Scytl

Innovating Democracy

Scytl sVote

Privacy Formal Proof Report

Software version 2.1

Document version 1.1

Scytl – Secure Electronic Voting

STRICTLY CONFIDENTIAL

Table of contents

1 Summary

In 2013 the Federal Chancellery published a new regulation for the authorization of Internet voting systems (VEleS)[1] that became enforced at the beginning of 2014. The VEleS regulation sets up a framework for the authorization of voting systems according to three different levels, which are directly linked to the amount of electorate that is able to vote through them. From the cantonal point of view, these levels limit the electorate up to 30%, 50% or 100%. To reach the two higher levels, VEleS requires the voting system to pass a certification process based on the security and verifiability properties of such system. In this aim, the certification process includes an examination of the cryptographic protocol, to guarantee it is compliant with the ordinance security requirements of a specific level (abstract model assumptions), by means of verifying a cryptographic and symbolic proof of the voting system to be certified (Req. 5.1.1 of the VEleS Technical Annex[2]).

For up to 50% electorate level, the voting system needs to provide cryptographic and symbolic proofs to demonstrate that the implemented cryptographic protocol provides **individual verifiability** under the reduced abstract model trust assumptions defined in section 4.1 of the VEleS Technical Annex.

However, for up to 100% electorate level, the system must provide proofs that demonstrate the protocol provides **complete verifiability** (including individual one) under the complete abstract model defined in section 4.3 of the VEleS Technical Annex. Furthermore, certification for the 100% electorate level requires not only proving verifiability properties (as in the 50% electorate level) but also voter privacy ones.

In 2017, Scytl sVote Voting Protocol was certified according to the 50% electorate level as compliant with the individual verifiability requirements of the VEleS regulation. To this end, cryptographic[3] and symbolic[4] proofs of the individual verifiability properties were provided. However, to achieve the 100% electoral level, Scytl sVote Voting Protocol has being revised[5] to be in line to the complete verifiability requirements. Therefore, new cryptographic and symbolic proofs of complete verifiability, and cryptographic and symbolic proofs of voter privacy, have been generated to prove the complete verifiability requirements.

In this document, we are introducing the paper that provides the symbolic proof of voter privacy of Scytl sVote Voting Protocol, according to the complete abstract model defined in the VEleS ordinance. The paper has been led by Véronique Cortier (CNRS/LORIA) and Mathieu Turuani (INRIA/LORIA), with the collaboration of Scytl's R&S department.

---

1 Swiss Federal Chancellery. Federal Chancellery Ordinance on Electronic Voting (VEleS) of 13 December 2013. (Status as of 1 July 2018)

2 Swiss Federal Chancellery. Technical and administrative requirements for electronic vote casting. Annex to the FCh Ordinance of 13 December 2013 on Electronic Voting (OEV, SR 161.116)

3 Scytl Secure Electronic Voting. Swiss Online Voting System Cryptographic proof of Individual Verifiability. 2017

4 Véronique Cortier, David Galindo, Mathieu Turuan. Analysis of Cast-as-Intended Verifiability and Ballot Privacy Properties for Scytl's Swiss On-line Voting Protocol using ProVerif (2017)

5 Scytl Secure Electronic. Scytl sVote. Protocol Specifications. V5.1 (2018)

2 Appendix

2.1 EV Solution Intellectual Property Rights Notice (the Notice)

Scytl sVote is part of a larger system called EV Solution, developed under the "Framework Agreement" entered into by and between Post CH Ltd (Swiss Post) and Scytl Secure Electronic Voting, S.A. (Scytl) on September 30th, 2015.

Parts of this EV Solution system and other relevant details are defined below.

2.1.1 Definitions

The following terms shall have the meanings specified below:

"**EV Solution**" means an online voting system consisting of the Scytl Standard Software (also referred to as Scytl sVote or Scytl Online Voting 2.0) in combination with the Swiss Post-Scytl Software, and all the associated middleware provided by Scytl as a bundle with the Scytl Standard Software and the Swiss Post-Scytl Software. Software below middleware (e.g. Linux OS and Windows OS and Oracle software) that are needed to run the EV Solution are not part of the EV Solution.

"**Intellectual Property Rights**" or "**IPRs**", for the purposes of this Notice and pursuant to the Framework Agreement, means copyright and patent rights (if any), know-how and trade secrets, performance rights and entitlements to such rights.

"**Scytl Online Voting 2.0**" is the brand name that was used to identify Scytl Standard Software in the market.

"**Scytl Standard Software**" means all software developed by Scytl for the EV Solution, whose architecture, specifications and capabilities are described in Scytl sVote documents, excluding Swiss Post-Scytl Software and software developed by Scytl independently to the EV Solution.

"**Software**" means software code (source code and object code), user interfaces and documentation (preparatory documentation and manuals) and including releases and patches etc.

"**Scytl sVote**" means the registered trademark proprietary to Scytl, that identifies Scytl Standard Software in the market.

"**Swiss Post-Scytl Software**" means the software developed for the EV Solution (excluding Scytl Standard Software) pursuant to the Framework Agreement. Swiss Post-Scytl Software comprises of the following:

i. Key Translation Module: A mapping service that translates external IDs to internal IDs for specific entities so that external systems can integrate with sVote.

ii. Swiss Post Integration Tools: A group of applications that allow the integration between Swiss Post's applications and sVote through file conversions.

iii. Swiss Post Voting Portal Frontend: Frontend application that guides the voters throughout all the voting steps enabling them to successfully cast a vote for a particular election.

2.1.2 Copyright notice

2.1.2.1 Scytl Standard Software

All intellectual property rights in the Scytl Standard Software are Scytl's sole property. Scytl owns and shall retain all rights, title and interest in and to the Scytl Standard Software. Scytl Standard Software is licensed to Swiss Post under the terms and conditions described in the Framework Agreement.

2.1.2.2 Swiss Post-Scytl Software

All intellectual property rights in the Swiss Post-Scytl Software are the joint property of Scytl and Swiss Post (Joint IP).

2.1.2.3 EV Solution

All intellectual property rights in the EV Solution other than Joint IP will be owned by Scytl or by third parties as applicable.

3 Annex: "Analysis of Ballot Privacy Property for Scytl sVote Protocol using ProVerif"

Report

Analysis of Ballot Privacy Property for Scytl sVote Protocol using ProVerif.

R&S Scytl

January 2, 2019

Acknowledgment

This report is the companion to the ProVerif modelisation of Scytl sVote Protocol led by Véronique Cortier (CNRS/LORIA) and Mathieu Turuani (INRIA/LORIA). For completeness, the source code of the symbolic models can be found in the Appendix.

Contents

Chapter 1

Introduction

Switzerland has a long history of the direct participation of its citizens in decision-making processes. Besides traditional elections where voters choose their representatives in the Federal Assembly, citizens can participate in several other voting events. Citizens can propose popular voting initiatives on their own (after having obtained enough popular support by collecting signatures), and the parties and governments themselves (at the communal, cantonal or federal level) can organize referendums in order to ask the citizens for their opinion on a new law or a modification of the Constitution, among others. Thus, on average Swiss citizens have the chance to participate in 3-4 voting processes a year.

Remote electronic voting was first introduced in Switzerland in three cantons: Geneva, Zurich and Neuchâtel [11]. The first binding trials were held in 2004. By 2019, 10 cantons will have offered the electronic voting channel to their electors. However, until recently the participation rate has been restricted to be up to 10% of the eligible voters. In 2011 the Federal Council of Switzerland started a task force for studying the security issues of electronic voting. As a result, the Federal Council published, in 2013, a report with the requirements for extending the use of the electronic voting systems to a larger part of the electorate. This framework [5], which became binding in January 2014, provides requirements of functionality, security, verifiability and testing/certification which allow the electronic voting systems to be extended to 30%, 50% or up to 100% of the electorate. More specifically, while current electronic voting systems may be allowed to be used for up to 30% of the electorate provided that they fulfil a certain set of functional and security requirements, systems to be used for up to 100% of the electorate are required to additionally provide verifiability features. Although the modality of electronic voting (DRE, remote, ...) is not specified in the report, it refers to electronic voting systems where the vote is cast electronically. In this paper, we will talk specifically of remote electronic voting systems.

Verifiability in remote electronic voting is traditionally divided in three types, which are related to the phase of the voting process which is verified [1]. The first step to audit is the vote preparation at the voting client application run in the voter's device. This application is usually in charge of encrypting the selections made by the voter prior to casting them to a remote server so that their secrecy is ensured. *Cast-as-intended* verification methods provide the voters with means to audit that the vote prepared and encrypted by the voting client application contains what they selected, and that no changes have been performed. *Recorded-as-cast* verification methods provide voters with mechanisms to ensure that, once cast, their votes have been correctly received and stored at the remote voting server. Finally, *counted-as-recorded* verification allows voters, auditors and third party observers to check that the result of the tally corresponds to the votes which were received and stored at the remote voting server during the voting phase.

Classically, cast-as-intended and recorded-as-cast verifiability are known as *individually verifiable* mechanisms, while counted-as-recorded is considered to be a *universally* verifiable method. The explanation is simple: only the voter knows that she had actually cast a vote, and the intended content. On the other hand, anybody should be able to verify the correct outcome of the election given the votes in the ballot box.

However, the trust model and verifiability requirements defined by the Federal Council differ from these well-known properties. Specifically, the Federal Council defines two types of verifiability in the regulation for e-voting:

Individual verifiability is defined according to the following trust model:

- The server side of the voting platform is trusted.
- A part of the voters may not be trustworthy.
- The client side and the communication channel between the client side and the server side is not trusted.

Under this scope, the Federal Council requirement regarding verifiability is that an attacker cannot change the voter intention, prevent a vote from being stored, or cast a vote on its own, without detection from an honest voter that follows the verification protocol.

While this seems similar to the *usual* union of cast-as-intended and recorded-as-cast verifiability done in the literature, it differs from it due to the fact that in this model the server side is trusted, which is not the case when talking in general about recorded as cast mechanisms [7]. We can refer to it as a "weak" recorded as cast verification.

Complete verifiability considers the following trust model:

- The server side of the voting platform is not trusted. Instead, there exists a group of so called control components which interact with it and which is trusted as a whole, under the assumption that at least one of them is reliable (each sole control component is not trusted).
- Same assumptions than for individual verifiability apply for voters, print office, client side, and channel between the client side and the server side.
- Given proofs generated by the system that will be verified by auditors, at least one of the auditors and her technical aids (software or hardware tools) are trusted to behave properly.

Taking into account this trust model, the Federal Council verifiability requirements for this type are many: an attacker cannot change a vote before/after it is stored, or prevent a vote from being stored, delete it from the ballot box, as well as insert new votes, without voters or auditors noticing it. These correspond to the previous requirements for individual verifiability, taking into account that the trusted part of the system is not the server, but the control components which interact with it. Additionally, voters must have to be able to verify whether their voting credentials have been used to cast a vote in the system. Finally, auditors must receive a proof that the result of the election corresponds to the votes cast by eligible voters and accepted by the system during the voting phase. All these requirements have to be fullfilled while vote and intermediate results secrecy is preserved.

In this case, the requirements for complete verifiability cover the classic cast-as-intended, recorded-as-cast and counted-as-recorded concepts, plus additional features (such as that each voter can verify her participation or not in the election). Note that, by the definition provided, the recorded-as-cast verification may not be restricted to be verified by the voter, but also by auditors which inspect the votes registered by the trusted part of the system (the control components).

According to the report by the Federal Council, systems to be used for up to the 50% of electors are required to provide methods for individual verifiability, and systems for up to 100% of the electorate are required to provide complete verifiability, while also enforcing the separation of duties on operations impacting the privacy, integrity and verifiability of the system.

Besides its requirements, the different electorate extents for which the electronic voting system can be used define the level of certification to be passed. Specifically, systems to be used for more than the 50% of the electors have to provide both security and symbolic proofs which demonstrate that the system fulfills the claimed properties.

This report covers the automatic verification using ProVerif of the *ballot privacy* property of the *sVote Protocol Specifications* by Scytl [10]. This verification effort was led by Véronique Cortier (CNRS/LORIA) and Mathieu Turuani (INRIA/LORIA). The analysis has been performed using symbolic cryptography and

can be verified automatically using a well-known state-of-the-art automated verification tool called ProVerif [*] [4].

Limitations. As it is well-known, symbolic proofs of cryptographic protocols deal with abstractions thereof, omitting numerous cryptographic and mathematical properties of the underlying primitives. Symbolic proofs are widely accepted as a good indication that the design of a cryptographic protocol is not flawed, and it is considered to be a good sanitization method for complex cryptographic protocols, such as e-voting protocols. *However, symbolic proofs do not cover actual implementations of the security protocols, and might overlook special attacks that make use of specialized properties of the cryptographic primitives.* Our analysis is not an exception to this rule.

sVote Protocol Specification [10] reports the use of several groups of control components 4 Return Codes Control Components, 4 Mixing Control Component[†] as well as Electoral Board members that keep shares of the last CCM's private key. Our model makes use of only 2 such components in each group. Also, instead of relying on the fact that the last CCM's private key is split among Electoral board members and reconstructed only if cleansing and all first three mixing and partial decryption procedures are valid, we model both CCM as entities that also perform verification and outputs nothing in case it fails. Please refer to the Section 4.1 for the justification as to why this modeling is equal to the case when the last CCM's private key is split among Electoral Board members and can be reconstructed only after successful verification. The simplification is done due to several reasons: to have a simpler ProVerif model and hence improved readability, shortening the running time of ProVerif analysing the corresponding models. This type of simplification is often found in both computational and symbolic models/proofs in the literature. We have no reason to expect that the properties proven do not extend to the case of 4 control components assuming the presence of Electoral Board.

Organization

In Chapter 2, the trust model and security requirements are discussed. In Chapter 3, we define the building blocks used later in Chapter 4.2 to construct our abstraction of Scytl sVote Protocol [10]. Our symbolic model is built from this abstraction. In Chapter 4, we discuss some ProVerif limitations and the difference that might arise between the symbolic model and a concrete implementation of the protocol. In Chapter 5, we define our symbolic privacy property for a bounded number of candidates n and a bounded number of voting choices ψ under the given trust assumptions. In Chapter 6, we discuss our findings.

Finally, in the Appendix we include the source code of our main *extended* ProVerif specification. More precisely, our model is described in the following files: `spec_CCM2-3-4.pv` and `spec_CCM1.pv`

[*]Please note, that Chancellery's requirements [6] does not specify particular tool for automated analysis. Therefore, any widely accepted verifier is permitted. In our analysis we used well-known automatic cryptographic protocol verifier called ProVerif.

[†]Please note, that the last CCM does not know its private key as it is distributed among members of Electoral Board and reconstructed only after successful verification of cleansing and mixing performed by the first three CCMs.

Chapter 2

Threat model and security goals

In this section, we derive precise, formal security goals from the informal description of the model for complete verifiability given by the Swiss Federal Chancellery. Our interpretation of the threat model is supported by quotes from and references to relevant excerpts of the Federal Chancellery's requirements [6]. The extraction of precise properties from the legislation's informally stated goals, is an important step for justifying that the model used throughout the security proofs of sVote is indeed the same up to the differences in notations.

