correctly), so the votes corresponding to those ballots can be discarded when computing the tally.

As a consequence of the universal composability theorem and the realizability results of the functionalities in Section 4.1, the ideal calls in Π_{ZV} can be replaced by the local subroutines of the real ZEROVOTE protocol outlined in Section 3.2 and we get the following result.

Theorem 4.3.1. Π_{ZV} *UC-emulates* ZEROVOTE.

Chapter 5

Proving privacy

We now have all the building blocks required to complete the analysis of our protocol. In this chapter, we are concerned with the final step: to describe an ideal functionality that captures the goal of preserving vote privacy in a self-tallying protocol in the presence of static adversaries, and give a proof that Π_{ZV} realizes it in the UC framework.

5.1 Ideal self-tallying voting protocol ($\mathcal{F}_{\text{vote}}$)

Recall that in the UC framework, we need to match the external structure of our real and hybrid protocols to even have a chance at realizability (i.e. the number of stages and the format of calls have to be the same), but that aside we are defining an ideal service independent of the work that we have done so far. Much of the complexity of the definition that we will give is simply related to having to deal with corrupted voters on a public channel.

For these reasons, an intuitively more appealing functionality like the one below would *not* work for us, even though it might be enough to capture the same concerns in the standalone model:

Functionality $\mathcal{F}_{\text{priv}}$:

- On first input $(\texttt{Setup}, sid, E)$ from **EA**:

  Store E and output $(\texttt{Setup}, sid, E)$ to $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

- On input $(\texttt{Cast}, sid, o_i)$ from $\mathbf{V_i}$:

  If $\mathbf{V_i} \in E$, add o_i to T and output $(\texttt{Vote}, sid, o_i)$ to $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

- On input $(\texttt{Tally}, sid)$ from P:

  Output $(\texttt{Tally}, T)$ to P.

Functionality $\mathcal{F}_{\text{vote}}$:

We split the definition by the individual stages for clarity, and discuss the properties that it tries to achieve throughout the definition as well as in a summary at the end.

1. Setup

- On first input $(\texttt{Setup}, sid, E)$ from **EA**:
  i. Store E, the list of eligible voters.
  ii. Initialize the lists $\mathcal{L}_C$ (credentials), $\mathcal{L}_B$ (ballots) and $\mathcal{L}_O$ (votes) as empty.
  iii. Hand $(\texttt{Setup}, sid, E)$ to $\mathcal{S}$.
  iv. Wait to receive $(\texttt{Setup}, sid, algs)$ from $\mathcal{S}$.
  v. Store $algs = (\texttt{GenCredential}, \texttt{GenBallot})$, a tuple of PPT algorithms.
  vi. Obtain the set of corrupted voters $\mathbb{V}_{corr}$.
  vii. Set the election status to 'Credential'.
  viii. Output $(\texttt{Setup}, sid, E)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$.

As in the previous section, the election authority **EA** is responsible for initiating the protocol as well as giving commands to proceed from stage to stage. $\mathcal{F}_{\text{VOTE}}$ will make sure to only accept calls relevant to the stage it is currently in.

The list of eligible voters E is public information and so it is given to the adversary, who then supplies the algorithms for generating credentials and ballots. Since we are only considering static corruption (i.e. parties must be corrupted at the outset of the execution), $\mathcal{F}_{\text{VOTE}}$ also obtains the set of corrupted voters.

Internally, the functionality will also keep track of the submitted credentials, ballots and votes.

2. Credential generation

If the election status is not 'Credential', ignore `GenCredential` queries.

- On input $(\texttt{GenCredential}, sid)$ from $\mathbf{V_i}$:

  For honest and eligible voters (i.e. if $\mathbf{V_i} \notin \mathbb{V}_{corr}$ and $\mathbf{V_i} \in E$):

  i. If there is $(\mathbf{V_i}, c_i', 1)$ in $\mathcal{L}_C$ for some c_i', ignore the query. [A single credential per honest voter.]

ii. Compute $c_i \leftarrow$ `GenCredential`().

iii. If $c_i = c_i'$ for some $(\mathbf{V_i'}, c_i', 1)$ in $\mathcal{L}_C$, abort. [Uniqueness of credentials.]

iv. Add $(\mathbf{V_i}, c_i, 1)$ to $\mathcal{L}_C$ and output $(\texttt{Credential}, sid, \mathbf{V_i}, c_i)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

For corrupted voters (i.e. $\mathbf{V_i} \in \mathbb{V}_{corr}$):

i. Hand $(\texttt{GenCredential}, sid, \mathbf{V_i})$ to $\mathcal{S}$.

ii. Wait to receive $(\texttt{GenCredential}, sid, \mathbf{V_i}, \mathbf{V_i'}, c_i')$ from $\mathcal{S}$.

iii. Set $v = 0$ if any of the following are true:

(a) $\mathbf{V_i} \notin E$. [Credentials of ineligible voters are not valid.]

(b) $\mathbf{V_i'} \neq \mathbf{V_i}$. [Forged identities are not valid.]

(c) There is $(\mathbf{V_i}, c_i, 1)$ in $\mathcal{L}_C$ for some c_i. [More than one credential cannot be valid.]

(d) $c_i' = c_i$ for some $(\mathbf{V_i''}, c_i, 1)$ in $\mathcal{L}_C$. [A non-unique credential is not valid.]

iv. Otherwise set $v = 1$.

v. Add $(\mathbf{V_i'}, c_i', v)$ to $\mathcal{L}_C$ and output $(\texttt{Credential}, sid, \mathbf{V_i}, c_i')$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$. [All messages get posted.]

Credentials for honest voters are computed using the supplied `GenCredential` algorithm which is not allowed to take any input, and therefore the credentials cannot leak any auxiliary information (although we could let them depend on the identity, since we are not trying to hide that in this stage). $\mathcal{F}_{\text{VOTE}}$ makes sure that the credentials are unique, and outputs them along with the identity of the voter to everybody else.

In the case of corrupted voters, there are quite a few checks to perform to ensure that all the properties are satisfied, but note that these are internal and only to establish a correct tally, and the potentially invalid credentials are also output to everybody else. This is because we want to model a public channel that nobody has control over.

3. Ballot casting

- On input $(\texttt{CastBallot}, sid)$ from **EA**:

i. If the election status is 'Credential', set it to 'Cast', otherwise ignore.

ii. Output $(\texttt{CastBallot}, sid)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

If the election status is not 'Cast', ignore all other `CastBallot` queries.

