Analysis of Cast-as-Intended Verifiability and Ballot Privacy Properties for Scytl's Swiss On-line Voting Protocol using ProVerif (version 2)

April 7, 2017

SCYTL SECURE ELECTRONIC VOTING

© 2017 SCYTL SECURE ELECTRONIC VOTING, S.A. All rights reserved

This Document is proprietary to SCYTL SECURE ELECTRONIC VOTING, S.A. (SCYTL) and is protected by the Spanish laws on copyright and by the applicable International Conventions.

No part of this Document may be: (i) communicated to the public, by any means including the right of making available; (ii) distributed including but not limited to sale, rental or lending; (iii) reproduced whether direct or indirectly, temporary or permanently by any means; and/or (iv) adapted, modified or otherwise transformed.

Notwithstanding the foregoing the Document may be printed and/or downloaded.

The cryptographic mechanisms and protocols described in this Document may be protected by SCYTL SECURE ELECTRONIC VOTING, S.A. and/or other third parties' patents.

Report

Analysis of Cast-as-Intended Verifiability and Ballot Privacy Properties for Scytl's Swiss On-line Voting Protocol using ProVerif (version 2)

David Galindo
University of Birmingham, UK

April 7, 2017

Abstract

This report is the companion to the ProVerif modelisation of the so-called Scytl's Swiss Online Voting protocol lead by Véronique Cortier (CNRS/LORIA) and Mathieu Turuani (INRIA/LORIA), with the collaboration of David Galindo (University of Birmingham). For completeness, the source code of the symbolic models can be found in the Appendix. This document file path is `report/report2017full.pdf`.

Contents

1 Introduction **1**

2 Abstract Building Blocks **2**

3 Abstract Model of Swiss Online Voting Protocol **5**

4 Verifiability Properties for an infinite number of candidates **10**

5 Ballot Privacy Property **15**

6 Results **19**

7 Lessons Learned and Limitations **20**

A ProVerif source file specs/spec_v13-REA.pve before expansion **24**

B ProVerif source file specs/study_v13-OBS_k=1.pv **33**

Chapter 1

Introduction

This report covers the automatic verification using ProVerif of the *cast-as-intended* verifiability and *ballot privacy* properties of the *Swiss Online Voting Protocol* by Scytl. This verification effort was performed by teams of LORIA in France and University of Birmingham in UK. In particular, the ProVerif analysis was lead by Véronique Cortier (CNRS) and Mathieu Turuani (INRIA). 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].

This model builds on a previous model by the same authors as described in [7] and has been produced as a response to the auditors requests in [2].

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.

Organization

In Chapter 2 we define the building blocks used later in Chapter 3 to construct our abstraction of Scytl's Swiss Online Voting Protocol. Our symbolic model is built from this abstraction. In Chapter 4 we define our symbolic verifiability properties for an unbounded number of candidates, while in Chapter 5 we define the ballot privacy property. In Chapter 6 we discuss our findings. In Chapter 7 we discuss the differences that might arise between the symbolic model and a concrete implementation of the protocol, pointing out a few attacks that we found in case an implementation of the protocol does not comply with our model.

Finally, in the Appendix we include the source code of our main *extended* ProVerif specifications. Some of these are files with extension `.pve` to be found in folder `/specs`, containing a quasi ProVerif code that allows to specify a symbolic model expressing elections of type (k, ∞), where k is the number of choices cast by a voter in the election, and the number n of total voting options (equivalently, number of candidates) is unbounded. More precisely, the Appendix contains the code corresponding to the file

- `specs/study_v13-REA.pve`
- `specs/study_v13-OBS_k=1.pv`

A standard ProVerif specification (with extension `.pv`) can automatically be obtained from the file `specs/study_v13-REA.pve` by particularizing k to a concrete value k_0, as explained in Chapter 6. The corresponding ProVerif files can be found in the folder `/analysis`.

Chapter 2

Abstract Building Blocks

Let us recall the main ingredients of a symbolic model [5]. Given a set $\mathcal{X}$ of *variables* and a set $\mathcal{N}$ of *names*, the set of *terms* associated to a set of function symbols $\mathcal{F}$, the variables $\mathcal{X}$ and the 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. For an adequate understanding of the following objects and/or notation, we assume the reader familiar with Scytl's cryptographic specification of the voting protocol [9, 10].

Functions

Function		**Type**	**Meaning**	**Properties**
ske	:	agent_id → priv_ekey	$\mathsf{ske}_{\mathsf{id}}$, the private decryption key of agent id	private
pube	:	priv_ekey → pub_ekey	builds public encryption key $\mathsf{pke}_{\mathsf{id}}$ from $\mathsf{ske}_{\mathsf{id}}$, i.e. $\mathsf{pke}_{\mathsf{id}} := \mathsf{pub}(\mathsf{ske}_{\mathsf{id}})$	public, non-invertible
sks	:	agent_id → priv_skey	$\mathsf{sks}_{\mathsf{id}}$, the private signing key of agent id	private
pubs	:	priv_skey → pub_skey	builds public signing key $\mathsf{pks}_{\mathsf{id}}$ from $\mathsf{sks}_{\mathsf{id}}$, i.e. $\mathsf{pks}_{\mathsf{id}} := \mathsf{pub}(\mathsf{sks}_{\mathsf{id}})$	public, non-invertible
f	:	sym_ekey×bitstr → sym_ekey	keyed pseudo-random function	public, non-invertible
pCC	:	priv_ekey×nat → nat	partial Choice Codes function $\mathsf{pCC}(\mathsf{ske}, v) := (v)^{\mathsf{ske}}$	public, non-invertible
Sig	:	priv_skey×bitstr → bitstr	digital signature primitive	public
Enc_s	:	sym_ekey → bitstr	symmetric encryption	public, non-invertible
Enc_{c_1}	:	bitstr → bitstr	randomized asymmetric encryption: first component	public, non-invertible
Enc_{c_2}	:	pub_ekey×bitstr ×bitstr → bitstr	randomized asymmetric encryption: second component	public, non-invertible
Enc	:	pub_ekey×bitstr ×bitstr → bitstr	$\mathsf{Enc}(\mathsf{pke}, \mathsf{m}, \mathsf{r}) := (\mathsf{Enc}_{c_1}(\mathsf{r}), \mathsf{Enc}_{c_2}(\mathsf{pke}, \mathsf{m}, \mathsf{r})) = \mathsf{ctxt}$	public, non-invertible
ϕ	:	bitstr_set → bitstr	aggregation function ϕ	public invertible
tild	:	bitstr×priv_ekey → nat	builds $(\tilde{c}_1, \tilde{c}_2) := \mathsf{tild}(\mathsf{ctxt}, \mathsf{ske})$ as $(c_1^{\mathsf{ske}}, c_2^{\mathsf{ske}})$ for ZKPs	public, non-invertible

Notation

Object		Type	Meaning	Properties
$\mathtt{EB}_{\mathtt{sk}}$	:	priv_ekey	electoral board private key	private
$\mathtt{EB}_{\mathtt{pk}}$	:	pub_ekey	electoral board public key ($= \mathsf{pube}(\mathtt{EB}_{\mathtt{sk}})$)	public
$\mathtt{VCCs}_{\mathtt{sk}}$	:	priv_skey	Vote Cast Code signing key	private
$\mathtt{VCCs}_{\mathtt{pk}}$	:	pub_skey	Vote Cast Code verification key ($= \mathsf{pubs}(\mathtt{VCCs}_{\mathtt{sk}})$)	public
$\mathtt{C}_{\mathtt{sk}}$	:	sym_ekey	Codes Secret Key	private
$\mathtt{SVK}_{\mathsf{id}}$	:	password	voter password	private
$\mathtt{VC}_{\mathsf{id}}$	:	agent_id	Verification Card ID associated to id	public
$\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}$	:	priv_ekey	Verification Card $\mathtt{VC}_{\mathsf{id}}$'s private key	private
$\mathtt{VC}^{\mathsf{id}}_{\mathtt{pk}}$	:	pub_ekey	Verification Card $\mathtt{VC}_{\mathsf{id}}$'s public key $\left(= \mathsf{pube}(\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}})\right)$	public
$\mathtt{BCK}^{\mathsf{id}}$	:	nat	Ballot Casting Key associated to id	private
$\mathtt{VCks}_{\mathsf{id}}$	:	bitstr	Key Store associated to id	private
$\mathtt{VCC}^{\mathsf{id}}$	:	sym_ekey	long Vote Cast Code for id	private
$\mathtt{sVCC}^{\mathsf{id}}$	:	bitstr	short Vote Cast Code for id	private
$S_{\mathtt{VCC}^{\mathsf{id}}}$	:	bitstr	validity proof for Vote Cast Code $\mathtt{sVCC}^{\mathsf{id}}$	private invertible
$j_1, \ldots, j_n$	:	nat	voting options available in the election	public
$J_1, \ldots, J_k$	:	nat	voter individual k choices in the election	public
v	:	nat → nat	$v_i := \mathsf{v}(j_i)$, the encoding of voting option j_i	public, invertible
$\mathtt{sCC}^{\mathsf{id}}_i$	:	bitstr	i-th Short Choice Code associated to id	private
$\mathtt{pCC}^{\mathsf{id}}_i$	:	nat	i-th Partial Choice Code associated to id	private
bb	:	table(agent_id,bitstr)	Ballot Box, storing ballots for each agent id	private
cb	:	table(agent_id,bitstr)	Confirmation Box, storing codes $\mathtt{sVCC}^{\mathsf{id}}/S_{\mathtt{VCC}^{\mathsf{id}}}$	private

These objects are specific to the Swiss OV protocol. Our notation tries to be as close as possible to the notation in [9, 10].

Attacker capabilities (Dolev-Yao)

The attacker capabilities consist on:

- Computing and inverting public invertible functions, e.g.
  - ◇ $\mathsf{pk}_{\mathsf{id}} \vdash \mathsf{id}$ and $\mathsf{id} \vdash \mathsf{pk}_{\mathsf{id}}$ $\forall \mathsf{id}$ (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.
  - ◇ $\mathsf{m} \vdash \mathsf{h}(\mathsf{m})$ $\forall \mathsf{m}$ (i.e. h is a efficiently computable hash function that cannot be inverted; in particular the rule $\mathsf{h}(\mathsf{m}) \vdash \mathsf{m}$ $\forall \mathsf{m}$ is *not available* to the attacker)
  - ◇ $\mathsf{sk}, \mathsf{m} \vdash \mathsf{f}(\mathsf{sk}, \mathsf{m}) \quad \forall \mathsf{sk}, \mathsf{m}$ (i.e. f is an efficiently computable pseudorandom function on knowledge the secret sk; for the sake of clarity, let us stress that the rule $\mathsf{m} \vdash \mathsf{f}(\mathsf{sk}, \mathsf{m})$ is *not available* to the attacker)
  - ◇ $\mathsf{sks}, \mathsf{m} \vdash \mathsf{Sig}(\mathsf{sks}, \mathsf{m}) \quad \forall \mathsf{sks}, \mathsf{m}$
  - ◇ $\mathsf{Enc}(\mathsf{pke}, \mathsf{m}, \mathsf{r}) \quad \forall \mathsf{pke}, \mathsf{m}, \mathsf{r}$
  - ◇ $\mathsf{Enc}_s(\mathsf{sk}, \mathsf{m}) \quad \forall \mathsf{sk}, \mathsf{m}$
- Computing the following functions associated to symmetric encryption:

- $\diamond$ $\mathsf{Dec}_s\big(\mathsf{sk}, \mathsf{Enc}_s(\mathsf{sk},\mathsf{m})\big) \vdash \mathsf{m} \quad \forall \mathsf{sk},\mathsf{m}$ (i.e. decrypting a symmetric encryption of a message returns the original message)

- Computing the following functions associated to public key encryption:

  - $\diamond$ $\mathsf{Dec}\big(\mathsf{ske}, \mathsf{Enc}(\mathsf{pub}(\mathsf{ske}),\mathsf{m},\mathsf{r})\big) \vdash m \quad \forall \mathsf{ske},\mathsf{m},\mathsf{r}$ (i.e. decrypting an asymmetric encryption of a message returns the original message)

- Computing the following functions associated to digital signatures:

  - $\diamond$ $\mathsf{verif}\big(\mathsf{pub}(\mathsf{sks}),\mathsf{m},\mathsf{Sig}(\mathsf{sks},\mathsf{m})\big) \vdash \mathsf{true} \quad \forall \mathsf{sks},\mathsf{m}$ (i.e. any well-formed signature is accepted)

- 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:

  - $\diamond$ zkp : pub_ekey $\times$ pub_ekey $\times$ bitstr $\times$ $\text{nat}_1 \times \ldots \times \text{nat}_k$ $\times$ nat $\times$ priv_ekey $\to$ bitstr;
    models a non-interactive zero-knowledge proof
  - $\diamond$ verifP : pub_ekey $\times$ pub_ekey $\times$ bitstr $\times$ $\text{nat}_1 \times \ldots \times \text{nat}_k$ $\times$ bitstr $\to$ bool;
    models the verification equation of a zero-knowledge proof
  - $\diamond$ $$\mathsf{verifP}\Big(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}^{\mathsf{id}}_{\mathtt{pk}}, \Big(\mathsf{Enc}_{c_1}(\mathsf{r}), \mathsf{Enc}_{c_2}\big(\mathtt{EB}_{\mathtt{pk}}, \phi(v_1,\ldots,v_k), \mathsf{r}\big)\Big), \mathsf{pCC}(\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}, v_1), \ldots, \mathsf{pCC}(\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}, v_k),$$
    $$\mathsf{zkp}\Big(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}^{\mathsf{id}}_{\mathtt{pk}}, \Big(\mathsf{Enc}_{c_1}(\mathsf{r}), \mathsf{Enc}_{c_2}\big(\mathtt{EB}_{\mathtt{pk}}, \phi(v_1,\ldots,v_k), \mathsf{r}\big)\Big), \mathsf{pCC}(\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}, v_1), \ldots, \mathsf{pCC}(\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}, v_k),$$
    $$\mathsf{r}, \mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}\Big)$$
    $$\Big) = \mathsf{true}$$
    where $\mathsf{pCC}(\mathsf{ske}, v) := (v)^{\mathsf{ske}}$ for any encoded voting option v.

The equation above is aimed at capturing the three non-interactive zero-knowledge proofs $\pi_{\mathtt{sch}}, \pi_{\mathtt{exp}}$ and $\pi_{\mathtt{pleq}}$ computed in the algorithm $\mathsf{CreateVote}$ as defined in [10]. Roughly speaking, $\pi_{\mathtt{sch}}$ proves knowledge of nonce r, while $\pi_{\mathtt{exp}}$ and $\pi_{\mathtt{pleq}}$ prove that the partial choice codes $\mathtt{pCC}_1, \ldots, \mathtt{pCC}_k$ are linked to the ciphertext $\mathsf{Enc}\big(\mathtt{EB}_{\mathtt{pk}}, \phi(v_1,\ldots,v_k), \mathsf{r}\big)$, that encrypts the voting options, and the voters' verification card secret key $\mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}}$. The most important limitation of this modelling of the zero-knowledge proofs is that the compacting function $\phi(\cdot)$ enjoys commutative properties that cannot be captured in Proverif as of yet.

Chapter 3

Abstract Model of Swiss Online Voting Protocol

A high-level account of our abstract model for the Swiss Online Voting Protocol is given in Figure 3.1. This model is the basis of our ProVerif analysis. The code in the files `specs/study_v13-REA.pve` and `specs/study_v13-OBS_k=1.pv` closely follows its description. Our abstract model cannot in any case replace, for verification purposes, the careful examination of the aforementioned files.[*] 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.

Data Initialisation

The following initialisation data is computed by the Registrar off-line and securely transmitted to the corresponding agents (see Figure 3.1):

- $\mathsf{init}_{\mathsf{id}}$ 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

To build the above data, the registrar uses knowledge of secrets $\mathtt{C}_{\mathtt{sk}}$ (the codes secret key) and $\mathtt{VCCs}_{\mathtt{sk}}$ (the Vote Cast Code private signing key).

Processes

We make use of the following processes, that are built from the functions that were described in Chapter 2:

- $\mathsf{AliceData}(\mathtt{SVK}_{\mathsf{id}}, \mathtt{C}_{\mathtt{sk}})$ is run by the Registrar and initialises the voter's secret data, where $\mathtt{SVK}_{\mathsf{id}}$ is a secret password (only known to Registrar and the corresponding voter associated to id):
  1. derive a voters' Voting Card ID as $\mathtt{VC}_{\mathsf{id}} := \mathsf{deltaId}(\mathtt{SVK})^{\dagger}$.
  2. assign a random Verification Card key pair $(\mathtt{VC}^{\mathsf{id}}_{\mathtt{pk}}, \mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}})$ to voter id
  3. assign a random Ballot Casting Key $\mathtt{BCK}^{\mathsf{id}}$ to voter id
  4. assign a key store $\mathtt{VCks}_{\mathsf{id}} := \mathsf{Enc}_s(\mathsf{deltaKey}(\mathtt{SVK}), \mathtt{VC}^{\mathsf{id}}_{\mathtt{sk}})$ to voter $\mathsf{id}^{\ddagger}$

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

[†] The function $\mathsf{deltaId}(\cdot) = \delta(\cdot, \mathtt{IDseed})$, where δ is a Key Derivation Function (cf [10]).

[‡] The function $\mathsf{deltaKey}(\cdot) = \delta(\cdot, \mathtt{KEYseed})$, where δ is a Key Derivation Function (cf [10]).