To precisely see this, we review the Federal Chancellery's requirements for complete verifiability in Section 2.1. Then, we introduce the model used in sVote in Section 2.2. Last, in Section 2.3 we provide concrete mappings between the system components and the communication channels in both models.

2.1 Security Assumptions and Threat Model

2.1.1 Assumptions on parties

According to section 4.3 of the Ferederal Chancellery's requirements [6], control components, auditors and auditors' technical aid are referred as ***additional*** system components. Similarly, section 4.3 defines ***additional*** communication channels referring to system components listed in section 4.1. Based on that and also the fact that complete verifiability in practice uses the same provisions as for individual verifiability **we treat the complete abstract model defined in section 4.3 as the extension of the reduced abstract model defined in section 4.1**.

Thus, we assume, that the full list of system components consists of components mentioned in section 4.1 plus additional components from section 4.3. Also, we assume that the full list of possible communication channels consists of those defined in 4.1 plus additional communication channels from section 4.3. Using the same logic, we defined trusted elements as the combination of trusted components from section 4.1 (if section 4.3 do not state otherwise) and trusted assumption of section 4.3.

The full list of the system components of the complete abstract model is defined in Table 2.1. Please note, that the term 'system component' is introduced by the legislation and **consists of the System** itself as well as **Voters, Print office** etc.

2.1.2 Assumptions on communication channels

Federal Chancellery's reduced abstract model (section 4.1) defines as trustworthy the technical aids, the system, and the print office and also all channels except User platform $\leftrightarrow$ system and System $\leftrightarrow$ print office. The complete abstract model (section 4.3) regards the system and **only the system** as untrustworthy, introducing a set of control components trusted as the whole instead. Also section 4.3 states "*Of the **additional** communications channels, only those between the auditors and their technical aids may be deemed trustworthy.*"

System components	Trust assumption	Section
Voters	**significant proportion** of voters are **non-trustworthy**	4.1
User platform	**untrustworthy** for individual and complete verifiability **trustworthy** for privacy	4.1 4.3
Trusted technical aids for voters	**trustworthy**	4.1
System (server-side)	**untrustworthy**	4.3
Print office	**trustworthy**	4.1
Control Components	**trustworthy only as the whole**	4.3
Auditors	at least **one is trustworthy**	4.3
Auditor's technical aid	at least one honest auditor has a **trustworthy** aid	4.3

Table 2.1: Assumptions on parties of the complete abstract model defined in VEleS [6]

The list of the all possible communication channels is presented in Table 2.2.

Communication channel	Trust assumption	Section
Voters ↔ user platform	**trustworthy**	4.1
Voters ↔ trustworthy technical aids	**trustworthy**	4.1
Trustworthy technical aids ↔ user platform	**trustworthy**	4.1
User platform ↔ system	**untrustworthy**	4.1
System ↔ print office	**untrustworthy**	4.1
Print office → voter	**trustworthy** *	4.1
Control component ↔ system	**untrustworthy**	4.3
System ↔ auditor's technical aids	**untrustworthy**	4.3
Auditors' technical aid ↔ auditors	**trustworthy**	4.3
Bidirectional channels for communication between control components	**untrustworthy**	4.3

Table 2.2: Assumptions on communication channels of the complete abstract model defined in VEleS [6]

2.1.3 Additional assumptions on parties regarding voting secrecy

According to [6, Section 4.3], voting secrecy does not account for scenarios where the attacker corrupts the user platform: "*Under the trust assumptions for complete verifiability of the protocol, the attacker is unable to breach voting secrecy or to obtain early provisional results without changing* † *the voters or their user platforms maliciously.*"

Please note that, according to 4.4.8 [6], vote secrecy should be preserved only for trustworthy voters: *"It must be ensured that the voting secrecy of trustworthy voters cannot be breached without maliciously changing their user platform through the server-sided manipulation of the application."*

Additionally, the sections 4.3 and supplementary provision 4.4.8 and 4.4.9 imply that for privacy the user platform of trustworthy voters is considered to be trustworthy. Provision 4.4.8 states the following *"Voters should therefore be able, using a trustworthy platform, to satisfy themselves that the application is sending their vote in encrypted form with the correct key."*. Provision 4.4.9 is *"it must be ensured that the server-sided system cannot find out the content of a vote cast in cooperation with an untrustworthy voter."*

*This channel may only be regarded as trustworthy if the information has been sent by Swiss Post (section 4.2.9 page 23)

†In the French version the word corrupting is used: "compte tenu des hypothèses de confiance qui ont été formulées à propos de la vérifiabilité complète du protocole, l'attaquant ne peut ni violer le secret du vote, ni établir des résultats partiels de manière anticipé sans corrompre les électeurs ou leurs plates-formes utilisateurs respectives."

2.2 Trust model in sVote

2.2.1 Protocol participants in sVote

The sVote Protocol Specifications [10] uses slightly different notations (Protocol Specifications section 2) and defines the participants of the voting protocol as follows:

- *Voter*: they participate in the election by choosing their preferred options.
- *Voting Client*: is the device used by the voter to cast their vote given the voting options selected by the voters.
- *Voting Server*: it receives, processes and stores the votes cast by the voters in the ballot box BB.
- *Control Components* are separated in two groups, one is participating in choice return codes the other in mixing:

  Choice Return Codes Control Components (CCR's): they collaborate with the Print office indirectly (via the Voting Server) in the setup phase, and directly with the Voting Server in the voting phase, to compute the so-called long Choice Return Codes.

  Mixing Control Components (CCM's): they perform the mixing and partial decryption of the ciphertexts in the ballot box.

- *Print Office*: It is responsible for generating, printing and delivering the voting cards to the voters as well as for generating the required election keys [*].
- *Election Administrators*: they are responsible for generating the election configuration, verifying it, computing the results and publishing them. In the Protocol Specification this entity is divided into:
  - *Administration Board* and *Administration Portal*: Both compoenents are used to set-up and sign the configuration and therefore, can perform some cryptographic operations. However for the privacy proof we do not distinguish between those two.
  - *Electoral Board*: This entity owns a key pair whose private key is shared among the Board members and is used to partially decrypt the votes in the last Control Component execution.
- *Global Bulletin board*: is the entity used to store all the information generated during the election to verify the entire process. It stores election configuration, votes, confirmations and keeps track of all the actions performed by each entity. The Bulletin Board is implemented as a distributed system, that includes: election configuration (maintained by *Print office*), Secure Logger (maintained by *CCRs*) and Ballot Box (maintained by *Voting Server*). In this document we refer to Secure Logger as CCR's logs.
- *Auditors*: they are responsible for verifying the integrity of the procedures run in the counting phase. They are a crucial part of ensuring the verifiability properties as set up by the Chancellery's requirements, in so auditors can be leveraged to detect misbehavior.
- *Verifier*: is the component used to verify the correctness of the entire election process, the integrity of the data processed through different voting system components, and that these processes are accurate and fair.

[*] For generating cryptographic material, *Print Office* runs a software called Secure Data Manger (SDM). This software is executed in a controlled, offline environment on the canton's premises. All operations on the SDM are subject to very strict 4-eyes principles and are executed on laptops with special access rights and hardened laptops.

2.2.2 Trust assumptions in sVote

Privacy is proven under assumption required for complete verifiability using the following trust model:

- The *Voting Server* is not trusted. Instead, there exists two groups of so called control components *CCM's* and *CCR's* which interact with *Voting Server* directly and indirectly with *Print office* (via Voting Server). Each group of control components is trusted as a whole, under the assumption that at least one of them is reliable. However, each sole control component is not trusted).
- Credential delivery channel (postal channel between *Print office* and voters) is considered to be trustworthy.
- *Print office* is trusted.
- The *Voting Client* of honest voters is considered to be trusted for privacy, and not trusted for individual and universal verifiability.
- Initial election configuration (number and names of the candidates, number of voters, number of allowed options etc) generated by *Election Administrators* is assumed to be correct as the *Print office* has no means for verifying this information.
- The communication channel between the client side and the server side is not trusted.
- A part of the voters may not be trustworthy.
- *Electoral Board* is treated as set of control component and therefore is trusted as whole, i.e. at least one Electoral Board member is assumed to be trustworthy.
- At least one of the auditors and her technical aids (software or hardware tools) are trusted to behave properly.

2.3 Correspondence between both security models

Now we can align the security model defined by the Chancellery (see Section 2.1) and the security model of sVote defined in Section 2.2. A mapping for the different protocol participants is given in Table 2.3. Similarly, a mapping regarding the communication channels is given in Table 2.4.

sVote's system component	Chancellery's system component	Trust assumption
Voters	Voters	**significant proportion** of voters are **non-trustworthy**
Voting Client	User platform	**untrustworthy** for individual and complete verifiability **trustworthy** for privacy
Voting Card	Trusted technical aids for voters	**trustworthy**
Voting Server	System (server-side)	**untrustworthy**
Print office	Print office	**trustworthy**
CCM CCR	Control Components	**trustworthy only as the whole**
Auditors	Auditors	at least **one is trustworthy**
Verifier	Auditor's technical aid	at least one honest auditor has a **trustworthy** aid

Table 2.3: Correspondence between sVote and the Chancellery assumptions made on the protocol's participants

sVote communication channels	Chancellery's communication channel	Trust assumption
Voters ↔ Voting Client	Voters ↔ user platform	**trustworthy**
Voters ↔ Voting Cards	Voters ↔ Trustworthy technical aids	**trustworthy**
no channel exists	Trustworthy technical aids ↔ User platform	**trustworthy**
Voting Client ↔ Voting Server	User platform ↔ system	**untrustworthy**
Voting Server ↔ Print office	System ↔ print office	**untrustworthy**
Print office → Voter	Print office → voter	**trustworthy** [a]
CCM ↔ Voting Server CCR ↔ Voting Server	Control component ↔ system	**untrustworthy**
Voting Server ↔ Verifier CCM ↔ Verifier CCR ↔ Verifier	System ↔ auditor's technical aids	**untrustworthy**
Verifier ↔ auditor	Auditors' technical aid ↔ auditors	**trustworthy**
no channel exists	Bidirectional channels for communication between control components	**untrustworthy**

Table 2.4: Correspondence between the sVote and the Chancellery assumptions made on the communication channels

[a] This channel may only be regarded as trustworthy if the information has been sent by Swiss Post (section 4.2.9 page 23).

Chapter 3

Abstract Building Blocks

In symbolic models [8], the set of *terms* associated to a set of *function* symbols $\mathcal{F}$, a set of variables $\mathcal{X}$ and a set of names $\mathcal{N}$ is inductively defined as the names, variables, and function symbols applied to other terms. Terms are equipped with *inference rules* of the form $a \vdash b$, that define which messages b can be computed from an a priori given set of messages a. We start by describing our modelling of the basic cryptographic functions used in the protocol (we assume the reader to have some degree of familiarity with Scytl's cryptographic specification of the voting protocol [10]). We use sans serif format for functions and typewriter format for `variables`.

Functions

Function		Type	Meaning	Properties
ske	:	agent_id → priv_ekey	$\mathtt{ske}_{\mathsf{id}}$, the private encryption key associated with an agent id	private
pube	:	priv_ekey → pub_ekey	builds public key $\mathtt{pke}_{\mathsf{id}}$ from $\mathtt{ske}_{\mathsf{id}}$, i.e. $\mathtt{pke}_{\mathsf{id}} := \mathsf{pube}(\mathtt{ske}_{\mathsf{id}})$	public, non-invertible
sq	:	nat → nat	a square function	public,invertible
pubs	:	priv_skey → pub_skey	builds public signing key from the private one	public, non-invertible
bck	:	agent_id → nat	$\mathtt{BCK}^{\mathsf{id}}$, ballot casting key	private
f	:	prf_keys×bitstr → sym_ekey	keyed pseudo-random function	public, non-invertible
pCC	:	priv_ekey×bitstr → bitstr	partial Choice Return Codes function $\mathsf{pCC}(\mathtt{ske}, v) := (v)^{\mathtt{ske}}$	public, non-invertible
Sig	:	priv_skey×bitstr → bitstr	digital signature primitive	public
Enc_s	:	sym_ekey × bitstr → bitstr	symmetric encryption	public, non-invertible
Enc	:	pub_ekey×bitstr×nat → bitstr	randomized asymmetric encryption	public, non-invertible
ϕ	:	nat_set → bitstr	aggregation function ϕ	public invertible
tild	:	priv_ekey×bitstr → nat	$\mathsf{tild}(\mathtt{E1}, \mathtt{k}_{\mathsf{id}})$ is $\mathtt{E1}^{\mathtt{k}_{\mathsf{id}}}$	public, non-invertible
KSkey	:	password → sym_key	Models $\mathsf{PBKDF2}(\mathtt{SVK}_{\mathsf{id}}, \mathsf{KEYseed})$ with a KEYseed fixed.	public, non-invertible
δ	:	priv_ekey×bitstr →priv_ekey	the key derivation function	public invertible
H	:	bitstr→ nat	The hash function	public, non-invertible
mergepk	:	pub_ekey → pub_ekey	merge public keys	public,invertible

Functions

Function		Type	Meaning	Properties
deltaX	:	nat → sym_ekey	merge public keys	public,invertible
merge_kc	:	bitstr × bitstr → bitstr	models merging of replies from CCRs for computing lVCC	public, invertible
merge_k	:	bitstr × bitstr → bitstr	models merging replies from CCRs for computing lCC	public, invertible
pCC	:	priv_ekey × nat → nat	function for computing partial Choice Return Codes	public
ZKP	:	pub_ekey×pub_ekey×bitstr× bitstr × bitstr×nat×priv_ekey→ bitstr	function for modeling non-interactive zero-knowledge proofs of knowledge	public
VC	:	agent_id → bitstr	get the Verification Card for an agent id	public
SVK	:	agent_id → password	get the Start Voting Key for an agent id	private
BCK	:	agent_id → nat	get the Ballot Casting Key for an agent id	private
CC	:	agent_id → bitstr	get the short Choice Return Codes for an agent id	private
VCC	:	agent_id → bitstr	get the short Vote Cast Return Code for an agent id	private
ZKPexp	:	pub_ekey×bitstr× nat ×nat×priv_ekey→ bitstr	function for modeling non-interactive zero-knowledge proof of the correct exponentiation	public

