Scytl sVote

Complete Verifiability Formal Proof Report

Software version 2.1
Document version 1.0
# Table of contents

1 Summary ................................................................................................................. 4

2 Appendix .................................................................................................................. 5
   2.1 EV Solution Intellectual Property Rights Notice (the Notice)................................. 5
       2.1.1 Definitions ........................................................................................................... 5
       2.1.2 Copyright notice .................................................................................................. 6

1 Summary

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

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

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

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

In this document, we are introducing the paper that provides the symbolic proof of complete Verifiability of Scytl sVote Voting Protocol, according to the complete abstract model defined in the VEleS ordinance. The paper has been led by David Galindo, PhD (Crypto in Motion Ltd) with assistance from Eike Ritter, PhD. and Scytl’s R&S department.

---

1 Swiss Federal Chancellery. Federal Chancellery Ordinance on Electronic Voting (VEleS) of 13 December 2013. (Status as of 1 July 2018)
3 Scytl Secure Electronic Voting. Swiss Online Voting System Cryptographic proof of Individual Verifiability. 2017
2 Appendix

2.1 EV Solution Intellectual Property Rights Notice (the Notice)

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

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

2.1.1 Definitions

The following terms shall have the meanings specified below:

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

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

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

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

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

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

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

i. Key Translation Module: A mapping service that translates external IDs to internal IDs for specific entities so that external systems can integrate with sVote.
ii. Swiss Post Integration Tools: A group of applications that allow the integration between Swiss Post’s applications and sVote through file conversions.

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

2.1.2 Copyright notice

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

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

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

Report

Analysis of Cast-as-Intended and Counted-as-Recorded Verifiability for Scytl sVote Protocol using ProVerif

David Galindo* and Scytl R&S

November 27, 2018

*Crypto in Motion Ltd, UK
Acknowledgment

This report is the companion to the ProVerif modelisation of Scytl sVote Protocol led by David Galindo, PhD (Crypto in Motion Ltd) with assistance from Eike Ritter, PhD. For completeness, the source code of the main symbolic model can be found in the Appendix.
# Contents

1 Introduction ........................................... 1

2 Threat model and security goals ....................... 2

   2.1 Security Assumptions and Threat Model ............ 2

      2.1.1 Assumptions on parties ..................... 2

      2.1.2 Assumptions on communication channels ....... 2

      2.1.3 Additional assumptions on parties regarding voting secrecy 3

   2.2 Trust model in sVote ............................. 4

      2.2.1 Protocol participants in sVote ............... 4

      2.2.2 Trust assumptions in sVote ............... 5

   2.3 Correspondence between both security models ....... 5

3 Abstract Building Blocks .................................. 7

4 Abstract Model of sVote with Control Components Protocol .... 11

   4.1 Abstractions and relation to the base protocol .... 11

      4.1.1 System setup ................................ 11

      4.1.2 Election setup and voting phase .............. 12

   4.2 sVote Model in ProVerif .......................... 14

5 Cast-as-Intended Verifiability in the Presence of Dishonest Parties for sVote 22

6 Counted-as-Recorded Verifiability in the Presence of Dishonest Parties for sVote 26

7 Conclusions ............................................ 32

A ProVerif source file /sources/cast-as-intended-2ccrs-Proverifv3.1.pve before expansion 34

B ProVerif source file /sources/UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve before expansion 51
Chapter 1

Introduction

This report covers the automatic verification using ProVerif of Cast-as-Intended and Counted-as-Recorded verifiability properties of the Scytl sVote Protocol by Scytl [5]. This verification effort was led by David Galindo, PhD (Crypto in Motion Ltd) with assistance from Eike Ritter, PhD. 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 [2]. This model builds on previous models as described in [6, 7].

Organization

In Chapter 2 we discuss the security model posed by [3]. In Chapter 3 we define the building blocks used later in Chapter 4 to construct our abstraction of Scytl sVote Protocol Specifications [5]. In Chapters 5 and 6 we define our symbolic Cast-as-Intended and Counted-as-Recorded verifiability properties respectively for a bounded number of candidates $n$ and a bounded number of voting choices $\psi$, for one trustworthy Mixing Control Component and one trustworthy Return Code Control Component. In Chapter 7 we present the conclusions and the limitations of our model.

Finally, in the Appendix we include the source code of our main extended ProVerif specification. Some of these are files with extension .pve to be found in folder /sources, containing a quasi ProVerif code that allows to specify a symbolic model expressing elections of type $(\psi, n)$, where $\psi$ is the number of choices cast by a voter in the election, and the number $n$ of total voting options (or candidates). More precisely, our model is contained in the files

- sources/cast-as-intended-2ccrs-Proverifv3.1.pve
- sources/cast-as-intended-2ccrs-psi1-Proverifv3.1.pve
- sources/UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve
- sources/UniversalVerifiability2CCMs-DishonestCCM2v3.1.pve

In the Appendix the source of code of the first and third files are provided. A standard ProVerif specification (with extension .pv) can automatically be obtained from the file sources/filename**.pve by particularizing $\psi, n$ to concrete values $\psi_0, n_0$, as explained in Chapters 5 and 6. Examples of the corresponding ProVerif files can be found in the folder /analysis. In sVote the Choice Return Codes are computed interactively between trustworthy and untrustworthy components.
Chapter 2

Threat model and security goals

In this section, we derive precise, formal security goals from the informal description of the model for complete verifiability given by the Swiss Federal Chancellery. Our interpretation of the threat model is supported by quotes from and references to relevant excerpts of the Federal Chancellery’s requirements [3]. The extraction of precise properties from the legislations informally stated goals is an important step for justifying that the model used throughout the security proofs of sVote are indeed the same up to the differences in notations.

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

2.1 Security Assumptions and Threat Model

2.1.1 Assumptions on parties

According to section 4.3 of the Federal Chancellery’s requirements [3], control components, auditors and auditors’ technical aid are referred as additional system components. Similarly, section 4.3 defines additional communication channels referring to system components listed in section 4.1. Based on that and also the fact that complete verifiability in practice uses the same provisions as for individual verifiability we treat the complete abstract model defined in section 4.3 as the extension of the reduced abstract model defined in section 4.1.

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

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

2.1.2 Assumptions on communication channels

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

<table>
<thead>
<tr>
<th>System components</th>
<th>Trust assumption</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voters</td>
<td>significant proportion of voters are non-trustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>User platform</td>
<td>untrustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>Trusted technical aids for voters</td>
<td>trustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>System (server-side)</td>
<td>untrustworthy</td>
<td>4.3</td>
</tr>
<tr>
<td>Control Components</td>
<td>trustworthy only as the whole</td>
<td>4.3</td>
</tr>
<tr>
<td>Auditors</td>
<td>at least one is trustworthy</td>
<td>4.3</td>
</tr>
<tr>
<td>Auditor’s technical aid</td>
<td>at least one honest auditor has a trustworthy aid</td>
<td>4.3</td>
</tr>
</tbody>
</table>