5. let $\mathtt{pCC}_i^{\mathsf{id}} := (v_i)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}}$, voter id's i-th partial Choice Code $\forall i = 1, \ldots, n$
6. let $\mathtt{CC}_i^{\mathsf{id}} = \mathsf{f}(\mathtt{C}_{\mathtt{sk}}, \mathtt{pCC}_i^{\mathsf{id}})$, voter id's i-th Choice Code $\forall i = 1, \ldots, n$
7. assign short Choice Codes $\mathtt{sCC}_i^{\mathsf{id}}$ at random for voter id, $\forall i = 1, \ldots, n$
8. let $\mathtt{VCC}^{\mathsf{id}} = \mathsf{f}(\mathtt{C}_{\mathtt{sk}}, (\mathtt{BCK}^{\mathsf{id}})^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}})$, the voter id's Vote Cast Code
9. assign a random short Vote Cast Code $\mathtt{sVCC}^{\mathsf{id}}$ to voter id
10. voter's data is initialised as $\mathsf{init}_{\mathsf{id}} := (\mathtt{SVK}_{\mathsf{id}}, \mathtt{BCK}^{\mathsf{id}}, \mathtt{sVCC}^{\mathsf{id}}, \{j_i, \mathtt{sCC}_i^{\mathsf{id}}\}_{i=1}^{n})$

- $\mathsf{ServData}(\mathtt{EB}_{\mathtt{pk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$ is run by the Registrar and initialises the Server's secret data as follows:

  1. let $\mathsf{CMtable} = \{(\mathtt{CM}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}})\}^{\mathsf{id's}}$, where
     (a) $\mathtt{CM}^{\mathsf{id}} = \left(\left\{\left[\mathsf{h}(\mathtt{CC}_i^{\mathsf{id}}), \mathsf{Enc}_s(\mathtt{CC}_i^{\mathsf{id}}, \mathtt{sCC}_i^{\mathsf{id}})\right]\right\}_{i=1}^{n}, \left[\mathsf{h}(\mathtt{VCC}^{\mathsf{id}}), \mathsf{Enc}_s(\mathtt{VCC}^{\mathsf{id}}, \mathtt{sVCC}^{\mathsf{id}})\right]\right)$
     (b) $S_{\mathtt{VCC}^{\mathsf{id}}} := \mathsf{Sig}(\mathtt{VCCs}_{\mathtt{sk}}, \mathtt{sVCC}^{\mathsf{id}})$, is the validity proof for the short code $\mathtt{sVCC}^{\mathsf{id}}$
  2. let $\mathsf{KeySt} := \{\mathtt{VCks}_{\mathsf{id}}\}^{\mathsf{id's}}$, the collection of all voters' key stores
  3. server's data is initialised as $\mathsf{init}_{\mathsf{S}} := (\mathtt{EB}_{\mathtt{pk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}} := \mathsf{pubs}(\mathtt{VCCs}_{\mathtt{sk}}), \mathsf{CMtable}, \mathsf{KeySt})$

- $\mathsf{GetKey}(\mathtt{SVK}_{\mathsf{id}}, \mathtt{VCks}_{\mathsf{id}})$ recovers the Verification Card private key $\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}$ as follows:

  1. output $\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}} := \mathsf{Dec}_s(\mathsf{deltaKey}(\mathtt{SVK}_{\mathsf{id}}), \mathtt{VCks}_{\mathsf{id}})$

- $\mathsf{CreateVote}(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}_{\mathsf{id}}, J_1, \ldots, J_k, \mathtt{VC}_{\mathtt{pk}}^{\mathsf{id}}, \mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}})$ consists of the following steps:

  1. let $v_i = \mathsf{v}(J_i)$ for $i = 1, \ldots, k$
  2. let $V = \phi(v_1, \ldots, v_k)$
  3. let $\mathsf{ctxt} = \mathsf{Enc}(\mathtt{EB}_{\mathtt{pk}}, V, \mathsf{r})$ for a fresh nonce r
  4. let $\mathsf{P} = \mathsf{zkp}\big(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}_{\mathtt{pk}}^{\mathsf{id}}, \mathsf{ctxt}, (v_1)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}}, \ldots, (v_k)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}}, \mathsf{r}, \mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}\big)$
  5. output $\mathsf{b} = (\mathsf{ctxt}, (v_1)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}}, \ldots, (v_k)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}}, \mathsf{tild}(\mathsf{ctxt}, \mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}), \mathtt{VC}_{\mathtt{pk}}^{\mathsf{id}}, \mathsf{P})$ a ballot§

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

- $\mathsf{ProcessVoteCheck}(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}_{\mathsf{id}}, \mathsf{b})$ outputs a boolean and consists of the following steps:

  1. let $\mathsf{b} = (\mathsf{ctxt}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{ec}, \mathbf{VC}_{\mathtt{pk}}^{\mathsf{id}}, \mathsf{P})$¶
  2. output $\mathsf{verifP}(\mathtt{EB}_{\mathtt{pk}}, \mathbf{VC}_{\mathtt{pk}}^{\mathsf{id}}, \mathsf{ctxt}, \mathsf{ec}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{P})$

  This is the algorithm used by the voting server to test a ballot.

- $\mathsf{CreateRC}(\mathsf{b}, \mathtt{C}_{\mathtt{sk}}, \mathsf{CMtable})$ might output a set of choice codes and consists of the following steps:

  1. let $\mathsf{b} = (\mathsf{ctxt}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{ec}, \mathbf{VC}_{\mathtt{pk}}^{\mathsf{id}}, \mathsf{P})$
  2. let $\mathbf{CC}_1 = \mathsf{f}(\mathtt{C}_{\mathtt{sk}}, \mathsf{w}_1), \ldots, \mathbf{CC}_k = \mathsf{f}(\mathtt{C}_{\mathtt{sk}}, \mathsf{w}_k)$
  3. output $(\mathtt{sCC}_1 := \mathsf{Dec}_s\left(\mathbf{CC}_1, \mathsf{Enc}_s(\mathtt{CC}_1, \mathtt{sCC}_1)\right), \ldots, \mathtt{sCC}_k := \mathsf{Dec}_s\left(\mathbf{CC}_k, \mathsf{Enc}_s(\mathtt{CC}_k, \mathtt{sCC}_k)\right))$ iff $[\mathsf{h}(\mathbf{CC}_i), \mathsf{Enc}_s(\mathtt{CC}_i, \mathtt{sCC}_i)]$ exists as an entry in $\mathsf{CMtable}$ $\forall i = 1, \ldots, k$
  4. else output void

  This is the algorithm used by the voting server to compute the return codes to be send to the voter's device.

- $\mathsf{AuditCodes}((\mathbf{sCC}_1, \ldots, \mathbf{sCC}_k), (\mathtt{sCC}_1^{\mathsf{id}}, \ldots, \mathtt{sCC}_k^{\mathsf{id}}))$ outputs a boolean and consists of the following steps:

§ $\mathsf{tild}(\mathsf{ctxt}, \mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}})$ equals $(c_1^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}}, c_2^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{id}}})$ as defined in CreateVote in [10]
¶ $\mathbf{VC}_{\mathtt{pk}}^{\mathsf{id}}$ as obtained from ballot b might not be equal to the legitimate $\mathtt{VC}_{\mathtt{pk}}^{\mathsf{id}}$

1. output true iff $\mathbf{sCC}_i \in \{\mathtt{sCC}_1^{\mathsf{id}}, \ldots, \mathtt{sCC}_k^{\mathsf{id}}\}\ \forall i = 1, \ldots, k$
2. else output false

This is the algorithm used by the voter to check whether all expected short Choice Codes $\{(J_i^{\mathsf{id}}, \mathtt{sCC}_i^{\mathsf{id}})\}_{i=1}^k$ corresponding to the voter's intended choices $\{J_i^{\mathsf{id}}\}_{i=1}^k$ were indeed received.

- $\mathsf{Tally}(\mathtt{EB}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{bb}, \mathsf{cb})$ is run by the tallying authority and proceeds as follows:

  1. let $\mathsf{bb} = \{(\mathtt{VC}_{\mathsf{id}}, \mathsf{b})\}_{\mathsf{id}}$ and $\mathsf{cb} = \{(\mathtt{VC}_{\mathsf{id}}, \mathtt{sVCC}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}})\}_{\mathsf{id}}$
  2. it creates a list L_{checked} containing all $(\mathtt{VC}_{\mathsf{id}}, \mathsf{b}) \in \mathsf{bb}$ such that $\mathsf{ProcessVoteCheck}(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}_{\mathsf{id}}, \mathsf{b}) = \mathsf{true}$ and $\mathsf{verif}(\mathtt{VCCs}_{\mathtt{pk}}, \mathtt{sVCC}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}}) = \mathsf{true}$, where $(\mathtt{VC}_{\mathsf{id}}, \mathtt{sVCC}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}}) \in \mathsf{cb}$
  3. if there are duplicates wrt. $\mathtt{VC}_{\mathsf{id}}$ in L_{checked} outputs void
  4. from each $(\mathtt{VC}_{\mathsf{id}}, \mathsf{b}) \in L_{\mathsf{checked}}$, obtains ctxt from b, next obtains ciphertext components (c_1, c_2) from ctxt (cf. [9, 10]) and adds (c_1, c_2) to a list L
  5. from the list $L = \{(c_1, c_2)\}$ computes a shuffled list $L_{\mathtt{mix}} = \{(c_1, c_2)\}_{\mathtt{mix}}$
  6. finally, outputs as a result a list $L_v = \{\mathsf{Dec}(\mathsf{ske}, (c_1, c_2))\}_{\mathtt{mix}}$ together with $L_{\mathtt{mix}}$

Swiss Online Voting Protocol - Abstraction

In Figure 3.1 we can find the abstraction of the Swiss Online Voting protocol that has been used as the basis of our ProVerif symbolic model. In the following we give some intuition thereof, and the reader is referred to [9, 10] for the original description of the protocol. Notice that the numbered items next have a correspondence with the items as numbered in Figure 3.1:

1. The Registrar initializes the voter's data $\mathsf{init}_{\mathsf{A}}$ and the server's data $\mathsf{init}_{\mathsf{S}}$ off-line:

   - The Registrar runs $\mathsf{AliceData}(\mathtt{SVK}_{\mathsf{A}}, \mathtt{C}_{\mathtt{sk}})$ to initialise Alice's secret data as $\mathsf{init}_{\mathsf{A}} := (\mathtt{SVK}_{\mathsf{A}}, \mathtt{BCK}^{\mathsf{A}}, \mathtt{sVCC}^{\mathsf{A}}, \{(j_i, \mathtt{sCC}_i^{\mathsf{A}})\}_{i=1}^n)$, where $\mathtt{sCC}_i^{\mathsf{A}}$ is Alice's *expected short choice code* for the choice j_i for each $i \in \{1, \ldots, n\}$.
   - The Registrar runs $\mathsf{ServData}(\mathtt{EB}_{\mathtt{pk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$ to initialise the Server's secret data for the election as $\mathsf{init}_{\mathsf{S}} := (\mathtt{EB}_{\mathtt{pk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}} := \mathsf{pubs}(\mathtt{VCCs}_{\mathtt{sk}}), \mathsf{CMtable}, \mathsf{KeySt})$, where

     $$\mathsf{CMtable} = \left\{ \left(\left\{ \left[\mathsf{h}(\mathtt{CC}_i^{\mathsf{id}}), \mathsf{Enc}_s(\mathtt{CC}_i^{\mathsf{id}}, \mathtt{sCC}_i^{\mathsf{id}}) \right] \right\}_{i=1}^n, \left[\mathsf{h}(\mathtt{VCC}^{\mathsf{id}}), \mathsf{Enc}_s(\mathtt{VCC}^{\mathsf{id}}, \mathtt{sVCC}^{\mathsf{id}}) \right], S_{\mathtt{VCC}^{\mathsf{id}}} \right) \right\}^{\mathsf{id's}}$$

   - $\mathsf{init}_{\mathsf{DEV}} := \mathtt{EB}_{\mathtt{pk}}$ is distributed to the Voting Devices.

     Roughly speaking, the information stored in $\mathsf{CMtable}$ will allow the Voting Server to compute a set of short Choice Codes corresponding to any voter id choices $\{J_i^{\mathsf{id}}\}_{i=1}^k$, without learning what the values of those choices. KeySt is a collection of key stores containing voters' secret credentials.

2. Alice the Voter, on input $\mathsf{init}_{\mathsf{A}}$, enters her voting choices $\{J_i\}_{i=1}^k \subseteq \{j_i\}_{i=1}^n$ in her Voting Device, followed by her password $\mathtt{SVK}_{\mathsf{A}}$, iff those voting choices are pairwise different and there are exactly k of them. Crucially, Alice keeps $\{(j_i, \mathtt{sCC}_i^{\mathsf{A}})\}_{i=1}^n$ and $\mathtt{sVCC}^{\mathsf{A}}$ secret from her Voting Device, where $\mathtt{sCC}_i^{\mathsf{A}}$ stands for Alice's short choice code for voting option j_i.

3. Alice's Voting Device, after establishing an authenticated connection with the Voting Server, computes Alice's voting card ID as $\mathtt{VC}_{\mathsf{A}} := \mathsf{deltaId}(\mathtt{SVK}_{\mathsf{A}})$ and sends it 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.

4. Alice's Voting Device obtains Alice's Voting Card private key $\mathtt{VC}_{\mathtt{sk}}^{\mathsf{A}}$ by running $\mathsf{GetKey}(\mathtt{SVK}_{\mathsf{A}}, \mathtt{VCks}_{\mathsf{A}})$. Next the voting device runs $\mathsf{CreateVote}(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VC}_{\mathsf{A}}, J_1, \ldots, J_k, \mathtt{VC}_{\mathtt{pk}}^{\mathsf{A}}, \mathtt{VC}_{\mathtt{sk}}^{\mathsf{A}})$, obtaining a ballot

   $$\mathsf{b} = (\mathsf{ctxt}, (v_1)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{A}}}, \ldots, (v_k)^{\mathtt{VC}_{\mathtt{sk}}^{\mathsf{A}}}, \mathsf{tild}(\mathsf{ctxt}, \mathtt{VC}_{\mathtt{sk}}^{\mathsf{A}}), \mathtt{VC}_{\mathtt{pk}}^{\mathsf{A}}, \mathsf{P})$$

that is received by the Voting Server. The ballot consists of several parts: ctxt is an encryption of Alice's voting choices $\{J_i\}_{i=1}^k$; $\{(v_i)^{\mathtt{VC}^{\mathsf{A}}_{\mathsf{sk}}}\}_{i=1}^k$ allow the Voting Server to compute the (short) Choice Codes corresponding to Alice's vote; the remaining components are used to prove consistency between ctxt and $\{(v_i)^{\mathtt{VC}^{\mathsf{A}}_{\mathsf{sk}}}\}_{i=1}^k$.

5. The Voting Server runs $\mathsf{ProcessVoteCheck}(\mathtt{EB}_{\mathsf{pk}}, \mathtt{VC}_{\mathsf{A}}, \mathsf{b})$ on input Alice's ballot, to decide whether the ballot is accepted. If ballot b passes the test and there is not an entry for $\mathtt{VC}_{\mathsf{A}}$ in the ballot box bb, then the server adds an entry $(\mathtt{VC}_{\mathsf{A}}, \mathsf{b})$ to bb and runs $\mathsf{CreateRC}(\mathsf{b}, \mathtt{C}_{\mathsf{sk}}, \mathsf{CMtable})$ to compute a set of short choice codes $\{\mathbf{sCC}_i\}_{i=1}^k$ that are sent back to Alice's voting device.

6. Alice's Voting Device displays to her the set of choice codes $\{\mathbf{sCC}_i\}_{i=1}^k$ received from the Voting Server. Alice then runs the human computable algorithm $\mathsf{AuditCodes}((\mathbf{sCC}_1, \ldots, \mathbf{sCC}_k), (\mathtt{sCC}^{\mathsf{A}}_1, \ldots, \mathtt{sCC}^{\mathsf{A}}_k))$, that checks Alice's expected return codes $\{(J_i, \mathtt{sCC}^{\mathsf{A}}_i)\}_{i=1}^k$ against the codes $\{\mathbf{sCC}_i\}_{i=1}^k$ displayed by the Voting Device.

7. If $\mathsf{AuditCodes}(\{\mathbf{sCC}_i\}_{i=1}^k, \{\mathtt{sCC}^{\mathsf{A}}_i\}_{i=1}^k)$ outputs true then Alice enters her Ballot Casting Key $\mathtt{BCK}^{\mathsf{A}}$ in the Voting Device, which computes and forwards to the Voting Server the value $(\mathtt{BCK}^{\mathsf{A}})^{\mathtt{VC}^{\mathsf{A}}_{\mathsf{sk}}}$.

8. After some checks, the Voting Server returns to the Voting Device a short Vote Cast Code $\mathbf{sVCC}$.

9. The Voting Device displays $\mathbf{sVCC}$ to Alice, who then checks if $\mathbf{sVCC} = \mathtt{sVCC}^{\mathsf{A}}$.

Once the election is closed, the tallying authority runs $\mathsf{Tally}(\mathtt{EB}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathsf{pk}}, \mathsf{bb}, \mathsf{cb})$ and outputs the result of the election.

Events

Our model includes a number of events. Events are simply annotations to make statements about which part of the protocol has been reached [5].

- $\mathsf{Confirmed}(\mathsf{init}_{\mathsf{A}}, J_1, \ldots, J_k)$ is reached (or holds) iff Alice has checked the short Choice Codes displayed to her by her voting device and they match the codes corresponding to her voting options as displayed in her voter's voting card $\mathtt{VC}_{\mathsf{A}}$. However, Alice has not yet finalised her vote.

- $\mathsf{HasVoted}(\mathsf{init}_{\mathsf{S}}, \mathtt{VC}_{\mathsf{A}}, \mathsf{b}, \mathtt{sVCC}^{\mathsf{A}}, S_{\mathtt{VCC}^{\mathsf{A}}})$ is reached when the server has processed and accepted Alice's ballot and Ballot Casting Key.