Notation

Object		Type	Meaning	Properties
$\mathtt{pk}_{\mathtt{EL}}^{(j)}$	:	priv_ekey	CCM$_j$'s public election key share $\left(= \mathsf{pube}(\mathtt{sk}_{\mathtt{EL}}^{(j)})\right)$	public
$\mathtt{pk}_{\mathtt{EL}}$	:	pub_ekey	Election public key $\left(= \mathsf{mergepk}(\mathtt{pk}_{\mathtt{EL}}^{(j)})\right)$	public
$\mathtt{VCCs}_{\mathtt{sk}}$	:	priv_skey	Vote Cast Return Code signing key	private
$\mathtt{VCCs}_{\mathtt{pk}}$	:	pub_skey	Vote Cast Return Code verification key $(= \mathsf{pubs}(\mathtt{VCCs}_{\mathtt{sk}}))$	public
$\mathtt{SVK}_{\mathsf{id}}$	:	password	Start Voting Key $\left(= \mathsf{SVK}(\mathsf{id})\right)$	private
$\mathtt{VC}_{\mathtt{id}}$	:	agent_id	Verification Card ID associated to id $\left(= \mathsf{VC}(\mathsf{id})\right)$	public
$\mathtt{k}_{\mathsf{id}}$	:	priv_ekey	Verification Card $\mathtt{VC}_{\mathtt{id}}$'s private key $\left(= \mathtt{ske}(\mathsf{id})\right)$	private
$\mathtt{K}_{\mathsf{id}}$	:	pub_ekey	Verification Card $\mathtt{VC}_{\mathtt{id}}$'s public key $\left(= \mathtt{pke}(\mathsf{id})\right)$	public
$\mathtt{BCK}^{\mathsf{id}}$	:	nat	Ballot Casting Key associated to id $\left(= \mathsf{BCK}(\mathsf{id})\right)$	private
$\mathtt{VCks}_{\mathsf{id}}$	:	bitstr	Key Store associated to id	private
$\mathtt{lVCC}^{\mathsf{id}}$	:	sym_ekey	Long Vote Cast Return Code for id	private
$\mathtt{pVCC}^{\mathsf{id}}$	:	nat	Pre-Vote Cast Return Code associated to id	private
$\mathtt{VCC}^{\mathsf{id}}$	:	bitstr	Vote Cast Return Code for id $\left(= \mathsf{VCC}(\mathsf{id})\right)$	private
$J_1, \ldots, J_n$	:	nat	voting options available in the election	public
$J_1, \ldots, J_\psi$	:	nat	voter individual ψ choices in the election	public
v	:	nat → bitstr	$v_i := \mathsf{v}(J_i)$, the encoding of voting option (candidate) J_i	public, invertible
$\mathtt{CC}_i^{\mathsf{id}}$	:	bitstr	i-th (short) Choice Return Code associated to id $\left(= \mathsf{CC}(\mathsf{id}, J_i)\right)$	private
$\mathtt{lCC}_i^{\mathsf{id}}$	:	bitstr	i-th long Choice Return Code associated to id	private
$\mathtt{pCC}_i^{\mathsf{id}}$	:	nat	i-th Partial Choice Return Code associated to id	private
$\mathtt{pC}_i^{\mathsf{id}}$	:	nat	i-th pre-Choice Return Code associated to id	private
$\mathtt{k}_{\mathsf{id}}^{\mathrm{CCR}}$	:	pub_ekey	CCR's voter choice return code generation private key	public
$\mathtt{K}_{\mathsf{id}}^{\mathrm{CCR}}$	:	pub_ekey	CCR's voter choice return code generation public key $\left(= \mathsf{pube}(\mathtt{k}_{\mathsf{id}}^{\mathrm{CCR}})\right)$	public

These objects are specific to the sVote protocol. Our notation tries to be as close as possible to the notation in [10]. However, please note, that it deviates on several occasions: (1) Election public key defined as $\mathsf{EL}_{\mathtt{pk}}$ in [10] is referred as $\mathtt{pk}_{\mathtt{EL}}$ throughout this report; (2) Election public and private key shares of CCM_j defined as $(\mathsf{EL}_{\mathtt{pk}}^{(j)}, \mathsf{EL}_{\mathtt{sk}}^{(j)})$ in [10] are referred as $\mathtt{pk}_{\mathtt{EL}}^{(j)}, \mathtt{sk}_{\mathtt{EL}}^{(j)}$ throughout this report; (3) CCR's voter choice return code generation public and private key pair defined as $(\mathtt{K}_{\mathsf{id}}^{j}, \mathtt{k}_{\mathsf{id}}^{j})$ in [10] is referred as $(\mathtt{K}_{\mathsf{id}}^{\mathrm{CCR}}, \mathtt{k}_{\mathsf{id}}^{\mathrm{CCR}})$ throughout this report to avoid confusions with Verification Card $\mathtt{VC}_{\mathtt{id}}$'s public private key pair $(\mathtt{K}_{\mathsf{id}}, \mathtt{k}_{\mathsf{id}})$ in the code.

Attacker capabilities (Dolev-Yao)

The attacker capabilities consist on:

- Computing and inverting public invertible functions, e.g.
  - $\mathtt{pk}_{\mathsf{id}} \vdash \mathsf{id}$ and $\mathsf{id} \vdash \mathtt{pk}_{\mathsf{id}}$ $\forall \mathsf{id}$ (i.e. the link between any public key and the corresponding agent's pseudo identity id is publicly known)
- Computing public non-invertible functions, e.g.
  - $\mathtt{m} \vdash \mathsf{H}(\mathtt{m})$ $\forall \mathtt{m}$ (i.e. H is a efficiently computable hash function that cannot be inverted; in particular the rule $\mathsf{H}(\mathtt{m}) \vdash \mathtt{m}$ $\forall \mathtt{m}$ is *not available* to the attacker)
  - $\mathtt{sk}, \mathtt{m} \vdash f(\mathtt{sk}, \mathtt{m}) \quad \forall \mathtt{sk}, \mathtt{m}$ (i.e. f is an efficiently computable pseudorandom function on knowledge the secret $\mathtt{sk}$; for the sake of clarity, let us stress that the rule $\mathtt{m} \vdash f(\mathtt{sk}, \mathtt{m})$ is *not available* to the attacker)
  - $\mathtt{sks}, \mathtt{m} \vdash \mathsf{Sig}(\mathtt{sks}, \mathtt{m}) \quad \forall \mathtt{sks}, \mathtt{m}$
  - $\mathsf{Enc}(\mathtt{pke}, \mathtt{m}, \mathtt{r}) \quad \forall \mathtt{pke}, \mathtt{m}, \mathtt{r}$
  - $\mathsf{Enc}_s(\mathsf{sk}, \mathtt{m}) \quad \forall \mathsf{sk}, \mathtt{m}$
  - $\mathtt{pke} \vdash \mathsf{mergepk}(\mathtt{pke}) \quad \forall \mathtt{pke}$
  - $v_1, \ldots, v_\psi, \mathtt{r} \vdash \mathsf{Enc}\big((v_1, \ldots, v_\psi), \mathtt{r}\big) \quad \forall v_1, \ldots, v_\psi, \mathtt{r}$
- Computing the following functions associated to symmetric encryption:
  - $\mathsf{Dec}_s\big(\mathsf{sk}, \mathsf{Enc}_s(\mathsf{sk}, \mathtt{m})\big) \vdash \mathtt{m} \quad \forall \mathsf{sk}, \mathtt{m}$ (i.e. decrypting a symmetric encryption of a message returns the original message)
- Computing the following functions associated to public key encryption:
  - $\mathsf{Dec}\big(\mathtt{ske}, \mathsf{Enc}(\mathsf{pubs}(\mathtt{ske}), \mathtt{m}, \mathtt{r})\big) \vdash \mathtt{m} \quad \forall \mathtt{ske}, \mathtt{m}, \mathtt{r}$ (i.e. decrypting an asymmetric encryption of a set ot natural numbers returns the original set)
- Computing the following functions associated to digital signatures:
  - $\mathsf{Verif}\big(\mathsf{pube}(\mathtt{sks}), \mathtt{m}, \mathsf{Sig}(\mathtt{sks}, \mathtt{m})\big) \vdash \mathsf{true} \quad \forall \mathtt{sks}, \mathtt{m}$ (i.e. any well-formed signature is accepted)
- Use Homomorphic property of the double exponentiation for any k_1, k_2, v:
  - $\mathsf{Homo}\Big(\mathtt{pCC}\big(k_1, \mathtt{pCC}(k_2, v)\big)\Big) = \mathtt{pCC}\big(k_2, \mathtt{pCC}(k_1, v)\big)$
- The following are specialised functions and rules with which we have abstracted away the zero-knowledge proofs from the protocol. Let $\mathsf{ZKP}, \mathsf{VerifP}$ be functions with the following types:
  - ZKP : pub_ekey×pub_ekey×bitstr× bitstr× bitstr×nat×priv_ekey→ bitstr; models a non-interactive zero-knowledge proof
  - VerifP : pub_ekey × pub_ekey × bitstr × bitstr × bitstr → bool; models the verification equation of a zero-knowledge proof

◇ $\mathsf{VerifP}\Big(\mathtt{pk}_{\mathtt{EL}}, \mathtt{K}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{id}}, \mathsf{Enc}\big(\mathtt{pk}_{\mathtt{EL}}, (v_1, \ldots, v_\psi), \mathtt{r}\big), \big((v_1)^{\mathtt{k}_{\mathtt{id}}}, \ldots, (v_\psi)^{\mathtt{k}_{\mathtt{id}}}\big)$
$\mathsf{ZKP}\Big(\mathtt{pk}_{\mathtt{EL}}, \mathtt{K}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{id}}, \mathsf{Enc}\big(\mathtt{pk}_{\mathtt{EL}}, (v_1, \ldots, v_\psi), \mathtt{r}\big)\ \big((v_1)^{\mathtt{k}_{\mathtt{id}}}, \ldots, (v_\psi)^{\mathtt{k}_{\mathtt{id}}}\big)\Big),$
$\mathtt{r}, \mathtt{k}_{\mathtt{id}}\Big) = \mathsf{true}$

The equation above is aimed at capturing the three non-interactive zero-knowledge proofs (Schnorr proof, Exponentiation Proof, Plaintext Equality Proof) computed in the algorithm CreateVote as defined in [10]. Roughly speaking, the Schnorr proof proves knowledge of nonce $\mathtt{r}$, while the Exponentiation and Plaintext Equality proofs prove that the ciphertext $\mathsf{Enc}\big(\mathtt{pk}_{\mathtt{EL}}, (v_1, \ldots, v_\psi), \mathtt{r}\big)$, that encrypts the voting options; the partial Choice Return Codes $\big((v_1)^{\mathtt{k}_{\mathtt{id}}}, \ldots, (v_\psi)^{\mathtt{k}_{\mathtt{id}}}\big)$ and the voters' verification card private key $\mathtt{k}_{\mathtt{id}}$ and verification card $\mathtt{VC}_{\mathtt{id}}$ are all linked.

◇ ZKPexp : pub_ekey×bitstr× nat× nat×priv_ekey→ bitstr;
models a non-interactive zero-knowledge proof of correct exponentiation

◇ VerifExp : pub_ekey × bitstr × nat × nat × bitstr → bool;
models the verification equation of a zero-knowledge proof

◇ $\mathsf{VerifExp}\Big(\mathtt{K}_{\mathtt{id}}^{\mathrm{CCR}}, \mathtt{VC}_{\mathtt{id}}, v_i, v_i^{\mathtt{k}_{\mathtt{id}}^{\mathrm{CCR}}}, \mathsf{ZKPexp}\big(\mathtt{K}_{\mathtt{id}}^{\mathrm{CCR}}, \mathtt{VC}_{\mathtt{id}}, v_i, v_i^{\mathtt{k}_{\mathtt{id}}^{\mathrm{CCR}}}, \mathtt{k}_{\mathtt{id}}^{\mathrm{CCR}}\big)\Big) = \mathsf{true}$

The equation above is aimed at capturing the Exponentiation Proof computed by CCRs during the Setup as defined in [10]. Roughly speaking, it proves that the ciphertext $v_i^{\mathtt{k}_{\mathtt{id}}^{\mathrm{CCR}}}$ is indeed v_i raised into the CCR's Voter choice return code generation private key that corresponds to the Voter choice return code generation public key $\mathtt{K}_{\mathtt{id}}^{\mathrm{CCR}}$

◇ PDec2 : priv_ekey× bitstr→bitstr
Partial decryption plus re-encryption

◇ PMix2 : priv_ekey×bitst×bitstr→bitstr.
Proof of correct Mixing

◇ MixVerify2 : bitstr×bitstr×bitstr→ bool returns true if one of the following statements is true:

- $\mathsf{MixVerify2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}), (\mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E3})), \mathsf{PMix2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}),$
  $(\mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E3})))) = \mathsf{true}$
- $\mathsf{MixVerify2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}), (\mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E2})), \mathsf{PMix2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}),$
  $(\mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E2})))) = \mathsf{true}$
- $\mathsf{MixVerify2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}), (\mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E3})), \mathsf{PMix2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}),$
  $(\mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E3})))) = \mathsf{true}$
- $\mathsf{MixVerify2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}), (\mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E1})), \mathsf{PMix2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}),$
  $(\mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E1})))) = \mathsf{true}$
- $\mathsf{MixVerify2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}), (\mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E2}, \mathsf{PDec2}(\mathtt{E2})), \mathsf{PMix2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}),$
  $(\mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E1}), \mathsf{PDec2}(\mathtt{E2})))) = \mathsf{true}$
- $\mathsf{MixVerify2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}), (\mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E1})), \mathsf{PMix2}((\mathtt{E1}, \mathtt{E2}, \mathtt{E3}),$
  $(\mathsf{PDec2}(\mathtt{E3}), \mathsf{PDec2}(\mathtt{E2}), \mathsf{PDec2}(\mathtt{E1})))) = \mathsf{true}$

The equation above is aimed at capturing merged together non-interactive zero-knowledge proofs of shuffling and partial decryption performed by dishonest CCMs prior to the honest one. The private keys of these CCMs are public and therefore abstracted.

Chapter 4

Abstract Model of sVote with Control Components Protocol

4.1 Abstractions and relation to the base protocol

In this section we point out the abstractions and simplifications that have been done in the protocol description presented in Section 4 in relation to the implementation of the system used in the Swiss Online Voting System.

4.1.1 System setup

We assume that global configuration, independent of specific election events, is set in advance and it is ready to use.

CA HIERARCHY. Constitution of a platform root CA, and generation of credentials for the different system contexts and tenants that wish to run an election is omitted. For more details, see [10, Sections 3.1, 3.2, 3.3]

NUMBER OF CONTROL COMPONENTS. For simplicity we consider only two CCRs and two CCMs for Verifiability model, while still keeping four CCMs for the privacy one. Also, we do not model the fact that the last key being split among members of Electoral Board as specified in [10].

Without loss of generality we can use only two CCRs in our model because, the role of the CCRs is symmetric. Indeed, in the security analysis the only assumption we made on the number of the control components is that at least one member of each group is trusted.

As for CCMs, even though those components are executed in a sequence, we also claim that our reduction does not affect proof structure due to the mandatory audit performed before the last CCM is executed and the fact that the last key is distributed among members of Electoral Board.

Consider a case of N CCMs where only one of them is honest. This consideration can be done without loss of generality as any other scenario can be mapped to this extreme case. According to [10], the mandatory verification would be performed after $N-1$ CCMs shuffled and partially decrypted votes. Verification would fail if at least one of the following is true: a) cleansing procedure is not correct b) one of the mixing proofs is invalid or c) one of the decryption proofs is invalid.

If the verification holds, Electoral Board members would submit their private shares so the last CCM would reconstruct its key. Taking into account that Electoral Board can be viewed as a set of 'human control

components'[*], at least one of the members is honest and refuses to submit the decryption key share if validation fails. Thus, the last CCM would be able to reconstruct the last CCM's decryption key and perform the decryption process if and only if verification holds.

Table 4.1 shows all possible corruption scenarios in case of N CCMs and Electoral Board. Please bear in mind, that the table was constructed assuming that there is only one honest CCM in the whole chain and Honest Electoral Board members submit their private keyshares if and only if verification holds.