**Table 2.1: Assumptions on parties of the complete abstract model defined in VEleS [3]**

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

<table>
<thead>
<tr>
<th>Communication channel</th>
<th>Trust assumption</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voters ↔ user platform</td>
<td>trustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>Voters ↔ trustworthy technical aids</td>
<td>trustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>Trustworthy technical aids ↔ user platform</td>
<td>trustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>User platform ↔ system</td>
<td>untrustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>System ↔ print office</td>
<td>untrustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>Print office → voter</td>
<td>trustworthy</td>
<td>4.1</td>
</tr>
<tr>
<td>Control component ↔ system</td>
<td>untrustworthy</td>
<td>4.3</td>
</tr>
<tr>
<td>System ↔ auditor’s technical aids</td>
<td>untrustworthy</td>
<td>4.3</td>
</tr>
<tr>
<td>Auditors’ technical aid ↔ auditors</td>
<td>trustworthy</td>
<td>4.3</td>
</tr>
<tr>
<td>Bidirectional channels for communication between control components</td>
<td>untrustworthy</td>
<td>4.3</td>
</tr>
</tbody>
</table>

**Table 2.2: Assumptions on communication channels of the complete abstract model defined in VEleS [3]**

### 2.1.3 Additional assumptions on parties regarding voting secrecy

According to [3, Section 4.3], voting secrecy does not account for scenarios where the attacker corrupts user platform: "Under the trust assumptions for complete verifiability of the protocol, the attacker is unable to breach voting secrecy or to obtain early provisional results without changing the voters or their user platforms maliciously."

Please note that, according to 4.4.8 [3], vote secrecy should be preserved only for trustworthy voters: “It must be ensured that the voting secrecy of trustworthy voters cannot be breached without maliciously changing their user platform through the server-sided manipulation of the application.”

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

---

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

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

2.2.1 Protocol participants in sVote

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

- **Voter**: they participate in the election by choosing their preferred options.

- **Voting Client**: is the device used by the voter to cast their vote given the voting options selected by the voters.

- **Voting Server**: it receives, processes and stores the votes cast by the voters in the ballot box BB.

- **Control Components** are separated in two groups, one is participating in choice return codes the other in mixing:
  
  - **Choice Return Codes Control Components (CCR’s)**: they collaborate with the Print office indirectly (via the Voting Server) in the setup phase, and directly with the Voting Server in the voting phase, to compute the so-called long Choice Return Codes.
  
  - **Mixing Control Components (CCM’s)**: they perform the mixing and partial decryption of the ciphertexts in the ballot box.

- **Print Office**: It is responsible for generating, printing and delivering the voting cards to the voters as well as for generating the required election keys.

- **Election Administrators**: they are responsible for generating the election configuration, verifying it, computing the results and publishing them. In the Protocol Specification this entity is divided into:
  
  - **Administration Board** and **Administration Portal**: Both components are used to set-up and sign the configuration and therefore, can perform some cryptographic operations. However for the privacy proof we do not distinguish between those two.
  
  - **Electoral Board**: This entity owns a key pair whose private key is shared among the Board members and is used to partially decrypt the votes in the last Control Component execution.

- **Global Bulletin board**: is the entity used to store all the information generated during the election to verify the entire process. It stores election configuration, votes, confirmations and keeps track of all the actions performed by each entity. The Bulletin Board is implemented as a distributed system, that includes: election configuration (maintained by Print office), Secure Logger (maintained by CCRs) and Ballot Box (maintained by Voting Server). In this document we refer to Secure Logger as CCR’s logs.

- **Electoral Board**: This entity owns a key pair whose private key is shared among the Board members and is used to partially decrypt the votes in the last Control Component execution.

- **Auditors**: they are responsible for verifying the integrity of the procedures run in the counting phase. They are a crucial part of ensuring the verifiability properties as set up by the Chancellery’s requirements, in so auditors can be leveraged to detect misbehavior.

- **Verifier**: is the component used to verify the correctness of the entire election process, the integrity of the data processed through different voting system components, and that these processes are accurate and fair.

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

Complete verifiability uses the following trust model:

- The Voting Server is not trusted. Instead, there exists two groups of so called control components CCM’s and CCR’s which interact with Voting Server directly and indirectly with Print office (via Voting Server). Each group of control components is trusted as a whole, under the assumption that at least one of them is reliable. However, each sole control component is not trusted.

- Credential delivery channel (postal channel between Print office and voters) is considered to be trustworthy.

- Print office is trusted.

- The Voting Client of honest voters is considered to be trusted for privacy, and not trusted for individual and universal verifiability.

- Initial election configuration (number and names of the candidates, number of voters, number of allowed options etc) generated by Election Administrators is assumed to be correct as the Print office has no means for verifying this information.

- The communication channel between the client side and the server side is not trusted.

- A part of the voters may not be trustworthy.

- Electoral Board is treated as set of control component and therefore is trusted as whole, i.e. at least one Electoral Board member is assumed to be trustworthy.

- At least one of the auditors and her technical aids (software or hardware tools) are trusted to behave properly.

2.3 Correspondence between both security models

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

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

Table 2.3: Correspondence between sVote and the Chancellery’s assumption made on the protocol’s participants

*this channel may only be regarded as trustworthy if the information has been sent by Swiss Post (section 4.2.9 page 23)
<table>
<thead>
<tr>
<th>sVote communication channels</th>
<th>Chancellery’s communication channel</th>
<th>Trust assumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voters ↔ Voting Client</td>
<td>Voters ↔ user platform</td>
<td>trustworthy</td>
</tr>
<tr>
<td>Voters ↔ Voting Cards</td>
<td>Voters ↔ Trustworthy technical aids</td>
<td>trustworthy</td>
</tr>
<tr>
<td>no channel exists</td>
<td>Trustworthy technical aids ↔ User platform</td>
<td>trustworthy</td>
</tr>
<tr>
<td>Voting Client ↔ Voting Server</td>
<td>User platform ↔ system</td>
<td>untrustworthy</td>
</tr>
<tr>
<td>Voting Server ↔ Print office</td>
<td>System ↔ print office</td>
<td>untrustworthy</td>
</tr>
<tr>
<td>Print office → Voter</td>
<td>Print office → voter</td>
<td>trustworthy[^1]</td>
</tr>
<tr>
<td>CCM ↔ Voting Server</td>
<td>Control component ↔ system</td>
<td>untrustworthy</td>
</tr>
<tr>
<td>CCR ↔ Voting Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voting Server ↔ Verifier</td>
<td>System ↔ auditor’s technical aids</td>
<td>untrustworthy</td>
</tr>
<tr>
<td>CCM ↔ Verifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR ↔ Verifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verifier ↔ auditor</td>
<td>Auditors’ technical aid ↔ auditors</td>
<td>trustworthy</td>
</tr>
<tr>
<td>no channel exists</td>
<td>Bidirectional channels for communication between control components</td>
<td>untrustworthy</td>
</tr>
</tbody>
</table>

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

[^1]This channel may only be regarded as trustworthy if the information has been sent by Swiss Post (section 4.2.9 page 23).
Chapter 3