- $\mathsf{HappyUser}(\mathsf{init}_{\mathsf{A}}, J_1, \ldots, J_k)$ is reached when $\mathsf{Confirmed}(\mathsf{init}_{\mathsf{A}}, J_1, \ldots, J_k) = \mathsf{true}$ and Alice has checked and accepted the short Vote Cast Code displayed by her voting device. Thus, Alice has finalised her vote.

- $\mathsf{InsertBB}(\mathtt{VC}_{\mathsf{id}}, \text{bitstring})$ is reached when the Server registers an entry bitstring for voter id in the Ballot Box.

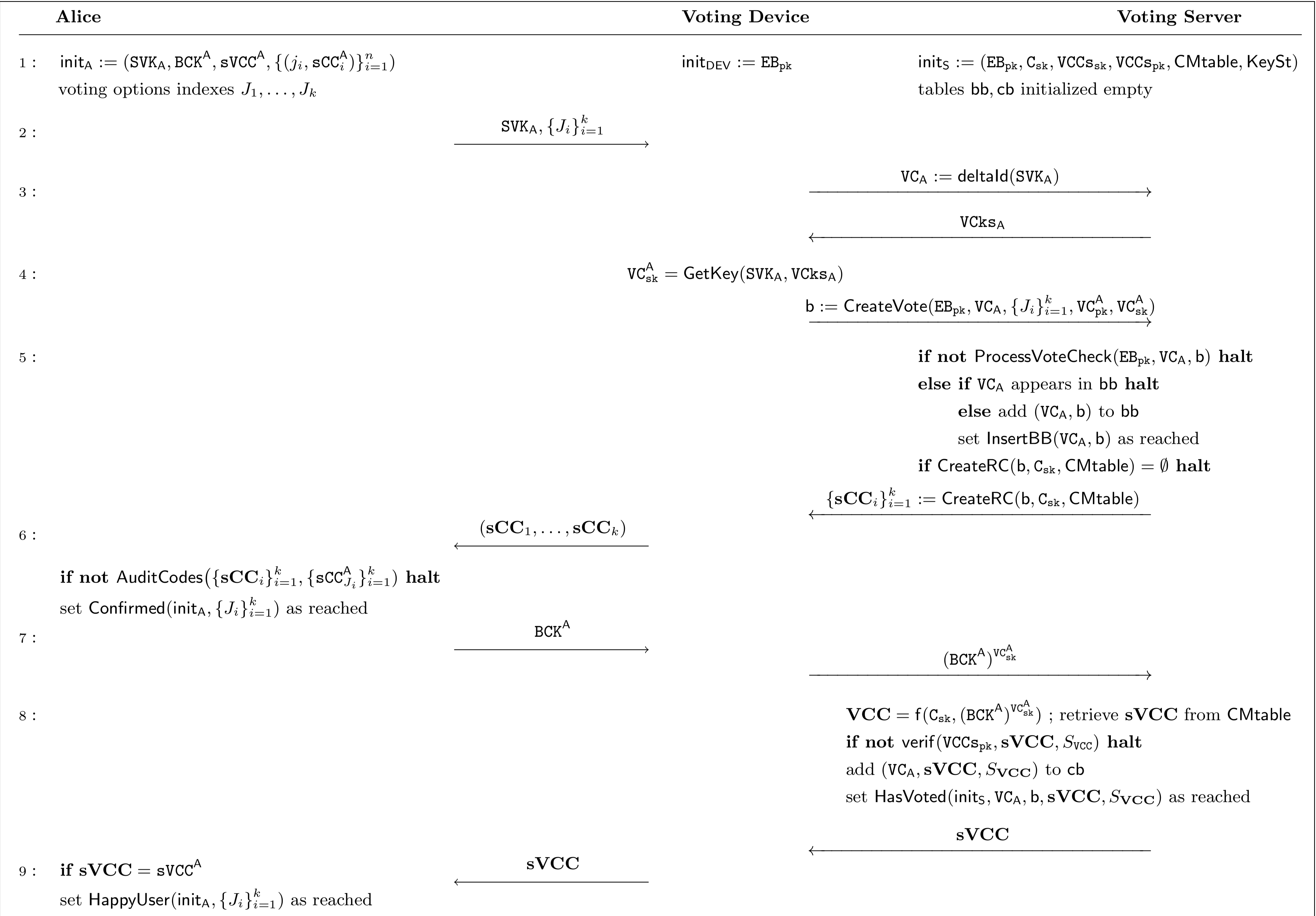

Figure 3.1: Message Sequence Chart of Swiss Online Voting Protocol

Chapter 4

Verifiability Properties for an infinite number of candidates

Let an election be of type (k, n) if the total number of options available in the election is n (i.e. $j_1, \ldots, j_n$) and to vote in the election a voter needs to choose exactly k of those choices different pairwise. We consider *two* different *types of voters*:

- honest voters with compromised 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, i.e.

- she enters in her device k pairwise different valid voting options $J_1, \ldots, J_k$;
- she confirms a ballot iff the codes returned to her by her voting device match the codes corresponding to her choices as displayed on her private audit data;
- she enters the ballot casting key $\mathsf{BCK}^{\mathsf{id}}$ as displayed on her private audit data after confirming her ballot.

Voter id's voting device $\mathsf{VD}_{\mathsf{id}}$ is honest (uncompromised, non-corrupted) iff its behaviour corresponds to the behaviour described in the protocol specification. $\mathsf{VD}_{\mathsf{id}}$ is a *compromised voting device* or dishonest/corrupted if it behaves otherwise.

Attacker model

In the next two models, an attacker is given the following capabilities:

- It can schedule an unbounded number of voters of any of the two types described above
- It can choose the choices $J_1, \ldots, J_k$ for every voter
- It controls all corrupted voting devices and corrupted voters

An *attacker does not*:

- Control the voting server nor the registrar nor uncorrupted voting devices
- Have access to honest voters private audit data

Our verifiability properties consider a number n infinite of voting options. Obviously, no election has an infinite number n of candidates. However, it is not difficult to see that if there is no attack against a given security property for $n = \infty$, then there is no attack for n' bounded (from an attack against n' bounded we can obtain an attack for an infinite n by only using the n' first options of the infinite voting options list). Thus showing the security for n infinite implies security for n finite. Our verifiability properties are still dependant of k, i.e. the number of voting choices that a voter casts in a election of type (k, n), as it is actually the case for the ballot privacy specification.

To ease the analysis as much as we could, we now consider *only one honest voter*, and her voting choices are constant and *not chosen by the adversary*. The adversary controls an unbounded number of other dishonest voters and the honest voter's voting device. The Voting Server is still honest, and runs an unbounded number of instances concurrently. We set the unique honest voter choices for the election to be publicly known constants $j_1, \ldots, j_k$.

Our verifiability properties must also ignore the traces where the Voting Server registered more than one ballot for the same voter id. The reason is that the behaviour whereby the voting server ends up storing more than one ballot per voter in bb cannot be avoided in ProVerif, since the tool does not allow us to synchronize concurrent threads accessing the server database, so we cannot prevent an attacker from submitting concurrent requests $(\mathsf{id}, \mathsf{b}_1)$ and $(\mathsf{id}, \mathsf{b}_2)$ with $\mathsf{b}_1 \neq \mathsf{b}_2$ to the server so that at the end both entries are recorded in bb.

We deal with those traces by stating that there might exist at most $k+1$ ballots in bb for the same voter id (maybe the same ballots, maybe not) such that:

- one of them contains the votes registered by the Voting Server for this voter id, and this ballot is ready to be sent to tallying
- for each intended voting option $j \in \{j_1, \ldots, j_k\}$ cast by the voter, there exists one ballot in bb that contains j, for this voter id

Consequently, if the real implementation of the protocol can guarantee that the server never stores two different ballots for the same id, then necessarily all these $k+1$ ballots are equal and thus, the ballot recorded by the server in bb contains the voter's intended voting options $j_1, \ldots, j_k$. Therefore, the only difference between the choices contained in the ballot as stored by the server and the voter's intentions $j_1, \ldots, j_k$ consist on the order in which these choices are presented in the ballot, which is not relevant.

In the following our verifiability properties are formally stated.

Verifiability properties

Property Cast-as-Intended

Formally, *Cast-as-Intended* is captured by asserting the following property for every trace at any moment and for every *honest* voter id:

$$\begin{aligned}
&\mathsf{HasVoted}(\mathsf{init_S}, \mathsf{VC_{id}}, \mathsf{b}, \mathsf{sVCC^{id}}, S_{\mathsf{VCC^{id}}}) \text{ is reached} \Longrightarrow \mathsf{Confirmed}(\mathsf{init_{id}}, j_1, \ldots, j_k) \text{ is reached}\\
&\quad \text{AND } \mathsf{b} = (\mathsf{ctxt}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{ec}, \mathsf{VC^{id}_{pk}}, \mathsf{P})\\
&\quad \text{AND } \mathsf{ctxt} = \mathsf{Enc}(\mathsf{pke}, \phi(v_1, \ldots, v_k), \mathsf{r})\\
&\quad \text{AND } \{v_1, \ldots, v_k\} = \{\mathsf{v}(J^{\mathsf{id}}_1), \ldots, \mathsf{v}(J^{\mathsf{id}}_k)\}\\
&\quad \text{AND } (\mathsf{id}, \mathsf{b}) \in \mathsf{bb} \text{ AND } \forall i \in 1..k,\ (\mathsf{id}, \mathsf{b}_i) \in \mathsf{bb} \text{ AND } j_i \text{ is in } \mathsf{b}_i
\end{aligned} \tag{4.1}$$

The first part is read as: if for any honest voter id the event $\mathsf{HasVoted}(\mathsf{init_S}, \mathsf{VC_{id}}, \mathsf{b}, \mathsf{sVCC^{id}}, S_{\mathsf{VCC^{id}}})$ is reached, then there exists a ballot $\mathsf{b} = (\mathsf{ctxt}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{ec}, \mathsf{VC^{id}_{pk}}, \mathsf{P})$ stored in the ballot box bb on behalf of voter id, such that ctxt contains the voting options $J^{\mathsf{id}}_1, \ldots, J^{\mathsf{id}}_k$. A link needs to established between $j_1, \ldots, j_k$ (i.e. the voter id intended voting choices) and $J^{\mathsf{id}}_1, \ldots, J^{\mathsf{id}}_k$ (the voting choices contained in the entry $(\cdot, \mathsf{b})$ recorded by the server). This link is given in Line 4.1, where it is ensured that:

```
309  (* Cast-as-Intended Property : The server registers a vote for the *honest* voter if and
       only if the voter confirmed his vote to him. *)
310  query Id:agent_id, Csk:symmetric_key, {W$i:nat|$i=1..$k}, EBpk:public_ekey, VCCssk:private
       _skey, EC:nat, P:bitstring, R:nat,
311  sVCCid:bitstring, C:bitstring, S_VCC:bitstring, {J$i:nat|$i=1..$k}, SVK:password,
312  { {J$iJ$j:nat, W$iW$j:nat|$i=1..$k|, }, R$j:nat, EC$j:nat, P$j:bitstring|$j=1..$k|,\n
       };
313  event(HasVoted(ServData(EBpk,Csk,VCCssk), deltaId(honest(SVK)), (C,{W$i|$i=1..$k},EC,pube(
       ske(deltaId(honest(SVK)))),P), sVCCid, S_VCC))
314  ==>  event(Confirmed(AliceData(honest(SVK),Csk), {ja$i|$i=1..$k}))  && Id = deltaId(honest
       (SVK))                       (* alias VCid *)
315  && C = (Enc_c1(R),Enc_c2(EBpk,phi({v(J$i )|$i=1..$k}),R))
316  (* Linking J1..Jk to j1..jk  *)
317
318  && event(InsertBB(Id, ((Enc_c1(R  ),Enc_c2(EBpk,phi({v(J$i   )|$i=1..$k}),R  )),{W$i   |$i
       =1..$k},EC  ,pube(ske(Id)),P  )))
319  {     && event(InsertBB(Id, ((Enc_c1(R$j),Enc_c2(EBpk,phi({v(J$iJ$j)|$i=1..$k}),R$j)),{W$
       iW$j|$i=1..$k},EC$j,pube(ske(Id)),P$j))) |$j=1..$k|\n}
320  && {({J$iJ$j = ja$j|$i=1..$k| || })|$j=1..$k| && }
321
```

Figure 4.1: Extended ProVerif process for Cast-as-Intended from file /specs/study_v13-REA.pve with comments inline

- the ballot b has been stored in bb as $(\mathsf{id}, \mathsf{b})$
- each $j_i \in j_1, \ldots, j_k$ exists inside a ballot b_i stored in bb with id

As mentioned previously, if a real implementation of the protocol can guarantee some single-vote property ensuring that two different ballots cannot coexist in bb with the same id, then the $k+1$ ballots $\mathsf{b}, \mathsf{b}_1, \ldots, \mathsf{b}_k$ collapse to a single ballot and thus, $\{j_1, \ldots, j_k\} \subseteq \{J_1^{\mathsf{id}}, \ldots, J_k^{\mathsf{id}}\}$. Since $j_1, \ldots, j_k$ are all different, it follows that $j_1, \ldots, j_k$ and $J_1^{\mathsf{id}}, \ldots, J_k^{\mathsf{id}}$ are equal modulo re-ordering. The ProVerif code corresponding to this is given in Figure 4.1, and can be found in the file /specs/study_v13-REA.pve

Property Tallied-as-Cast

Formally, *Tallied-as-Cast* is captured by asserting the following property for every trace at any moment and for every honest voter id:

$$\begin{aligned}
\mathsf{HappyUser}(\mathsf{init}_{\mathsf{id}}, j_1, \ldots, j_k) \text{ is reached} \implies & \mathsf{HasVoted}(\mathsf{init}_{\mathsf{S}}, \mathtt{VC}_{\mathsf{id}}, \mathsf{b}, \mathtt{sVCC}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}}) \text{ is reached} \\
& \text{AND } \mathsf{b} = (\mathsf{ctxt}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{ec}, \mathtt{VC}^{\mathsf{id}}_{\mathsf{pk}}, \mathsf{P}) \\
& \text{AND } \mathsf{ctxt} = \mathsf{Enc}(\mathsf{pke}, \phi(v_1, \ldots, v_k), \mathsf{r}) \\
& \text{AND } \{v_1, \ldots, v_k\} = \{\mathsf{v}(J_1^{\mathsf{id}}), \ldots, \mathsf{v}(J_k^{\mathsf{id}})\} \\
& \text{AND } \mathsf{verifP}(\mathsf{EB}_{\mathsf{pk}}, \mathtt{VC}^{\mathsf{id}}_{\mathsf{pk}}, \mathsf{ctxt}, \mathsf{ec}, \mathsf{w}_1, \ldots, \mathsf{w}_k, \mathsf{P}) \\
& \text{AND } \mathsf{verif}(\mathtt{VCCs}_{\mathsf{pk}}, \mathtt{sVCC}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}}) \\
& \text{AND } (\mathsf{id}, \mathsf{b}) \in \mathsf{bb} \text{ AND } \forall i \in 1..k,\ (\mathsf{id}, \mathsf{b}_i) \in \mathsf{bb} \text{ AND } j_i \text{ is in } \mathsf{b}_i
\end{aligned}$$

Notice that the Tallied-as-Cast property guarantees that if $\mathsf{HappyUser}(\mathsf{init}_{\mathsf{id}}, j_1^{\mathsf{id}}, \ldots, j_k^{\mathsf{id}})$ is reached then $\mathsf{HasVoted}(\mathsf{init}_{\mathsf{S}}, \mathtt{VC}_{\mathsf{id}}, \mathsf{b}, \mathtt{sVCC}^{\mathsf{id}}, S_{\mathtt{VCC}^{\mathsf{id}}})$ is reached. Thus by using the discussion form the Cast-as-Intended property, it follows that a ballot b has been stored on behalf of id, and that b that validates all the expected properties (i.e. is accepted by verifP) and contains voting options $J_1^{\mathsf{id}}, \ldots, J_k^{\mathsf{id}}$. No link $j_1, \ldots, j_k$ with $J_1^{\mathsf{id}}, \ldots, J_k^{\mathsf{id}}$ is yet established. Finally, the ingredient providing this link is actually the same as in Cast-as-Intended above, and thus ensures that if a real implementation validates a single-vote property, then $j_1, \ldots, j_k$ and $J_1^{\mathsf{id}}, \ldots, J_k^{\mathsf{id}}$ are equal, modulo reordering. The ProVerif code corresponding to this is given in Figure 4.2, and can be found in the file /specs/study_v13-REA.pve

```
328   (* Tallied-as-Cast Property : The voter finishes and is happy iff the server has his vote
        (same choices) and the Tally will accept it.*)
329   query Id:agent_id, Csk:symmetric_key, {W$i:nat|$i=1..$k}, EBpk:public_ekey, VCCssk:private
        _skey, EC:nat, P:bitstring, R:nat,
330   sVCCid:bitstring, C:bitstring, S_VCC:bitstring, {J$i:nat|$i=1..$k}, SVK:password,
331   { {J$iJ$j:nat, W$iW$j:nat|$i=1..$k|, }, R$j:nat, EC$j:nat, P$j:bitstring|$j=1..$k|,\n
        };
332   event(HappyUser(AliceData(SVK,Csk), {ja$i|$i=1..$k}))
333   ==>  event(HasVoted(ServData(EBpk,Csk,VCCssk), Id, (C, {W$i|$i=1..$k}, EC, pube(ske(Id)),
        P), sVCCid, S_VCC))  && Id = deltaId(SVK)
334   (* The ballot is present in the table *)
335   (* The ballot is confirmed in the table *)
336   (* Runs VerifP from ProcessVoteCheck *)
337   && C = (Enc_c1(R),Enc_c2(EBpk,phi({v(J$i)|$i=1..$k}),R))
338   && {W$i=pCC(ske(Id),v(J$i))|$i=1..$k| && }
339   && P = ZKP(EBpk,pube(ske(Id)),C,{W$i|$i=1..$k}, R,ske(Id))
340
341   (* Runs Verify to check the confirmation *)
342   && S_VCC = Sign(VCCssk,sVCCid)
343   (* Linking J1..Jk to j1..jk  *)
344
345   && event(InsertBB(Id, ((Enc_c1(R  ),Enc_c2(EBpk,phi({v(J$i   )|$i=1..$k}),R  )),{W$i   |$i
        =1..$k},EC  ,pube(ske(Id)),P  )))
346   {   && event(InsertBB(Id, ((Enc_c1(R$j),Enc_c2(EBpk,phi({v(J$iJ$j)|$i=1..$k}),R$j)),{W$iW$
        j|$i=1..$k},EC$j,pube(ske(Id)),P$j))) |$j=1..$k|\n}
347   && {({J$iJ$j = ja$j|$i=1..$k| || })|$j=1..$k| && }
348
```

Figure 4.2: Extended ProVerif process for Tallied-as-Cast from file `/specs/study_v13-REA.pve` with comments inline

Main process

Given that our analysis with an unbounded number of voting options involves one single honest voter, the main process for this specification will:

1. give the election parameters to the attacker
2. define an implicit pseudonym `ida` for the honest voter by assigning password `svka`, and start its subprocess A with the initialization data from AliceData(·) and its pre-established voting choices $j_1, \ldots, j_k$
3. define and start concurrently an unbounded number of dishonest voters by giving their initialization data to the intruder
4. define and start an unbounded number of honest servers, all sharing the same initialization data obtained from ServData(·).