	Description	Honest CCM is among first N-1	Verification holds	Honest EB members submit their keyshares	Last CCM knows its key
Case 1	Last CCM is corrupted, but can't reconstruct its key.	Yes	No	No	No
Case 2	Last CCM is corrupted and can reconstruct its key.		Yes	Yes	Yes
Case 3	Last CCM is honest, but can't reconstruct its key.	No	No	No	No
Case 4	Last CCM is honest and knows its key.		Yes	Yes	Yes

Table 4.1: Possible corruption scenarios in case of N CCMs and Electoral Board

Due to simplicity reasons, in proofs we abstract from Electoral Board members and assume that the second CCM's is no different from the first one. Please also note, that in the cryptographic games, the challenger executes both a CCM and mandatory verification in a single function and outputs the result only if verification holds. This is done without loss of generality because, all possible outcomes in such case are consistent with outcomes in case of N CCMs and Electoral Board i.e.:

1. The Challenger runs the first CCM
   This case covers **Case 2** as in this scenario the last CCM is corrupted and can reconstruct its key.
   Also **Case 1** can be viewed as a strong version of **Case 2**, where Attacker controls the last node but is not able to reconstruct the key due to at least one missing share.

2. The Challenger runs the second CCM
   This case covers both **Case 3** and **Case 4** as in those scenarios only the last CCM is honest.

As for the privacy model we can combine all corrupt CCMs into a single entity and use the argumentation given above.

4.1.2 Election setup and voting phase

Modeling setup. Recall that in the real implementation all cryptographic operations performed by the control components are enhanced with zero-knowledge proofs that are checked before the voting phase starts[†]. Therefore, malicious CCRs are in practice forced to behave as honest-but-curious entities. In our model, the Print Office first defines the mapping table using a constructor/destructor approach, and so it is defined for an unbounded number of (honest and dishonest) voters, and secondly, for the honest voters for which the ballot privacy property is tested it allows an attacker to contribute in the generation of the partial choice return codes, as in the real implementation. On the other hand, since the vote cast code is independent of the voting options the way they are generated does not affect privacy, and hence we opted for doing it statically in the Print Office. The reason for having some parts done statically is efficiency, but we emphasize again that the *pieces of information* depending on the voting options (for honest voters) are generated *with the collaboration of the CCRs* as in the real protocol.

[*] According to section 4.4.10 of [6], it is also permitted to implement a group of control components so that they take the form of people.

[†] The check must be done by a trusted entity. In the protocol's specifications [10] it is intentionally left open who is in charge of performing the check: either the auditor's technical aid or delegated to the Print Office. Here, for simplicity, the check is done in the *trusted* environment of the Print Office.

Communication between Print Office and CCRs. We have strenghtened the model by allowing *full* corruption in the CCRs during setup and voting phases. This is reflected, in the fact that communication between the Print Office and the CCRs is not encrypted (thus, the corresponding encryption keys can be omitted), nor are the pre-choice return codes generated in the voting device. Also, the attacker gets all secret keys corresponding to the CCRs (e.g. the verification card secret keys) However during the tally phase, an honest CCM expects all votes from honest voters to be present in the cleansed ballot box. The latter is consistent with the real implementation due to auditors checking consistency of CCRs' transcripts and the server transcript.

Offline mixing control component. We have abstracted away the generation and recovery of the election key private key share $\mathtt{sk}_{\mathtt{EL}}^{(2)}$ corresponding to the last CCM. In the actual implementation, during setup this private key is secret-shared using Shamir secret sharing scheme among the electoral board members, and during tally, it is reconstructed accordingly. However, this details are irrelevant in the security analysis: it only matters that the adversary gets to know the key $\mathtt{sk}_{\mathtt{EL}}^{(2)}$ in case the last CCM is compromised. See [10, Sections 4.5, 6.1.2, Annex 9.1.9, 9.1.10] for more details.

Generation of system parameters. Parameters not relevant to the security analysis are assumed to be generated ahead of time. These include election rules, ballot generation, initialization of the ballot box, and necessary public parameters needed to encrypt the voting options.

Voting options. We denote the set of valid votes by Ω, which is composed by any combination of voting options $\{v_1, \ldots, v_\psi\}$ which is valid according to the election rules. Both the set of valid votes and the counting function $\rho : (\Omega \cup \{\perp\})^* \to R$ are assumed to be defined in advance. As already mentioned, the set of possible results R is given by the multiset function ρ, which provides the cleartext votes cast by the voters in a random order [3]. Additionally, if write-in values are permitted in the election, cast-as-intended for the write-in content cannot be provided, since it is not possible to provide the voter with a mapping of all possible write-in values to a choice return code. Therefore, we will not cover write-in values in our security and formal analysis.

Authentication. We have not covered details on how eligible voters authenticate in the system. This might happen via a dedicated protocol, or via a third party. In any case, the focus in this document is to show that privacy is maintained regardless of authentication procedure, so how the user authenticates is considered out of the scope. This is consistent with the current state-of-the-art in computational security proofs for e-voting systems, where eligibility verifiability [12] is very rarely analyzed.

In the actual implementation, users authenticate in the system using a challenge-response mechanism. The goal of the authentication procedure is to ensure that only a voter who proved that he opened an encrypted key store gets a valid authentication token. Namely, a voter sends a request that includes his Credentials ID to the server. The server sends back the corresponding encrypted Verification Card keystore and a challenge. Voter replies by sending a client message, which includes the server's challenge signed with the key retrieved from the encrypted keystore. If the reply to the challenge is valid, Server generates an Authentication Token containing the Voter Information, a timestamp etc. and sends it to the voter. During the voting phase, the voter going to include this token to every request it sends to the server. See [10, Sections 5.1, 5.2] for more details.

Details about authentication layer have been deliberately omitted in the proofs for the sake of clarity, and given the fact that they are not relevant for proving cast-as-intended verifiability, universal verifiability or privacy. We emphasize that our model is independent of the way encrypted Credential Data Keystore is delivered to the voters. Our only requirements are: Verification Card Keystore is generated and encrypted the way it is described in our model and Voting Cards are delivered to the voters via a trusted channel (i.e. post office).

Please notice, that a Verification Card Keystore can only be opened by the person in possession of the voting card: the keystore is encrypted with the key that is derived from the Start Voting Key ($\texttt{SVK}_{\texttt{id}}$) printed in the voting card.

In the protocol model we assume, that all encrypted Verification Card Keystore are public. Moreover, the Attacker can open a keystore if he controls the Voting Client or corrupts a voter and access his voting card. In case, when a voter is honest and Voting Client is not leaking any information to the Attacker, it is assumed that voter's Start Voting Key ($\texttt{SVK}_{\texttt{id}}$) that is printed in the voting card is private.

State machine of control components and bulletin board. Security properties in symbolic models need to hold for all possible protocol executions and attacker behaviors. In particular, an attacker does not have to follow the normal thread of execution and can try to replay messages, skip steps within the protocol or try to use components as oracles to learn critical values (e.g. choice return codes).

Our symbolic model explicitly allows for such attacker behavior. However, there are certain limitations of the attacker capabilities with regard to the state machine of control components and bulletin board.

- The attacker cannot prevent a control component of recording every action related to the choice return codes calculation to the secure log (CCR log). Therefore, an attacker is unable to bring a control component (for instance by sending concurrent requests) to record a different pre choice return code in the CCR log than he returned to the voting server controlled by the attacker. This limitation of the attacker's capabilities is justified by the supplementary provision 4.4.11 of the technical annex of the VEleS :

  > **Supplementary provision 4.4.11, Annex of VEleS :** *[...]The control components are often called "trustees" in English in the abstraction. In the abstraction, trustees are described as entities that can initiate complex calculations and keep private elements secret. The calculations may include the provably correct mixing and re-encryption of votes (re. their anonymisation; each trustee corresponds to a mixing node of a re-encryption network),* ***the running of a trustworthy electronic public board*** *or the creation of the PKI and, with the aid of their parts of the distributed private key, the provably correct decryption of votes[...]*

  The secure logs are implemented as an immutable chain of logs, that are signed by a private key of the control component. The secure logs are stored on the same machine that runs the control component. Therefore, the calculation of the control components and the secure logs are not separated by a network that could potentially be controlled by an attacker.

- The attacker cannot interfere with the serialized execution of computations within an honest control component. For instance, it is not within the attacker's reach to bring an honest control component to calculate an exponentiation without calculating the corresponding encryption proof. For the same reason, we safely assume that the honest control component sends and logs the complete set of information of a computation. For example, we assume that the honest control component never sends and logs an exponentiation without sending and logging the corresponding exponentiation proof and signature.

In the protocol specifications, the control components perform an additional check to make sure that only one set of choice return codes is computed per verification card id. Therefore, the control components keep state about which voting cards were already processed. However, due to the difficulty to capture state in symbolic models and in ProVerif in particular (there exists just very recent extensions of ProVerif to capture stateful properties of cryptographic properties [**?**]), we omitted this check in the symbolic model. Since the control components record any choice return code generation in the CCR log, the auditor will anyway detect if a control component processed two different votes for the same verification card id.

4.2 sVote Model in Proverif

Our abstract model for sVote with Control Components Protocol is given next. This model is the basis of our ProVerif analysis. Our abstract model cannot in any case replace, for verification purposes, the careful examination of the Proverif code.* Sometimes the code slightly differs from the abstract model, mainly for reasons of improving the performance of ProVerif when running the corresponding verification tests over the given piece of code (in those cases, the modified code is functionally equivalent to the corresponding algorithms of the abstract models), or because certain operations are not available in ProVerif. The abstract model source code has been included in Appendix A.

Data Initialisation

The following initialisation data is computed by the Print office and it is stored off-line until is securely transmitted to the corresponding agents:

- $\mathsf{InitData}_{\mathsf{id}}$ is the initial data corresponding to voter id
- $\mathsf{init}_{\mathsf{DEV}}$ is the initial data corresponding to any Voting Device
- $\mathsf{init}_{\mathsf{S}}$ is the initial data corresponding to the Voting Server

Processes

We make use of the following processes, that are built from the functions that were described in Chapter 3. Let $J_1, \ldots, J_n$ be the universe of voting choices (candidates) available in the election.

- $\mathsf{SetupElKey}(\mathtt{pk}_{\mathtt{EL}}^{(j)})$ is run by the Print Office on inputs the Election Public Key shares $\mathtt{pk}_{\mathtt{EL}}^{(j)}$ created by an honest CCM_j and computes the Election Public Key as $\mathtt{pk}_{\mathtt{EL}} := \mathsf{mergepk}(\mathtt{pk}_{\mathtt{EL}}^{(j)})$
- $\mathsf{AliceData}(\mathsf{id})$ is the function that models the delivery of a Voting Card $\mathsf{InitData}_{\mathsf{id}}$ to the corresponding voter id. In Proverif this information is assumed to be private.
- $\mathsf{GetAliceData}(\mathsf{InitData}_{\mathsf{id}}, \{J_i\}_{i=1}^{\psi})$ allows the voter id to retrieve the codes

$$(\mathtt{SVK}_{\mathsf{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{BCK}^{\mathsf{id}}, \mathtt{VCC}^{\mathsf{id}}, \{\mathtt{CC}_i^{\mathsf{id}}\}_{i=1}^{\psi})$$

  he needs in order to vote w.r.t. his voting choices $\{J_i\}_{i=1}^{\psi}$:

  1. $\mathtt{VC}_{\mathtt{id}}$ is a random Verification Card ID associated to the voter id, $\mathtt{VC}_{\mathtt{id}} = \mathtt{VC}(\mathsf{id})$
  2. $\mathtt{SVK}_{\mathsf{id}}$ is a random start voting key associated to the voter id, $\mathtt{SVK}_{\mathsf{id}} = \mathtt{SVK}(\mathsf{id})$
  3. $\mathtt{BCK}^{\mathsf{id}}$ is a random Ballot Casting Key associated to voter id, $\mathtt{BCK}^{\mathsf{id}} = \mathtt{BCK}(\mathsf{id})$
  4. let $\mathtt{CC}_i^{\mathsf{id}}$ be the short Choice Return Codes assigned at random to voter's id choice J_i $\forall i = 1, \ldots, n$, $\mathtt{CC}_i^{\mathsf{id}} = \mathtt{CC}(\mathsf{id}, J_i)$
  5. let $\mathtt{VCC}^{\mathsf{id}}$ be the random short Vote Cast Return Code assigned to voter's id , $\mathtt{VCC}^{\mathsf{id}} = \mathtt{VCC}(\mathsf{id})$
  6. voter's data is $(\mathtt{SVK}_{\mathsf{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{BCK}^{\mathsf{id}}, \mathtt{VCC}^{\mathsf{id}}, \{\mathtt{CC}_i^{\mathsf{id}}\}_{i=1}^{\psi})$
- $\mathsf{GetKey}(\mathtt{SVK}_{\mathsf{id}}, \mathtt{VCks}_{\mathsf{id}})$ recovers the Verification Card private key $\mathtt{k}_{\mathsf{id}}$ as follows:

  1. output $\mathtt{k}_{\mathsf{id}} := \mathsf{Dec}_s(\mathtt{VCks}_{\mathsf{id}}, \mathtt{KSkey}(\mathtt{SVK}_{\mathsf{id}}))$
- $\mathsf{CreateVote}(\mathtt{pk}_{\mathtt{EL}}, \mathtt{VC}_{\mathtt{id}}, \{J_i\}_{i=1}^{\psi}, \mathtt{K}_{\mathsf{id}}, \mathtt{k}_{\mathsf{id}})$ consists of the following steps:

  1. let $V = \phi(v(J_1), \ldots, v(J_\psi))$

*In case there is any discrepancy between the abstract model in this chapter and the definitions in the Proverif files, the latter prevail.

2. let $\mathtt{E1} = \mathsf{Enc}(\mathtt{pk}_{\mathtt{EL}}, V, \mathtt{r})$ for a fresh nonce $\mathtt{r}$
3. let $\mathtt{E2} = \mathtt{pCC}(\mathtt{k}_{\mathsf{id}}, v(J_i), \ldots, v(J_\psi))$
4. let $\mathtt{P} = \mathsf{ZKP}(\mathtt{pk}_{\mathtt{EL}}, \mathtt{K}_{\mathsf{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{E1}, \mathtt{E2}, \mathtt{r}, \mathtt{k}_{\mathsf{id}})$
5. output $\mathtt{b} = (\mathtt{E1}, \mathtt{E2}, \mathsf{tild}(\mathtt{k}_{\mathsf{id}}, \mathtt{E1}), \mathtt{K}_{\mathsf{id}}, \mathtt{P})$ a ballot

This is the algorithm that the voter's device runs to create a ballot containing the voter's intended voting options $J_1, \ldots, J_\psi$

- $\mathsf{ProcessVote}(\mathtt{pk}_{\mathtt{EL}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{b})$ outputs a boolean and consists of the following steps:
  1. let $\mathtt{b} = (\mathtt{E1}, \mathtt{E2}, \mathtt{EC}, \mathtt{pk}_{\mathtt{EL}}, \mathtt{P})$
  2. output $\mathsf{VerifP}(\mathtt{pk}_{\mathtt{EL}}, \mathtt{K}_{\mathsf{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{E1}, \mathtt{E2}, \mathtt{P})$

  This is the algorithm used for testing a ballot.

sVote Voting Protocol - Abstraction

In the following we describe our abstract model for sVote and the reader is referred to [10] for the specification of sVote.