Abstract Building Blocks

In symbolic models [4], the set of terms associated to a set of function symbols $F$, a set of variables $X$ and a set of names $N$ is inductively defined as the names, variables, and function symbols applied to other terms. Terms are equipped with inference rules of the form $a \vdash b$, that define which messages $b$ can be computed from an a priori given set of messages $a$. We start by describing our modelling of the basic cryptographic functions used in the protocol (we assume the reader to have some degree of familiarity with Scytl’s cryptographic specification of the voting protocol [5]). We use Sans Serif format for functions and typewriter format for variables.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Meaning</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ske$</td>
<td>$\text{agent}_id \rightarrow \text{priv}_key$</td>
<td>$ske_{id}$, the private decryption key of $\text{agent}_id$</td>
<td>private</td>
</tr>
<tr>
<td>$pube$</td>
<td>$\text{priv}_key \rightarrow \text{pub}_key$</td>
<td>builds public encryption key $\text{pke}<em>{id}$ from $ske</em>{id}$, i.e. $\text{pke}<em>{id} := \text{pub}(ske</em>{id})$</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$sks$</td>
<td>$\text{agent}_id \rightarrow \text{priv}_skey$</td>
<td>$sks_{id}$, the private signing key of $\text{agent}_id$</td>
<td>private</td>
</tr>
<tr>
<td>$pubs$</td>
<td>$\text{priv}_skey \rightarrow \text{pub}_skey$</td>
<td>builds public signing key $\text{pks}<em>{id}$ from $sks</em>{id}$, i.e. $\text{pks}<em>{id} := \text{pub}(sks</em>{id})$</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$bck$</td>
<td>$\text{agent}_id \rightarrow \text{bitstr}$</td>
<td>$\text{BCK}^{id}$, ballot casting key</td>
<td>private</td>
</tr>
<tr>
<td>$f$</td>
<td>$\text{sym}_key \times \text{bitstr} \rightarrow \text{sym}_key$</td>
<td>keyed pseudo-random function</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$pCC$</td>
<td>$\text{priv}_key \times \text{bitstr} \rightarrow \text{bitstr}$</td>
<td>partial Choice Return Codes function</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$\text{Sig}$</td>
<td>$\text{priv}_skey \times \text{bitstr} \rightarrow \text{bitstr}$</td>
<td>digital signature primitive</td>
<td>public</td>
</tr>
<tr>
<td>$\text{Enc}_a$</td>
<td>$\text{sym}_key \times \text{bitstr} \rightarrow \text{bitstr}$</td>
<td>symmetric encryption</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$\text{Enc}_{c_1}$</td>
<td>$\text{bitstr} \rightarrow \text{bitstr}$</td>
<td>randomized asymmetric encryption: first component</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$\text{Enc}_{c_2}$</td>
<td>$\text{pub}_key \times \text{bitstr} \times \text{bitstr} \rightarrow \text{bitstr}$</td>
<td>randomized asymmetric encryption: second component</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$\text{Enc}$</td>
<td>$\text{pub}_key \times \text{bitstr} \times \text{bitstr} \rightarrow \text{bitstr}$</td>
<td>$\text{Enc}(\text{pke}, m, r) := (\text{Enc}<em>{c_1}(r), \text{Enc}</em>{c_2}(\text{pke}, m, r)) = \text{ctxt}$</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$\text{tild}$</td>
<td>$\text{bitstr} \times \text{priv}_key \rightarrow \text{bitstr}$</td>
<td>$\text{tild}((c_1, c_2), \text{ske})$ is $(c_1^{\text{ske}}, c_2^{\text{ske}})$ for setup phase</td>
<td>public, non-invertible</td>
</tr>
<tr>
<td>$\text{tildPsi}$</td>
<td>$\text{bitstr} \times \text{priv}_key \rightarrow \text{bitstr}$</td>
<td>to exponentiate a ciphertext vector of $\psi$ components</td>
<td>public, non-invertible</td>
</tr>
</tbody>
</table>

Additional public non-invertible functions include:
- \( \text{mEnc}_{c_1} : \text{bitstr} \rightarrow \text{bitstr} \) (randomized asymmetric multiple encryption first component)

- \( \text{mEnc}_{c_\psi} : \text{pub}_\text{ekey} \times \ldots \times \text{pub}_\text{ekey} \times \text{nat} \times \ldots \times \text{nat} \times \text{bitstr} \rightarrow \text{bitstr} \) (randomized asymmetric multiple encryption second component)

- \( \text{mEnc} : \text{pub}_\text{ekey} \times \ldots \times \text{pub}_\text{ekey} \times \text{nat} \times \ldots \times \text{nat} \times \text{bitstr} \rightarrow \text{bitstr} \) is defined as
  \[
  \text{mEnc}((\text{pke}_1, \ldots, \text{pke}_\psi), (v_1, \ldots, v_\psi), r) := (\text{mEnc}_1(r), \text{mEnc}_{\psi}((\text{pke}_1, \ldots, \text{pke}_\psi), (v_1, \ldots, v_\psi), r)) = (\text{mctxt}_t, \text{mctxt}_s)
  \]

- \( \text{tildPsi} : \text{bitstr} \times \text{priv}_\text{ekey} \rightarrow \text{bitstr} \) builds \( \text{tildPsi}(\text{mctxt}_2, \text{ske}) := \text{mctxt}_2^{\psi} \) for computing encrypted pre-Choice Return Codes from encrypted partial Choice Return Codes

- \( \text{tildPCC} : \text{priv}_\text{ekey} \times \text{bitstr} \times \text{bitstr} \rightarrow \text{bitstr} \)

- \( \text{mergepk} : \text{bitstr} \times \text{pub}_\text{ekey} \times \text{pub}_\text{ekey} \times \text{pub}_\text{ekey} \rightarrow \text{pub}_\text{ekey} \)

- \( \text{mergeExpo} : \text{bitstr} \times \text{bitstr} \times \text{bitstr} \times \text{bitstr} \rightarrow \text{bitstr} \)

- \( \text{ mergemExpo } : \text{bitstr} \times \text{bitstr} \times \text{bitstr} \times \text{bitstr} \rightarrow \text{bitstr} \)

**Notation**

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Meaning</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELak</td>
<td>priv_ekey</td>
<td>Election private key</td>
<td>private</td>
</tr>
<tr>
<td>ELpk</td>
<td>pub_ekey</td>
<td>Election public key (= pub_ekey(ELak))</td>
<td>public</td>
</tr>
<tr>
<td>VCCsak</td>
<td>priv_skey</td>
<td>Vote Cast Return Code signing key</td>
<td>private</td>
</tr>
<tr>
<td>VCCspk</td>
<td>pub_skey</td>
<td>Vote Cast Return Code verification key (= pubs(VCCsak))</td>
<td>public</td>
</tr>
<tr>
<td>C_{\text{id}}</td>
<td>sym_ekey</td>
<td>Codes Secret Key</td>
<td>private</td>
</tr>
<tr>
<td>SVK_{\text{id}}</td>
<td></td>
<td>Start voting key associated to \text{id}</td>
<td>private</td>
</tr>
<tr>
<td>V_{\text{id}}</td>
<td>agent_{\text{id}}</td>
<td>Verification Card ID associated to \text{id}</td>
<td>public</td>
</tr>
<tr>
<td>\text{k}<em>\text{id}(\text{VCC}</em>{\text{id}})</td>
<td>priv_ekey</td>
<td>Verification Card \text{VCC}_{\text{id}}'s private key</td>
<td>private</td>
</tr>
<tr>
<td>\text{k}<em>\text{id}(\text{VCC}</em>{\text{id}})</td>
<td>pub_ekey</td>
<td>Verification Card \text{VCC}<em>{\text{id}}'s public key (= pube(\text{k}</em>\text{id}))</td>
<td>public</td>
</tr>
<tr>
<td>BCK_{\text{id}}</td>
<td>bitstr</td>
<td>Ballot Casting Key associated to \text{id}</td>
<td>private</td>
</tr>
<tr>
<td>VCCs_{\text{id}}</td>
<td>bitstr</td>
<td>Key Store associated to \text{id}</td>
<td>private</td>
</tr>
<tr>
<td>\text{iVCC}_{\text{id}}</td>
<td>sym_ekey</td>
<td>long Vote Cast Return Code for \text{id}</td>
<td>private</td>
</tr>
<tr>
<td>pVCC_{\text{id}}</td>
<td>nat</td>
<td>pre-Vote Cast Return Code associated to \text{id}</td>
<td>private</td>
</tr>
<tr>
<td>VCC_{\text{id}}</td>
<td>bitstr</td>
<td>short Vote Cast Return Code for \text{id}</td>
<td>private</td>
</tr>
<tr>
<td>S_{\text{VCC}_{\text{id}}}</td>
<td>bitstr</td>
<td>validity proof for Vote Cast Return Code \text{VCC}_{\text{id}}</td>
<td>private</td>
</tr>
</tbody>
</table>