- On input $(\texttt{CastBallot}, sid, o_i)$ from $\mathbf{V_i}$:

  For honest and eligible voters:

  i. If there is $(\mathbf{V_i}, b'_i, o_i, 1)$ in $\mathcal{L}_B$ for some b'_i or there is no $(\mathbf{V_i}, c'_i, 1)$ for some c'_i in $\mathcal{L}_C$, ignore the query. [A single ballot per honest voter with a credential.]

  ii. Compute $b_i \leftarrow \texttt{GenBallot}()$.

  iii. If $b_i = b'_i$ for some $(\mathbf{V'_i}, b'_i, o'_i, 1)$ in $\mathcal{L}_B$, abort. [Uniqueness of ballots.]

  iv. Add $(\mathbf{V_i}, b_i, o_i, 1)$ to $\mathcal{L}_B$ and output $(\texttt{Ballot}, sid, b_i)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

  For corrupted voters:

  i. Hand $(\texttt{CastBallot}, sid, \mathbf{V_i})$ to $\mathcal{S}$.

  ii. Wait to receive $(\texttt{CastBallot}, sid, \mathbf{V_i}, o'_i, b'_i)$ from $\mathcal{S}$.

  iii. Set $v = 0$ if any of the following are true:

  (a) $\mathbf{V_i} \notin E$. [Ballots of ineligible voters are not valid.]

  (b) There is $(\mathbf{V_i}, b_i, o_i, 1)$ in $\mathcal{L}_B$ for some b_i, o_i. [More than one ballot per voter cannot be valid.]

  (c) $b_i = b'_i$ for some $(\mathbf{V_i}, b_i, o_i, 1)$ in $\mathcal{L}_B$. [A non-unique ballot is not valid.]

  (d) There is no $(\mathbf{V_i}, c_i, 1)$ in $\mathcal{L}_C$ for some c_i. [A ballot from a voter without a valid credential is not valid.]

  iv. Otherwise set $v = 1$.

  v. Add $(\mathbf{V_i}, b'_i, o'_i, v)$ to $\mathcal{L}_B$ and output $(\texttt{Ballot}, sid, b'_i)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

In the third stage, it is crucial that the ballot does not depend on the voter's identity nor their vote, since it will be output to all parties including the adversary and we want $\mathcal{F}_{\text{VOTE}}$ to preserve voter privacy. At the same time, the ballots have to be bound to votes and also have to be unique.

4. Ballot opening (tally)

- On input $(\texttt{OpenBallot}, sid)$ from $\mathbf{EA}$:

  i. If the election status is 'Cast', set it to 'Open', otherwise ignore.

  ii. Output $(\texttt{OpenBallot}, sid)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

If the election status is not 'Open', ignore all other commands.

- On input $(\texttt{OpenBallot}, sid)$ from $\mathbf{V_i}$:

  For honest and eligible voters:

  i. If there is $(\mathbf{V_i}, o'_i, 1)$ in $\mathcal{L}_O$ for some o'_i, or there is no $(\mathbf{V_i}, b'_i, o_i, 1)$ for some b'_i in $\mathcal{L}_B$, or there is no $(\mathbf{V_i}, c'_i, 1)$ for some c'_i in $\mathcal{L}_C$, ignore the query. [A single vote per honest voter with a credential and a ballot.]

  ii. Take o_i from $(\mathbf{V_i}, b_i, o_i, 1)$ in $\mathcal{L}_B$.

  iii. Add $(\mathbf{V_i}, o_i, 1)$ to $\mathcal{L}_O$ and output $(\texttt{Vote}, sid, o_i)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

  For corrupted voters:

  i. Hand $(\texttt{OpenBallot}, sid, \mathbf{V_i})$ to $\mathcal{S}$.

  ii. Wait to receive $(\texttt{OpenBallot}, sid, \mathbf{V_i}, o'_i)$ from $\mathcal{S}$.

  iii. Set $v = 0$ if any of the following are true:

  (a) $\mathbf{V_i} \notin E$. [Votes of ineligible voters are not valid.]

  (b) There is $(\mathbf{V_i}, o_i, 1)$ in $\mathcal{L}_O$ for some o_i. [More than one vote per voter cannot be valid.]

  (c) There is no $(\mathbf{V_i}, c_i, 1)$ in $\mathcal{L}_C$ for some c_i or there is no $(\mathbf{V_i}, b_i, o_i, 1)$ for some b_i in $\mathcal{L}_B$. [A vote from a voter without a valid credential or ballot is not valid.]

  (d) There is $o_i \neq o'_i$ in some $(\mathbf{V_i}, b_i, o_i, 1)$ in $\mathcal{L}_B$. [A vote different than the one cast is not valid.]

  iv. Otherwise set $v = 1$.

  v. Add $(\mathbf{V_i}, o'_i, v)$ to $\mathcal{L}_O$ and output $(\texttt{Ballot}, sid, o'_i)$ to all voters $\mathbf{V_1}, \ldots, \mathbf{V_n}$ and $\mathcal{S}$.

- On input $(\texttt{Tally}, sid)$ from party P:

  1. Compute $T = \{o_i\}$ for o_i from $\mathcal{L}_O$ of the form $(\mathbf{V_i}, o_i, 1)$.

  2. Output $(\texttt{Tally}, sid, T)$ to P.

In the final stage, the voters can call $\mathcal{F}_{\text{VOTE}}$ to reveal their votes and therefore have them counted in the tally. As in the earlier stages, invalid openings (such as for different votes than the ones cast) will be output but will be marked as invalid internally so that at any point, $\mathcal{F}_{\text{VOTE}}$ can return the correct tally.

5.2 Discussion of properties

No delayed output. The adversary cannot block messages from appearing on the broadcast channel. This is not strictly required but it models our network assumptions that the adversary cannot target a party to stop it from participating in the protocol.

The stages do not overlap. The election authority commands when stages begin, with the exception of the transition between 'Setup' and 'Credential' which happens automatically once $\mathcal{F}_{\text{VOTE}}$ has the algorithms. In either case it is certain that a new stage cannot begin before the previous one has ended.

Honest voters are assumed to be eligible. We do not need to explicitly model ineligible honest voters since they would simply not participate in the protocol (except as passive observers).

One unique credential, ballot and vote per voter. Honest voters only submit a single query during each stage. Corrupted voters are free to post anything as many times as they wish, but invalid posts will not verify as valid.

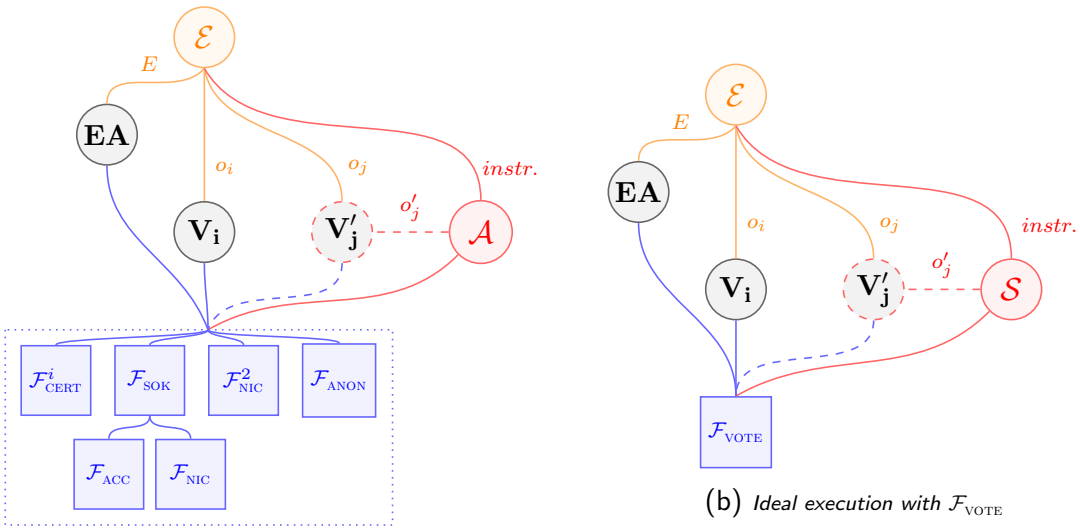
Adversarial algorithms. `GenCredential` and `GenBallot` take no input, and so cannot leak anything about the voter.