We note that the threat model with respect to cryptographic key management used in our model follows an *all-or-nothing* approach: we assume that if an attacker compromises a service, it gets hold of all the cryptographic key material of the corresponding service (even if it the corresponding service may have a trusted hardware module where cryptographic keys could still be safeguarded). On the contrary, we assume that if a component is trustworthy, none of the cryptographic keys that are unique to that component are compromised. In particular, a trustworthy component cannot be impersonated by an adversary (whereas any compromised component is modelled as fully controlled by the adversary).

We think this modelling is appropriate for analysing sVote verifiability properties. Indeed, the cast-as-recorded verifiability of sVote does not rely on the security of a complex key management scheme, but rather on the integrity of the ballot box as computed by the voting server and the soundness of several zero-knowledge proof systems, circumstances that are well-aligned with the aforementioned all-or-nothing approach to cryptographic key compromise.

1. The first part of Setup phase is static and information generated as follows:
   (a) Honest CCM_j:
      i. Generates its private election key share $\mathtt{sk}_{\mathtt{EL}}^{(j)}$, the corresponding public key is $\mathtt{pk}_{\mathtt{EL}}^{(j)} = \mathsf{pube}(\mathtt{sk}_{\mathtt{EL}}^{(j)})$. Keys of dishonest CCMs are fixed and public, so they are abstracted away.
   (b) Print Office:
      i. Merges all the public keys to construct an election public key: $\mathtt{pk}_{\mathtt{EL}} = \mathsf{mergepk}(\mathtt{pk}_{\mathtt{EL}}^{(j)})$.
      ii. $(\mathtt{pk}_{\mathtt{SDM}}, \mathtt{sk}_{\mathtt{SDM}})$ are abstracted (see paragraph on communication between Print Office and CCRs in Section 4.1.2).
      iii. Codes Secret Key $\mathtt{C}_{\mathtt{sk}}$ is fixed and public and hence abstracted away.
      iv. Generates a random Vote Cast Return Code Signer private key $\mathtt{VCCs}_{\mathtt{sk}}$ and computes the corresponding public key as $\mathtt{VCCs}_{\mathtt{pk}} = \mathsf{pubs}(\mathtt{VCCs}_{\mathtt{sk}})$.
      For every voter:
         A. Generates a random the Ballot Casting Key $\mathtt{BCK}^{\mathsf{id}}$.
         B. Generates a random Start Voting Key $\mathtt{SVK}_{\mathsf{id}}$.
         C. Generates the short Choice Return Codes for all options $i = 1, \ldots, n$: $\mathtt{CC}_i^{\mathsf{id}}$.
         D. Generates the short Vote Cast Return Code $\mathtt{VCC}^{\mathsf{id}}$ and the signature $\mathtt{sVCC}^{\mathsf{id}} = \mathsf{Sign}(\mathtt{VCC}^{\mathsf{id}}, \mathtt{VCCs}_{\mathtt{sk}})$.
         E. Computes an Encrypted Verification Card Private key store $\mathsf{Enc}_s(\mathtt{t\_private\_ekey}(\mathtt{ske}(\mathsf{id})), \mathtt{KSkey}(\mathtt{SVK}_{\mathsf{id}}))$

F. Computes mapping table as follows:
$\mathsf{readCC}(\mathsf{H}(\mathsf{t\_nat}(\mathsf{H}(\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_k}(\mathsf{pCC}(\mathsf{ske}(\mathsf{id}), v(J)), \mathtt{VC}_{\mathsf{id}})))(, \mathsf{CMtable}(\mathtt{VCCs}_{\mathsf{sk}}))$
$= \mathsf{Enc}_s(\mathtt{CC}_i^{\mathsf{id}}, \mathsf{deltaX}(\mathsf{H}(\mathsf{t\_nat}(\mathsf{H}(\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_k}(\mathsf{pCC}(\mathsf{ske}(\mathsf{id}), v(J)), \mathtt{VC}_{\mathsf{id}})))))).$
$\mathsf{readVCC}(\mathsf{H}(\mathsf{t\_nat}(\mathsf{H}(\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_kc}(\mathsf{sq}(\mathsf{H}(\mathsf{pCC}(\mathsf{ske}(\mathsf{id}), \mathsf{sq}(\mathtt{BCK}^{\mathsf{id}})))), \mathtt{VC}_{\mathsf{id}})))), \mathsf{CMtable}(\mathtt{VCCs}_{\mathsf{sk}}))$
$= \mathsf{Enc}_s(\mathtt{VC}_{\mathsf{id}}, \mathsf{deltaX}(\mathsf{H}(\mathsf{t\_nat}(\mathsf{H}(\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_kc}(\mathsf{sq}(\mathsf{H}(\mathsf{pCC}(\mathsf{ske}(\mathsf{id}), \mathsf{sq}(\mathtt{BCK}^{\mathsf{id}})))), \mathtt{VC}_{\mathsf{id}})))))).$
$\mathsf{readS\_VCC}(\mathsf{H}(\mathsf{t\_nat}(\mathsf{H}(\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_kc}(\mathsf{sq}(\mathsf{H}(\mathsf{pCC}(\mathsf{ske}(\mathsf{id}), \mathsf{sq}(\mathtt{BCK}^{\mathsf{id}})), \mathtt{VC}_{\mathsf{id}}))))))), \mathsf{CMtable}(\mathtt{VCCs}_{\mathsf{sk}}))$
$= \mathsf{Sign}(\mathtt{VC}_{\mathsf{id}}, \mathtt{VCCs}_{\mathsf{sk}}).$

2. The second part of the Setup is dynamic (see paragraph on modeling setup in Section 4.1.2). It is done as follows: On input $\{\mathtt{K}_{\mathsf{id}}^{\mathrm{CCR}}, \mathtt{pC}_i, \pi_i\}_{i=1}^{\phi}$ from all CCRs for voter id, Print Office verifies proofs of the correct exponentiation $\mathsf{VerifExp}(\mathsf{pube}(\mathtt{k}_{\mathsf{id}}^{\mathrm{CCR}}), \mathtt{VC}_{\mathsf{id}}, v(J_i), \mathtt{pC}_i, \pi_i)$ for all voting choices. If the validation holds, it outputs the static table $\mathsf{CMtable}(\mathtt{VCCs}_{\mathsf{sk}})$ and computes the dynamic mapping table for the voter id as follows:
$\Big[\mathsf{H}\Big(\mathsf{t\_nat}(\mathsf{H}(\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_k}\big(\mathsf{pCC}(\mathsf{ske}(\mathsf{id}), \mathtt{pC}_i), \mathtt{VC}_{\mathsf{id}})))\Big), \mathsf{CMtable}(\mathtt{VCCs}_{\mathsf{sk}}),$
$\mathsf{Enc}_s\Big(\mathtt{CC}_i^{\mathsf{id}}, \mathsf{deltaX}\big(\mathsf{H}(\mathsf{t\_nat}(\mathsf{H}\big[\mathtt{VC}_{\mathsf{id}}, \mathsf{merge\_k}(\mathsf{pCC}\big(\mathsf{ske}(\mathsf{id}), \mathtt{pC}_i\big), \mathtt{VC}_{\mathsf{id}})\big]))\big)\Big)\Big]$
At the end, print office shuffles entries in the dynamic mapping table and outputs the result.*

3. Alice the Voter, on input $\mathsf{init}_{\mathsf{A}} = \mathsf{AliceData}(\mathsf{A})$ and her voting choices $\{J_i^{\mathsf{A}}\}_{i=1}^{\psi} \subseteq \{J_i\}_{i=1}^{n}$ runs $\mathsf{GetAliceData}$ to obtain codes she needs to vote
$$(\mathtt{SVK}_{\mathsf{id}}, \mathtt{VC}_{\mathsf{id}}, \mathtt{BCK}^{\mathsf{id}}, \mathtt{VCC}^{\mathsf{id}}, \mathtt{CC}_i^{\mathsf{id}})$$
w.r.t. her voting choices $\{J_i^{\mathsf{A}}\}_{i=1}^{\psi} \subseteq \{J_i\}_{i=1}^{n}$.

4. Alice's Voting Device, after establishing an authenticated connection with the Voting Server, sends Alice's verification card ID $\mathtt{VC}_{\mathsf{A}}$ to the Voting Server. The Voting Server on input $\mathtt{VC}_{\mathsf{A}}$ retrieves Alice's key store as $\mathtt{VCks}_{\mathsf{A}}$ and sends it to Alice's Voting Device.

5. Alice's Voting Device obtains Alice's Verification Card private key $\mathtt{k}_{\mathsf{A}}$ by running $\mathsf{GetKey}(\mathtt{SVK}_{\mathsf{A}}, \mathtt{VCks}_{\mathsf{A}})$. Next the Voting Device runs $\mathsf{CreateVote}(\mathtt{pk}_{\mathtt{EL}}, \mathtt{VC}_{\mathsf{A}}, \{J_i^{\mathsf{A}}\}_{i=1}^{\psi}, \mathtt{K}_{\mathsf{A}}, \mathtt{k}_{\mathsf{A}})$, obtaining a ballot
$$\mathtt{b} = (\mathtt{E1}, \mathtt{E2}, \mathsf{tild}(\mathtt{k}_{\mathsf{id}}, \mathtt{E1}), \mathtt{K}_{\mathsf{A}}, \mathtt{P})$$
that is received by the Voting Server. The ballot consists of several parts: $\mathtt{E1}$ is an encryption of Alice's voting choices $\{J_i^{\mathsf{A}}\}_{i=1}^{\psi}$; $\mathtt{E2}$ allows the Voting Server to compute the (short) Choice Return Codes corresponding to Alice's vote; the remaining components are used to prove consistency between ciphertexts $\mathtt{E1}$ and $\mathtt{E2}$.

6. The Voting Server interacts with CCRs to generate Choice Return Codes corresponding to Alice's selections $(\mathtt{CC}_1, \ldots, \mathtt{CC}_\psi)$ and sends them back to Alice's Voting Device.

7. Alice's Voting Device displays to her the set of Choice Return Codes $\{\mathtt{CC}_i\}_{i=1}^{\psi}$ received from the Voting Server. Alice checks that the received codes match the expected return codes $\{(J_i^{\mathsf{A}}, \mathtt{CC}_i^{\mathsf{A}})\}_{i=1}^{\psi}$.

8. If the codes match, Alice enters her Ballot Casting Key $\mathtt{BCK}^{\mathsf{A}}$ in the Voting Device to confirm the vote, which computes and forwards to the Voting Server the Confirmation Message $\mathtt{CM}^{\mathsf{A}} := \mathsf{pCC}(\mathtt{VCks}_{\mathsf{A}}, \mathsf{sq}(\mathtt{BCK}^{\mathsf{A}}))$

9. The Voting Server communicates with the CCRs to generate the Vote Confirmation Code $\mathtt{VCC}$ and sends it back to Alice, who then checks if $\mathtt{VCC} = \mathtt{VCC}^{\mathsf{A}}$.

10. Once the election is closed, the Voting Server proceeds to do the cleansing of the ballot box and outputs the list of encrypted voting options $L = \{(c_1, c_2)\}$ as a result.

*In the real protocol, the verification of the output of the CCRs is done after the generation of the mapping table, as opposed to here. The order is irrelevant, as long as the table is published iff the verification is successful.

11. On input $L = \{(c_1, c_2)\}$, Mixing Control Component CCM_1 verifies the cleansing by running ProcessVote on each ballot and checking that the votes of honest voters are present in the bb. If the verification succeeds, it runs the mixing and partial decryption algorithms and computes the proof of correct shuffling and decryption.

12. On input $\big(L = \big\{(c_1, c_2)\big\}, L_1 = \big\{(c_1^{(1)}, c_2^{(1)})\big\}_{\mathsf{mix}}, \mathtt{P_m}\big)$, Mixing Control Component CCM_j where $j \in \{2, 3, 4\}$ verifies the cleansing by running ProcessVote on each ballot and checking that the votes of honest voters are present in the bb and also verifies previous CCMs correct mixing and partial decryption proofs. If the verification succeeds, it runs the mixing and partial decryption algorithms and computes the proof of correct shuffling and decryption.

Chapter 5

Ballot Privacy Property

Roughly speaking, a remote voting protocol satisfies ballot privacy if no one can learn other information about the voting options of an honest voter, with a honest voting device, who has followed all the steps as described in the protocol, than what can be learned from the election result alone [9],[2]. Intuitively, in symbolic models ballot privacy is captured by asking that an attacker should not be able to distinguish the situation where Alice votes 0 and Bob votes 1 form the situation where the votes are swapped:

$$V_A(0)|V_B(1) \approx V_A(1)|V_B(0)$$

We recall that an election is of type (k, n) if the total number of options available in the election is n, and the number of any voters choices is k. For ballot privacy we consider two different types of voters:

- honest voters with uncompromised voting devices
- corrupted voters who are under the control of an attacker, including their voting devices

Let id be an honest voter iff she follows the instructions given to her. Note that for ballot privacy we need to assume that honest voters devices are uncompromised. For this reason, when compared to verifiability, for privacy honest voters use honest voting devices (i.e. the case honest voter with corrupted voting device is excluded).

Attacker model In our ballot privacy model, an attacker is given the following capabilities:

1. It can schedule an unbounded number of voters of any of the two types described above
2. It controls corrupted voting devices and corrupted voters
3. It controls the voting server
4. It controls one of Choice Return Code Control Components, say CCR_2
5. It controls all but one of the Mixing Control Components

An attacker does not:

1. Control the Print Office nor the uncorrupted Voting Devices
2. Have access to honest voters' private audit data
3. Control the trustworthy Choice Return Code Control Component, say CCR_1
4. Have access to CCR_1 private data
5. Control the trustworthy CCM
6. Cannot choose the choices $J_1, \ldots, J_k$ for honest voters

Thanks to the recent work [2], proving ballot privacy for an unbounded number of voters (i.e. the attacker can schedule an unbounded number of honest and corrupted voters) can be simplified to proving ballot privacy for 3 voters. More precisely, if there is a privacy attack against a given voting protocol with no revote of type (k, n) for an unbounded number of voters m, then there is an attack against the same protocol for $m = 3$ consisting of 2 honest voters (e.g. Alice and Bob) and a dishonest voter (e.g. Charlie) or for $m = 2$ consisting of 2 honest voters [2, Theorem 1]. Formally, we are able to verify that for an election with 3 voters Alice, Bob and Charlie, where Alice and Bob are honest but Charlie is corrupted, or for an election with 2 honest voters Alice and Bob, the situations where Alice and Bob swap their votes cannot be distinguished, i.e.

$$V_A(J_1^A, \ldots, J_k^A)|V_B(J_1^B, \ldots, J_k^B) \approx V_A(J_1^B, \ldots, J_k^B)|V_B(J_1^A, \ldots, J_k^A)$$

where $\{J_1^A, \ldots, J_k^A\}, \{J_1^B, \ldots, J_k^B\} \subseteq \{j_1, \ldots, j_n\}$.

The above property is captured by the ProVerif code given in Figure 5.1 and to be found in the files `spec_CCM1.pv` as given in Appendix A and in `spec_CCM2-3-4.pv`. The former file corresponds to an honest CCM_1 and the latter for an honest CCM_2, CCM_3 or CCM_4. Please note, that we are only able to prove the case $k = 1$. In the following we guide the reader through the ProVerif code that defines the ballot privacy process:

`process`
This command is used to initiate the ballot privacy process. In particular, it initiates the election.

```
out(c, pube(sk_EL_x)); out(c, pk_EL);
out(c, VCCs_pk)
out(c,pke(idA)); out(c,pke(idB));
```

Those commands are run to give the election parameters to the attacker.