\( J_1, \ldots, J_n \) : nat
voting options available in the election

\( J_1, \ldots, J_\psi \) : nat
voter individual \( \psi \) choices in the election

\( v \) : nat \rightarrow \text{bitstr}
\( v := v(J_1) \), the encoding of voting option (candidate) \( J_1 \)

\( \text{CC}_{\text{id}}^i \) : bitstr
\( i \)-th (short) Choice Return Code associated to \text{id}

\( \text{iCC}_{\text{id}}^i \) : bitstr
\( i \)-th long Choice Return Code associated to \text{id}

\( \text{pCC}_{\text{id}}^i \) : nat
\( i \)-th Partial Choice Return Code associated to \text{id}

\( \text{pC}_{\text{id}}^i \) : nat
\( i \)-th pre-Choice Return Code associated to \text{id}

\( \text{bb} \) : table(agent_id, bitstr)
Ballot Box, storing ballots for each agent

\( \text{cb} \) : table(agent_id, bitstr)
Confirmation Box, storing codes \( \text{VCC}_{\text{id}}^i / S_{\text{VCC}_{\text{id}}}^i \)
These objects are specific to the sVote protocol. Our notation tries to be as close as possible to the notation in [5].

**Attacker capabilities (Dolev-Yao)**

The attacker capabilities consist on:

- Computing and inverting public invertible functions, e.g.
  - \( p_{sk} \vdash id \text{ and } id \vdash p_{id} \) (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.
  - \( m \vdash h(m) \) (i.e. \( h \) is a efficiently computable hash function that cannot be inverted; in particular the rule \( h(m) \vdash m \) is not available to the attacker)
  - \( m \vdash H(m) \) (i.e. \( H \) is a special hash function that maps strings to a special subset [4]; the rule \( H(m) \vdash m \) is not available to the attacker)
  - \( sk, m \vdash f(sk, m) \) (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 \( m \vdash f(sk, m) \) is not available to the attacker)
  - \( sks, m \vdash \text{Sig}(sks, m) \) \( \forall sks, m \)
  - \( \text{Enc}(pke, m, r) \) \( \forall pke, m, r \)
  - \( \text{Enc}(sk, m) \) \( \forall sk, m \)
  - \( pke_1, pke_2 \vdash \text{mergepk}(pke_1, pke_2) \) \( \forall pke_1, pke_2 \)
  - \( ctxt_1, ctxt_2 \vdash \text{mergeExpo}(ctxt_1, ctxt_2) \) \( \forall ctxt_1, ctxt_2 \)
  - \( pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(\psi)}, v_1, \ldots, v_\psi, r \vdash m\text{Enc}\left(\left(pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(\psi)}, \left(v_1, \ldots, v_\psi\right), r\right) \right) \) \( \forall pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(\psi)} \)

- Computing the following functions associated to symmetric encryption:
  - \( \text{Dec}(sk, \text{Enc}(sk, m)) \vdash m \) \( \forall sk, m \) (i.e. decrypting a symmetric encryption of a message returns the original message)

- Computing the following functions associated to public key encryption:
  - \( \text{Dec}(ske, \text{Enc}(ske, m, r)) \vdash m \) \( \forall ske, m, r \) (i.e. decrypting an asymmetric encryption of a set of natural numbers returns the original set)

- Computing the following functions associated to digital signatures:
  - \( \text{verif}(\text{pub}(sk), m, \text{Sig}(sk, m)) \vdash \text{true} \) \( \forall sks, m \) (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 \( \text{zkp}, \text{verifP} \) be functions with the following types:
  - \( \text{zkp} : m \times (\text{pub}_{\text{ek}} \times \text{pub}_{\text{ek}} \times \text{bitstr} \times \ldots) \rightarrow \text{bitstr} \) models a non-interactive zero-knowledge proof
  - \( \text{verifP} : m \times (\text{pub}_{\text{ek}} \times \text{pub}_{\text{ek}} \times \text{bitstr} \times \ldots) \rightarrow \text{bool} \) models the verification equation of a zero-knowledge proof

*In cryptographic terms, this is the subset \( QR_N \) of Quadratic Residues modulo an RSA modulus.*
• Sequential composition of partial decryptions:

\[ \diamond \text{verifP}\left(\text{EL}_{pk}, K_{id}, \text{VC}_{id}, p_{k_{CCR}}^{(1)} \ldots, p_{k_{CCR}}^{(p)}, \left(\text{Enc}_{c_1}(r), \text{Enc}_{c_2}(\text{EL}_{pk}, (v_1, \ldots, v_\psi), r)\right),\right) \]
\[ \left(m_{\text{Enc}}_{c_1}(r'), m_{\text{Enc}}_{c_2}\left(p_{k_{CCR}}^{(1)} \ldots, p_{k_{CCR}}^{(p)}, \left(\text{Enc}_{c_1}(r), \text{Enc}_{c_2}(\text{EL}_{pk}, (v_1, \ldots, v_\psi), r)\right)\right),\right) \]
\[ \zeta_{\text{pk}}\left(\text{EL}_{pk}, K_{id}, \text{VC}_{id}, p_{k_{CCR}}^{(1)} \ldots, p_{k_{CCR}}^{(p)}, \left(\text{Enc}_{c_1}(r), \text{Enc}_{c_2}(\text{EL}_{pk}, (v_1, \ldots, v_\psi), r)\right),\right) \]
\[ \left(m_{\text{Enc}}_{c_1}(r'), m_{\text{Enc}}_{c_2}\left(p_{k_{CCR}}^{(1)} \ldots, p_{k_{CCR}}^{(p)}, \left(\text{Enc}_{c_1}(r), \text{Enc}_{c_2}(\text{EL}_{pk}, (v_1, \ldots, v_\psi), r)\right)\right),\right) \]
\[ r, r', K_{id} \right) = \text{true} \]

The equation above is aimed at capturing the three non-interactive zero-knowledge proofs (Schnorr proof, Exponentiation Proof, Plaintext Equality Proof) computed in the algorithm \text{CreateVote} as defined in [5]. Roughly speaking, the Schnorr proof proves knowledge of nonce \( r \), while the Exponentiation and Plaintext Equality proofs prove that the ciphertext \( \text{Enc}(\text{EL}_{pk}, (v_1, \ldots, v_\psi), r) \), that encrypts the voting options; the ciphertext \( m_{\text{Enc}}\left(p_{k_{CCR}}^{(1)} \ldots, p_{k_{CCR}}^{(p)}, \left(\text{Enc}_{c_1}(r), \text{Enc}_{c_2}(\text{EL}_{pk}, (v_1, \ldots, v_\psi), r)\right)\right) \) that encrypts the partial Choice Return Codes; and the voters’ verification card private key \( k_{id} \) and verification card identifier \( \text{VC}_{id} \) are all linked.