The corresponding extended ProVerif code (i.e. dependant on the value of k) is given in Figure 4.3, and can be found in the file `/specs/study_v13-REA.pve`

```
356  (* 8. Main process  -- initiates the election *)
357
358  process
359  (* Public output from Setup(..)  --  Gives the election's parameters to the Intruder. *)
360  out(c, ebpk); out(c, vccspk);
361
362  (* Gives the honest voter's public data to the Intruder *)
363  out(c, deltaId(honest(svka))); out(c,pke(deltaId(honest(svka))));
364  out(c, deltaId(honest(svkb))); out(c,pke(deltaId(honest(svkb))));
365
366  (* Roles for honest voter *)
367  Alice( c, AliceData(honest(svka),csk),{ja$i|$i=1..$k} )
368
369  (* Dishonest voter(s) : As many as possible for Reachability *)
370  | !(new svki:password; out(c,svki); out(c,AliceData(svki,csk)))
371
372  (* Bulletin Board (ie. Server) : is Honest for Reachability *)
373  | !Serv(c,ServData(ebpk,csk,vccssk),c)
374
```

Figure 4.3: Main process for both Cast-as-Intended and Tallied-as-Cast in `/specs/study_v13-REA.pve`

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 [6, 3, 1], than what can be learned from the election result alone. 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:

$$V_{\mathsf{A}}(0) \mid V_{\mathsf{B}}(1) \;\approx\; V_{\mathsf{A}}(1) \mid V_{\mathsf{B}}(0)$$

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 voter's choices is k of. 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:

- It can schedule an unbounded number of voters of any of the two types described above
- It controls corrupted voting devices and corrupted voters
- It controls the voting server

An *attacker*:

- Does not control the tallying authority nor the registrar nor uncorrupted voting devices
- Cannot launch Tally unless all honest ballots are received
- Cannot have access to honest voters private audit data
- Cannot choose the choices $J_1, \ldots, J_k$ for honest voters

Ballot Privacy Property

Thanks to the recent work [1], 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 [1, 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.

$$V_{\mathsf{A}}(J_1^{\mathsf{A}}, \ldots, J_k^{\mathsf{A}}) \mid V_{\mathsf{B}}(J_1^{\mathsf{B}}, \ldots, J_k^{\mathsf{B}}) \approx V_{\mathsf{A}}(J_1^{\mathsf{B}}, \ldots, J_k^{\mathsf{B}}) \mid V_{\mathsf{B}}(J_1^{\mathsf{A}}, \ldots, J_k^{\mathsf{A}})$$

where $\{J_1^{\mathsf{A}}, \ldots, J_k^{\mathsf{A}}\}, \{J_1^{\mathsf{B}}, \ldots, J_k^{\mathsf{B}}\} \subseteq \{j_1, \ldots, j_n\}$, provided that the mixing process is only applied to honest ballots (i.e. in the election with 3 voters the ciphertext $(c_1^{\mathsf{C}}, c_2^{\mathsf{C}})$ obtained from $\mathsf{b_C}$ is not entered into the mixing process).

The above property is captured by the ProVerif code given in Figure 5.1 and to be found in the file `/specs/study_v13-OBS_k=1.pv` as given in Appendix B.

In contrast to verifiability, for ballot privacy 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, ebpk); out(c, vccspk);
```

This command is run to give the election parameters to the attacker.

```
out(c, deltaId(honest(svka))); out(c,pke(deltaId(honest(svka))));
out(c, deltaId(honest(svka))); out(c,pke(deltaId(honest(svka))));
```

These lines initiate honest voters with pseudonyms $\mathsf{id_A}$ and $\mathsf{id_B}$. In the voters' respective public channels it is written that the voters with corresponding Voting Card ID's $\mathtt{VC_A}$ and $\mathtt{VC_B}$ are honest.

```
Alice_Cmp(AliceData(honest(svka),csk),choice[ja1,jb1],c,ebpk)
| Alice_Cmp(AliceData(honest(svkb),csk),choice[jb1,ja1],c,ebpk)
```

These lines set the indistinghishability property

$$V_{\mathsf{A}}(J_1^{\mathsf{A}}) \mid V_{\mathsf{B}}(J_1^{\mathsf{B}}) \approx V_{\mathsf{A}}(J_1^{\mathsf{B}}) \mid V_{\mathsf{B}}(J_1^{\mathsf{A}})$$

discussed above.

```
| (new svki:password; out(c,svki); out(c,AliceData(svki,csk)))
```

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

```
| out(c,ServData(ebpk,csk,vccssk))
```

Likewise this gives the attacker total control of the Voting Server.

```
| Tally(c,ebsk,vccspk,deltaId(honest(svka)),deltaId(honest(svkb)))
```

The tally phase starts once all the previous processes have stopped.

To complete the verification of ballot privacy for our symbolic model of the Swiss OV protocol, we additionally need to verify ballot privacy on the file

- `/specs/study_v13-OBS_HH_k=1.pv`

```
333  (* 8. Main process  -- initiates the election *)
334
335  process
336  (* Public output from Setup(..)  --  Gives the election's parameters to the Intruder. *)
337  out(c, ebpk); out(c, vccspk);
338
339  (* Gives the honest voter's public data to the Intruder *)
340  out(c, deltaId(honest(svka))); out(c,pke(deltaId(honest(svka))));
341  out(c, deltaId(honest(svkb))); out(c,pke(deltaId(honest(svkb))));
342
343  (* Roles for honest voters  --  two for Observational Equivalence. *)
344  Alice_Cmp(AliceData(honest(svka),csk),choice[ja1,jb1],c,ebpk)
345  | Alice_Cmp(AliceData(honest(svkb),csk),choice[jb1,ja1],c,ebpk)
346  (* Alice_Cmp(AliceData(honest(svka),csk),       ja1     ,c,ebpk)*)   (* For Runnability
       check *)
347  (*|Alice_Cmp(AliceData(honest(svkb),csk),       jb1     ,c,ebpk)*)   (* Same
         *)
348
349  (* Dishonest voter(s) : only one for Obervational equivalence. *)
350  |  (new svki:password; out(c,svki); out(c,AliceData(svki,csk)))
351
352  (* Bulletin Board (ie. Server) :  Dishonest for Observational Equivalence. *)
353  |  out(c,ServData(ebpk,csk,vccssk))
354
355  (* The Tally phase  --  only after the honest voters have voted. *)
356  | Tally(c,ebsk,vccspk,deltaId(honest(svka)),deltaId(honest(svkb)))
357
```

Figure 5.1: ProVerif main process for Ballot Privacy from file /specs/study_v13-OBS.k=1.pv

to cover the case where there is no dishonest voter (since the cases with two honest voters and one dishonest voter HDH and DHH are straightforwardly equivalent to the case HHD which is captured in the file study_v13-OBS.k=1.pv).

The difference between the files /specs/study_v13-OBS_HH_k=1.pv and study_v13-OBS.k=1.pv can be seen in how the algorithm $\mathsf{Tally}(\mathsf{EB}_{\mathtt{sk}}, \mathsf{VCCs}_{\mathtt{pk}}, \mathsf{bb}, \mathsf{cb})$ is defined therein. In the case HH, we find

```
(*  in(CTally, (VCid3:agent_id,B3:bitstring,sVCC3:bitstring,S_VCC3:bitstring)); *)
```

has been commented out and therefore only the ballots of the honest voters Alice and Bob are tallied.

```
233  (* Tally -- the election tally *)
234  free mix:channel [private].
235  let Tally(CTally:channel,Ske:private_key,Id1:agent_id,Id2:agent_id) =
236  in(CTally, (=Id1,B1:bitstring)); in(CTally, (=Id2,B2:bitstring)); in(CTally, (Id3:agent_id
       ,B3:bitstring));
237  let Ok1 = ProcessBallotCheck(pub(Ske),Id1,B1) in
238  let Ok2 = ProcessBallotCheck(pub(Ske),Id2,B2) in
239  let Ok3 = ProcessBallotCheck(pub(Ske),Id3,B3) in
240  if Id1 <> Id2 && Id1 <> Id3 && Id2 <> Id3 then
241  let (C1:bitstring, {W1$i:nat|$i=1..$k}, EC1:nat, =pk(Id1), P1:bitstring) = B1 in
242  let (C2:bitstring, {W2$i:nat|$i=1..$k}, EC2:nat, =pk(Id2), P2:bitstring) = B2 in
243  let (C3:bitstring, {W3$i:nat|$i=1..$k}, EC3:nat, =pk(Id3), P3:bitstring) = B3 in
244  (* MixNet Modeling -- The data to be mixed is sent concurrently on a specific channel *)
245  out(mix, choice[C1,C2]) | out(mix, choice[C2,C1]) | in(mix, MC1:bitstring); in(mix, MC2:
       bitstring);
246
247  (* Decrypting -- The mixed ciphers are opened and decrypted. *)
248  let phi({v(J1$i)|$i=1..$k}) = dec(Ske,MC1) in
249  let phi({v(J2$i)|$i=1..$k}) = dec(Ske,MC2) in
250  let phi({v(J3$i)|$i=1..$k}) = dec(Ske, C3) in
251  (* Publishing -- The intruder receices the election's result. *)
252
253  out(c, ({J1$i|$i=1..$k},{J2$i|$i=1..$k},{J3$i|$i=1..$k}))
```

Figure 5.2: ProVerif extended code for Tally from file /specs/study_v13-OBS.k=1.pv

Chapter 6

Results

A summary of the verified security properties can be seen in Table 6.1. Regarding verifiability, ProVerif has automatically verified that the properties *Cast-as-Intended* and *Tallied-as-Cast* defined in Chapter 4 *hold* for elections of type (k, ∞), for $k = 1, \ldots, 4$ and an unbounded number of voters, against an attacker controlling every voter except for one honest voter. Regarding ballot privacy, 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 [1, Theorem 1], imply ballot privacy for elections of type $(1, n)$ for n unbounded, and an unbounded number of voters m.

Property	k #voter choices	n #voting options	m #voters
Cast-as-Intended	$1, \ldots, 4$	unbounded	unbounded
Tallied-as-Cast	$1, \ldots, 4$	unbounded	unbounded
Ballot Privacy	1 1	unbounded unbounded	2,3 unbounded†

Table 6.1: Summary of Verified Security Properties of Our Symbolic Model of Swiss Online Voting Protocol. † obtained by virtue of [1, Theorem 1] plus the ProVerif verification of Ballot Privacy for number of voters $m = 2, 3$ and number of candidates n unbounded.

The corresponding ProVerif files containing the source code for the automatic verification of the security properties are to be found in the folder `analysis/`. For instance the file `analysis/study_v13-REA.k=4.pv` contains the ProVerif source code for verifying the security properties of our model for $k = 4$, and it is obtained by running the command

```
analysis$ ./run study_v13-REA.pve k=4
```

The file `specs/study_v13.pve` serves as a 'root' model file from which the verifiability and privacy models can be generated. For instance,

- `analysis$ ./run study_v13.pve OBS k=1` generates the file `study_v13-OBS.k=1.pv` for Observational Equivalence
- `analysis$ ./run study_v13.pve OBS HH k=1` generates the file `study_v13-OBS_HH_k=1.pv`
- `analysis$ ./run study_v13.pve REA k=3` generates the file `study_v13-REA.k=3.pv`

Chapter 7

Lessons Learned and Limitations

Our model places some constraints on any implementation of the Swiss Online Voting protocol. This translates into a set of requirements as follows.

Requirements to be Satisfied by any Implementation

Requirement 1

In an election of type (k, n) the voters enter *pairwise different* voting options.

Requirement 2

In an election of type (k, n) the voters enter *exactly* k voting options.

Requirement 3

Any implementation needs to ensure that the voting server does not register in the ballot box two different ballots for the same voter.

Attacks Found if the Requirements above are Not Satisfied

In discussion with the vendor, it has become apparent that Requirement 1 and Requirement 2 were already met by the design and implementation of the Swiss OV protocol. The vendor has also put in place implementation measures to satisfy Requirement 3. Therefore, *we are not claiming the warnings below apply to the current implementation of the Swiss OV protocol*, as the evidence given to us indicates that it is likely the current implementation meets those requirements and hence it is not vulnerable to the attacks described next (but obviously we cannot guarantee that).

If an implementation of the Swiss OV protocol fails to ensure some of the previous requirements, then either there are attacks against the cast-intended verifiability property or our analysis might not guarantee any security property. We enumerate next a set of warnings that to take into account if some of the above requirements is not met by an instantiation of the voting protocol.

Warning 1

If Requirement 3 is not met ProVerif exhibits attacks against verifiability. Such an attack occurs when the corrupted voting device submits ballots $(\mathsf{VC}_{\mathsf{id}}, \mathsf{b}_1)$ and $(\mathsf{VC}_{\mathsf{id}}, \mathsf{b}_2)$, with $\mathsf{b}_1 \neq \mathsf{b}_2$, concurrently on different threads to the server so that the server:

1. first checks that no ballot already exists for that $\mathsf{VC}_{\mathsf{id}}$, for both ballots in both threads; then

2. accepts and adds the two new ballots concurrently, both after the two tests. This behaviour cannot be avoided in the ProVerif specification, because it would rely on some form of thread synchronization to ensure that no server thread can move between the moment when one thread tests the existence of a ballot in the database, and the moment when it consequently adds a new ballot to the database.

Likewise storing different ballots in the ballot box for the same $\mathsf{VC_{id}}$ and then choosing only one of them to be tallied does not prevent this attack or variations thereof.

Warning 2

If Requirement 1 is not met, then ProVerif finds an attack. Indeed, let us assume a voter that enters two equal voting options a, a, so the voter is expecting to see twice the same choice codes $\mathtt{sCC}_a, \mathtt{sCC}_a$. The intruder votes a, b (instead of a, a) and when it receives from the server the codes $\mathtt{sCC}_a, \mathtt{sCC}_b$, it displays the duplicate codes $\mathtt{sCC}_a, \mathtt{sCC}_a$ to the voter. As a result, the voter confirms her vote while she is actually voting for an another unexpected voting option.

Warning 3

If Requirement 2 is not met, then ProVerif finds similarly an attack. Indeed, let us assume an election of type (k, n) and a voter that enters k' voting options $J_1, \ldots, J_{k'}$ with $k' < k$. Then the voter is expecting to see k' choice codes $\mathtt{sCC}_1, \ldots, \mathtt{sCC}_{k'}$. Now, the intruder votes $J_1, \ldots, J_{k'}, J_{k'+1}, \ldots, J_k$, and when it receives from the server the codes $\mathtt{sCC}_1, \ldots, \mathtt{sCC}_k$, it only displays the codes $\mathtt{sCC}_1, \ldots, \mathtt{sCC}_{k'}$ to make the voter confirm her vote while she is voting for a few extra unexpected voting options.

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 specification of $\mathsf{Tally}(\cdot)$ that we made to accommodate our symbolic models to ProVerif and with the motivation of improving the running time of the verification of the security properties::

- In our abstraction of the algorithm $\mathsf{Tally}(\cdot)$ (cf. Chapter 3) we have deliberately omitted that proofs of correct mixing and proofs of correct decryption are part of the output. The reason is that proving that a given mixing protocol and a given proof system for correct decryption were outside of the scope of our analysis. We do not expect this limitation to hinder the validity of our symbolic verification of ballot privacy, were these primitives be included in our model. Intuitively, those proof systems are *zero-knowledge*, so they are guaranteed not to leak information on the voting choices. On the other hand, even in demanding computational cryptographic frameworks such as Universal Composability those primitives can be securely composed with generic tallying algorithms [8]. However, we have not formally verified this in our symbolic model.

- Likewise, some of the verifications done inside $\mathsf{Tally}(\cdot)$ in the original description of the protocol have been moved from our abstraction of $\mathsf{Tally}(\cdot)$ and incorporated inside the Tallied-as-Cast properties.

- Our tally process does not include the dishonest voters' ballots into the mixing. In contrast, in the original protocol description, all ballots in bb are entered into the mixnet. This however does not

represent a limitation of our model, as by doing so we give more information to the attacker, compared to a real instantiation of the protocol, so we are actually verifying a stronger property.

Lessons Learned

From the discussion above, we derive the following recommendations:

1. The voter must be made aware that sending the same choice more than once (i.e. repeating a choice) not only probably makes her vote invalid following the election rules but, most of all, makes it vulnerable to an attack where the intruder uses the duplicated choice to make her vote for an other unexpected voting option the voter did not agree upon.

2. A similar problem occurs when the voter votes for less voting options than the actual number of voting options defined by the election. For that reason, if the election allows a voter to abstain for several questions, multiple different blank voting options must be provided to voters, each corresponding to each question that allows abstention. Those blank voting options shall have different individual return codes, and voters shall be advised that they need to check a return code for each blank option.

3. The voting server must implement some form of thread synchronization to guarantee that two different ballots will never be accepted for the same id. The tests implemented by the Voting Server or during tallying in the specification we were given are not enough to protect against this attack.

Bibliography

[1] Myrto Arapinis, Véronique Cortier, and Steve Kremer. When are three voters enough for privacy properties? In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*, volume 9879 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2016.

[2] David Basin and Srdjan Capkun. Review of electronic voting protocol models and proofs. 2017. Draft v1.0.

[3] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. SoK: A comprehensive analysis of game-based ballot privacy definitions. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 499–516. IEEE Computer Society, 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] 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.

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

[7] David Galindo. Analysis of cast-as-intended verifiability and ballot privacy properties for Scytl's Swiss On-line Voting protocol using ProVerif, November 2016.

[8] Shahram Khazaei, Tal Moran, and Douglas Wikström. A mix-net from any CCA2 secure cryptosystem. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658, pages 607–625. Springer, 2012.

[9] Scytl Research. Swiss On-line Voting Protocol. 2016. Manuscript.

[10] Scytl Research. Swiss On-line Voting Protocol. March 2017. Revision 1.2. Manuscript.

Appendix A

ProVerif source file specs/spec_v13-REA.pve before expansion