Corruption model. The standard corruption model of [Can13] is not enough because it only allows for corruption of the parties' input, but we want to let the adversary post anything on the public channel, so we expand what happens upon corruption accordingly.

Checks on corrupted voters. Allowing the adversary to supply the output means that some checks need to be performed on the values to determine which should be counted as valid.

Capturing validity. A single validity bit represents several properties of credentials/ballots/votes (saved during generation to distinguish honest voters and corrupted voters trying to impersonate them, and checked again during verification in case of credentials/ballots that have not been posted yet).

Verification is not modeled. Although voters in the hybrid protocol have to verify the contents of the board in order to compute the correct tally, the ideal functionality is trusted to compute it directly. This is acceptable since we were aiming to only capture privacy and correctness properties.



(a) *Hybrid execution with* Π_{ZV}

(b) *Ideal execution with* $\mathcal{F}_{\text{VOTE}}$

Figure 5.1: Execution in the real vs. ideal model.

5.3 Proof of Π_{zv} realizing $\mathcal{F}_{\text{vote}}$

We will show the following theorem.

Theorem 5.3.1. *Let* $\mathcal{H}$ *=(*$\mathcal{F}_{\text{ANON}}$, $\{\mathcal{F}^i_{\text{CERT}}\}$, $\mathcal{F}_{\text{SOK}}$ ($\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ACC}}$), $\mathcal{F}^2_{\text{NIC}}$*).* Π_{ZV} *UC-realizes* $\mathcal{F}_{\text{VOTE}}$ *in* $\mathcal{H}$*-hybrid model in the presence of static adversaries.*

Proof. We refer to the standard definitions and theorems of the UC framework outlined in Section 2.2. By Definition 2.2.3, the theorem that we want to prove is equivalent to Π_{ZV} UC-emulating the ideal protocol for $\mathcal{F}_{\text{VOTE}}$. Denote this protocol by $\mathcal{I}_{\mathcal{F}}$. What we want to show is the following claim (by Definition 2.2.2):

For all PPT adversaries $\mathcal{A}$ *there exists a PPT simulator* $\mathcal{S}$ *such that for all PPT environments* $\mathcal{E}$ *we have*

$$\text{EXEC}_{\mathcal{I}_{\mathcal{F}},\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\Pi_{\text{ZV}},\mathcal{A},\mathcal{E}}.$$

The environment $\mathcal{E}$ will supply the inputs o_i of all voters $\mathbf{V_i}$ in both the ideal and real execution. We assume that $t < n$ out of the n eligible voters will be corrupted from the outset of the execution, according to the definition of a static adversary.

$\mathcal{S}$ will internally simulate the real execution for $\mathcal{A}$ by playing the part of the honest parties and hybrid functionalities and forwarding all communication from $\mathcal{E}$. In the ideal execution, it will submit $\mathcal{A}$'s corruption requests to $\mathcal{F}_{\text{VOTE}}$ and forward messages from $\mathcal{A}$ to $\mathcal{E}$.

We construct $\mathcal{S}$ according to the following:

1. Wait to receive $(\texttt{Setup}, sid, E)$ from $\mathcal{F}_{\text{VOTE}}$.

2. Simulate the **Setup** stage of Π_{ZV}:
   - Play the part of **EA** in announcing the stages of the protocol.
   - Simulate initializing $\mathcal{F}_{\text{CERT}}^{EA}$, $\{\mathcal{F}_{\text{CERT}}^{i}\}$ for honest $\mathbf{V_i}$ by handing (`Setup`, sid') to $\mathcal{A}$ for suitable sid'. Receive the signing and verification algorithms $\texttt{Sign}_{\text{CERT}}^{i}$ and $\texttt{Verify}_{\text{CERT}}^{i}$ (if not, abort the ideal execution).
   - Respond to `Setup` queries for $\mathcal{F}_{\text{ANON}}$ and $\mathcal{F}_{\text{CERT}}^{j}$ for corrupted $\mathbf{V'_j}$.
   - Similarly, simulate the initialization of $\mathcal{F}_{\text{SOK}}$ ($\mathcal{F}_{\text{NIC}}$,$\mathcal{F}_{\text{ACC}}$) by accepting the verification algorithms and parameters of commitment and accumulator schemes given by $\mathcal{A}$.
   - Simulate **EA**'s interaction with $\mathcal{F}_{\text{ANON}}$ by sending ($\texttt{Post}, sid_a, \{E\}_{\mathbf{EA}}$) to $\mathcal{A}$ where $\{E\}_{\mathbf{EA}} = (E, \mathbf{EA}, \texttt{Sign}_{\text{CERT}}^{EA}(E))$.
3. Define the algorithms `GenCredential`(), `GenBallot`() as `TrapCom`(*params*, *trapdoor*) from the provided commitment scheme algorithms[1].

   Send (`Setup`, sid, `GenCredential`, `GenBallot`) to $\mathcal{F}_{\text{VOTE}}$.
4. Begin receiving ($\texttt{Credential}, sid, \mathbf{V_i}, c_i$) from $\mathcal{F}_{\text{VOTE}}$ for honest $\mathbf{V_i}$ and ($\texttt{GenCredential}, sid, \mathbf{V_i}$) for corrupted voters.
5. Simulate the **Credential generation** stage of Π_{ZV}:
   - For honest $\mathbf{V_i}$: generate a random S, sign each credential, simulate posting $\{c_i\}_{\mathbf{V_i}}$ in the same way as outlined earlier for **EA** and correctly respond to verification requests.
   - For corrupted $\mathbf{V_i}$: respond to all $\mathcal{F}_{\text{NIC}}$-`Commit` queries with valid (c, r) pairs. Sign and simulate posting anything requested by $\mathcal{A}$ via the corrupted voters (which may include invalid signatures).
6. Begin sending ($\texttt{GenCredential}, sid, \mathbf{V_i}, \mathbf{V'_i}, c'_i$) to $\mathcal{F}_{\text{VOTE}}$ where c'_i is the output received from $\mathcal{A}$ on behalf of a corrupted voter $\mathbf{V_i}$ and $\mathbf{V'_i}$ is a potentially different identity (in case $\mathcal{A}$ tries to impersonate a different voter).
7. Wait to be notified of the status change by $\mathcal{F}_{\text{VOTE}}$ with ($\texttt{CastBallot}, sid$). Collate all valid credentials received into the set C.
8. Begin receiving ($\texttt{Ballot}, sid, b_i$) from $\mathcal{F}_{\text{VOTE}}$ on behalf of honest voters and ($\texttt{GenBallot}, sid, \mathbf{V_i}$) for corrupted voters.
9. Simulate the **Ballot casting** stage of Π_{ZV}:
   - For honest ballots: randomly pair each b_i with some honest $\mathbf{V_i}$ (that is associated with c_i and S) and compute $a \leftarrow \texttt{Acc}(\texttt{Acc}(u, C \backslash \{c_i\}), \{c_i\})$ and $\sigma \leftarrow \texttt{SimSign}(b_i, (a, S))$. If $\texttt{Verify}_{\text{SOK}}(b_i, (a, S), \sigma) \neq 1$, choose a