`PrintOffice`
This line runs generation of the dynamic mapping table that is based on the CCRs input:

These lines initiate honest voters with pseudonyms id_A and id_B. In the voters respective public channels it is written that the voters with corresponding Verification Card IDs $\mathtt{VC}_A$ and $\mathtt{VC}_B$ are honest.

```
 |Alice_Cmp(AliceData(idA),choice[jA1,jB1])
 |Alice_Cmp(AliceData(idB),choice[jB1,jA1])}
```

These lines set the indistinghishability property discussed above:

$$V_A(J_1^A)|V_B(J_1^B) \approx V_A(J_1^B)|V_B(J_1^A)$$

```
    | (new idC:agent_id; out(c,AliceData(idC)))
```

These commands set the attacker to play and control a corrupted voter by giving the attacker the corresponding voters credentials to the attacker.

In case when one of the $\text{CCM}_2, \text{CCM}_3, \text{CCM}_4$ is honest, the following line concludes the game:

```
| CCMx(idA,idB)
```

Otherwise, if CCM_1 is honest, the game ends as follows:

```
| CCM1(idA,idB)
| out(c, sk_EL_2)
| out(c, sk_EL_3)
| out(c, sk_EL_4)
```

```
free   idA, idB : agent_id .      (* Defines the identity of two honest voters Alice and Bob
      # public *)
free   jA1, jB1 : nat      .      (* Voting choices, for the two Honest voters Alice and Bob
      # public *)


reduc forall k1: private_ekey, k2: private_ekey, J:nat;
homo(pCC(k1,pCC(k2,v(J)))) = pCC(k2,pCC(k1,v(J)))
.

let PrintOffice =
in(c, (pc1:nat,pr1:bitstring,pc2:nat,pr2:bitstring,pc3:nat,pr3:bitstring,pc4:nat,pr4:
    bitstring));
if VerifExp(pube(skccr1),VC(idA),v(jA1), pc1, pr1) = true then
if VerifExp(pube(skccr1),VC(idA),v(jB1), pc2, pr2) = true then
if VerifExp(pube(skccr2),VC(idB),v(jA1), pc3, pr3) = true then
if VerifExp(pube(skccr2),VC(idB),v(jB1), pc4, pr4) = true then
out(c, CMtable(VCCs_sk));
let t1 = (H(t_nat(H( (VC(idA), merge_k(pCC(ske(idA), pc1 ) , VC(idA) )) ))), CMtable(VCCs_sk
    ),
Enc_s(CC(idA,jA1), deltaX(H(t_nat(H( (VC(idA), merge_k( pCC(ske(idA), pc1 ) ,VC(idA) )  , VC
    (idA) )) ))))  ) in
let t2 = (H(t_nat(H( (VC(idA), merge_k( pCC(ske(idA), pc2 )  , VC(idA) )) ))), CMtable(VCCs_
    sk),
Enc_s(CC(idA,jB1), deltaX(H(t_nat(H( (VC(idA), merge_k( pCC(ske(idA), pc2 )  ,VC(idA) )  ,
    VC(idA) )) ))))  ) in
let t3 = (H(t_nat(H( (VC(idB), merge_k(pCC(ske(idB), pc3 ) , VC(idB) )) ))), CMtable(VCCs_sk
    ),
Enc_s(CC(idB,jA1), deltaX(H(t_nat(H( (VC(idB), merge_k( pCC(ske(idB), pc3 ) ,VC(idB) )  , VC
    (idB) )) ))))  ) in
let t4 = (H(t_nat(H( (VC(idB), merge_k(pCC(ske(idB), pc4 ) , VC(idB) )) ))), CMtable(VCCs_sk
    ),
Enc_s(CC(idB,jB1), deltaX(H(t_nat(H( (VC(idB), merge_k( pCC(ske(idB), pc4 ) , VC(idB) )  ,
    VC(idB) )) ))))  ) in
out( c,(choice[t1,t2], choice[t2,t1], choice[t3,t4],choice[t4,t3]) ).

(* 5. Main process, with the choices of honest voters. *)

(* Honest agents and their vote choices for the Observational Equivalence property (privacy)
    . *)
process
(* Publishes public data that are not already tagged as public. *)
out(c, pube(sk_EL_x)); out(c, pk_EL);   (* Publishes the public key of the honest CCM, and
    the pk_EL composed key.   *)
out(c, VCCs_pk);        (* Publish the public part of VCCs key pair.           *)
out(c, pke(idA)); out(c, pke(idB));   (* Publishes the honest voter's public keys.
     *)
PrintOffice
(* Roles for honest voters. *)
|  Alice_Cmp(AliceData(idA),choice[jA1,jB1])
| Alice_Cmp(AliceData(idB),choice[jB1,jA1])

(* Dishonest voter(s) : need only one for Obervational equivalence. *)
| (new idC:agent_id; out(c,AliceData(idC)))

(* The honest or dishonest CCMs *)
| CCMx(idA,idB)
```

Figure 5.1: Interactive mapping table generation and the main process in `spec_CCM2-3-4.pv` for Ballot Privacy property

Chapter 6

Conclusion

ProVerif has automatically verified that the property Ballot Privacy as defined in Chapter 5 holds for elections of type (1, n) for n unbounded, and $m = 3$ voters. This result, together with Theorem1 defined in [2], implies ballot privacy for elections of type (1, n) for n unbounded, and an unbounded number of voters m.

The corresponding ProVerif file containing the source code for the automatic verification of the security properties are to be found in the folder `analysis/`.

Limitations of Our Analysis

First of all we would like to remind the reader that symbolic models of cryptographic protocols deal with abstractions thereof. As a result, they omit numerous cryptographic and mathematical properties of the underlying primitives, but present the advantage of allowing, in some cases, for the automatic verification of security properties held by those symbolic models. Symbolic security arguments are widely accepted as a good indication that the design of a cryptographic protocol is not flawed, and it is considered to be a good sanitization method for complex cryptographic protocols, such as e-voting protocols. Symbolic proofs do not cover actual implementations of the security protocols, and might overlook special attacks that make use of specialized properties of the cryptographic primitives. Notwithstanding, as a result of our analysis we have been able to draw some recommendations and requirements that need to be satisfied by an actual implementation of the protocol analysed.

We point out next some simplifications of the original sVote specification that we made to help with the readability of the model and given the time constraints:

- In our abstraction we have modeled 2 CCRs instead of 4 CCRs. This has helped enormously our work with the model and significantly improves the readability of the model and this report. See discussion on number of control components in Section 4.1.1.
- We have not made use in our model of a series of non-interactive zero-knowledge proofs present in the sVote specification (namely, those computed by CCRs regarding the vote cast code generation) that are aimed at limiting the adversarial capabilities due to efficiency reasons. See a thorough discussion about the setup in Section 4.1.2.
- In our abstraction we have modeled the last CCM_4 without using the fact that its private key is split among Electoral Board members as the sVote specification define. Instead we assumed, that an honest CCM performs verification of cleansing and mixing performed by the nodes prior and outputs nothing in case verification fails. To see why this assumption can be done without loss of generality please refer to discussion on offline mixing component in Section 4.1.2.

Bibliography

[1] Ben Adida and C. Andrew Neff. Ballot casting assurance. In Dan S. Wallach and Ronald L. Rivest, editors, *2006 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT'06, Vancouver, BC, Canada, August 1, 2006*. USENIX Association, 2006.

[2] Myrto Arapinis, Véronique Cortier, and Steve Kremer. When are three voters enough for privacy properties? *IACR Cryptology ePrint Archive*, 2016:690, 2016.

[3] 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 2015*. IEEE Computer Society, 5 2015.

[4] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2014. prosecco.gforge.inria.fr/personal/bblanche/proverif.

[5] Swiss Federal Chancellery. Federal chancellery ordinance on electronic voting (VEleS) 2.0. Available at https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html, July 2018.

[6] Swiss Federal Chancellery. Technical and administrative requirements for electronic vote casting. Annex of the Federal Chancellery Ordinance on Electronic Voting 2.0 . https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html, July 2018.

[7] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachène. Election verifiability for Helios under weaker trust assumptions. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014 Proceedings, Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 327–344. Springer, 2014.

[8] Véronique Cortier and Steve Kremer. Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends in Programming Languages*, 1(3):151–267, 2014.

[9] Stéphanie Delaune, Steve Kremer, and Mark Ryan. *Verifying Privacy-Type Properties of Electronic Voting Protocols: A Taster*, pages 289–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[10] Scytl Secure Electronic. Scytl sVote. Protocol Specifications., 2018. V5.1.

[11] Jan Gerlach and Urs Gasser. Three case studies from Switzerland: E-voting, 2009.

[12] Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345, pages 389–404. Springer, 2010.

Appendix A

ProVerif source file spec_CCM2-3-4.pv

```
(* ProVerif specification, version 8, for Scytl's Voting Protocol and the privacy property.
    Tabulation size : 8 spaces.                    *)
(* Observational Equivalence : For Ballot Privacy; 2 honest + 1 or 0 dishonest in the
    challenge;             *)

(* This ProVerif model corresponds to the specification of the Scytl protocol, as described
    in the document:            *)
(* "sVote Voting with Control Components Protocol-Cryptographic proof of Privacy" as
    available on October 23th.           *)

(*                                     *)
(* Trust assumptions:                                  *)
(* - Print Office is honest                                        *)
(* - voting client is honest                                    *)
(*   with the trust assumptions as required with the level "Universal Verifiability" of the
    Swiss Chancelerie                     *)
(* - at least one CCM is honest                                                          *)
(* - all CCR are dishonest when the voting phase start: they simply provide all their
    material to the adversary           *)
(* - the voting server is dishonest
         *)
(* - all but two voters are dishonest                             *)


set ignoreTypes=false.


(* 1. Objects and Types *)

(* Types *)
type   agent_id.      fun t_agent_id(agent_id)       :    bitstring   [data,typeConverter].
     (* The type for any voter pseudonym, from 'ID' *)
type   password.      fun t_password(password)       :    bitstring   [data,typeConverter].
     (* The type for user passwords, from 'A_svk'.  *)
type   nat     .      fun t_nat(nat)                 :    bitstring   [data,typeConverter].
     (* The type for natural numbers and codes,     *)

(* Utilities *)
free   c          : channel              .    (* Public Channel for communications with the
    intruder    # public    *)
fun    sq(nat)          : nat   [data]     .    (* a square function, for the bck(..)
      # public    *)
free   a1                     : nat        .    (* MODELING : an indexed set of nat atoms,
    for disquality tests  -- only in Test. *)


(* 2. Intruder capabilities and functions extracted from the CryptoPaper. *)