```
1  (* Post-review: Unified ProVerif specification, version 11, for Scytl's Voting Protocol. All
      variants included. Tabulation size : 8 spaces. *)
2  (* Reachability ........... : For Cast as Intended and Tallied as Cast properties; from
     study_v13.pve with the 'REA' option.              *)
3  (* Observational Equivalence : For Ballot Provivacy; 2 honest + 1 or 0 dishonest in the
     challenge; [..] with the 'OBS' option w/ or w/o HH. *)
4  (* Syntactical remark ...... : To avoid confusions, the Vote V of the voter is named here
     the Ballot B of the voter (it uses encryption).   *)
5
6  set ignoreTypes=true.
7
8  (* 1. Objects and Types *)
9
10 type   agent_id.      fun t_agent_id(agent_id)       :    bitstring   [data,typeConverter].
      (* The type for any IDs like e.g. voting card. *)
11 type   password.      fun t_password(password)       :    bitstring   [data,typeConverter].
      (* The type for user passwords.                *)
12 type   nat     .      fun t_nat(nat)                 :    bitstring   [data,typeConverter].
      (* The type for natural numbers.               *)
13 free   c          : channel              .    (* Public Channel for communications with the
     intruder   # public                 *)
14 fun    v(nat)                  : nat          [data]   .    (* The function from voting
     choices to voting options    # public ,    invertible *)
15 table  bb(agent_id,bitstring)                          .    (* The Ballot Box, storing the
     ballots for each agent    # private               *)
16 table  cb(agent_id,bitstring,bitstring)                .    (* The Confirm. Box, storing VCC/S
     _VCC for each agent id # private                *)
17 free   svka, svkb              : password              .    (* Defines two SVK for two honest
     voters Alice and Bob   # public                *)
18 free   {ja$i,jb$i|$i=1..$k}    : nat                   .    (* Voting choices for the Honest
     voters -- REA or OBS    # public                *)
19 free   {a$i|$i=1..$k}          : nat         .    (* MODELING : an indexed set of nat atoms,
     for disquality tests  -- only in Test. *)
20
21
22
23 (* 2. Intruder capabilities and functions extracted from the CryptoPaper. *)
24
25 (* Encryption scheme  --  from CryptoPaper, page 4  --  expected to be ElGamal  --  Key
     pairs generated through Gen_e, which is implicit.   *)
26 type   private_ekey.  fun t_private_ekey(private_ekey):   bitstring   [data,typeConverter].
      (* The type for private keys + type converter. *)
```

```
27 type   public_ekey .  fun t_public_ekey(public_ekey)  :   bitstring   [data,typeConverter].
      (* The type for public keys  + type converter. *)
28 fun    ske(agent_id)          : private_ekey [private].   (* The private enc. key associated
      to an agent_id        # private                 *)
29 fun    pube(private_ekey)     : public_ekey           .   (* The function to rebuild a
    public key from the private # public , noninvertible *)
30 letfun pke(Id:agent_id)       = pube(ske(Id))         .   (* The public  enc. key associated
      to an agent id        # public ,    invertible *)
31 fun    Enc_c1(                         nat) : nat        .   (* The c1 part of the asymmetric
    encryption function with explicit random number. *)
32 fun    Enc_c2(public_ekey,bitstring,nat) : bitstring  .   (* The c2 part of the asymmetric
    encryption function with explicit random number. *)
33 letfun Enc(Pk:public_ekey,M:bitstring,R:nat) = (Enc_c1(R),Enc_c2(Pk,M,R)) .              (*
      Encryption *)
34 reduc  forall Sk:private_ekey, M:bitstring, R:nat;    Dec(Sk,(Enc_c1(R),Enc_c2(pube(Sk),M,R)
    )) = M   .                  (* Decryption *)
35 reduc  forall Pk:public_ekey,  M:bitstring, R:nat;    VerifE(Pk,(Enc_c1(R),Enc_c2(Pk,M,R)))
       = true.                  (* Checks key *)
36 reduc  forall Id:agent_id;                            Get_Id(pube(ske(Id)))
       = Id  .                  (* Extract Id *)
37
38
39 (* Signature scheme  --  from CryptoPaper, page 4 & 5  --  expected to be the RSA
    Probabilistic Signature Scheme (PSS)  --  Gen_s implicit. *)
40 type   private_skey.  fun t_private_skey(private_skey):   bitstring   [data,typeConverter].
      (* The type for private keys + type converter. *)
41 type   public_skey .  fun t_public_skey(public_skey)  :   bitstring   [data,typeConverter].
      (* The type for public keys  + type converter. *)
42 fun    sks(agent_id)          : private_skey [private].   (* The private sig. key associated
      to an agent_id        # private                 *)
43 fun    pubs(private_skey)     : public_skey           .   (* The function to rebuild a
    public key from the private # public , noninvertible *)
44 letfun pks(Id:agent_id)       = pubs(sks(Id))         .   (* The public  sig. key associated
      to an agent id        # public ,    invertible *)
45 fun    Sign(private_skey,bitstring)   : bitstring     .   (* The digital signature function
    with explicit random number. (Signature)        *)
46 reduc  forall Sk:private_skey, M:bitstring;           Verify(pubs(Sk),M,Sign(Sk,M)) = true.
                    (* Verification *)
47 reduc  forall Sk:private_skey, M:bitstring;           Checks(pubs(Sk),Sign(Sk,M))   = M   .
              (* More expressive than Verify. *)
48 reduc  forall Sk:private_skey, M:bitstring;           Get_Message(Sign(Sk,M))       = M   .
              (* Worst case for modeling only *)
49
50
51 (* Modeling of the Non-Interactive Zero-Knowledge Proofs of Knowledge  -- from CryptoPaper,
    pages 5 & 6  -- Names changed for abstractions. *)
52 (* Both the 'Equality of discrete logarithms' (used in two variants) and the 'Knowledge of
    encryption exponent' are abstracted inside the   *)
53 (* ZKP and VerifP operators. Consequently, the c1~ and c2~, which are intermediate values,
    are not needed anymore and thus abstracted away. *)
54 (* Moreover, the 'Correct decryption' is not needed for this modeling w.r.t the expected
    security properties.                                    *)
55 fun    pCC(private_ekey,nat)       : nat              .   (* The built of partial Choice
    Codes (ski,vi) = vi^ski   # public , noninvertible *)
56 fun    tild(private_ekey,bitstring): nat              .   (* The built of c1~ and c2~
    abstracted from ZKP/VerifP   # public , noninvertible *)
57 fun    phi({nat|$i=1..$k})         : bitstring [data] .   (* The agregation function (prod)
    over a tuple of votes  # public ,    invertible *)
58 reduc  forall Sk:private_ekey, V:nat;                 Get_Vote1(pCC(Sk,V),Sk)      = V
     .        (* Worst case for modeling only *)
59 reduc  forall Sk:private_ekey, C:bitstring;           Get_Vote2(tild(Sk,C),Sk)     = C
     .        (* Worst case for modeling only *)
60 fun    ZKP(public_ekey, public_ekey, bitstring, {nat|$i=1..$k}, nat, private_ekey) :
    bitstring.       (* Modeling of the proof generation   *)
61 reduc  forall EBpk:public_ekey, VCidsk:private_ekey, {v$i:nat|$i=1..$k}, R:nat;
              (* Modeling of the proof verification *)
62        VerifP(EBpk, pube(VCidsk), (Enc_c1(R),Enc_c2(EBpk,phi({v$i|$i=1..$k}),R)), {pCC(
    VCidsk,v$i)|$i=1..$k},
```

```
63          ZKP(EBpk, pube(VCidsk), (Enc_c1(R),Enc_c2(EBpk,phi({v$i|$i=1..$k}),R)), {pCC(
    VCidsk,v$i)|$i=1..$k}, R,VCidsk)) = true.
64 (* Consequently to abstracting c1~ and c2~ away from ZKP/VerifP, we do not need anymore the
    4 variants of this reduction shown previously.  *)
65 (* Remaining limitation : the agregation function phi(..) is commutative, but modeling it
    generates far too many cases for ProVerif.        *)
66
67
68 (* Symmetric encryption scheme  --  from CryptoPaper Draft update, page 3  --  expected to
    be based on AES                                  *)
69 type   symmetric_ekey. fun t_symmetric_ekey(symmetric_ekey): bitstring [data,typeConverter].
     (* The type for symetric encryption keys.      *)
70 fun    Enc_s(symmetric_ekey,bitstring) : bitstring   .
     (* Encryption (symmetric key) *)
71 reduc  forall SKey:symmetric_ekey, M:bitstring;      Dec_s(SKey,Enc_s(SKey,M)) = M.
     (* Decyption  (symmetric key) *)
72
73
74 (* Key and IDs derivation scheme  --  from CryptoPaper Draft update, page 3  *)
75 (* free   IDseed              : bitstring            . *)  (* A public seed for IDs   --
    Not needed: value fixed  # public                 *)
76 (* free   KEYseed             : bitstring            . *)  (* A public seed for KEYs  --
    Not needed: value fixed  # public                 *)
77 fun    deltaId(password)      : agent_id             .     (* The delta function, for agents
     IDs                   # public                 *)
78 fun    deltaKey(password)     : symmetric_ekey       .     (* The delta function, for
    symmetric encryption keys    # public                *)
79 fun    honest(password)       : password [private]   .     (* MODELING : a function to
    separate honest and dishonest agents (only in Prop). *)
80
81
82 (* Keyed Pseudo-Random and hash functions  --  from cryptoPaper, pages 7 & 8  --  Note: Not
    the same keys as for symmetric encryption       *)
83 type   symmetric_key. fun t_symmetric_key(symmetric_key): bitstring   [data,typeConverter].
     (* The type for symetric keys for KPR f(...).  *)
84 fun    H(symmetric_ekey)      : bitstring            .    (* The hash function  --  from
    CryptoPaper page 5        # public , noninvertible *)
85 fun    f(symmetric_key,nat)   : symmetric_ekey       .    (* The keyed pseudo-random
    function f  --  page 7        # public , noninvertible *)
86
87
88
89 (* 3. Initialization sequence (done off-line in this modeling). *)
90
91 (* Setup(...) i.e. the initialisation step from the Registrar  --  done off-line in this
    modeling  --  from CryptoPaper page 9              *)
92 free   election             : agent_id             .    (* The election id - used to
    derive public/private keys  # public (sk is private) *)
93 free   signature            : agent_id             .    (* The signature id - used to
    derive public/private keys # public (sk is private) *)
94 letfun ebpk                 = pke(election)         .    (* The Electorial Board public
    key in pair (EBpk,EBsk)  # public                 *)
95 letfun ebsk                 = ske(election)         .    (* The Electorial Board private
    key in pair (EBpk,EBsk)  # private                *)
96 free   csk                  : symmetric_key [private] .    (* The Codes secret key 'Csk' (for
     f(...) function)      # private                *)
97 letfun vccspk               = pks(signature)         .    (* The Vote Cast Code Signer
    public  key in (VCCspk,VCCssk) # public              *)
98 letfun vccssk               = sks(signature)         .    (* The Vote Cast Code Signer
    private key in (VSSspk,VCCssk) # private             *)
99
100
101 (* Register(..,id,csk,vccssk) ie. the registration step for any new voter  --  done off-line
     through functions  --  from CryptoPaper page 9 *)
102 fun    bck(agent_id)                    : nat       [private].  (* The Ballot Casting Key '
    BCK^id'  of an agent id                 # private *)
103 fun    sCC(symmetric_key,agent_id,nat) : bitstring [private].  (* The short choice code 'sCC
    ^id_i' of an agent id + voting option # private *)
```

```
104 fun    sVCC(symmetric_key,agent_id)      : bitstring [private]. (* The short vote cast code '
    sVCC^id' of an agent id               # private *)
105 reduc  forall SVK:password; GetVCks(deltaId(SVK)) = Enc_s(deltaKey(SVK), t_private_ekey(ske(
    deltaId(SVK)))). (* CryptoPaperDraft upd.page 4 *)
106 fun    CM_table(symmetric_key,agent_id) : bitstring  .        (* The Codes Mapping Table ('CM
    _id', unbounded)  --  opened with readCC/VCC(). *)
107 reduc  forall Csk:symmetric_key, VCid:agent_id, J:nat;
                                        (* Enc_s(CC,sCC) *)
108        readCC(H(f(Csk, pCC(ske(VCid),v(J)  ))), CM_table(Csk,VCid)) = Enc_s(f(Csk, pCC(ske(
    VCid),v(J)  )), sCC(Csk,VCid,v(J))).
109 reduc  forall Csk:symmetric_key, VCid:agent_id;
                                      (* Enc_s(VCC,sVCC) *)
110        readVCC(H(f(Csk, pCC(ske(VCid),bck(VCid)))), CM_table(Csk,VCid)) = Enc_s(f(Csk, pCC(
    ske(VCid),bck(VCid))), sVCC(Csk,VCid)) .
111 (* For any voter id, generates VCidpk,VCidsk ........ : pke(id) and ske(id) and GetVCks(id)
        --  distributed in encrypted private data.  *)
112 (* For any voter id, chooses a ballot casting key ... : bck(id) as defined above;
        --  distributed in AliceData and Mapping.   *)
113 (* For any voter id, computes a list of Choice Codes CC(id,i) ........ : with f(..) and pCC
    (..) and v(i)      --  used to build the Mapping. *)
114 (* for any voter id, computes the Vote Cast Code  VCC(id) ........... : with f(..) and pCC
    (..) and bck(id)  --  used to build the Mapping. *)
115 (* for any voter id, chooses the short CC and short VCC ............. : with sCC(..) and
    sVCC(..) -- distributed in AliceData and Mapping. *)
116 (* For any voter id, computes the signature of the Vote Cast Code .... : Sign(vccssk,sVCC(id
    ))                --  distributed in ServData.   *)
117 (* For any voter id, stores a list of hashed choice codes {H(CC(id,i))}_i=1..inf ..... :
    through MakeRFList  --  distributed in ServData.   *)
118
119
120 (* Registration data for any voter id  --  AliceData(svk,csk) produced by the Registrar for
    Alice   --  opened with a reduction             *)
121 fun   AliceData(password,symmetric_key) : bitstring [private].
122 reduc forall Csk:symmetric_key, SVK:password, {J$i:nat|$i=1..$k};
123       GetAliceData(AliceData(SVK,Csk),{J$i|$i=1..$k}) = (SVK, bck(deltaId(SVK)), sVCC(Csk,
    deltaId(SVK)), {sCC(Csk,deltaId(SVK),v(J$i))|$i=1..$k}).
124 (*    Note: The set of sCC(id,v(J)) is a selection of sCC corresponding to the {J$i|$i=1..$k
    }, decided at Alice's initialisation.          *)
125
126
127 (* Registration data for the server (Bulletin Board)  --  ServData(pke,Csk,sks) produced by
    the Registrar  -- opened with a reduction       *)
128 fun   ServData(public_ekey,symmetric_key,private_skey) : bitstring [private].
129 reduc forall EBpk:public_ekey, Csk:symmetric_key, VCCssk:private_skey, SVK:password;
                       (* Note : VCid = deltaId(SVK) *)
130       GetServData(ServData(EBpk,Csk,VCCssk),deltaId(SVK)) = (EBpk,Csk,pubs(VCCssk), Enc_s(
    deltaKey(SVK),t_private_ekey(ske(deltaId(SVK)))),
131                                                               CM_table(Csk,deltaId(SVK)),
    Sign(VCCssk,sVCC(Csk,deltaId(SVK))) ).
132
133
134
135 (* 4. Algebraic properties and List of Events *)
136
137 (* Equational theory commented out  --  Kept for reference but this property is not
    supported by ProVerif. *)
138 (* equation forall {V$i:nat|$i=1..$k};  phi({V$i|$i=1..$k}) = phi({V$i|$i=1..$k 'mixed' }) .
                   *)              (* Commutativity *)
139
140 event Confirmed(bitstring, {nat|$i=1..$k}).                                     (* Issued by
    the voter when he/she confirms his/her vote.    *)
141 event HappyUser(bitstring, {nat|$i=1..$k}).                                     (* Issued by
    the voter when he/she terminates successfuly.   *)
142 event InsertBB( agent_id, bitstring).                                           (* Issued by
    the server when it adds someting in BB.         *)
143 event HasVoted( bitstring, agent_id, bitstring, bitstring, bitstring).         (* Issued by
    the server when it gets a voter's confirmation. *)
```