[1] We do distinguish between $\mathcal{F}_{\text{NIC}}$ within $\mathcal{F}_{\text{SOK}}$ and $\mathcal{F}_{\text{NIC}}^{2}$, so to be precise if $\mathcal{A}$ provides different algorithms or parameters for these then `GenCredential` and `GenBallot` will also use different versions of `TrapCom`.

different pair and repeat until all verifications that $\mathcal{F}_{\text{SOK}}$ would have done pass[2]. Simulate posting $(b_i, (\sigma, S))$ to $\mathcal{A}$.

- For corrupted $\mathbf{V_i}$: respond to requests simulating $\mathcal{F}_{\text{ACC}}$ or $\mathcal{F}_{\text{SOK}}$.

10. Begin sending $(\texttt{CastBallot}, sid, \mathbf{V_i}, o_i', b_i')$ to $\mathcal{F}_{\text{VOTE}}$ where o_i', b_i' is the output provided by $\mathcal{A}$ on behalf of the corrupted voters.
11. Wait to be notified of the status change by $\mathcal{F}_{\text{VOTE}}$ with $(\texttt{OpenBallot}, sid)$.
12. Begin receiving $(\texttt{Vote}, sid, o_i)$ from $\mathcal{F}_{\text{VOTE}}$ on behalf of honest voters and $(\texttt{OpenBallot}, sid, \mathbf{V_i}, o_i')$ for corrupted voters.
13. Simulate the **Ballot opening** stage of Π_{ZV}:
    - For honest votes: randomly pair each o_i with some honest b_i received in the previous phase and like $\mathcal{F}_{\text{NIC}}$ check that $\texttt{Verify}_{\text{NIC}}(b_i, (o_i, r_i)) = 1$ for some $r_i \leftarrow \texttt{TrapOpen}(o_i, info)$. If not, choose a different pairing. Post $(b_i, (o_i, r_i))$ to $\mathcal{A}$.
    - For corrupted $\mathbf{V_i}$: respond to $\mathcal{A}$'s requests to the ideal functionalities.
14. Begin sending $(\texttt{OpenBallot}, sid, \mathbf{V_i}, o_i')$ to $\mathcal{F}_{\text{VOTE}}$ where o_i' is the output provided by $\mathcal{A}$ on behalf of the corrupted voters.
15. Within the simulation, whenever $\mathcal{A}$ requests the corrupted voters to output the **tally**, relay the request to $\mathcal{F}_{\text{VOTE}}$.

We can show that for any fixed sets of inputs and corrupted voters (which may or may not comply with the protocol), the messages observed on the anonymous channel in a run of Π_{ZV} with $\mathcal{A}$ are the same as in a run of $\mathcal{F}_{\text{VOTE}}$ with $\mathcal{S}$.

Within each stage of both the real and ideal executions, the contents of the bulletin board are summarized in the following list (where we do not show invalid entries but assume that some may have been published). Note that all parties have access to `Setup` parameters and algorithms of the commitment and accumulator schemes. They are not distributed directly through the broadcast channel, but they are available to any party upon request.

Public view (developed in real time):

1. *Setup*
2. $C = [\,\{c\}_{\mathbf{V_i}} \mid$ credential c for eligible $\mathbf{V_i}\,]$
3. $B = [\,(b_i, (\sigma, S)) \mid$ signature σ on ballot b_i proving knowledge of $c \in C$ such that c is a commitment to $S\,]$
4. $T = [\,(b_i, (o_i, r_i)) \mid$ vote o_i committed to in $b_i\,]$

[2]Since these are honest voters, eventually at least one pairing will have to verify. If none does, the simulator can abort just as $\mathcal{F}_{\text{SOK}}$ would have.

First, note that what happens to ineligible voters in the ideal execution is the same as in the hybrid protocol. Anyone acting as a verifier in Π_{ZV} will be able to determine the invalid credentials by the signatures and hence check which votes are invalid at the end of the protocol. Similarly, $\mathcal{S}$ will ensure that the simulated data it posts is not valid for those voters, and $\mathcal{F}_{\text{VOTE}}$ will not include their votes in the tally.

The real adversary $\mathcal{A}$ can see all inputs and outputs of the voters it corrupts, as well as modify the contents of their messages (within the context of the protocol). Even if $\mathcal{A}$ collaborates with the environment $\mathcal{E}$, this does not give $\mathcal{E}$ any advantage in distinguishing the executions. The honest votes published by $\mathcal{S}$ in the ideal execution are assigned to a random permutation of voter identities, but the properties of $\mathcal{F}_{\text{ACC}}$ and $\mathcal{F}_{\text{SOK}}$ guarantee that the signatures of knowledge verify (since $\mathcal{S}$ chooses valid credentials from the set to compute them) and no other information is leaked in the process.

Note that this comparison only works in the presence of static adversaries, as otherwise the environment could corrupt a voter after $\mathcal{S}$ executes stage 3, and $\mathcal{S}$ would not be able to fake the internal "state" of the voter correctly since its pairings between votes and voters are completely random. □

Combining Theorem 5.3.1 that we have just proven with Theorem 4.3.1 via the transitive property of UC-emulation (and noting the assumptions that we made at the beginning of Chapter 3), we obtain the result that we sought:

Theorem 5.3.2. ZEROVOTE *UC-realizes* $\mathcal{F}_{\text{VOTE}}$.

Chapter 6

Conclusion

6.1 Summary and limitations

The main objective of this project was to set Zerovote in a formal model and prove it preserves voter privacy, which has been completed. A number of obstacles encountered on the way have shaped the path taken towards this goal: we judged that it is better to use an intermediate step in the form of a hybrid protocol, and consequently had to search for suitable composable functionalities as well as give a definition for and prove a functionality for one of the primitives, the secure accumulator. Modularity is at the heart of the motivation for the UC framework, and for this reason we believe that the accumulator functionality might be of independent interest.