\[ \diamond \text{verifP}_1\left(\text{EL}_{pk}, \text{EL}_{pk}^{(1)}, L = \{(c_1, c_2)\}, \text{MixDec}_1(\text{EL}_{pk}^{(1)}, L = \{(c_1, c_2)\})\right) = \text{true} \]

The equation above is aimed at capturing the two non-interactive zero-knowledge proofs (Mixing Proof and Partial Decryption Proof) computed by CCM\(_1\). Roughly speaking, if the above equation is verified, then it is guaranteed that the ElGamal ciphertext list \( L \) output by the Voting Server after cleansing, and the ciphertext list \( L_1 \) output by CCM\(_1\) after mixing and partial decryption, both encrypt the same plaintext set, modulo a secret permutation, under public keys \( \text{EL}_{pk} := \text{mergepk}(\text{EL}_{pk}^{(1)}, \text{EL}_{pk}^{(2)}) \) and \( \text{EL}_{pk} \) respectively.

\[ \diamond \text{verifP}_2\left(\text{EL}_{pk}^{(1)}, \text{EL}_{pk}^{(2)}, L = \{(c_1, c_2)\}, \text{MixDec}_2(\text{EL}_{pk}^{(2)}, \text{MixDec}_1(\text{EL}_{pk}^{(1)}, L = \{(c_1, c_2)\})\) \right) = \text{true} \]

The equation above is aimed at capturing the two non-interactive zero-knowledge proofs (Mixing Proof and Decryption Proof) computed by CCM\(_2\). Roughly speaking, if the above equation is verified, then it is guaranteed that the ElGamal ciphertext list \( L \) output by the Voting Server after cleansing, and the plaintext list \( L_e \) output by CCM\(_2\) after mixing and decryption, correspond to the same plaintext set, modulo a secret permutation.

- Sequential composition of partial decryptions:

\[ \diamond \text{PDec}_2(\text{sk}_2, \text{PDec}_1(\text{sk}_1, (\text{Enc}_{c_1}(r), \text{Enc}_{c_2}(\text{mergepk}(\text{pube}(\text{sk}_1), \text{pube}(\text{sk}_2)), m, r)))) \vdash m \quad \forall \text{sk}_1, \text{sk}_2, m, r \]

(i.e. sequentially composed partial decryption of an asymmetric encryption of a set of natural numbers returns the original set)
Chapter 4

Abstract Model of sVote with Control Components Protocol

4.1 Abstractions and relation to the base protocol

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

4.1.1 System setup

In this section, we point out the abstractions and simplifications that have been done in the protocol description presented in Section 4 in relation to the implementation of Scytl sVote Protocol in an Online Voting System.

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

CA hierarchy. Constitution of a platform root CA, and generation of credentials for the different system contexts and tenants that wish to run an election is omitted. For more details, see [5, Sections 3.1, 3.2, 3.3]

Bulletin board, ballot box and logs. In sVote the ballot box is maintained by the voting server. In addition to this, both groups of control components log all the transcript they see during setup, voting and tally phases. The ballot box and the logs are later handed to the auditors. We model this situation as a distributed bulletin board $\mathcal{B}$, comprised of the ballot box denoted with $bb$, the transcript of the $j$-th choice return control component $CCR_j$ denoted with $bb^{CCR}_j$, and the transcript of the $j$-th mixing control component $CCM_j$ denoted with $bb^{CCM}_j$. Throughout this document, we sometimes refer to $bb^{CCR}_j$ and $bb^{CCM}_j$ as the secure log of the control components. For details on how transcript is logged securely see [5, Section 3.5]. For example the secure log of $CCR_j$ includes the voter’s ballot submitted by the voting device, and the part of the transcript corresponding to the generation of the choice codes and the vote cast codes.

Number of control components. For simplicity we consider two CCRs and two CCMs, whereas in [5] four components per group are specified.

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

As for CCMs, even though those components are executed in a sequence, we also claim that our reduction does not affect proof structure due to the mandatory audit performed before the last CCM is executed and the fact that the last key is distributed among members of Electoral Board.
Consider a case of $N$ CCMs where only one of them is honest. This consideration can be done without loss of generality as any other scenario can be mapped to this extreme case. According to [5], the mandatory verification would be performed after $N - 1$ CCMs shuffled and partially decrypted votes. Verification would fail if at least one of the following is true: a) cleansing procedure is not correct b) one of mixing proofs is invalid or c) one of the decryption proofs is invalid.

If the verification holds, Electoral Board members would submit their private shares so the last CCM would reconstruct its key. Taking into account that Electoral Board can be viewed as a set of ‘human control components’ at least one of the members is honest and refuses to submit the decryption key share if validation fails. Thus, the last CCM would be able to reconstruct the last CCM’s decryption key and perform the decryption process if and only if verification holds.

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Honest CCM is among first N-1</th>
<th>Verification holds</th>
<th>Honest EB members submit their keyshares</th>
<th>Last CCM knows its key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>Last CCM is corrupted, but can’t reconstruct its key.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Case 2</td>
<td>Last CCM is corrupted and can reconstruct its key.</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Case 3</td>
<td>Last CCM is honest, but can’t reconstruct its key.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Case 4</td>
<td>Last CCM is honest and knows its key.</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

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

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

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

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

4.1.2 Election setup and voting phase

**Modeling auditors and sVote logic.** In our model we assume the following statements are always true. (1) In an election of type $(\psi, n)$ the voters enter exactly $\psi$ voting options. (2) The Voting Server does not register in the ballot box two different ballots for the same voter. (3) The Voting Server queries only once the CCRs during the retrieval of the choice return codes during the voting phase. (4) Every ballot processed by the CCRs belongs to the ballot box submitted to cleansing. (5) The cleansed ballot box is the input to the CCMs. These simplifications are justified by the fact that either the checks are part of the logic implemented in sVote, or are verified by auditors before the election is declared valid.

*According to section 4.4.10 of [3], it is also permitted to implement a group of control components so that they take the form of people.*
**Offline mixing control component.** We have abstracted away the generation and recovery of the election key private key share $E_{sk}^{(2)}$ corresponding to the last CCM. In the actual implementation, during setup this private key is secret-shared using Shamir secret sharing scheme among the electoral board members, and during tally, it is reconstructed accordingly. However, this details are irrelevant in the security analysis: it only matters that the adversary gets to know the key $E_{sk}^{(2)}$ in case the last CCM is compromised. See [5, Sections 4.5, 6.1.2, Annex 9.1.9, 9.1.10] for more details.