```

```
36
37 (* Public Key Encryption Scheme, from CryptoPaper section 3.1;    Note: Enc(k,m,r) i.e. with
       key first, then message and randomness.        *)
38 type   private_ekey.  fun t_private_ekey(private_ekey):   bitstring   [data,typeConverter].
       (* The type for private keys + type converter. *)
39 type   public_ekey .  fun t_public_ekey(public_ekey)  :   bitstring   [data,typeConverter].
       (* The type for public keys  + type converter. *)
40 fun    ske(agent_id)          : private_ekey [private].   (* The private enc. key associated
       to an agent_id        # private                *)
41 fun    pube(private_ekey)     : public_ekey           .   (* The function to get a public
      key from the private one # public , noninvertible *)
42 letfun pke(Id:agent_id)       = pube(ske(Id))         .   (* The public  enc. key associated
       to an agent id        # public ,    invertible *)
43 (* Modeling : In this model, we do not need anymore the split of Enc(K,M,R) into two sub
      operators Enc_c1(R),Enc_c2(K,M,R) since no function*)
44 (*        in this model refers to these parts separatly. Therefore, we use the very
      classical modeling of Enc(..) as one operator.      *)
45 fun    Enc(public_ekey,bitstring,nat) : bitstring.        (* The asymmetric encryption
      function with explicit random. *)
46 reduc  forall Sk:private_ekey, M:bitstring, R:nat;    Dec(Sk,Enc(pube(Sk),M,R)) = M   .
              (* Decryption *)
47 reduc  forall Pk:public_ekey,  M:bitstring, R:nat;    VerifE(Pk,Enc(Pk,M,R))    = true.
              (* Checks key *)
48 reduc  forall Id:agent_id;                            Get_Id(pube(ske(Id)))     = Id  .
              (* Extract Id *)
49
50
51 (* Symmetric key encryption scheme - from CryptoPaper section 3.1 *)                  (*
      Without ghash(c,a) *)
52 type   symmetric_ekey. fun t_symmetric_ekey(symmetric_ekey): bitstring [data,typeConverter].
       (* The type for symetric encryption keys.      *)
53 fun    Enc_s(bitstring,symmetric_ekey) : bitstring   .
       (* Encryption (symmetric key)    message, key  *)
54 reduc  forall SKey:symmetric_ekey, M:bitstring;      Dec_s(Enc_s(M,SKey),SKey) = M.
       (* Decyption  (symmetric key)    cypher , key  *)
55
56
57 (* Key derivation functions, from CryptoPaper section 3.2 *)
58 fun    delta(private_ekey,bitstring) : symmetric_ekey [data]. (* The Key Derivation Function
       '\delta' based on PBKDF2(SVK_id,KEYseed), with salt. # public, invertible *)
59 fun    KSkey(password)         : symmetric_ekey [data]. (* The key PBKDF2(SVK_id,KEYseed)
      with a KEYseed fixed. # public invertible                                  *)
60 fun    deltaX(nat)                  : symmetric_ekey [data]. (* The Key Derivation Function
       '\delta' with a fixed 'salt' for Setup.SDM(..).      # public, invertible *)
61
62 (* Representation of the voting options, from CryptoPaper section 3.3 *)
63 fun    v(nat)                 : nat   [data] .    (* The function from voting choices to
      voting options    # public ,    invertible *)
64 fun    phi(nat)               : bitstring [data] .    (* The agregation function (prod) over
       a tuple of votes  # public ,    invertible *)
65
66
67 (* Hash function, from CryptoPaper section 3.4 *)
68 fun    H(bitstring) : nat .             (* The hash function H(..). It produces numbers of
      size <|p|. # public, noninvertible *)
69
70
71 (* Signature scheme, from CryptoPaper section 3.5 *)
72 type   private_skey.  fun t_private_skey(private_skey):   bitstring   [data,typeConverter].
       (* The type for private keys + type converter. *)
73 type   public_skey .  fun t_public_skey(public_skey)  :   bitstring   [data,typeConverter].
       (* The type for public keys  + type converter. *)
74 fun    pubs(private_skey)           : public_skey     .   (* The function to rebuild a
      public key from the private # public , noninvertible *)
75 fun    Sign(bitstring,private_skey) : bitstring       .   (* The digital signature function
      with explicit random number. (Signature)        *)
76 reduc  forall Sk:private_skey, M:bitstring;           Verify(pubs(Sk),M,Sign(M,Sk)) = true.
              (* Verification *)
```

```
reduc  forall Sk:private_skey, M:bitstring;           Checks(pubs(Sk),Sign(M,Sk))   = M   .
          (* More expressive than Verify. *)
reduc  forall Sk:private_skey, M:bitstring;           Get_Message(Sign(M,Sk))       = M   .
          (* Worst case for modeling only *)


(* Non-interractive zero-knowledge proofs of knowledge, from CryptoPaper section 3.6 *)
(* The 'Equality of Discrete Logarithm', 'Knowledge of Encryption Exponent' and 'Proof of
    Equality of Encryption' are abstracted inside the *)
(* ZKP and VerifP operators.     NOTE : no pk_CCR is used here, because it is assumed in
    this model that sk_CCR is publicly known.  *)
(* Modeling : consequently to abstracting pk_CCR and sk_CCR, this ZKP does not (need to)
    guarantee the fair use of pk_CCR in CreateVote(..).*)
fun    pCC(private_ekey,nat) : nat.      (* partial Choice Return Codes, i.e. pCC('k_id','v_
    i') models 'v_i^k_id'  # public *)
fun    tild(private_ekey,bitstring): nat.
fun    VC(agent_id)  : bitstring  [data].    (* the random Verification Card for id
            # public *)
fun    ZKP(public_ekey, public_ekey, bitstring, bitstring, bitstring, nat, private_ekey) :
    bitstring.
reduc  forall pk_EL:public_ekey, Id:agent_id, J1:nat, R:nat;
       VerifP(pk_EL, pube(ske(Id)), VC(Id), Enc(pk_EL,phi(v(J1)),R), t_nat(pCC(ske(Id),v(J1)
    )),
          ZKP(pk_EL, pube(ske(Id)), VC(Id), Enc(pk_EL,phi(v(J1)),R), t_nat(pCC(ske(Id),v(J1)
    )), R,ske(Id)))
       = true
.

fun    ZKPexp(public_ekey,bitstring,  nat, nat, private_ekey) : bitstring.
reduc  forall kccr_id: private_ekey, Id:agent_id, J:nat;
       VerifExp(pube(kccr_id),VC(Id), v(J), pCC(kccr_id,v(J)),
          ZKPexp(pube(kccr_id),VC(Id),v(J), pCC(kccr_id,v(J)),kccr_id))
       = true
.

(* Verifiable mixnet, from CryptoPaper section 3.7 *)
(* Three votes are considered, two honest plus one dishonest. *)

fun PDec(private_ekey, bitstring):bitstring.         (* Partial decryption plus re-
    encryption, with pub. enc. key abstracted. *)
fun PMix(private_ekey, bitstring, bitstring):bitstring.          (* Proof of correct Mixing,
     with pi_mix and pi_dec *)
fun MixVerify(public_ekey, bitstring, bitstring, bitstring) : bool       (* Checks the
    validity of PMix(..).            *)
reduc     forall k:private_ekey, E1:bitstring, E2:bitstring, E3:bitstring;
          MixVerify(pube(k), (E1,E2,E3), (PDec(k,E1),PDec(k,E2),PDec(k,E3)), PMix(k, (E1,E2,
    E3), (PDec(k,E1),PDec(k,E2),PDec(k,E3)))) = true
otherwise forall k:private_ekey, E1:bitstring, E2:bitstring, E3:bitstring;
          MixVerify(pube(k), (E1,E2,E3), (PDec(k,E1),PDec(k,E3),PDec(k,E2)), PMix(k, (E1,E2,
    E3), (PDec(k,E1),PDec(k,E3),PDec(k,E2)))) = true
otherwise forall k:private_ekey, E1:bitstring, E2:bitstring, E3:bitstring;
          MixVerify(pube(k), (E1,E2,E3), (PDec(k,E2),PDec(k,E1),PDec(k,E3)), PMix(k, (E1,E2,
    E3), (PDec(k,E2),PDec(k,E1),PDec(k,E3)))) = true
otherwise forall k:private_ekey, E1:bitstring, E2:bitstring, E3:bitstring;
          MixVerify(pube(k), (E1,E2,E3), (PDec(k,E2),PDec(k,E3),PDec(k,E1)), PMix(k, (E1,E2,
    E3), (PDec(k,E2),PDec(k,E3),PDec(k,E1)))) = true
otherwise forall k:private_ekey, E1:bitstring, E2:bitstring, E3:bitstring;
          MixVerify(pube(k), (E1,E2,E3), (PDec(k,E3),PDec(k,E1),PDec(k,E2)), PMix(k, (E1,E2,
    E3), (PDec(k,E3),PDec(k,E1),PDec(k,E2)))) = true
otherwise forall k:private_ekey, E1:bitstring, E2:bitstring, E3:bitstring;
          MixVerify(pube(k), (E1,E2,E3), (PDec(k,E3),PDec(k,E2),PDec(k,E1)), PMix(k, (E1,E2,
    E3), (PDec(k,E3),PDec(k,E2),PDec(k,E1)))) = true
.


(* 3. Methods and Agents processes *)