```
144 event NeverTrue.                                                              (* An event
        that is never activated, and thus, never true.    *)
145 event Results({nat|$i=1..$k},{nat|$i=1..$k},{nat|$i=1..$k}).                 (* Issued by
        the Tally when it publishes the results.        *)
146
147
148
149 (* 5. Methods and Agents processes *)
150
151 (* GetID(SVKid)  --  Computer generates the Voting Card ID  --  Directly replaced by deltaId
        (SVKid)  --  from CryptoPaper update page 4.    *)
152 (* letfun GetID(SVKid:password) = deltaId(SVKid). *)
153
154 (* GetKey(SVKid,VCksid)  --  Computer retreives the Verification Card private key from the
        keystore  --  from CryptoPaper update page 4.    *)
155 letfun GetKey(SVKid:password,VCksid:bitstring) =
156     let KSpwd = deltaKey(SVKid) in
157     let t_private_ekey(VCidsk:private_ekey) = Dec_s(deltaKey(SVKid),VCksid) in
158     VCidsk
159 .
160
161 (* CreateVote(EBpk,VCid,Vopt,VCidsk)  --  Computer creates the ballot  --  from CryptoPaper
        pages 9 & 10.                                    *)
162 (* NOTE : Change w.r.t. the CryptoPaper Update page 4 : EBpk added as 1st argument, because
        it is needed by this method.                     *)
163 letfun CreateVote(EBpk:public_ekey,VCid:agent_id,{J$i:nat|$i=1..$k},VCidpk:public_ekey,
        VCidsk:private_ekey) =
164     let V = phi({v(J$i)|$i=1..$k}) in new R:nat; let C = Enc(EBpk, V, R) in
165     let P = ZKP(EBpk,VCidpk,C,{pCC(VCidsk,v(J$i))|$i=1..$k}, R,VCidsk) in
166     (C, {pCC(VCidsk,v(J$i))|$i=1..$k}, tild(VCidsk,C), VCidpk, P)
167 .
168
169 (* ProcessVote(bb,VCid,B)  --  Server checks and processes the ballot  --  bb tested and
        filled outside  --  from CryptoPaper page 10.      *)
170 letfun ProcessVoteCheck(EBpk:public_ekey,VCid:agent_id,B:bitstring) =
171     let (C:bitstring, {W$i:nat|$i=1..$k}, EC:nat, =pke(VCid), P:bitstring) = B in
172     let Ok1 = VerifP(EBpk,pke(VCid),C,{W$i|$i=1..$k}, P) in
173 (*  let Ok2 = VerifE(EBpk, C) in           NOTE : Commented out because this is not shown in
         the last version of the CryptoPaper (02/2017). *)
174     true
175 .
176
177 (* CreateRC(B,csk,CMtable)  --  Server prepares the Choice Codes  --  aka. CreatePallotProof
          --  from CryptoPaper page 10.                  *)
178 letfun CreateRC(B:bitstring,Csk:symmetric_key,CMtable:bitstring) =
179     let (C:bitstring, {W$i:nat|$i=1..$k}, EC:nat, VCidpk:public_ekey, P:bitstring) = B in
180     let ({CC$i:symmetric_ekey|$i=1..$k}) = ({f(Csk,W$i)|$i=1..$k}) in
181 {     let sCC$i = Dec_s(CC$i,readCC(H(CC$i),CMtable)) in |$i=1..$k|\n}
182     ({sCC$i|$i=1..$k})
183 .
184
185 (* Confirm(VCid,B,VCidsk,BCKid)  --  Computer generates a Confirmation Message  --  Done
        directly inside 'Cmp' and 'Alice_Cmp'.              *)
186
187 (* AuditBallotProof(({sCC_Received$i|$i=1..$k}),({sCCid$i|$i=1..$k}))  --  Alice checks if
        all expected CC were indeed received.             *)
188 (* According to the CryptoPaper update, instead of a method, this is done directly inside
        the 'Alice' and 'Alice_Cmp' processes.            *)
189
190 (* ProcessConfirm(bb,VCid,CMid,Csk,VCCssk,S_VCCid)  --  Server checks the retreived short
        Vote Cast Code.                                   *)
191 (* Instead of a method, this is done directly inside the 'Serv' process.
                                                          *)
192
193
194 (* Typing of the messages -- between Voter and his Computer only *)
195 free mAC1, mAC2, mCA1, mCA2 : bitstring.
196
```

```
197 (* Alice  --  The client process *)
198 let Alice(Ch1:channel,InitData:bitstring,{J$i:nat|$i=1..$k}) =
199     (* Checks that the voting choices are all different *)
200 {     if {(J$i = J$j && a$i <> a$j)|$j=1..$k| || } then 0 else |$i=1..$k|\n}     (* No honest
       voter can use twice the same option *)
201     (* Retrieves registration data obtained from the Registrar  --  Set of initial data
        given to Alice by the Registrar.              *)
202     let (SVKid:password, BCKid:nat, sVCCid:bitstring, {sCCid$i:bitstring|$i=1..$k}) =
        GetAliceData(InitData,{J$i|$i=1..$k}) in
203     (* Voting part -- The voting process followed by agent Alice *)
204     out(Ch1, ( mAC1,SVKid,{J$i|$i=1..$k}));
205     in( Ch1, (=mCA1,({sCC_Received$i:bitstring|$i=1..$k})));
206     if {({(sCC_Received$i=sCCid$j)|$i=1..$k| || })|$j=1..$k| && } then        (* Compares
        the short Choice Codes.               *)
207     ( event Confirmed(InitData, {J$i|$i=1..$k});
208       out(Ch1, ( mAC2,BCKid));
209       in( Ch1, (=mCA2,=sVCCid));                (* Alice checks the Vote Cast Code's value.
          *)
210       event HappyUser(InitData, {J$i|$i=1..$k})
211     )
212 .
213
214 (* Computer  --  The Alice's computer *)
215 let Cmp(Ch1:channel,Ch2:channel,EBpk:public_ekey) =
216     in( Ch1, (=mAC1,SVKid:password,{J$i:nat|$i=1..$k}));          (* The Start Voting Key
        plus the Voting options. *)
217     let VCid:agent_id = deltaId(SVKid) in                                          (* The
        GetID method; CryptoPaperUpdate page 4.   *)
218     out(Ch2, VCid);                                                                (* Send
        the Voting Card ID to the Bulletin Board *)
219
220     in( Ch2, VCksid:bitstring);                                                    (*
        Receives the asso. Verification Card keystore *)
221     let VCidsk = GetKey(SVKid,VCksid) in              (* Recover the asso. the Voting Card
        private key *)
222     out(Ch2, CreateVote(EBpk,VCid,{J$i|$i=1..$k},pube(VCidsk),VCidsk));       (* Sends the
        ballot (ie. 'Vote') to the server.  *)
223
224     in( Ch2, sCC_Set:bitstring);                (* Transmits the Choice Codes to the voter.
          *)
225     out(Ch1, ( mCA1,sCC_Set));
226
227     in( Ch1, (=mAC2,BCKid:nat));
228     out(Ch2, pCC(VCidsk,BCKid));                (* The Confirm(VCid,_,VCidsk,BCK) voter's
        method *)
229
230     in( Ch2, sVCCid:bitstring);                 (* Transmits the Vote Cast Code to the voter
        .    *)
231     out(Ch1, ( mCA2,sVCCid))
232 .
233
234 (* MODELING  --  Alice plus her Computer together (if both honest, to avoid useless secure
        communications).                        *)
235 let Alice_Cmp(InitData:bitstring,{J$i:nat|$i=1..$k},Ch2:channel,EBpk:public_ekey) =
236     (* Checks that the voting choices are all different *)
237 {     if {(J$i = J$j && a$i <> a$j)|$j=1..$k| || } then 0 else |$i=1..$k|\n}     (* No honest
       voter can use twice the same option *)
238     (* Retrieves registration data obtained from the Registrar  --  Set of initial data
        given to Alice by the Registrar.              *)
239     let (SVKid:password, BCKid:nat, sVCCid:bitstring, {sCCid$i:bitstring|$i=1..$k}) =
        GetAliceData(InitData,{J$i|$i=1..$k}) in
240     (* Voting part -- The voting process followed by agent Alice *)
241     let VCid:agent_id = deltaId(SVKid) in                                          (* The
        GetID method; CryptoPaperUpdate page 4.   *)
242     out(Ch2, VCid);                                                                (* Send
        the Voting Card ID to the Bulletin Board *)
243
```

```
244     in( Ch2, VCksid:bitstring);                                                       (*
    Receives the asso. Verification Card keystore *)
245     let VCidsk = GetKey(SVKid,VCksid) in              (* Recover the asso. the Voting Card
    private key *)
246     out(Ch2, CreateVote(EBpk,VCid,{J$i|$i=1..$k},pube(VCidsk),VCidsk));       (* Sends the
    ballot (ie. 'Vote') to the server.  *)
247 
248     in( Ch2, ({sCC_Received$i:bitstring|$i=1..$k}));             (* Receives the short Choice
     Codes from server.  *)
249     if {({(sCC_Received$i=sCCid$j)|$i=1..$k| || })|$j=1..$k| && } then        (* Compares
    the short Choice Codes.                 *)
250     ( event Confirmed(InitData, {J$i|$i=1..$k});
251       out(Ch2, pCC(VCidsk,BCKid));                 (* The Confirm(VCid,_,VCidsk,BCK) voter's
    method *)
252       in( Ch2, =sVCCid);                   (* Alice checks the Vote Cast Code's value.
    *)
253       event HappyUser(InitData, {J$i|$i=1..$k})
254     )
255 .
256 
257 (* Server  --  The election server  --  With infinitly iterated Vote Cast Code (ie.
    Finalization/ProcessConfirm) part.                *)
258 let Serv(Ch2:channel,InitData:bitstring,CTally:channel) =
259     in(Ch2, VCid:agent_id);
260     (* Retrieves Registration data for this Voting Card Id *)
261     let (EBpk:public_ekey,Csk:symmetric_key,VCCspk:public_skey,VCksid:bitstring,CMtable:
    bitstring,S_VCC:bitstring) = GetServData(InitData,VCid) in
262     (* Provides VCksid to the voting device, and asks for the ballot (alias 'vote') *)
263     out(Ch2, VCksid);
264     in( Ch2, B:bitstring);
265     (* Voting part -- the Ballot processing and confirmation followed by the Server *)
266     let Ok1 = ProcessVoteCheck(EBpk,VCid,B) in
267     ( get bb(=VCid,B2) in 0 else
268       ( event InsertBB(VCid,B); insert bb(VCid,B);        (* Add ballot to the Ballot Box '
    bb'                 *)
269   let sCC_Set = CreateRC(B,Csk,CMtable) in              (* Gets the set of CC; Blocks the
    process if this fails.  *)
270   ( out(Ch2, sCC_Set);
271     !( in( Ch2, CM:nat);
272        let VCCid:symmetric_ekey = f(Csk,CM) in        (* The ProcessConfirm(bb,VCid,CM,Csk,
    VCCspk,S_VCC) method *)
273        let sVCCid:bitstring = Dec_s(VCCid, readVCC(H(VCCid),CMtable)) in
274        let Ok2 = Verify(VCCspk,sVCCid,S_VCC) in
275        ( insert cb(VCid,sVCCid,S_VCC);
276          event HasVoted(InitData,VCid,B,sVCCid,S_VCC);      (* Add confirmation to the
    Confirmation Box 'cb'          *)
277          out(Ch2, sVCCid); out(CTally, (VCid,B,sVCCid,S_VCC))   (* Phase 1 -- Sends the
    ballot to the Tally                *)
278        )
279            )
280         )
281       )
282     )
283 .
284 
285 
286 
287 (* 6. Second Phase -- The Tally after all votes are collected. *)
288 
289 (* No Tally for the reachability properties, only for Observational equivalence. *)
290 
291 
292 (* 7. Security properties *)
293 
294 (* Successfull-run Properties : Properties false iff the agents could drive the protocol to
    a successful state; Expected normal run. *)
295 #query SVK:password, Csk:symmetric_key, EBpk:public_ekey, VCCssk:private_skey, B :bitstring,
     VCC:bitstring, S_VCC:bitstring;
```

```
296 #                                    event(HasVoted(ServData(EBpk,Csk,VCCssk), deltaId(honest(
    SVK)), B, VCC, S_VCC)) ==> event(NeverTrue).
297 #query SVK:password, Csk:symmetric_key, {J$i:nat|$i=1..$k};
298 #                                    event(HappyUser(AliceData(honest(SVK),Csk), {J$i|$i=1..$k
    }))                         ==> event(NeverTrue).
299
300 (* Single-vote Property : Only one ballot is stored in the table for each agent Id  --
    Impossible with ProVerif, need thread exclusion.    *)
301 (*                       Moreover, the protocol specification we used did not ensure this
    property, which is implementation dependant.     *)
302 (* Remark : If we combine this prop with our cast-as-intended and tallied-as-cast proved
    properties, then the expected cast-as-intended     *)
303 (*          and tallied-as-cast, with only one ballot, immediately follows. If however the
    single-vote property cannot be guarantied by     *)
304 (*          the implementation, then the protocol is vulnerable: a dishonest voter's
    computer may cast multiple ballots concurrently so     *)
305 (*          that one with other choices that the voter's ones is finally accepted. The
    simple table's tests do not prevent this behaviour.  *)
306 (* query VCid:agent_id, B1:bitstring, B2:bitstring;   event(InsertBB(VCid,B1)) && event(
    InsertBB(VCid,B2)) ==> B1 = B2.     *)
307
308
309 (* Cast-as-Intended Property : The server registers a vote for the *honest* voter if and
    only if the voter confirmed his vote to him. *)
310 query Id:agent_id, Csk:symmetric_key, {W$i:nat|$i=1..$k}, EBpk:public_ekey, VCCssk:private_
    skey, EC:nat, P:bitstring, R:nat,
311       sVCCid:bitstring, C:bitstring, S_VCC:bitstring, {J$i:nat|$i=1..$k}, SVK:password,
312      { {J$iJ$j:nat, W$iW$j:nat|$i=1..$k|, }, R$j:nat, EC$j:nat, P$j:bitstring|$j=1..$k|,\n
        };
313   event(HasVoted(ServData(EBpk,Csk,VCCssk), deltaId(honest(SVK)), (C,{W$i|$i=1..$k},EC,pube(
      ske(deltaId(honest(SVK)))),P), sVCCid, S_VCC))
314     ==>  event(Confirmed(AliceData(honest(SVK),Csk), {ja$i|$i=1..$k}))  && Id = deltaId(
      honest(SVK))                       (* alias VCid *)
315       && C = (Enc_c1(R),Enc_c2(EBpk,phi({v(J$i )|$i=1..$k}),R))
316   (* Linking J1..Jk to j1..jk : There exists one ballot for J1..Jk, plus k ballots
      containing every j in j1..jk; Equal if Single-vote *)
317   (* Consequently, if the voting choices in HasVoted(..) does not match the voter's
      intentions, then the Server misbehaved.     *)
318       && event(InsertBB(Id, ((Enc_c1(R  ),Enc_c2(EBpk,phi({v(J$i   )|$i=1..$k}),R  )),{W$i
     |$i=1..$k},EC  ,pube(ske(Id)),P  )))
319 {      && event(InsertBB(Id, ((Enc_c1(R$j),Enc_c2(EBpk,phi({v(J$iJ$j)|$i=1..$k}),R$j)),{W$iW$
    j|$i=1..$k},EC$j,pube(ske(Id)),P$j))) |$j=1..$k|\n}
320       && {({J$iJ$j = ja$j|$i=1..$k| || })|$j=1..$k| && }
321 .
322
323 (* REMARK: If the Server did not misbehave, then all these InsertBB(...) must be equal all
    together, since only one could be cast for *)
324 (*         each Id. It follows that the ballot shown by HasVoted(..) contains all the honest
     voter's choices j1..jk, and thus no more *)
325 (*         that these since the j1..jk differ two-by-two and each ballot contains exactly k
    voting choices.                           *)
326
327
328 (* Tallied-as-Cast Property : The voter finishes and is happy iff the server has his vote (
    same choices) and the Tally will accept it.*)
329 query Id:agent_id, Csk:symmetric_key, {W$i:nat|$i=1..$k}, EBpk:public_ekey, VCCssk:private_
    skey, EC:nat, P:bitstring, R:nat,
330       sVCCid:bitstring, C:bitstring, S_VCC:bitstring, {J$i:nat|$i=1..$k}, SVK:password,
331      { {J$iJ$j:nat, W$iW$j:nat|$i=1..$k|, }, R$j:nat, EC$j:nat, P$j:bitstring|$j=1..$k|,\n
        };
332   event(HappyUser(AliceData(SVK,Csk), {ja$i|$i=1..$k}))
333     ==>  event(HasVoted(ServData(EBpk,Csk,VCCssk), Id, (C, {W$i|$i=1..$k}, EC, pube(ske(Id))
      , P), sVCCid, S_VCC))  && Id = deltaId(SVK)
334   (* The ballot is present in the table     -- Expected : table(bb(Id,B))           --
      REMOVED : subsumed by HasVoted(..)      *)
335   (* The ballot is confirmed in the table   -- Expected : table(cb(Id,sVCCid,S_VCC))  --
      REMOVED : subsumed by HasVoted(..)      *)
```