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

**Voting options.** We denote the set of valid votes by $\Omega$, which is composed by any combination of voting options $\{v_1, \ldots, v_\psi\}$ which is valid according to the election rules. Both the set of valid votes and the counting function $\rho : (\Omega \cup \{\bot\})^* \rightarrow R$ are assumed to be defined in advance. As already mentioned, the set of possible results $R$ is given by the multiset function $\rho$, which provides the cleartext votes cast by the voters in a random order. Additionally, if write-in values are permitted in the election, cast-as-intended for the write-in content cannot be provided, since it is not possible to provide the voter with a mapping of all possible write-in values to a choice return code. Therefore, we will not cover write-in values in our security and formal analysis. Last, vote correctness is also abstracted away since with the presence of auditors voting options not corresponding to the pre-defined prime numbers will be detected at the end of the election. Clearly, tampering with the range of voting options does not affect voter’s privacy.

**Authentication.** We have not covered details on how eligible voters authenticate in the system. This might happen via a dedicated protocol, or via a third party. In any case, the focus in this document is to show that privacy is maintained regardless of authentication procedure, so how the user authenticates is considered out of the scope. This is consistent with the current state-of-the-art in computational security proofs for e-voting systems, where eligibility verifiability [8] is very rarely analyzed.

In the actual implementation, users authenticate in the system using a challenge-response mechanism. The goal of the authentication procedure is to ensure that only a voter who proved that he opened an encrypted key store gets a valid authentication token. Namely, a voter sends a request that includes his Credentials ID to the server. The server sends back the corresponding encrypted Verification Card keystore and a challenge. Voter replies by sending a client message, which includes the server’s challenge signed with the key retrieved from the encrypted keystore. If the reply to the challenge is valid, Server generates an Authentication Token containing the Voter Information, a timestamp etc. and sends it to the voter. During the voting phase, the voter going to include this token to every request it sends to the server. See [5, Sections 5.1, 5.2] for more details.

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

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

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

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

Data Initialisation

The following initialisation data is computed by the Secure Data Manager (SDM) and is stored offline until it is securely transmitted to the corresponding agents:

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

Processes

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

- $\text{GenElKey}(\text{EL}_{\text{pk}}^{(1)}, \text{EL}_{\text{pk}}^{(2)})$ is run by the SDM on inputs the Election Public Key shares $\text{EL}_{\text{pk}}^{(1)}, \text{EL}_{\text{pk}}^{(2)}$ created by CCM1, CCM2 respectively, and computes the Election Public Key as $\text{EL}_{\text{pk}} := \text{mergepk}(\text{EL}_{\text{pk}}^{(1)}, \text{EL}_{\text{pk}}^{(2)})$
- $\text{AliceData}(C_{\text{ak}})$ is run by the SDM and initialises the voter’s secret data, where $C_{\text{ak}}$ is a secret value only known to the SDM and the Voting Server.

1. $\text{VC}_{\text{id}}$ is a random Verification Card ID associated to the voter id
2. $\text{SVK}_{\text{id}}$ is a random password associated to the voter id
3. $(\text{K}_{\text{id}}, \text{k}_{\text{id}})$ is a random Verification Card key pair associated to voter id, where $\text{k}_{\text{id}} := \text{pke}(\text{k}_{\text{id}})$
4. $\text{BCK}_{\text{id}}$ is a random Ballot Casting Key associated to voter id
5. $\text{VCxs}_{\text{id}} := \text{Enc}_{\text{k}}(\text{deltaKey}(\text{SVK}_{\text{id}}), \text{k}_{\text{id}})$ is a keystore assigned to voter id
6. let $p\text{CC}_{\text{id}} := (\nu_i)^{\text{k}_{\text{id}}}$, voter id’s $i$-th partial Choice Return Code $\forall i = 1, \ldots, n$
7. let $1\text{CC}_{\text{id}} = h(\text{VC}_{\text{id}}, p\text{CC}_{\text{id}})$, voter id’s $i$-th long Choice Return Code and Choice Return Code encryption symmetric key $\text{skcc}_{\text{id}} = \delta(C_{\text{sk}}, 1\text{CC}_{\text{id}})$ for $\forall i = 1, \ldots, n$
8. let $CC_{\text{id}}$ be the short Choice Return Codes assigned at random to voter id, $\forall i = 1, \ldots, n$
9. let $1\text{VC}_{\text{id}} = h(\text{VC}_{\text{id}}, p\text{VC}_{\text{id}})$, the voter id’s long Vote Cast Return Code and the $\text{skvcc} = \delta(C_{\text{sk}}, 1\text{VC}_{\text{id}})$ the Vote Cast Symmetric key
10. let $\text{VCC}_{\text{id}}$ be the random short Vote Cast Return Code assigned to voter id

---

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

1In the Proverif model $\text{VC}_{\text{id}}$ and id are linked through the private function $\text{deltaId}()$
2In the Proverif model $\text{SW}_{\text{id}}$ and id are linked through the private function $\text{deltaPassword}()$
3In the Proverif model $\text{k}_{\text{id}}$ and id are linked through the private function $\text{skel}()$
* The function $\text{deltaKey}() = \delta(\cdot, \text{KEYseed})$, where $\delta$ is a Key Derivation Function (cf Appendix A)
1$p\text{CC}_{\text{id}}$ is called pre-Choice Return Code and is computed by the SDM from $p\text{CC}_{\text{id}} := H((B\text{ck}_{\text{id}}^{(2)} k_{\text{id}})^2)$, in interaction with the CCR’s.
2The function $\text{deltaId}()$ is a Key Derivation Function
2$\text{pVC}_{\text{id}}$ is called pre-Vote Cast Return Code and is computed by the SDM from $\text{pVC}_{\text{id}} := H((B\text{ck}_{\text{id}}^{(2)} k_{\text{id}})^2)$, in interaction with the CCR’s.
11. voter’s data is initialised as init_{id} := (VC_{id}, SVK_{id}, BC_{id}, VCC_{id},\{ J_i, CO^i_{id}\}_{i=1}^\psi)

- ServData(EL_{pk}, C_{ak}, VCCs_{ak}) is run by the SDM and initialises the Voting Server secret data as follows:
  1. let KeySt := \{VCCs_{id}\}_{id=s}^\psi, the collection of all voters’ key stores
  2. Voting Server’s data is initialised as

\[
init_S := (EL_{pk}, C_{ak}, VCCs_{ak}, VCCs_{pk} := pubs(VCCs_{ak}), CMtable, KeySt)
\]

- GetKey(SVK_{id}, VCCs_{id}) recovers the verification card private key k_{id} as follows:
  1. output k_{id} := Dec_s(deltaKey(SVK_{id}), VCCs_{id})

- CreateVote(EL_{pk}, VC_{id}, \{ J_i \}_{i=1}^\psi, K_{id}, k_{id}, \{ pk_{CCR}^{(i)} \}_{i=1}^\psi) consists of the following steps:
  1. let v_i = v(J_i) for i = 1, \ldots, \psi
  2. let V = (v_1, \ldots, v_\psi)
  3. let E1 = Enc(EL_{pk}, V, r) for a fresh nonce r
  4. let E2 = mEnc((pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(1)}), ((v_1)_{k_{id}}, \ldots, (v_\psi)_{k_{id}}), r') for a fresh nonce r'
  5. let P = zkp(EL_{pk}, K_{id}, VC_{id}, pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(1)}, E1, E2, r', k_{id})
  6. output b = (E1, E2, K_{id}, P) a ballot

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

- ProcessVoteCheck(EL_{pk}, VC_{id}, b, \{ pk_{CCR}^{(i)} \}_{i=1}^\psi) outputs a boolean and consists of the following steps:
  1. let b = (xE1, xE2, xK_{VC_{id}}, xP)
  2. output verifP(EL_{pk}, xK_{VC_{id}}, VC_{id}, pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(1)}, xP, E1, xE2, xP)

This is the algorithm used by the Voting Server and the Return Codes Control Components to test a ballot.