However, the current specification also has a number of limitations:

- As it stands, the original UC framework requires that each instance of Zerovote uses a separate instance of the anonymous broadcast channel, which in real terms would mean that every election would require its own blockchain[1]. The certification scheme is affected in a similar way, with each instance tied to a new instance of the certification authority that has to be disjoint from all other instances. Getting around this while preserving the composition results is not simple. The *generalized UC* [CDPW07] is one attempt to formalize the security of protocols with global setup.
- In the last stage of Zerovote, anyone can compute a partial tally of votes in real time. Fairness is therefore not satisfied – though voters cannot change their vote based on the votes they see, they can still decide whether or not to open their ballot. This could be mitigated by having each voter generate shares of a keypair that would be associated with their credentials,

[1] As discussed at the beginning of Chapter 3, we have made the assumption that we can realize the anonymous broadcast channel on the blockchain, without modeling the details of the network. While blockchains seem like a natural way to obtain the channel, we stress that our protocol is not dependent on them and could make use of any future constructions as long as they can be proven to realize the channel in UC.

encrypting the votes to the large public key and requiring voters to post their private share after all (encrypted) votes have been submitted. Votes would then not be revealed until all voters submit their shares. While some voters could still block the elections by not participating, they could be identified and therefore held accountable.

- Another property not satisfied by Zerovote is coercion resistance, since coerced voters can release the secret information linking them to their credential and ballot which the coercer can easily verify. Some e-voting protocols solve this problem by providing voters with the means to give the coercer fake information which he cannot distinguish from the real secrets, but which is distinguishable for the tallying authorities if used to form ballots [JCJ05]. It is apparent that this solution would not work for self-tallying protocols, since by definition anyone can act as a tallier, including the adversary.
- The last issue that should be noted concerns the setup parameters. For simplicity, we assumed a trusted election authority that generates them and safeguards the corresponding secrets. In a real implementation, it would be desirable to generate the parameters in a more distributed way, but it is not clear to what extent this is feasible for all primitives used by Zerovote[2].

6.2 Future work

With the focus of this project being on privacy, we did not investigate other important properties such as verifiability. Given the public setting of the protocol, we believe that a proof of (individual and universal) verifiability would be possible and not too different from the treatment for privacy, though the need for incorporating verification requests could make it more complex. A less explored avenue for potential future work would also be in verifiability of eligibility (as opposed to only satisfying eligibility as part of the correctness properties).

More work could also be done in the areas outlined in the previous section where Zerovote falls short, i.e. provably obtaining fairness, coercion resistance and distributed setup. In our evaluation we also did not analyze the efficiency of the protocol, as it is a distinct issue from proving security and largely dependent on the implementation.

In a larger context, we have shown that blockchain-based self-tallying protocols are viable candidates for achieving open but privacy-preserving electronic elections, though they come with issues of their own. It remains to be seen whether their security guarantees could be improved and to what extent they could be adopted in practice.

[2]See [San99] for a proposal for accumulators without a trapdoor that use RSA moduli of unknown factorization (RSA-UFOs). However, the scheme is not practical.

Bibliography

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In *USENIX Security Symposium*, pages 335–348. USENIX Association, 2008.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014.

[BCG+15] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. SoK: A comprehensive analysis of game-based ballot privacy definitions. In *IEEE Symposium on Security and Privacy*, pages 499–516. IEEE Computer Society, 2015.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356. Springer, 2017.

[BPR+04] Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach. Hack-a-Vote: Security issues with electronic voting systems. *IEEE Security & Privacy*, 2(1):32–37, 2004.

[Can04] Ran Canetti. Universally composable signatures, certification and authentication. *IACR Cryptology ePrint Archive*, 2003:239, 2004.

[Can13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2005–2013. `https://eprint.iacr.org/2000/067` to access all versions.

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society, 2008.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.

[CDR16] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. UC commitments for modular protocol design and applications to revocation

and attribute tokens. In *CRYPTO (3)*, volume 9816 of *Lecture Notes in Computer Science*, pages 208–239. Springer, 2016.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[CGGI13] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachène. A generic construction for voting correctness at minimum cost - application to Helios. *IACR Cryptology ePrint Archive*, 2013:177, 2013.

[CGK+16] Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. SoK: Verifiability notions for e-voting protocols. In *IEEE Symposium on Security and Privacy*, pages 779–798. IEEE Computer Society, 2016.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2002.

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 78–96. Springer, 2006.

[Com02] The Electoral Commission. The implementation of electronic voting in the UK: Research summary. 2002. `http://www.electoralcommission.org.uk/__data/assets/electoral_commission_pdf_file/0005/16097/Implementationofe-votingsummary_6720-6268__E__N__S__W__.pdf`.

[CPS07] Ran Canetti, Rafael Pass, and Abhi Shelat. Cryptography from sunspots: How to use an imperfect reference string. In *FOCS*, pages 249–259. IEEE Computer Society, 2007.

[CSV16] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global PKI. In *Public Key Cryptography (2)*, volume 9615 of *Lecture Notes in Computer Science*, pages 265–296. Springer, 2016.

[DKR09] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[ED10] Saghar Estehghari and Yvo Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking Helios 2.0 as an example. In *EVT/WOTE*. USENIX Association, 2010.

[FOO92] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *AUSCRYPT*, volume

718 of *Lecture Notes in Computer Science*, pages 244–251. Springer, 1992.

[GKTP16] J. Paul Gibson, Robert Krimmer, Vanessa Teague, and Julia Pomares. A review of e-voting: the past, present and future. *Annales des Télécommunications*, 71(7-8):279–286, 2016.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[GRBR13] Gurchetan S. Grewal, Mark Dermot Ryan, Sergiu Bursuc, and Peter Y. A. Ryan. Caveat Coercitor: Coercion-evidence in electronic voting. In *IEEE Symposium on Security and Privacy*, pages 367–381. IEEE Computer Society, 2013.

[Gro04] Jens Groth. Evaluating security of voting schemes in the universal composability framework. In *ACNS*, volume 3089 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2004.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 444–459. Springer, 2006.

[Gro09] Jens Groth. Homomorphic trapdoor commitments to group elements. *IACR Cryptology ePrint Archive*, 2009:7, 2009.

[HBHW16] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2016. Latest version: `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`.

[JCJ05] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *WPES*, pages 61–70. ACM, 2005.

[Jus07] The government's response to the Electoral Commission's recommendations on the May 2007 electoral pilot schemes. 2007. `http://webarchive.nationalarchives.gov.uk/20110411090013/http://www.justice.gov.uk/publications/docs/gov-response-elec-comm.pdf`.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society, 2016.

[KY02] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 2002.

[Lin17] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society, 2013. `http://zerocoin.org/media/pdf/ZerocoinOakland.pdf`.

[MN06] Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 373–392. Springer, 2006.

[MN10] Tal Moran and Moni Naor. Split-ballot voting: Everlasting privacy with distributed trust. *ACM Trans. Inf. Syst. Secur.*, 13(2):16:1–16:43, 2010.

[MSH17] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *Financial Cryptography*, volume 10322 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2017.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. `https://bitcoin.org/bitcoin.pdf`.

[Ped91] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

[RG17] Robert Riemann and Stéphane Grumbach. Distributed protocols at the rescue for trustworthy online voting. In *ICISSP*, pages 499–505. SciTePress, 2017.

[RS06] Peter Y. A. Ryan and Steve A. Schneider. Prêt à voter with re-encryption mixes. In *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 313–326. Springer, 2006.