```
336   (* Runs VerifP from ProcessVoteCheck       -- Expected : VerifP(EBpk,pube(ske(Id)),C,{W$i|$
        i=1..$k}, P) = true          *)
337        && C = (Enc_c1(R),Enc_c2(EBpk,phi({v(J$i)|$i=1..$k}),R))
338        && {W$i=pCC(ske(Id),v(J$i))|$i=1..$k| && }
339        && P = ZKP(EBpk,pube(ske(Id)),C,{W$i|$i=1..$k}, R,ske(Id))
340   (* Runs verifE (but removed, 02/2017)      -- Expected : verifE(EBpk, C) = true  --  but
        done already through C = Enc(EBpk,....)      *)
341   (* Runs Verify to check the confirmation  -- Expected : Verify(pubs(VCCssk),sVCCid,S_VCC)
        = true             *)
342        && S_VCC = Sign(VCCssk,sVCCid)
343   (* Linking J1..Jk to j1..jk : There exists one ballot for J1..Jk, plus k ballots
        containing every j in j1..jk; Equal if Single-vote *)
344   (* Consequently, if the voting choices in HasVoted(..) does not match the voter's
        intentions, then the Server misbehaved.     *)
345        && event(InsertBB(Id, ((Enc_c1(R  ),Enc_c2(EBpk,phi({v(J$i   )|$i=1..$k}),R  )),{W$i
         |$i=1..$k},EC  ,pube(ske(Id)),P  )))
346   {    && event(InsertBB(Id, ((Enc_c1(R$j),Enc_c2(EBpk,phi({v(J$iJ$j)|$i=1..$k}),R$j)),{W$iW$
        j|$i=1..$k},EC$j,pube(ske(Id)),P$j))) |$j=1..$k|\n}
347        && {({J$iJ$j = ja$j|$i=1..$k| || })|$j=1..$k| && }
348 .
349
350 (* REMARK, as for Cast-as-intended : If the Server did not misbehave, then all these
        InsertBB(...) must be equal, and thus the ballot *)
351 (*         shown by HasVoted(..) contains all the honest voter's choices j1..jk, and no more
         since the j1..jk are unique and the       *)
352 (*         ballot contains k choices.  *)
353
354
355
356 (* 8. Main process  -- initiates the election *)
357
358 process
359 (* Public output from Setup(..)  --  Gives the election's parameters to the Intruder. *)
360   out(c, ebpk); out(c, vccspk);
361
362 (* Gives the honest voter's public data to the Intruder *)
363   out(c, deltaId(honest(svka))); out(c,pke(deltaId(honest(svka))));
364   out(c, deltaId(honest(svkb))); out(c,pke(deltaId(honest(svkb))));
365
366 (* Roles for honest voter(s)  --  one for Reachability, two for Observational Equivalence.
        *)
367     Alice( c, AliceData(honest(svka),csk),{ja$i|$i=1..$k} )
368
369 (* Dishonest voter(s) : As many as possible for Reachability, need only one for Obervational
         equivalence. *)
370  | !(new svki:password; out(c,svki); out(c,AliceData(svki,csk)))
371
372 (* Bulletin Board (ie. Server) : is Honest for Reachability, but Dishonest for Observational
         Equivalence. *)
373  | !Serv(c,ServData(ebpk,csk,vccssk),c)
```

../specs/study_v13–REA.pve

Appendix B

ProVerif source file specs/study_v13-OBS_k=1.pv

```
1 (* Post-review: Unified ProVerif specification, version 11, for Scytl's Voting Protocol. All
      variants included. Tabulation size : 8 spaces. *)
2 (* Reachability ........... : For Cast as Intended and Tallied as Cast properties; from
     study_v13.pve with the 'REA' option.            *)
3 (* Observational Equivalence : For Ballot Provivacy; 2 honest + 1 or 0 dishonest in the
     challenge; [..] with the 'OBS' option w/ or w/o HH. *)
4 (* Syntactical remark ...... : To avoid confusions, the Vote V of the voter is named here
     the Ballot B of the voter (it uses encryption).   *)
5
6 set ignoreTypes=false.
7
8 (* 1. Objects and Types *)
9
10 type   agent_id.      fun t_agent_id(agent_id)       :    bitstring   [data,typeConverter].
       (* The type for any IDs like e.g. voting card. *)
11 type   password.      fun t_password(password)       :    bitstring   [data,typeConverter].
       (* The type for user passwords.               *)
12 type   nat     .      fun t_nat(nat)                 :    bitstring   [data,typeConverter].
       (* The type for natural numbers.              *)
13 free   c          : channel              .    (* Public Channel for communications with the
     intruder   # public                 *)
14 fun    v(nat)                 : nat         [data]   .    (* The function from voting
     choices to voting options    # public ,    invertible *)
15 table  bb(agent_id,bitstring)                          .    (* The Ballot Box, storing the
     ballots for each agent    # private                 *)
16 table  cb(agent_id,bitstring,bitstring)                .    (* The Confirm. Box, storing VCC/S
     _VCC for each agent id # private                *)
17 free   svka, svkb             : password            .    (* Defines two SVK for two honest
     voters Alice and Bob   # public                *)
18 free   ja1,jb1                : nat                 .    (* Voting choices for the Honest
     voters -- REA or OBS    # public                *)
19 free   a1                     : nat        .    (* MODELING : an indexed set of nat atoms,
     for disquality tests  -- only in Test. *)
20
21
22
23 (* 2. Intruder capabilities and functions extracted from the CryptoPaper. *)
24
25 (* Encryption scheme  --  from CryptoPaper, page 4  --  expected to be ElGamal  --  Key
     pairs generated through Gen_e, which is implicit.   *)
26 type   private_ekey.  fun t_private_ekey(private_ekey):   bitstring   [data,typeConverter].
       (* The type for private keys + type converter. *)
27 type   public_ekey .  fun t_public_ekey(public_ekey)  :   bitstring   [data,typeConverter].
       (* The type for public keys  + type converter. *)
```

```
28 fun    ske(agent_id)          : private_ekey [private].   (* The private enc. key associated
      to an agent_id        # private                *)
29 fun    pube(private_ekey)     : public_ekey          .   (* The function to rebuild a
     public key from the private # public , noninvertible *)
30 letfun pke(Id:agent_id)       = pube(ske(Id))        .   (* The public  enc. key associated
      to an agent id        # public ,    invertible *)
31 fun    Enc_c1(                       nat) : nat        .   (* The c1 part of the asymmetric
     encryption function with explicit random number. *)
32 fun    Enc_c2(public_ekey,bitstring,nat) : bitstring  .   (* The c2 part of the asymmetric
     encryption function with explicit random number. *)
33 letfun Enc(Pk:public_ekey,M:bitstring,R:nat) = (Enc_c1(R),Enc_c2(Pk,M,R)) .             (*
      Encryption *)
34 reduc  forall Sk:private_ekey, M:bitstring, R:nat;    Dec(Sk,(Enc_c1(R),Enc_c2(pube(Sk),M,R)
     )) = M   .               (* Decryption *)
35 reduc  forall Pk:public_ekey,  M:bitstring, R:nat;    VerifE(Pk,(Enc_c1(R),Enc_c2(Pk,M,R)))
       = true.                (* Checks key *)
36 reduc  forall Id:agent_id;                            Get_Id(pube(ske(Id)))
       = Id  .                (* Extract Id *)
37
38
39 (* Signature scheme  --  from CryptoPaper, page 4 & 5  --  expected to be the RSA
     Probabilistic Signature Scheme (PSS)  --  Gen_s implicit. *)
40 type   private_skey.  fun t_private_skey(private_skey):   bitstring   [data,typeConverter].
      (* The type for private keys + type converter. *)
41 type   public_skey .  fun t_public_skey(public_skey)  :   bitstring   [data,typeConverter].
      (* The type for public keys  + type converter. *)
42 fun    sks(agent_id)          : private_skey [private].   (* The private sig. key associated
      to an agent_id        # private                *)
43 fun    pubs(private_skey)     : public_skey          .   (* The function to rebuild a
     public key from the private # public , noninvertible *)
44 letfun pks(Id:agent_id)       = pubs(sks(Id))        .   (* The public  sig. key associated
      to an agent id        # public ,    invertible *)
45 fun    Sign(private_skey,bitstring)   : bitstring     .   (* The digital signature function
     with explicit random number. (Signature)        *)
46 reduc  forall Sk:private_skey, M:bitstring;           Verify(pubs(Sk),M,Sign(Sk,M)) = true.
                      (* Verification *)
47 reduc  forall Sk:private_skey, M:bitstring;           Checks(pubs(Sk),Sign(Sk,M))   = M   .
              (* More expressive than Verify. *)
48 reduc  forall Sk:private_skey, M:bitstring;           Get_Message(Sign(Sk,M))       = M   .
              (* Worst case for modeling only *)
49
50
51 (* Modeling of the Non-Interactive Zero-Knowledge Proofs of Knowledge  -- from CryptoPaper,
     pages 5 & 6  -- Names changed for abstractions. *)
52 (* Both the 'Equality of discrete logarithms' (used in two variants) and the 'Knowledge of
     encryption exponent' are abstracted inside the   *)
53 (* ZKP and VerifP operators. Consequently, the c1~ and c2~, which are intermediate values,
     are not needed anymore and thus abstracted away. *)
54 (* Moreover, the 'Correct decryption' is not needed for this modeling w.r.t the expected
     security properties.                               *)
55 fun    pCC(private_ekey,nat)       : nat              .   (* The built of partial Choice
     Codes (ski,vi) = vi^ski   # public , noninvertible *)
56 fun    tild(private_ekey,bitstring): nat              .   (* The built of c1~ and c2~
     abstracted from ZKP/VerifP   # public , noninvertible *)
57 fun    phi(nat)                    : bitstring [data] .   (* The agregation function (prod)
     over a tuple of votes  # public ,    invertible *)
58 reduc  forall Sk:private_ekey, V:nat;                 Get_Vote1(pCC(Sk,V),Sk)      = V
       .       (* Worst case for modeling only *)
59 reduc  forall Sk:private_ekey, C:bitstring;           Get_Vote2(tild(Sk,C),Sk)     = C
       .       (* Worst case for modeling only *)
60 fun    ZKP(public_ekey, public_ekey, bitstring, nat, nat, private_ekey)           :
     bitstring.       (* Modeling of the proof generation   *)
61 reduc  forall EBpk:public_ekey, VCidsk:private_ekey, v1:nat, R:nat;
              (* Modeling of the proof verification *)
62        VerifP(EBpk, pube(VCidsk), (Enc_c1(R),Enc_c2(EBpk,phi(v1),R)), pCC(VCidsk,v1),
63           ZKP(EBpk, pube(VCidsk), (Enc_c1(R),Enc_c2(EBpk,phi(v1),R)), pCC(VCidsk,v1), R,
     VCidsk)) = true.
```

```
64 (* Consequently to abstracting c1~ and c2~ away from ZKP/VerifP, we do not need anymore the
     4 variants of this reduction shown previously.  *)
65 (* Remaining limitation : the agregation function phi(..) is commutative, but modeling it
     generates far too many cases for ProVerif.        *)
66
67
68 (* Symmetric encryption scheme  --  from CryptoPaper Draft update, page 3  --  expected to
     be based on AES                                   *)
69 type   symmetric_ekey. fun t_symmetric_ekey(symmetric_ekey): bitstring [data,typeConverter].
       (* The type for symetric encryption keys.      *)
70 fun    Enc_s(symmetric_ekey,bitstring) : bitstring   .
       (* Encryption (symmetric key) *)
71 reduc  forall SKey:symmetric_ekey, M:bitstring;      Dec_s(SKey,Enc_s(SKey,M)) = M.
       (* Decyption  (symmetric key) *)
72
73
74 (* Key and IDs derivation scheme  --  from CryptoPaper Draft update, page 3  *)
75 (* free   IDseed              : bitstring            . *)  (* A public seed for IDs   --
     Not needed: value fixed  # public                  *)
76 (* free   KEYseed             : bitstring            . *)  (* A public seed for KEYs  --
     Not needed: value fixed  # public                  *)
77 fun    deltaId(password)      : agent_id             .     (* The delta function, for agents
       IDs                    # public                  *)
78 fun    deltaKey(password)     : symmetric_ekey       .     (* The delta function, for
     symmetric encryption keys    # public               *)
79 fun    honest(password)       : password [private]   .     (* MODELING : a function to
     separate honest and dishonest agents (only in Prop). *)
80
81
82 (* Keyed Pseudo-Random and hash functions  --  from cryptoPaper, pages 7 & 8  --  Note: Not
     the same keys as for symmetric encryption       *)
83 type   symmetric_key. fun t_symmetric_key(symmetric_key): bitstring   [data,typeConverter].
       (* The type for symetric keys for KPR f(...).  *)
84 fun    H(symmetric_ekey)      : bitstring            .    (* The hash function  --  from
     CryptoPaper page 5         # public , noninvertible *)
85 fun    f(symmetric_key,nat)   : symmetric_ekey       .    (* The keyed pseudo-random
     function f  --  page 7        # public , noninvertible *)
86
87
88
89 (* 3. Initialization sequence (done off-line in this modeling). *)
90
91 (* Setup(...) i.e. the initialisation step from the Registrar  --  done off-line in this
     modeling  --  from CryptoPaper page 9              *)
92 free   election            : agent_id              .    (* The election id - used to
     derive public/private keys  # public (sk is private) *)
93 free   signature           : agent_id              .    (* The signature id - used to
     derive public/private keys # public (sk is private) *)
94 letfun ebpk                = pke(election)          .    (* The Electorial Board public
     key in pair (EBpk,EBsk)  # public                  *)
95 letfun ebsk                = ske(election)          .    (* The Electorial Board private
     key in pair (EBpk,EBsk)  # private                 *)
96 free   csk                 : symmetric_key [private] .    (* The Codes secret key 'Csk' (for
       f(...) function)      # private                 *)
97 letfun vccspk              = pks(signature)         .    (* The Vote Cast Code Signer
     public  key in (VCCspk,VCCssk) # public              *)
98 letfun vccssk              = sks(signature)         .    (* The Vote Cast Code Signer
     private key in (VSSspk,VCCssk) # private             *)
99
100
101 (* Register(..,id,csk,vccssk) ie. the registration step for any new voter  --  done off-line
       through functions  --  from CryptoPaper page 9 *)
102 fun    bck(agent_id)                    : nat       [private].  (* The Ballot Casting Key '
     BCK^id'  of an agent id                  # private *)
103 fun    sCC(symmetric_key,agent_id,nat) : bitstring [private].  (* The short choice code 'sCC
     ^id_i' of an agent id + voting option # private *)
104 fun    sVCC(symmetric_key,agent_id)     : bitstring [private]. (* The short vote cast code '
     sVCC^id' of an agent id               # private *)
```

```
105 reduc  forall SVK:password; GetVCks(deltaId(SVK)) = Enc_s(deltaKey(SVK), t_private_ekey(ske(
      deltaId(SVK)))). (* CryptoPaperDraft upd.page 4 *)
106 fun    CM_table(symmetric_key,agent_id) : bitstring  .       (* The Codes Mapping Table ('CM
      _id', unbounded)  --  opened with readCC/VCC(). *)
107 reduc  forall Csk:symmetric_key, VCid:agent_id, J:nat;
                                   (* Enc_s(CC,sCC) *)
108        readCC(H(f(Csk, pCC(ske(VCid),v(J)  ))), CM_table(Csk,VCid)) = Enc_s(f(Csk, pCC(ske(
      VCid),v(J)  )), sCC(Csk,VCid,v(J))).
109 reduc  forall Csk:symmetric_key, VCid:agent_id;
                                 (* Enc_s(VCC,sVCC) *)
110        readVCC(H(f(Csk, pCC(ske(VCid),bck(VCid)))), CM_table(Csk,VCid)) = Enc_s(f(Csk, pCC(
      ske(VCid),bck(VCid))), sVCC(Csk,VCid)) .
111 (* For any voter id, generates VCidpk,VCidsk ........ : pke(id) and ske(id) and GetVCks(id)
         --  distributed in encrypted private data.  *)
112 (* For any voter id, chooses a ballot casting key ... : bck(id) as defined above;
         --  distributed in AliceData and Mapping.   *)
113 (* For any voter id, computes a list of Choice Codes CC(id,i) ........ : with f(..) and pCC
      (..) and v(i)     --  used to build the Mapping. *)
114 (* for any voter id, computes the Vote Cast Code  VCC(id) ........... : with f(..) and pCC
      (..) and bck(id)  --  used to build the Mapping. *)
115 (* for any voter id, chooses the short CC and short VCC ............. : with sCC(..) and
      sVCC(..) -- distributed in AliceData and Mapping. *)
116 (* For any voter id, computes the signature of the Vote Cast Code .... : Sign(vccssk,sVCC(id
      ))               --  distributed in ServData.   *)
117 (* For any voter id, stores a list of hashed choice codes {H(CC(id,i))}_i=1..inf ..... :
      through MakeRFList  --  distributed in ServData.   *)
118
119
120 (* Registration data for any voter id  --  AliceData(svk,csk) produced by the Registrar for
      Alice   --  opened with a reduction            *)
121 fun   AliceData(password,symmetric_key) : bitstring [private].
122 reduc forall Csk:symmetric_key, SVK:password, J1:nat;
123       GetAliceData(AliceData(SVK,Csk),J1) = (SVK, bck(deltaId(SVK)), sVCC(Csk,deltaId(SVK)),
       sCC(Csk,deltaId(SVK),v(J1))).
124 (*    Note: The term sCC(id,v(J)) is a selection of one sCC corresponding to J1, decided at
      Alice's initialisation.                         *)
125
126
127 (* Registration data for the server (Bulletin Board)  --  ServData(pke,Csk,sks) produced by
      the Registrar  -- opened with a reduction       *)
128 fun   ServData(public_ekey,symmetric_key,private_skey) : bitstring [private].
129 reduc forall EBpk:public_ekey, Csk:symmetric_key, VCCssk:private_skey, SVK:password;
                      (* Note : VCid = deltaId(SVK) *)
130       GetServData(ServData(EBpk,Csk,VCCssk),deltaId(SVK)) = (EBpk,Csk,pubs(VCCssk), Enc_s(
      deltaKey(SVK),t_private_ekey(ske(deltaId(SVK)))),
131                                                             CM_table(Csk,deltaId(SVK)),
      Sign(VCCssk,sVCC(Csk,deltaId(SVK))) ).
132
133
134
135 (* 4. Algebraic properties and List of Events *)
136
137 (* Equational theory commented out  --  Kept for reference but this property is not
      supported by ProVerif. *)
138 (* equation forall {V$i:nat|$i=1..$k};  phi({V$i|$i=1..$k}) = phi({V$i|$i=1..$k 'mixed' }) .
                      *)              (* Commutativity *)
139
140 event Confirmed(bitstring, nat).                                            (* Issued by
      the voter when he/she confirms his/her vote.    *)
141 event HappyUser(bitstring, nat).                                            (* Issued by
      the voter when he/she terminates successfuly.   *)
142 event InsertBB( agent_id, bitstring).                                       (* Issued by
      the server when it adds someting in BB.        *)
143 event HasVoted( bitstring, agent_id, bitstring, bitstring, bitstring).      (* Issued by
      the server when it gets a voter's confirmation. *)
144 event NeverTrue.                                                            (* An event
      that is never activated, and thus, never true.   *)
```