- CreateRC(xC_{C_1}, \ldots, xC_{C_{\psi}}, VC_{id}, C_{ak}, CMtable) might output a set of Choice Return Codes and consists of the following steps:
  1. let x1CC_{id} = h(VC_{id}|xPC_{id}), and Choice Return Code encryption symmetric key skcc_{id} = \delta(C_{ak}, x1CC_{id}) for i = 1, \psi, where C_{ak} is the Code Secret key, and xPC_{i} are the pre-choice return codes computed by the voting server.
  2. output xCC_{id} = Dec^s(Enc^s(CC_{id}, skcc_{id}); skcc_{id}) \iff [h(x1CC_{id}), Enc^s(CC_{id}, skcc_{id})] exists as an entry in CMtable \forall i \leq \psi
  3. else output void

This is the algorithm used by the Voting Server to compute the short Choice Return Codes CC_{id} to be sent to the voter’s device.

- AuditCodes((xC_{C_1}, \ldots, xC_{C_{\psi}}), (CC_{id}, \ldots, CC_{id})) outputs a boolean and consists of the following steps:
  1. output true \iff xCC_{i} \in \{CC_{id}, \ldots, CC_{id}\} \forall i = 1, \ldots, \psi

\(^*\)For our analysis we have merged the Election Context, the Voting Workflow Context and the Vote Verification Context from [3] into a single agent called Voting Server. The rationale behind it is that all three contexts could be adversarially controlled, and hence we can group them into a single agent for security purposes.

\(^{1}\)xK_{VC_{id}} as obtained from ballot b might not be equal to the legitimate k_{id}
2. else output false

This is the algorithm used by the voter to check whether all expected short Choice Return Codes 
\( \{ (J^i, \text{CCR}^i) \}_{i=1}^v \) corresponding to the voter’s intended choices \( \{J^i\}_{i=1}^v \) were indeed received:

- **Cleansing**（\( \text{EL}_{pk}, \text{pk}^{(1)}_{\text{CCR}}, \ldots, \text{pk}^{(v)}_{\text{CCR}}, \text{VCCs}_{pk}, \text{bb}, \text{cb} \)） is run by the Voting Server and proceeds as follows:
  1. let \( \text{bb} = \{ (\text{VCD}_d, \text{b}) \}_{id} \) and \( \text{cb} = \{ (\text{VCD}_d, \text{VCC}^{id}, \text{VCCs}^{id}) \}_{id} \)
  2. it creates a list \( L_{\text{checked}} \) containing all \( (\text{VCD}_d, \text{b}) \in \text{bb} \) such that ProcessVoteCheck（\( \text{EL}_{pk}, \text{pk}^{(1)}_{\text{CCR}}, \ldots, \text{pk}^{(v)}_{\text{CCR}}, \text{VCD}_d, \text{b} \)） = true and verifVCCs（\( \text{VCCs}_{pk}, \text{VCD}_d, \text{VCCs}^{id} = \text{VCCs}^{id} \） = true, where \( (\text{VCD}_d, \text{VCC}^{id}, \text{VCCs}^{id}) \in \text{cb} \)
  3. from each \( (\text{VCD}_d, \text{b}) \in L_{\text{checked}} \), obtains E1 from b, next obtains ciphertext components \( (c_1, c_2) \) from E1 (cf. [5]) and adds \( (c_1, c_2) \) to a list \( L \)
  4. outputs list \( L = \{(c_1, c_2)\} \)

- **MixDec（\( \text{EL}^{(1)}_{pk} \), \( L = \{(c_1, c_2)\} \)） is run by CCM1 and proceeds as follows:
  1. from the ciphertext list \( L = \{(c_1, c_2)\} \) computes a shuffled and partially decrypted list \( L_1 = \{(c_1^{(1)}, c_2^{(1)})\}_{\text{mix}} \) and a zero-knowledge proof \( P_n \) of correct mixing and decryption.
  2. output \( L_1 = \{(c_1^{(1)}, c_2^{(1)})\}_{\text{mix}}, P_n \)

The output above is such that if we rewrite \( L = \{\text{Enc}(\text{EL}_{pk}, V)\} \) and \( L_1 = \{\text{Enc}(\text{EL}^{(2)}_{pk}, V')\}_{\text{mix}} \), then:

\[ L_1 = \{\text{Enc}(\text{EL}_{pk}, \{V\}_{\text{mix}})\}. \]

- **MixDec（\( \text{EL}^{(2)}_{pk} \), \( L = \{(c_1, c_2)\}, L_1 = \{(c_1^{(1)}, c_2^{(1)})\}_{\text{mix}}, P_n \)） is run by CCM2 and proceeds as follows:
  1. if verifP（\( \text{EL}_{pk} \), \( \text{EL}^{(1)}_{pk}, L = \{(c_1, c_2)\}, L_1 = \{(c_1^{(1)}, c_2^{(1)})\}_{\text{mix}}, P_n \)） = false abort
  2. from the ciphertext list \( L_1 = \{(c_1^{(1)}, c_2^{(1)})\}_{\text{mix}} \) computes a shuffled and final decrypted list \( L_v = \{V\}_{\text{mix}} \) and a zero-knowledge proof \( P_n \) of correct mixing and decryption.
  3. output \( L_v = \{V\}_{\text{mix}}, P_n \).

The output above is such that if we rewrite \( L = \{\text{Enc}(\text{EL}_{pk}, V)\} \) and \( L_1 = \{\text{Enc}(\text{EL}^{(2)}_{pk}, \{V\}_{\text{mix}})\} \), then:

\[ L_v = \{\{V\}_{\text{mix}}\}_{\text{mix}}. \]

**sVote Voting Protocol - Abstraction**

In the following section we describe our abstract model for sVote and the reader is referred to [5] for specification of sVote.

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

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

0. In the **Setup phase** the Secure Data Manager (SDM), together with the Return Codes Control Components CCR1, CCR2, computes the Verification Card’s data and several system parameters:
Recall, that according to Protocol Specifications \[5\] the computation of \( \text{ctxtbck} \).

The function \( \delta \) is defined as

\[
\delta(a, b) = \begin{cases} a \cdot b & \text{if } a \cdot b \leq n \text{ and } a \neq 0, \\ 0 & \text{otherwise}. \end{cases}
\]

During the verification process, the SDM performs the following steps for each voter identifier \( \text{Vcid} \):

1. Assigns a random Verification Card identifier \( \text{VCid} \).
2. Assigns \( \text{SVKid} \) as a start voting key.
3. Assigns a random Verification Card secret key \( k_d \) to verification card \( \text{Vcid} \).
4. Encrypts under the Secure Data Manager public key \( \text{pk}_{SDM} \) the complete set of voting options \( \{J_1, \ldots, J_n\} \) as

\[
\text{ctxtv}_i := (\text{Enc}(\text{pk}_{SDM}, v(J_1)), \ldots, \text{Enc}(\text{pk}_{SDM}, v(J_n)))
\]

with fresh nonces \( r_1, \ldots, r_n \).
5. Chooses a random Ballot Casting Key \( \text{Bck} \) and computes

\[
\text{ctxtbck} := \text{Enc}(\text{pk}_{SDM}, H((\text{Bck}^2)k_d), r)
\]

for a fresh nonce \( r \).