[San99] Tomas Sander. Efficient accumulators without trapdoor extended abstracts. In *ICICS*, volume 1726 of *Lecture Notes in Computer Science*, pages 252–262. Springer, 1999.

[SP15] Alan Szepieniec and Bart Preneel. New techniques for electronic voting. *IACR Cryptology ePrint Archive*, 2015:809, 2015.

[TT17] Pavel Tarasov and Hitesh Tewari. Internet voting using Zcash. *IACR Cryptology ePrint Archive*, 2017:585, 2017.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. Latest version: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[WWIH12] Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. Attacking the Washington, D.C. internet voting system. In *Financial Cryptography*, volume 7397 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2012.

Appendix A

Implementation of real primitives

The choice of specific implementations of the primitives in our protocol is not relevant to the security analysis as long as they are proven to satisfy certain properties, but for completeness we give the ones used by Zerocoin here.

A.1 Commitment scheme

The standard Pedersen's commitment scheme [Ped91] is used.

CSetup(1^λ)

1 : generate p, q prime such that $q \mid p-1$
2 : $\mathbb{G}_q \leftarrow$ subgroup of $\mathbb{Z}_p^*$ of order q
3 : $g, h \xleftarrow{R} \mathbb{G}_q$
4 : **return** (p, q, g, h)

Com($params, m$)

1 : $(p, q, g, h) \leftarrow params$
2 : $r \xleftarrow{R} \mathbb{Z}_q$
3 : **return** $(g^m h^r \mod p, r)$

VfCom($params, c, m, r$)

1 : $(p, q, g, h) \leftarrow params$
2 : **if** $c = g^m h^r \mod p$ **then return** 1
3 : **else return** 0

Figure A.1: Pedersen's commitment scheme

A.2 Secure accumulator

The Strong RSA accumulator construction [CL02] is given in Figure A.2.

`ASetup` generates the modulo N for accumulator operations and u which will serve as the basis for computing the credential accumulator in Zerovote. QR_N is the group of quadratic residues mod N. The generated safe primes may be discarded after obtaining N (if not, they could be used by the election authority to delete credentials from the accumulator, as we are using the dynamic accumulator

ASetup(1^λ)

1 : $N \leftarrow \texttt{GenRSA}(1^\lambda)$
2 : sample $u \in QR_N, u \neq 1$
3 : $\mathcal{X}_{A,B} \leftarrow \{e \text{ prime} : e \neq p', q', A \leq e \leq B\}$
4 : **return** $(N, u, \mathcal{X}_{A,B})$

Acc($params, w, C$)

1 : $(N, u, \mathcal{X}_{A,B}) \leftarrow params$
2 : $a \leftarrow w^{c_1 c_2 \cdots c_n} \mod N$ for $C = \{c_1, c_2, \ldots, c_n\}$
3 : **return** a

VfAcc($params, a, C, w$)

1 : $(N, u, \mathcal{X}_{A,B}) \leftarrow params$
2 : $(c_1, c_2, \ldots, c_n) \leftarrow C$
3 : $a' \leftarrow w^{c_1 c_2 \cdots c_n} \mod N$
4 : **if** $a' = a$ **and** $C \subset \mathcal{X}_{A,B}$ **then return** 1
5 : **else return** 0

Figure A.2: Strong RSA accumulator scheme

construction by [CL02]). The set $\mathcal{X}_{A,B}$ will serve as the set of permitted credential values in our protocol.

`Acc` and `VfAcc` essentially implement the accumulator function f except for accepting inputs directly as sets (so they can accumulate multiple items at a time), which is possible due to the quasi-commutativity property of the scheme and properties of exponentiation.

A.3 Signatures of knowledge

A skeleton for the construction proposed by [CL06] is given, adapted for our specific use with accumulators and commitments.

Sign($params, x, (a, S), (c, w, r)$)

1 : $\rho \leftarrow params$
2 : $enc \leftarrow \texttt{Enc}(PK, (x, (c, w, r)), R)$ for R random
3 : define $\mathcal{M}_L((a, S), (c, w, r))$ as the machine that accepts
4 : VfAcc($params, a, C \backslash \{c\}, w$) $= 1 \wedge$ VfCom($params, c, S, r$) $= 1$
5 : $\pi \leftarrow \texttt{NIZKProve}(\rho, (x, \mathcal{M}_L, (a, S), enc, PK),$
6 : $[\exists((c, w, r), R) : enc = \texttt{Enc}(PK, (x, (c, w, r)), R) \wedge \mathcal{M}_L((a, S), (c, w, r))], ((c, w, r), R))$
7 : **return** $\sigma = (enc, \pi)$

Verify($params, x, (a, S), \sigma$)

1 : $\rho \leftarrow params$
2 : $(enc, \pi) \leftarrow \sigma$
3 : $b \leftarrow \texttt{NIZKVerify}(\rho, \pi, (x, \mathcal{M}_L, (a, S), enc, PK),$
4 : $[\exists((c, w, r), R) : enc = \texttt{Enc}(PK, (x, (c, w, r)), R) \wedge \mathcal{M}_L((a, S), (c, w, r))])$
5 : **return** b

Figure A.3: Signatures of knowledge scheme

`Sign` produces a zero-knowledge signature of knowledge on x that can only be

made by someone with knowledge of c, w, r such that w is a witness to the credential c accumulated in a and c is a commitment to the serial number S and a random value r, using the verification algorithms `VfCom` and `VfAcc` that we defined in earlier sections to compose a language of input-witness pairs that satisfy the given requirements. All other inputs are assumed public.

`Verify` checks that σ is indeed a signature on x signed by someone with knowledge of some credential $c \in C$ such that c is a commitment to the serial number S and some random value.

The algorithms for generation and verification of σ must satisfy the conditions given in the construction in [CL06] (Section 3 and Appendix C). Namely, assuming a common random string ρ, we require ($\mathcal{G}$, `Enc`, `Dec`) to be a CPA secure dense cryptosystem[1] with a public key PK, and (`NIZKProve`, `NIZKSimsetup`, `NIZKSim`, `NIZKVerify`) to be a simulation-sound[2] non-interactive zero-knowledge proof system, both of which have implementations in the literature.

[1] Meaning an asymmetric encryption scheme secure against chosen-plaintext attacks even when using random public keys.

[2] A zero-knowledge proof system that satisfies *simulation-soundness* preserves soundness even against adversaries that have access to simulated proofs of arbitrary statements [Gro06].

Appendix B

On signatures, certificates and authenticated channels

B.1 Ideal signature functionality

We first give the signature functionality $\mathcal{F}_{\text{SIG}}$ as defined in [Can13] (2005 version) and also used in [CL06][1].