```
145 event Results(nat,nat,nat).                                              (* Issued by
      the Tally when it publishes the results.        *)
146
147
148
149 (* 5. Methods and Agents processes *)
150
151 (* GetID(SVKid)  --  Computer generates the Voting Card ID  --  Directly replaced by deltaId
      (SVKid)  --  from CryptoPaper update page 4.    *)
152 (* letfun GetID(SVKid:password) = deltaId(SVKid). *)
153
154 (* GetKey(SVKid,VCksid)  --  Computer retreives the Verification Card private key from the
      keystore  --  from CryptoPaper update page 4.    *)
155 letfun GetKey(SVKid:password,VCksid:bitstring) =
156     let KSpwd = deltaKey(SVKid) in
157     let t_private_ekey(VCidsk:private_ekey) = Dec_s(deltaKey(SVKid),VCksid) in
158     VCidsk
159 .
160
161 (* CreateVote(EBpk,VCid,Vopt,VCidsk)  --  Computer creates the ballot  --  from CryptoPaper
      pages 9 & 10.                                      *)
162 (* NOTE : Change w.r.t. the CryptoPaper Update page 4 : EBpk added as 1st argument, because
      it is needed by this method.                       *)
163 letfun CreateVote(EBpk:public_ekey,VCid:agent_id,J1:nat,VCidpk:public_ekey,VCidsk:private_
      ekey) =
164     let V = phi(v(J1)) in new R:nat; let C = Enc(EBpk, V, R) in
165     let P = ZKP(EBpk,VCidpk,C,pCC(VCidsk,v(J1)), R,VCidsk) in
166     (C, pCC(VCidsk,v(J1)), tild(VCidsk,C), VCidpk, P)
167 .
168
169 (* ProcessVote(bb,VCid,B)  --  Server checks and processes the ballot  --  bb tested and
      filled outside  --  from CryptoPaper page 10.      *)
170 letfun ProcessVoteCheck(EBpk:public_ekey,VCid:agent_id,B:bitstring) =
171     let (C:bitstring, W1:nat, EC:nat, =pke(VCid), P:bitstring) = B in
172     let Ok1 = VerifP(EBpk,pke(VCid),C,W1, P) in
173 (*  let Ok2 = VerifE(EBpk, C) in           NOTE : Commented out because this is not shown in
       the last version of the CryptoPaper (02/2017). *)
174     true
175 .
176
177 (* CreateRC(B,csk,CMtable)  --  Server prepares the Choice Codes  --  aka. CreatePallotProof
        --  from CryptoPaper page 10.                  *)
178 letfun CreateRC(B:bitstring,Csk:symmetric_key,CMtable:bitstring) =
179     let (C:bitstring, W1:nat, EC:nat, VCidpk:public_ekey, P:bitstring) = B in
180     let (CC1:symmetric_ekey) = (f(Csk,W1)) in
181     let sCC1 = Dec_s(CC1,readCC(H(CC1),CMtable)) in
182     (sCC1)
183 .
184
185 (* Confirm(VCid,B,VCidsk,BCKid)  --  Computer generates a Confirmation Message  --  Done
      directly inside 'Cmp' and 'Alice_Cmp'.             *)
186
187 (* AuditBallotProof((sCC_Received1),(sCCid1))  --  Alice checks if all expected CC were
      indeed received.                                  *)
188 (* According to the CryptoPaper update, instead of a method, this is done directly inside
      the 'Alice' and 'Alice_Cmp' processes.            *)
189
190 (* ProcessConfirm(bb,VCid,CMid,Csk,VCCssk,S_VCCid)  --  Server checks the retreived short
      Vote Cast Code.                                   *)
191 (* Instead of a method, this is done directly inside the 'Serv' process.
                                                        *)
192
193
194 (* Typing of the messages -- between Voter and his Computer only *)
195 free mAC1, mAC2, mCA1, mCA2 : bitstring.
196
197 (* Alice  --  The client process *)
198 let Alice(Ch1:channel,InitData:bitstring,J1:nat) =
```

```
199     (* Checks that the voting choices are all different *)
200     if (J1 = J1 && a1 <> a1) then 0 else                                    (* No honest
        voter can use twice the same option *)
201     (* Retrieves registration data obtained from the Registrar  --  Set of initial data
        given to Alice by the Registrar.              *)
202     let (SVKid:password, BCKid:nat, sVCCid:bitstring, sCCid1:bitstring) = GetAliceData(
        InitData,J1) in
203     (* Voting part -- The voting process followed by agent Alice *)
204     out(Ch1, ( mAC1,SVKid,J1));
205     in( Ch1, (=mCA1,(sCC_Received1:bitstring)));
206     if ((sCC_Received1=sCCid1)) then                                      (* Compares the
        short Choice Codes.            *)
207     ( event Confirmed(InitData, J1);
208       out(Ch1, ( mAC2,BCKid));
209       in( Ch1, (=mCA2,=sVCCid));                (* Alice checks the Vote Cast Code's value.
          *)
210       event HappyUser(InitData, J1)
211     )
212 .
213
214 (* Computer  --  The Alice's computer *)
215 let Cmp(Ch1:channel,Ch2:channel,EBpk:public_ekey) =
216     in( Ch1, (=mAC1,SVKid:password,J1:nat));                        (* The Start Voting
        Key plus the Voting options. *)
217     let VCid:agent_id = deltaId(SVKid) in                                      (* The
        GetID method; CryptoPaperUpdate page 4.   *)
218     out(Ch2, VCid);                                                           (* Send
        the Voting Card ID to the Bulletin Board *)
219
220     in( Ch2, VCksid:bitstring);                                               (*
        Receives the asso. Verification Card keystore *)
221     let VCidsk = GetKey(SVKid,VCksid) in              (* Recover the asso. the Voting Card
        private key *)
222     out(Ch2, CreateVote(EBpk,VCid,J1,pube(VCidsk),VCidsk));               (* Sends the
        ballot (ie. 'Vote') to the server.  *)
223
224     in( Ch2, sCC_Set:bitstring);                (* Transmits the Choice Codes to the voter.
          *)
225     out(Ch1, ( mCA1,sCC_Set));
226
227     in( Ch1, (=mAC2,BCKid:nat));
228     out(Ch2, pCC(VCidsk,BCKid));                (* The Confirm(VCid,_,VCidsk,BCK) voter's
        method *)
229
230     in( Ch2, sVCCid:bitstring);                 (* Transmits the Vote Cast Code to the voter
        .    *)
231     out(Ch1, ( mCA2,sVCCid))
232 .
233
234 (* MODELING  --  Alice plus her Computer together (if both honest, to avoid useless secure
        communications).                       *)
235 let Alice_Cmp(InitData:bitstring,J1:nat,Ch2:channel,EBpk:public_ekey) =
236     (* Checks that the voting choices are all different *)
237     if (J1 = J1 && a1 <> a1) then 0 else                                    (* No honest
        voter can use twice the same option *)
238     (* Retrieves registration data obtained from the Registrar  --  Set of initial data
        given to Alice by the Registrar.              *)
239     let (SVKid:password, BCKid:nat, sVCCid:bitstring, sCCid1:bitstring) = GetAliceData(
        InitData,J1) in
240     (* Voting part -- The voting process followed by agent Alice *)
241     let VCid:agent_id = deltaId(SVKid) in                                      (* The
        GetID method; CryptoPaperUpdate page 4.   *)
242     out(Ch2, VCid);                                                           (* Send
        the Voting Card ID to the Bulletin Board *)
243
244     in( Ch2, VCksid:bitstring);                                               (*
        Receives the asso. Verification Card keystore *)
```

```
245     let VCidsk = GetKey(SVKid,VCksid) in              (* Recover the asso. the Voting Card
        private key *)
246     out(Ch2, CreateVote(EBpk,VCid,J1,pube(VCidsk),VCidsk));                (* Sends the
        ballot (ie. 'Vote') to the server.  *)
247
248     in( Ch2, (sCC_Received1:bitstring));            (* Receives the short Choice Codes from
        server.  *)
249     if ((sCC_Received1=sCCid1)) then                                 (* Compares the short
        Choice Codes.             *)
250     ( event Confirmed(InitData, J1);
251       out(Ch2, pCC(VCidsk,BCKid));                 (* The Confirm(VCid,_,VCidsk,BCK) voter's
        method *)
252       in( Ch2, =sVCCid);                   (* Alice checks the Vote Cast Code's value.
        *)
253       event HappyUser(InitData, J1)
254     )
255 .
256
257 (* Server  --  The election server  --  With infinitly iterated Vote Cast Code (ie.
        Finalization/ProcessConfirm) part.                 *)
258 let Serv(Ch2:channel,InitData:bitstring,CTally:channel) =
259     in(Ch2, VCid:agent_id);
260     (* Retrieves Registration data for this Voting Card Id *)
261     let (EBpk:public_ekey,Csk:symmetric_key,VCCspk:public_skey,VCksid:bitstring,CMtable:
        bitstring,S_VCC:bitstring) = GetServData(InitData,VCid) in
262     (* Provides VCksid to the voting device, and asks for the ballot (alias 'vote') *)
263     out(Ch2, VCksid);
264     in( Ch2, B:bitstring);
265     (* Voting part -- the Ballot processing and confirmation followed by the Server *)
266     let Ok1 = ProcessVoteCheck(EBpk,VCid,B) in
267     ( get bb(=VCid,B2) in 0 else
268       ( event InsertBB(VCid,B); insert bb(VCid,B);        (* Add ballot to the Ballot Box '
        bb'                      *)
269   let sCC_Set = CreateRC(B,Csk,CMtable) in              (* Gets the set of CC; Blocks the
        process if this fails.  *)
270   ( out(Ch2, sCC_Set);
271     !( in( Ch2, CM:nat);
272        let VCCid:symmetric_ekey = f(Csk,CM) in        (* The ProcessConfirm(bb,VCid,CM,Csk,
        VCCspk,S_VCC) method *)
273        let sVCCid:bitstring = Dec_s(VCCid, readVCC(H(VCCid),CMtable)) in
274        let Ok2 = Verify(VCCspk,sVCCid,S_VCC) in
275        ( insert cb(VCid,sVCCid,S_VCC);
276          event HasVoted(InitData,VCid,B,sVCCid,S_VCC);      (* Add confirmation to the
        Confirmation Box 'cb'          *)
277          out(Ch2, sVCCid); out(CTally, (VCid,B,sVCCid,S_VCC))   (* Phase 1 -- Sends the
        ballot to the Tally               *)
278        )
279            )
280         )
281       )
282     )
283 .
284
285
286
287 (* 6. Second Phase -- The Tally after all votes are collected. *)
288
289 (* Runnability tests  --  Checks if we can reach a final Results(..) event.         *)
290 (*free  jci : nat .*)               (* Voting options for one Dishonest voter.       *)
291 (*query                             event(Results(ja1,jb1,jc1)) ==> event(NeverTrue).*)
292 (*query J11:nat, J21:nat, J31:nat; event(Results(J11,J21,J31)) ==> event(NeverTrue).*)
293 (*query SVK:password, Csk:symmetric_key, {J$i:nat,W$i:nat|$i=1..$k};                *)
294 (*             event(HappyUser(AliceData(honest(SVK),Csk), J1)) ==> event(NeverTrue).*)
295
296 (* Tally -- the election tally *)
297 free mix:channel [private].
298 let Tally(CTally:channel,EBsk:private_ekey,VCCspk:public_skey,VCid1:agent_id,VCid2:agent_id)
         =
```

```
299     (* Only one variant, called HHD previously; Others HDH and DHH simply swap the reception
         order, and thus are clearly equivalent.  *)
300     in(CTally, (=VCid1,B1:bitstring,sVCC1:bitstring,S_VCC1:bitstring));
301     in(CTally, (=VCid2,B2:bitstring,sVCC2:bitstring,S_VCC2:bitstring));
302     in(CTally, (VCid3:agent_id,B3:bitstring,sVCC3:bitstring,S_VCC3:bitstring));      (* use
        option HH to choose between HHD and HH *)
303     (* Validation of the Ballots in the bulletin board. *)
304     let Ok1 = ProcessVoteCheck(pube(EBsk),VCid1,B1) in let Ok1b = Verify(VCCspk,sVCC1,S_VCC
        1) in
305     let Ok2 = ProcessVoteCheck(pube(EBsk),VCid2,B2) in let Ok2b = Verify(VCCspk,sVCC2,S_VCC
        2) in
306     let Ok3 = ProcessVoteCheck(pube(EBsk),VCid3,B3) in let Ok3b = Verify(VCCspk,sVCC3,S_VCC
        3) in
307     if VCid1 <> VCid2  && VCid1 <> VCid3 && VCid2 <> VCid3
308     then let (C1:bitstring, W11:nat, EC1:nat, =pke(VCid1), P1:bitstring) = B1 in
309          let (C2:bitstring, W21:nat, EC2:nat, =pke(VCid2), P2:bitstring) = B2 in
310          let (C3:bitstring, W31:nat, EC3:nat, =pke(VCid3), P3:bitstring) = B3 in
311          (* MixNet Modeling -- The data to be mixed is sent concurrently on a specific
        channel *)
312          out(mix, choice[C1,C2]) | out(mix, choice[C2,C1]) | in(mix, MC1:bitstring); in(mix,
         MC2:bitstring);
313 (*       out(mix,        C1    ) | out(mix,        C2    ) | in(mix, MC1:bitstring); in(mix,
         MC2:bitstring);*) (* Only for Runnability. *)
314          (* Decrypting -- The mixed ciphers are opened and decrypted. *)
315          let phi(v(J11)) = Dec(EBsk,MC1) in
316          let phi(v(J21)) = Dec(EBsk,MC2) in
317          let phi(v(J31)) = Dec(EBsk, C3) in
318          (* Publishing -- The intruder receices the election's result. *)
319          event Results(J11,J21,J31);             (* Only for Runnability. *)
320          out(c, (J11));
321          out(c, (J21));
322    out(c, (J31))
323 .
324
325
326
327 (* 7. Security properties *)
328
329 (* For observational equivalence, this is done through the choice(..) method. *)
330
331
332
333 (* 8. Main process  -- initiates the election *)
334
335 process
336 (* Public output from Setup(..)  --  Gives the election's parameters to the Intruder. *)
337   out(c, ebpk); out(c, vccspk);
338
339 (* Gives the honest voter's public data to the Intruder *)
340   out(c, deltaId(honest(svka))); out(c,pke(deltaId(honest(svka))));
341   out(c, deltaId(honest(svkb))); out(c,pke(deltaId(honest(svkb))));
342
343 (* Roles for honest voter(s)  --  one for Reachability, two for Observational Equivalence.
        *)
344    Alice_Cmp(AliceData(honest(svka),csk),choice[ja1,jb1],c,ebpk)
345  | Alice_Cmp(AliceData(honest(svkb),csk),choice[jb1,ja1],c,ebpk)
346 (* Alice_Cmp(AliceData(honest(svka),csk),       ja1     ,c,ebpk)*)   (* For Runnability
        check *)
347 (*|Alice_Cmp(AliceData(honest(svkb),csk),       jb1     ,c,ebpk)*)   (* Same
          *)
348
349 (* Dishonest voter(s) : As many as possible for Reachability, need only one for Obervational
         equivalence. *)
350  |  (new svki:password; out(c,svki); out(c,AliceData(svki,csk)))
351
352 (* Bulletin Board (ie. Server) : is Honest for Reachability, but Dishonest for Observational
         Equivalence. *)
353  |  out(c,ServData(ebpk,csk,vccssk))
```

```
354
355 (* The Tally phase  --  only after the honest voters have voted. *)
356  | Tally(c,ebsk,vccspk,deltaId(honest(svka)),deltaId(honest(svkb)))
```

../specs/study_v13–OBS_k=1.pv