```

```
(* The derivation of the Verification Card private key, from CryptoPaper section 4.4.1 *)
letfun GetKey(SVK_id:password,VCks_id:bitstring) =
    let KSkey_id = KSkey(SVK_id) in
    let t_private_ekey(Sk_id:private_ekey) = Dec_s(VCks_id,KSkey_id) in
    Sk_id
.


(* The process of creating a ballot, from CryptoPaper section 4.4.2. *)       (* VCidpk ->
    Pk_id alias K_id; VCidsk -> Sk_id alias k_id *)
(* Modeling : consequently to abstracting pk_CCR and sk_CCR, the 'E2' part of the ballot
    directly contains the unencrypted partial Choice Return Codes.   *)
(* Modeling: The random R generated in this process is obtained as an extra argument due to
    ProVerif's language constraints.          *)
letfun CreateVote(pk_EL:public_ekey, VC_id:bitstring, J1:nat, Pk_id:public_ekey, Sk_id:
    private_ekey) =
    let V = phi(v(J1)) in new R:nat;
    let E1 = Enc(pk_EL, V, R) in
    let E2 = t_nat(pCC(Sk_id,v(J1))) in
    let P = ZKP(pk_EL,Pk_id,VC_id,E1,E2, R, Sk_id) in
    (E1, E2, tild(Sk_id,E1), Pk_id, P)
.


(* The verification algorithm to test a ballot, from CryptoPaper section 4.4.3 *)
(* Modeling : consequently to abstracting pk_CCR and sk_CCR, the ZKP does not (need to)
    guaranty the fair use of pk_CCR in the bulletin. *)
letfun ProcessVote(pk_EL:public_ekey, VC_id:bitstring, B:bitstring) =
    let (E1:bitstring, E2:bitstring, EC:nat, Pk_id:public_ekey, P:bitstring) = B in
    let Ok = VerifP(pk_EL, Pk_id, VC_id, E1, E2, P) in
    true
.


(* The algorithm to check the Choice Return Codes, from CryptoPaper section 4.4.6 *)
(* AuditCodes(....) : directly done by the voter in his process.              *)


(* SetupElKey.CCM_i - Generates CCM's secret keys. From CryptoPaper section 4.3.1   *)
free   sk_EL_x : private_ekey [private].    (* for CCMx honest   # private    *)
free   kccr_idA,kccr_idB : private_ekey .

(* MODELING : Other CCM's private keys are fixed and public, so they are abstracted away. *)
(* Consequently, no function in this model requires explicitly any of these as argument.  *)

(* SetupElKey.SDM   - Generates the Election's public key. From CryptoPaper section 4.3.1 *)
(* MODELING : Since all CCM's key pairs except one are both fixed and public, mergepk(Pk_x)
    models the   *)
(* merge of the public key Pk_x of the honest CCM with all the public keys of dishonest CCMs
    .    *)
fun    mergepk(public_ekey) : public_ekey [data]. (*   # public, invertible since other keys
     known *)
letfun pk_EL = mergepk(pube(sk_EL_x)).

(* Setup.SDM(ID) Part 1 - Initial Setup between the Print Office and the CCRs, based on an
    shared list of voter pseudonyms ID, fixed prior to this, from CryptoPaper section 4.3.2
    *)
(*     'sk_SDM' and 'pk_SDM' are abstracted                          *)
(*     'C_sk', alias the Codes secret key is abstracted too since it is fixed and public (it
     is public because it is transmitted to the voting server).               *)
free   VCCs_sk : private_skey [private] .   (* the Vote Cast Return Code Signer     #
    private *)
letfun VCCs_pk = pubs(VCCs_sk)          .   (* the pub. part of the VCCs, to be published
    *)

(* - Keys, Identifiers and Codes generated for each voter pseudonym *)
(*   Note : the Verification Card key pair (K_id,k_id) <- Gen_e(1^\lambda)  is denoted  (pke
    (id),ske(id)) *)
```

```
fun    SVK(agent_id) : password [private].    (* the Start Voting Key for id        # private *)
reduc  forall Id:agent_id; SVKtoID(SVK(Id)) = Id. (* the id retrieved from the Start Voting Key     *)
reduc  forall Id:agent_id; VCks(Id) = Enc_s(t_private_ekey(ske(Id)),KSkey(SVK(Id))).
         (* encrypted Verification Card private key, published. *)
fun    BCK(agent_id)    : nat       [private].    (* the Ballot Casting Key for id        # private *)
fun    CC(agent_id,nat) : bitstring [private].    (* the short Choice Return Codes for id # private *)   (* 'A_cc' *)
fun    VCC(agent_id)    : bitstring [private].    (* the short Vote Cast Return Code, id  # private *)   (* 'A_vcc' *)
(* letfun S_VCC(Id:agent_id) = Sign(VCC(Id),VCCs_sk). -- the signature value of VCC for id  # shortcut *)

(* Setup.CCR_i(init_CCR) - Computation of Long Choice Return Codes with the help of CCRs. *)
(* Modeling : the random secret keys 'k_1' to 'k_4' and 'kc_1' to 'kc_4' are both fixed and public, thus they too can be safely abstracted away.     *)
(* free   k_1 , k_2 , k_3 , k_4  : private_ekey . *)    (* the random secret key 'k_i'    # public  *)  (* public because transmitted to the CCRs. *)
(* free   kc_1, kc_2, kc_3, kc_4 : private_ekey . *)    (* the random secret key 'kc_i'   # public  *)  (* public because transmitted to the CCRs. *)

(* Setup.SDM(ID) Part 2 - After interracting with all CCRs for creating the Long Choice Return Codes. *)
(* Modeling : we need to define a function mergeCC(...) to combine contributions of CCRs to building the 'pC^id_i' and 'pVCC^id' codes, i.e. like :       *)
(*    fun mergeCC(nat,private_ekey,private_ekey,private_ekey,private_ekey) : bitstring.
   --    Modeling fct. for building the 'pC^id_i' pre-Choice Return codes.  *)
(* where: mergeCC( 'v_i^k_id' , 'k^1_id' , .. , 'k^4_id' ) models '(v_i^k_id)^(k^1_id) * .. * (v_i^k_id)^(k^4_id)', and thus                  *)
(*        mergeCC( 'v_i^k_id' , 'k^1_id' , .. , 'k^4_id' ) models '( v_i^(k^1_id) * .. * v_i^(k^4_id) )^(k_id)', i.e. 'pC^id_i' after a perfect Setup.SDM/CCR_i.    *)
(*        Note : 'v_i^k_id' is meant to be build with the pCC(..) function as defined for ZKP, i.e. pCC( 'k_id', 'v_i' ) models 'v_i^(k_id)'.         *)
(*        Note : 'v_i' is modeled with the v(..) function from voting choices to voting options, i.e. v( 'i' ) models 'v_i'.              *)
(*        Note : 'k_id' is the voter's Verification Card private key, and is modeled as ske( id) with id:agent_id.                      *)
(*        Note : 'k^i_id' is the 'delta(k_i,VC_id)' derived key of CCR_i, and is modeled as delta( k_i, VC(id) ) with i \in {1..4} and id:agent_id.       *)
(* Modeling : however, since the keys 'k_i' (and 'kc_i'), i=1..4, have been abstracted away, we do not want to (and cannot) use them as arguments to mergeCC(...).  *)
fun    merge_k( nat, bitstring) : bitstring [data].      (* where  merge_k('v_i^k_id',VC_id) models mergeCC('v_i^k_id', delta(k_1,VC_id), ..,delta(k_4,VC_id) ). *)
(* For the kc_i keys, this maleability is blocked through the use of the hash function H(..) in the spec. Therefore, we define :                *)
fun    merge_kc(nat, bitstring) : bitstring [data].
(*    Where merge_kc('H(bck^2^k_id)^2',VC_id) models 'H(bck^2^k_id)^2^delta(kc_1,VC_id) * .. * H(bck^2^k_id)^2^delta(kc_4,VC_id)' only.           *)
(*    Note : these powers cannot be mixed together anymore due to H(..).
        *)
(*    Note : '(BCK^id^2)^k_id' is meant to be built with the pCC(..) function too, as defined for ZKP, i.e. pCC('k_id', sq('BCK^id')) models '(BCK^id^2)^k_id'.     *)
(* This is now used to define lCC and lVCC. These are shortcuts, shown here for reference and which content allows to define the access to the CMtable.           *)
(* letfun lCC(Id:agent_id,J:nat)  = H( (VC(Id), merge_k(       pCC(ske(Id),   v(J)    )  , VC(Id) )) ).                    *)
(* letfun lVCC(Id:agent_id)       = H( (VC(Id), merge_kc( sq(H(t_nat(pCC(ske(Id),sq(BCK(Id))))))), VC(Id) )) ).                  *)
(* letfun hlCC(Id:agent_id,J:nat) = H(t_nat(lCC(Id,J))) with C_sk abstracted away, i.e. :
                          *)
(*            = H(t_nat(H( (VC(Id), merge_k(       pCC(ske(Id),   v(J)    )  , VC(Id) )) ))
   .                    *)
(* letfun hlVCC(Id:agent_id)      = H(t_nat(lVCC(Id))) with C_sk abstracted away, i.e. :
                          *)
(*                  = H(t_nat(H( (VC(Id), merge_kc( sq(H(t_nat(pCC(ske(Id),sq(BCK(Id))))))), VC(Id) )))).              *)
```

```
(* letfun skcc(Id:agent_id,J:nat) = deltaX(hlCC(Id,J) ) = deltaX(H(t_nat(H( (VC(Id), merge_k
    (       pCC(ske(Id),   v(J)    )  , VC(Id) )) ))).         *)
(* letfun skvcc(Id:agent_id)      = deltaX(hlVCC(Id,J)) = deltaX(H(t_nat(H( (VC(Id), merge_
    kc( sq(H(t_nat(pCC(ske(Id),sq(BCK(Id)))))), VC(Id) )) ))).           *)

(* - Reductions to access the CMtable. The (commented out) variants show these reduction
    using the lCC, lVCC and S_VCC shortcuts above which need to be inlined.    *)
(* reduc  forall Id:agent_id, J:nat; readCC( H(t_nat(lCC(Id,J))) ) = Enc_s(   CC(Id,J)
       , skcc(Id,J) ).          -- Access to the CMtable part for CC.  *)
(* reduc  forall Id:agent_id;       readVCC( H(t_nat(lVCC(Id)) ) ) = Enc_s( (VCC(Id),S_VCC(
    Id)) , skvcc(Id)  ).          -- Access to the CMtable part for VCC. *)
(* Modeling : this last one is splitted in two parts : one that produces the encryption of
    VCC(Id), the other one that simply produces S_VCC(Id) unencrypted.       *)
fun   CMtable(private_skey) : bitstring [private].                                (* The
    CMtable itself. *)
reduc forall VCCs_Sk:private_skey, Id:agent_id, J:nat;
      readCC( H(t_nat(H( (VC(Id), merge_k(       pCC(ske(Id),   v(J)    )  , VC(Id) )) ))),
    CMtable(VCCs_Sk))
            = Enc_s(CC(Id,J), deltaX(H(t_nat(H( (VC(Id), merge_k(       pCC(ske(Id),   v(J)
       )  , VC(Id) )) ))))).
reduc forall VCCs_Sk:private_skey, Id:agent_id;
      readVCC(H(t_nat(H( (VC(Id), merge_kc( sq(H(t_nat(pCC(ske(Id),sq(BCK(Id)))))), VC(Id) )
    ) ))), CMtable(VCCs_Sk))
            = Enc_s(VCC(Id) , deltaX(H(t_nat(H( (VC(Id), merge_kc( sq(H(t_nat(pCC(ske(Id),sq
    (BCK(Id)))))), VC(Id) )) ))))).
reduc forall VCCs_Sk:private_skey, Id:agent_id;
    readS_VCC(H(t_nat(H( (VC(Id), merge_kc( sq(H(t_nat(pCC(ske(Id),sq(BCK(Id)))))), VC(Id) )
    ) ))), CMtable(VCCs_Sk))
            = Sign(VCC(Id),VCCs_Sk).
(* Note : the CryptoPaper section 4.3.2 makes the Print Office output all secret data at the
     end of Setup.SDM. We assume that these are dispatched among agents.     *)



(* 4. Phases, from CryptoPaper section 4.7  --  aka. Protocol Roles *)

(* The public data published by the Print Office or CCRs are made available through public
    constants and public functions in Setup. *)


(* Voting Card for agent 'Id', as produced by the Print Office during the configuration
    phase, from CryptoPaper section 4.7 *)
(* Since the Voting Cards are transfered to the Voters is an way assumed to be secured, this
     data is made private.          *)
(* The GetAliceData reduction allows the voters can retreive the codes he needs to vote w.r.
    t. his voting choices.          *)
fun   AliceData(agent_id) : bitstring [private].
reduc forall Id:agent_id, J1:nat; GetAliceData(AliceData(Id),J1) = (SVK(Id), VC(Id), BCK(Id)
    , VCC(Id), CC(Id,J1)).



(* Voting Phase, Voter + Voting Device point of view, from CryptoPaper section 4.7 *)
let Alice_Cmp(InitData:bitstring,J1:nat) =
    (* Checks that the voting choices are all different *)
    if (J1 = J1 && a1 <> a1) then 0 else      (* No honest voter can use twice the same
    option *)

    (* Post-configuration phase, from CryptoPaper section 4.3  --  Gets the Voting Card from
     Print Office through postal channel *)
    let (SVK_id:password, VC_id:bitstring, BCK_id:nat, VCCid:bitstring, CCid1:bitstring) =
    GetAliceData(InitData,J1) in

    (* Authentication phase, from CryptoPaper section 4.3  --  the specific authentication
    method used is not modeled.  *)
    out(c, VC_id);            (* Send the Verification Card ID to the Voting System.  *)
    in( c, VCks_id:bitstring);          (* Receives the encrypted Verification Card keystore
    . *)
```

```
261
262     (* Voting phase, from CryptoPaper section 4.3  --  the voting process followed by agent
        Alice.      *)
263     let Sk_id = GetKey(SVK_id,VCks_id) in     (* Recovers the Verification Card private key.
            *)  (* Note: 'k_id' -> 'Sk_id' *)
264     out(c, CreateVote(pk_EL,VC_id,J1,pube(Sk_id),Sk_id)); (* Creates the voting ballot.
            *)
265     in( c, Rcv_CC_1:bitstring);         (* Receives the Choices Return Codes from the server
        . *)
266     if ((Rcv_CC_1=CCid1)) then (        (* Compares these codes.                          *)
267       out(c, pCC(Sk_id,sq(BCK_id)));        (* Confirms the Vote, from CryptoPaper section
        4.2.9  *)
268       in( c, =VCCid)            (* Receives and checks the Vote Cast Return Code. *)
269     )
270 .
271
272
273
274 (* The CCMs, from CryptoPaper section 4.7, 4.5.2, 4.5.3 *)
275 (* Note that our model assumes here that each CCM check the cleansing as well as all proof
        of correct mixing from previous CCMs *)
276 (* It is up to the implementation to guaranty that the final decryption occurs only if the
        Auditors have checked that all CCMs  *)
277 (* have produced valid proofs with valid input/outputs, and that no honest CCM has run twice
         it's process: otherwise the extra  *)
278 (* run could disclose privacy while the main, official run did not. The need to split the
        last sk_EL_4 key disapears with this. *)
279 (* We model one single honest CCM. *)
280
281 free mix:channel [private].         (* Private mixing channel for the (single) honest CCM.
          *)
282
283 (* Variants of the PDec, PMix and MixVerify merging together all the proofs produced by
        dishonest CCMs prior to the honest one. *)
284 (* The private keys of these CCM being fixed, public and abstrcted, these mothods do not
        take them as arguments.    *)
285 fun PDec2(bitstring):bitstring [data].      (* Partial decryption plus re-encryption, with
        pub. enc. key abstracted *)
286 fun PMix2(bitstring, bitstring):bitstring.    (* Two Proofs of correct Mixing in sequence,
        with pi_mix and pi_dec.  *)
287 fun MixVerify2(bitstring, bitstring, bitstring) : bool  (* Checks the validity of PMix2(..).
                *)
288 reduc     forall E1:bitstring, E2:bitstring, E3:bitstring;
289         MixVerify2((E1,E2,E3), (PDec2(E1),PDec2(E2),PDec2(E3)), PMix2((E1,E2,E3), (PDec2(E1)
        ,PDec2(E2),PDec2(E3)))) = true
290 otherwise forall E1:bitstring, E2:bitstring, E3:bitstring;
291         MixVerify2((E1,E2,E3), (PDec2(E1),PDec2(E3),PDec2(E2)), PMix2((E1,E2,E3), (PDec2(E1)
        ,PDec2(E3),PDec2(E2)))) = true
292 otherwise forall E1:bitstring, E2:bitstring, E3:bitstring;
293           MixVerify2((E1,E2,E3), (PDec2(E2),PDec2(E1),PDec2(E3)), PMix2((E1,E2,E3), (PDec2(E
        2),PDec2(E1),PDec2(E3)))) = true
294 otherwise forall E1:bitstring, E2:bitstring, E3:bitstring;
295           MixVerify2((E1,E2,E3), (PDec2(E2),PDec2(E3),PDec2(E1)), PMix2((E1,E2,E3), (PDec2(E
        2),PDec2(E3),PDec2(E1)))) = true
296 otherwise forall E1:bitstring, E2:bitstring, E3:bitstring;
297           MixVerify2((E1,E2,E3), (PDec2(E3),PDec2(E1),PDec2(E2)), PMix2((E1,E2,E3), (PDec2(E
        3),PDec2(E1),PDec2(E2)))) = true
298 otherwise forall E1:bitstring, E2:bitstring, E3:bitstring;
299           MixVerify2((E1,E2,E3), (PDec2(E3),PDec2(E2),PDec2(E1)), PMix2((E1,E2,E3), (PDec2(E
        3),PDec2(E2),PDec2(E1)))) = true
300 .
301
302 let CCMx(idA:agent_id,idB:agent_id) =
303    (* Cleansing + check that the ballots of the two honest voters were provided in the first
         list *)
304    in(c, (VC_id1:bitstring,B1:bitstring,VCC1:bitstring,S_VCC1:bitstring));  (* reading from
        bb *)
```

```
305    in(c, (VC_id2:bitstring,B2:bitstring,VCC2:bitstring,S_VCC2:bitstring));  (* reading from
        bb *)
306    in(c, (VC_id3:bitstring,B3:bitstring,VCC3:bitstring,S_VCC3:bitstring));  (* reading from
        bb *)
307    if VC_id1=VC(idA) then             (* Ensures that idA voted. *)
308    if VC_id2=VC(idB) then             (* Ensures that idB voted. *)
309    let Ok1 = ProcessVote(pk_EL,VC_id1,B1) in  let Ok1b = Verify(VCCs_pk,VCC1,S_VCC1) in
310    let Ok2 = ProcessVote(pk_EL,VC_id2,B2) in  let Ok2b = Verify(VCCs_pk,VCC2,S_VCC2) in
311    let Ok3 = ProcessVote(pk_EL,VC_id3,B3) in  let Ok3b = Verify(VCCs_pk,VCC3,S_VCC3) in
312    if VC_id1 <> VC_id2 && VC_id1 <> VC_id3 && VC_id2 <> VC_id3 then   (* Ensures distinct VC
        _id. *)
313    let (E1_1:bitstring, E2_1:bitstring, EC1:nat, Pk_id1:public_ekey, P1:bitstring) = B1 in
314    let (E1_2:bitstring, E2_2:bitstring, EC2:nat, Pk_id2:public_ekey, P2:bitstring) = B2 in
315    let (E1_3:bitstring, E2_3:bitstring, EC3:nat, Pk_id3:public_ekey, P3:bitstring) = B3 in
316    let L_0 = (E1_1,E1_2,E1_3) in
317    (* Checks previous CCM's correct mixing, re-encryption and partial decryption together,
        since they are all dishonest. *)
318    in(c, L:bitstring);    (* Requires a list of PDec2(..) of items from L_0, to be checked
        for MixVerify2(...).  *)
319    (* Modeling/Simplification : Unfortunatly, this makes ProVerif loop a lot, and raises the
         analysis time too high.   *)
320    (* Therefore, since all the private keys of dishonest CCMs are known, we assume that the
        intruder provides here a     *)
321    (* list L such that the ciphers of idA and idB are still in positions one and two. If a
        dishonest CCM were to mix the *)
322    (* list in an other order, the knowledge of it's private keys allows to reorder it with
        consistent mix/dec proofs.    *)
323    let (=PDec2(E1_1),=PDec2(E1_2),=PDec2(E1_3)) = L in
324    in(c, pi:bitstring); if MixVerify2(L_0,L,pi) = true then
325    (* Mixing, re-encryption and partial decryption *)
326    let (E1:bitstring,E2:bitstring,E3:bitstring) = L in
327    out(mix, choice[E1,E2]) | out(mix, choice[E2,E1]) |
328    in( mix, m1:bitstring ) ; in( mix, m2:bitstring ) ; let m3 = E3 in
329    out(c, (PDec(sk_EL_x,m1),PDec(sk_EL_x,m2),PDec(sk_EL_x,m3)));
330    out(c, PMix(sk_EL_x,(E1,E2,E3),(PDec(sk_EL_x,m1),PDec(sk_EL_x,m2),PDec(sk_EL_x,m3))))
331 .
332
333 (* Modeling : All partial decryptions after the honest CCM done together (if any), thus
        allowing to extract the votes.   *)
334 (*            Note that all the private keys needed to do this (if any) are both fixed and
        public, and thus abstracted.  *)
335 reduc forall sk:private_ekey, M:bitstring, R:nat;
336       Extract(PDec(sk,PDec2(Enc(mergepk(pube(sk)),M,R)))) = M
337 .
338
339 free   idA, idB : agent_id .      (* Defines the identity of two honest voters Alice and Bob
        # public *)
340 free   jA1, jB1 : nat      .      (* Voting choices, for the two Honest voters Alice and Bob
        # public *)
341
342
343 reduc forall k1: private_ekey, k2: private_ekey, J:nat;
344   homo(pCC(k1,pCC(k2,v(J)))) = pCC(k2,pCC(k1,v(J)))
345 .
346
347 let PrintOffice =
348   in(c, (pc1:nat,pr1:bitstring,pc2:nat,pr2:bitstring,pc3:nat,pr3:bitstring,pc4:nat,pr4:
        bitstring));
349   if VerifExp(pube(kccr_idA),VC(idA),v(jA1), pc1, pr1) = true then
350   if VerifExp(pube(kccr_idA),VC(idA),v(jB1), pc2, pr2) = true then
351   if VerifExp(pube(kccr_idB),VC(idB),v(jA1), pc3, pr3) = true then
352   if VerifExp(pube(kccr_idB),VC(idB),v(jB1), pc4, pr4) = true then
353   out(c, CMtable(VCCs_sk));
354   let t1 = (H(t_nat(H( (VC(idA), merge_k(pCC(ske(idA), pc1 ) , VC(idA) )) ))), CMtable(VCCs_
        sk),
355             Enc_s(CC(idA,jA1), deltaX(H(t_nat(H( (VC(idA), merge_k( pCC(ske(idA), pc1 ) ,
        VC(idA) )  , VC(idA) )) ))))  ) in
```

```
356   let t2 = (H(t_nat(H( (VC(idA), merge_k( pCC(ske(idA), pc2 )  , VC(idA) )) ))), CMtable(
        VCCs_sk),
357               Enc_s(CC(idA,jB1), deltaX(H(t_nat(H( (VC(idA), merge_k( pCC(ske(idA), pc2 )  ,
        VC(idA) )  , VC(idA) )) ))))  ) in
358   let t3 = (H(t_nat(H( (VC(idB), merge_k(pCC(ske(idB), pc3 ) , VC(idB) )) ))), CMtable(VCCs_
        sk),
359               Enc_s(CC(idB,jA1), deltaX(H(t_nat(H( (VC(idB), merge_k( pCC(ske(idB), pc3 ) ,
        VC(idB) )  , VC(idB) )) ))))  ) in
360   let t4 = (H(t_nat(H( (VC(idB), merge_k(pCC(ske(idB), pc4 ) , VC(idB) )) ))), CMtable(VCCs_
        sk),
361               Enc_s(CC(idB,jB1), deltaX(H(t_nat(H( (VC(idB), merge_k( pCC(ske(idB), pc4 ) ,
        VC(idB) )  , VC(idB) )) ))))  ) in
362   out( c,(choice[t1,t2], choice[t2,t1], choice[t3,t4],choice[t4,t3]) ).
363
364 (* 5. Main process, with the choices of honest voters. *)
365
366 (* Honest agents and their vote choices for the Observational Equivalence property (privacy)
      . *)
367
368
369 process
370   (* Publishes public data that are not already tagged as public. *)
371   out(c, pube(sk_EL_x)); out(c, pk_EL);   (* Publishes the public key of the honest CCM, and
        the pk_EL composed key.   *)
372   out(c, VCCs_pk);        (* Publish the public part of VCCs key pair.           *)
373   out(c, pke(idA)); out(c, pke(idB));   (* Publishes the honest voter's public keys.
          *)
374   PrintOffice
375 (* Roles for honest voters. *)
376  |  Alice_Cmp(AliceData(idA),choice[jA1,jB1])
377  | Alice_Cmp(AliceData(idB),choice[jB1,jA1])
378
379 (* Dishonest voter(s) : need only one for Obervational equivalence. *)
380  | (new idC:agent_id; out(c,AliceData(idC)))
381
382 (* The honest or dishonest CCMs *)
383  | CCMx(idA,idB)
```

"../interactive setup/spec_CCM2–3–4.pv"