Functionality $\mathcal{F}_{\text{sig}}$:

- **Setup (key generation)**
  - On first input $(\texttt{Setup}, sid)$ from party P:

    If $sid = (P, sid')$ for some sid', hand $(\texttt{Setup}, sid)$ to $\mathcal{A}$, else ignore.
  - On input $(\texttt{Algorithms}, sid, \texttt{Verify}, \texttt{Sign})$ from $\mathcal{A}$ for deterministic polynomial-time ITM `Verify` and PPT ITM `Sign`:

    Store the algorithms and output $(\texttt{VerificationAlgorithm}, sid, \texttt{Verify})$ to P.
  - On input $(\texttt{VerificationAlgorithm}, sid)$ from party V:

    Output $(\texttt{VerificationAlgorithm}, sid, \texttt{Verify})$ to V.
- **Signature generation**
  - On input $(\texttt{Sign}, sid, m)$ from party P:
    1. Set $\sigma \leftarrow \texttt{Sign}(m)$.
    2. If $\texttt{Verify}(m, \sigma) \neq 1$, abort.

[1] We change the format of errors in the definition to match our other functionalities, but this is purely a syntactical difference.

3. Record the entry (m, σ) and output $(\texttt{Signature}, sid, m, \sigma)$ to P.

- **Signature verification**
  - On input $(\texttt{Verify}, sid, m, \sigma, \texttt{Verify}')$ from party V:
    1. If $\texttt{Verify} = \texttt{Verify}'$, $\texttt{Verify}(m, \sigma) = 1$, the signer is not corrupted and no entry (m, σ') for any σ' is recorded, abort.
    2. Output $(\texttt{Verified}, sid, m, \texttt{Verify}'(m, \sigma))$ to V.

Of course, there are several ways to define a signature functionality, geared towards different uses and without a single accepted solution. We choose this one for a number of reasons: a generic equivalence result with traditional definitions of security for signature schemes, that it was defined with the goal of easy incorporation into larger protocols, and consistency with the signature of knowledge definition given in a later section.

It is important to note several properties of this definition, as discussed in more detail in [Can13] (2005, Section 7.2.1). A consequence of incorporating the identity of the signer into *sid* is that each party requires a new instance of the functionality, and in protocols with multiple instances we rely on the composition theorem to ensure its security properties still hold in this setting. However, it does not mean that the functionality provides a binding between signatures and identities, merely between signatures and keys (i.e. verification algorithms). The next subsection addresses the steps needed to achieve that.

The functionality can be realized by any EU-CMA secure signature scheme for a simple protocol Π_Σ given in [Can13] (we rephrase Claim 29).

Theorem B.1. *Π_Σ UC-realizes $\mathcal{F}_{\text{SIG}}$ if and only if the underlying signature scheme $\Sigma = (\mathit{Gen}, \mathit{Sig}, \mathit{Ver})$ is* EU-CMA.

B.2 On the need for an authenticated channel

Both [Can04] and [Can13] (2005 version) give definitions of $\mathcal{F}_{\text{SIG}}$, which they then use to build an authenticated channel functionality $\mathcal{F}_{\text{AUTH}}$. The formulations have certain differences (the key one being that only the later [Can13] definition uses algorithms supplied by the adversary[2] instead of letting the adversary directly compute the signatures), but they share a common structure and the proofs and other results that are built from them are similar.

[2] At first glance, it seems counterintuitive to use algorithms generated by the adversary, but this in fact strengthens the definition by showing that despite this, the security of the scheme is satisfied. The general concept is referred to as *allowed adversarial influence*, and when the functionality explicitly hands $\mathcal{A}$ some values, *allowed information leakage* [Can13].

Functionality $\mathcal{F}_{\text{auth}}$:

- On input $(\texttt{Send}, sid, m)$ from party P:

  If $sid = (P, R, sid')$ for some R and sid', then send a public delayed output $(\texttt{Sent}, sid, m)$ to R, otherwise ignore.

- On input $(\texttt{Corrupt-sender}, sid, m')$ from $\mathcal{A}$:

  If the delayed output is not delivered yet, then output $(\texttt{Sent}, sid, m')$ to R instead.

As shown in both cited sources, building $\mathcal{F}_{\text{AUTH}}$ is impossible in the bare model and so always requires some setup assumptions. A somewhat unintuitive path towards an ideal authenticated channel, presented in detail in [Can04], then consists of first defining a certificate authority functionality $\mathcal{F}_{\text{CA}}$, then defining a certification functionality $\mathcal{F}_{\text{CERT}}$ which can be realized in the $(\mathcal{F}_{\text{SIG}},\mathcal{F}_{\text{CA}})$-hybrid model, and finally defining the authentication functionality $\mathcal{F}_{\text{AUTH}}$ and showing that it can be realized in the $\mathcal{F}_{\text{CERT}}$-hybrid model.

In Zerovote, the only purpose of having an authenticated channel would be to model a medium through which the voters and the election authority can identify themselves with their public keys. In other words, we require our signatures to be bound to identities, which is exactly what $\mathcal{F}_{\text{CERT}}$ already provides. In the interface of $\mathcal{F}_{\text{CERT}}$ public keys become an "implementation detail" [Can04], so the correspondence with the real protocol becomes less intuitive, but still simpler than the essentially redundant solution of defining $\mathcal{F}_{\text{SIG}}$ separately and using $\mathcal{F}_{\text{AUTH}}$ only to distribute the keys.

Functionality $\mathcal{F}_{\text{ca}}$:

- On first input $(\texttt{Register}, sid, v)$ from party P:
  1. Hand $(\texttt{Registered}, sid, v)$ to $\mathcal{A}$ and wait for confirmation.
  2. If $sid = (P, sid')$ for some sid', record the pair (P, v).
- On input $(\texttt{Retrieve}, sid)$ from party P':
  1. Hand $(\texttt{Retrieve}, sid, P')$ to $\mathcal{A}$ and wait for confirmation.
  2. If there is a recorded pair (sid, v), output $(\texttt{Retrieve}, sid, v)$ to P', else output $(\texttt{Retrieve}, sid, \perp)$ to P'.

Appendix C

Alternative definition for secure accumulators

The relaxed definition of unforgeability is given below. The change consists of fixing the first parameter of f (hence we call it a "fixed basis") and understanding the accumulator purely in terms of sets of accumulated values. We will show it is weaker than the original one, since we require that the forged witness corresponds to a well-formed set rather than allowing the adversary to give us any object from $\mathcal{U}'_f$ (while it could still be a valid accumulator for some values, the original adversary does not need to know them to construct the forgery).

Definition C.1 (Fixed-basis accumulator)
$\text{SA} = (\mathcal{X}_\lambda, \mathtt{Gen})$ *is a fixed-basis accumulator scheme if it also satisfies:*

- ***Witness-set unforgeability***
  Let $\mathcal{U}'_f \times \mathcal{X}'_\lambda$ denote the domains for which the computational procedure for function $f \in F_\lambda$ is defined (so $\mathcal{U}_f \subseteq \mathcal{U}'_f, \mathcal{X}_\lambda \subseteq \mathcal{X}'_\lambda$). Let $g(x) = f(u, x)$ for fixed $u \in \mathcal{U}_f$.
  For all PPT adversaries $\mathcal{A}_\lambda$:

$$\Pr[f \leftarrow \mathtt{Gen}(1^\lambda); u \leftarrow \mathcal{U}_f; (x, W, X) \leftarrow \mathcal{A}_\lambda(g, \mathcal{U}_f) : X \subset \mathcal{X}_\lambda; W \subset \mathcal{X}'_\lambda; x \in \mathcal{X}'_\lambda; x \notin X; g(\{x\} \cup W) = g(X)] = \mathsf{negl}(\lambda)\,.$$

Theorem C.1. *If* SA *is a secure accumulator, then it is a fixed-basis accumulator.*

Proof. We will prove the contrapositive. Suppose $\text{SA} = (\mathcal{X}_\lambda, \mathtt{Gen})$ is not fixed-basis, i.e. it is not secure under witness-set unforgeability. Then there exists a forger $\mathcal{G}'$ that given $g, \mathcal{U}_f$ will produce a forged witness set W for $x \notin X$ with non-negligible probability. Then we can construct a forger $\mathcal{G}$ for witness unforgeability by taking the output of $\mathcal{G}'$ and computing $w = g(W)$, so that outputting (x, w, X) gives us a forgery $f(w, x) = g(\{x\} \cup W) = g(X) = f(u, X)$ with non-negligible probability. Hence SA is not a secure accumulator. □

Definition C.1 is not equal to Definition 3.2.2 (at least under the assumption of

hardness of the discrete logarithm). It can be shown for the strong RSA accumulator construction given in Definition 3.2.4 that proving the converse would require computing the discrete logarithm, since constructing a forger for the fixed-basis accumulator would necessitate finding the product of the elements of $W = (w_1, \ldots, w_m)$ for known w such that $w = u^{w_1 \cdots w_m} \mod n$.

We can now define a functionality $\mathcal{F}_{\text{ACC}}^{fb}$ based on the fixed-basis accumulator. It will store a table of records of the form [*accumulator*, *element*, *set*, *witness*], where *set* is understood to be the set that we wish to add *element* to, thereby obtaining an *accumulator* and a *witness* for *element* in $\{element\} \cup set$.

Functionality $\mathcal{F}_{\text{acc}}^{fb}$:

- **Setup**

  Same as $\mathcal{F}_{\text{ACC}}$, except the input of `Acc` is now only $X \subseteq \mathcal{X}_\lambda$ and u does not have to be in *params*.

- **Accumulation**

  - On input (`Accumulate`, *sid*, *elem*, *set*) from party P:

    1. If there is an entry [*accum*′, *elem*, *set*, *witness*′]:
       Output (`Accumulator`, *sid*, *accum*′, *witness*′) to P.
    2. If $elem \notin \mathcal{X}_\lambda$ or $set \not\subseteq \mathcal{X}_\lambda$, abort.
    3. Compute $witness \leftarrow \texttt{Acc}(set)$.
    4. If $witness \notin \mathcal{U}_{\texttt{Acc}}$, abort.
    5. If there is an entry [*accum*′, *elem*′, *set*′, *witness*′] that would violate either of the following conditions, abort:
       (a) Must have $witness = witness'$ if $set = set'$.
       (b) Must have $witness = accum'$ if $set = \{elem'\} \cup set'$.
    6. Compute $accum \leftarrow \texttt{Acc}(\{elem\} \cup set)$.
    7. If $accum \notin \mathcal{U}_{\texttt{Acc}}$ or $\texttt{Verify}(accum, elem, set, witness) \neq 1$, abort.
    8. If there is an entry [*accum*′, *elem*′, *set*′, *witness*′] that would violate either of the following conditions, abort:
       (a) Must have $accum = accum'$ if $\{elem\} \cup set = \{elem'\} \cup set'$.
       (b) Must have $accum = witness'$ if $\{elem\} \cup set = set'$.
    9. Record the entry [*accum*, *elem*, *set*, *witness*].
    10. Output (`Accumulator`, *sid*, *accum*, *witness*) to P.

- **Verification**
  - On input ($\texttt{Verify}, sid, accum, elem, set, witness$) from party P:
    1. If there is an entry $[accum, elem, set, witness]$:
       Output ($\texttt{Verified}, sid, 1$) to P.

       Else if $\texttt{Verify}(accum, elem, set, witness) = 1$ and there is an entry $[accum', elem', set', witness']$ that would violate either of the following conditions, abort:

       (a) If $witness = accum'$, then $elem' \in set$.

       (b) If $accum = witness'$, then $elem \in set'$.

       (c) If $accum = accum'$, then $elem \in \{elem'\} \cup set'$.

    2. Output ($\texttt{Verified}, sid, \texttt{Verify}(accum, elem, set, witness)$) to P.

Informally, in the accumulation part, the checks depending upon comparison with past entries ensure quasi-commutativity, while witness-set unforgeability is treated in the verification part. Consistency is implied by the determinism of `Verify` and by only allowing valid entries to be recorded in the table.

We can define a real protocol that makes use of the accumulator scheme:

Protocol $\Pi^{fb}_{\mathbf{acc}}$ for sa $= (\mathcal{X}_\lambda, \texttt{Gen})$ in the $\mathcal{F}^{\texttt{Gen}}_{\mathbf{crs}}$-hybrid model:

- **Setup**
  - On first input ($\texttt{Setup}, sid$), party P runs the following program:
    1. Send ($\texttt{CRS}, sid$) to $\mathcal{F}^{\texttt{Gen}}_{\text{CRS}}$ to receive ($\texttt{CRS}, sid, f, \mathcal{U}_f, u$).
    2. Store $params = (\mathcal{X}_\lambda, \mathcal{U}_\texttt{f})$ and define function $g(x) := f(u, x)$.
    3. Define $\texttt{Verify}(a, x, W, w) := w \stackrel{?}{=} g(W) \wedge a \stackrel{?}{=} g(\{x\} \cup W)$.
    4. Output ($\texttt{Algorithms}, sid, params, g, \texttt{Verify}$).
  - On input ($\texttt{Params}, sid$), P outputs ($\texttt{Params}, sid, params, g$).
  - On input ($\texttt{VerificationAlgorithm}, sid$), P outputs ($\texttt{VerificationAlgorithm}, sid, \texttt{Verify}$).
- **Accumulation**
  - On input ($\texttt{Accumulate}, sid, x, W$), P runs:
    1. If $x \notin \mathcal{X}_\lambda$ or $W \not\subseteq \mathcal{X}_\lambda$, abort.
    2. Compute $w \leftarrow g(W)$.
    3. Compute $a \leftarrow g(\{x\} \cup W)$.
    4. Output ($\texttt{Accumulator}, sid, a, w$).

- **Verification**
  - On input $(\texttt{Verify}, sid, a, x, W, w)$, P runs:
    1. If $x \notin \mathcal{X}_\lambda$, $W \not\subseteq \mathcal{X}_\lambda$, or $a, w \notin \mathcal{U}_f$, abort.
    2. Output $(\texttt{Verified}, sid, \texttt{Verify}(a, x, W, w))$.

We do not give proofs of realizability of $\mathcal{F}_{\text{ACC}}^{fb}$ as it is a weaker version and they would follow a similar structure as the proofs for $\mathcal{F}_{\text{ACC}}$ in Section 4.2.