CCRs’ data is initialized as \( \text{init}_{CCR} = (\psi, \text{pk}_{SDM}, (\text{ctxtv}_1^i, \ldots, \text{ctxtv}_n^i, \text{ctxtbck}^i))^{id} \).

for the Secure Data Manager public key \( \text{pk}_{SDM} := \text{pube}(\text{sk}_{SDM}) \).

The Secure Data Manager secret key \( \text{sk}_{SDM} \) corresponding to \( \text{pk}_{SDM} \) is only known to the SDM, while \( \psi \) stands for the number of voting options that a voter needs to choose for the current election.

Each Choice Return Code Control Component \( \text{CCR}_i \) runs \( \text{SetupCCR}(i, \text{init}_{CCR}) \) which is defined as follows:

\[\text{SetupCCR}(i, \text{init}_{CCR}):\]

1. Let \( \text{init}_{CCR} = (\psi, \text{pk}_{SDM}, (\text{ctxtv}_1^i, \ldots, \text{ctxtv}_n^i, \text{ctxtbck}^i))^{id} \).
2. Computes Control Components Choice Return Codes key pair \( \text{pk}_{CCR}(i, j)^{id} := \text{pube}(\text{sk}_{CCR}(i, j)^{id}) \) for \( j = 1, \ldots, \psi \).
3. Computes a Choice Return Codes Encryption key pair \( (K_i, k_i) \) as \( K_i := \text{pube}(k_i) \).
4. Computes the following for each Verification Card \( \text{Vcid} \):
   a. Derives a Voter Choice Return Code Generation private key \( k_i^i := \delta(k_i, \text{Vcid}) \) and a public key \( K_i := \text{pube}(k_i^i) \).
   b. Derives a Voter Vote Cast Return Code Generation private key \( k_c(i, \text{Vcid}) := \delta(k_c, \text{Vcid}) \) and a public key \( K_c(i, \text{Vcid}) := \text{pube}(k_c(i, \text{Vcid})) \).
   c. Computes \( \text{tild}((\text{ctxtv}_1^i, k_i^i), \ldots, \text{tild}((\text{ctxtv}_n^i, k_i^i)) \).
   d. Computes \( \text{tild}((\text{ctxtbck}^i, k_i^i)) \).

v. Sets

\[
\text{init}_{SDM}^i := (K_i, \{(\text{pk}_{CCR}(i, j), k_i^i, \text{tild}((\text{ctxtv}_1^i, k_i^i)), \ldots, \text{tild}((\text{ctxtv}_n^i, k_i^i)), \text{tild}((\text{ctxtbck}^i, k_i^i))\})^{id}
\]

That outputs \( \text{init}_{SDM}^i \) and key pairs \( (\text{pk}_{CCR}(i, 1)^{id}), \ldots, (\text{pk}_{CCR}(i, \psi)^{id}), (k_i, k_i^i) \) and \( (k_i, k_i^i) \) for \( i = 1, \ldots, n \).

For each Verification Card identifier \( \text{Vcid} \), the SDM computes

\[
\text{ctxtpc}(i, \text{Vcid}) := \text{tildPCC}(k_d, \text{mergeExpo}(\text{tild}((\text{ctxtv}_1^i, k_i^i)), \text{tild}((\text{ctxtv}_n^i, k_i^i))))
\]

for \( i = 1, \ldots, n \).

\textbf{Recall, that according to Protocol Specifications} \[5\], the computation of \( \text{ctxtbck}^i \) should have been the following:

\[
\text{ctxtbck} := \text{Enc}(\text{pk}_{SDM}, H((\text{Bck}^2)k_d), r)
\]

where \( H \) is a hash function, and the squaring is used to map into a set of quadratic residues. In our model this has been captured by using a special hash function \( H(\cdot) \).

In the Proverif model, \( k_i \) is obtained through \( \delta(\cdot, k_d) \).

The function \( \delta(\cdot, k_d) \) is defined as \( \delta(a, b) = a \cdot b \) if \( a \cdot b \leq n \) and \( a \neq 0 \), otherwise \( \delta(a, b) = 0 \).

In the Proverif model, \( k_i \) is obtained through \( \delta(\cdot, k_d) \).

In the Proverif model, \( k_i \) is obtained through \( \delta(\cdot, k_d) \).

The function \( \delta(\cdot, k_d) \) is defined as \( \delta(a, b) = a \cdot b \) if \( a \cdot b \leq n \) and \( a \neq 0 \), otherwise \( \delta(a, b) = 0 \).
ii. \( \text{ctxtbck}^{id} := \text{mergeExpo}(\text{tild}(\text{ctxtbck}^{id}, kC_i^{id}), \text{tild}(\text{ctxtbck}^{id}, kC_{id}^{id})) \)

iii. pre-Choice Return Codes as

\[
(pC_i^{id}, \ldots, pC_n^{id}) := (\text{Dec}(sk_{SDM}, \text{ctxtpc}^{id}_1), \ldots, \text{Dec}(sk_{SDM}, \text{ctxtpc}^{id}_n))
\]

iv. the pre Vote Cast Code as \( pVCC^{id} = \text{Dec}(sk_{SDM}, \text{ctxtbck}^{id}) \)

v. long Choice Return Codes \( lCC_i^{id} = h(VC_i^{id}|pC_i^{id}) \) and symmetric key \( skcC_i^{id} = \delta(C_{sk}, lCC_i^{id}) \)

vi. long Vote Cast Return Code \( lVCC^{id} = h(VC^{id}|pVCC^{id}) \) and symmetric key \( skvcc = \delta(C_{sk}, lVCC^{id}) \)

vii. assign short Choice Return Codes \( CC_i^{id} \subseteq \{cCC \} \) from her Voting Device, where \( VCC_{ks} \)

viii. the mapping between long and short codes

\[
\text{map}^{id} = \left( \left\{ [h(1CC_i^{id}), \text{Enc}_s(skcC_i^{id}, CC_i^{id})] \right\}_{i=1}^{n}, [h(1VCC^{id}), \text{Enc}_s(skvcc, VCC^{id}|S_{VCC^{id}})] \right)
\]

where \( S_{VCC^{id}} := \text{Sig}(VCC_{sk}, VCC^{id}) \) is the validity proof for the short code \( VCC^{id} \)

x. let \( CMtable := \{\text{map}^{id}\}^{id}_S \)

1. The SDM initialises the voter’s data \( \text{init}_A \) and the server’s data \( \text{init}_S \) offline:

   • The SDM runs \( AliceData(C_{sk}) \) to initialise Alice’s secret data as \( \text{init}_A := (VC_A, SVK_A, BCK^A, VCC_A, \{(J_i, CC_i^A)\}_{i=1}^{n}) \), where \( CC_i^A \) is Alice’s expected short Choice Return Code for the choice \( J_i \) for each \( i \in \{1, \ldots, n\} \)

   • The SDM runs \( ServData(EL_{pk}, C_{sk}, VCC_{sk}) \) to initialise the Voting Server’s secret data for the election as

\[
\text{init}_S := (EL_{pk}, C_{sk}, VCC_{sk}, VCC_{pk} := pubs(VCC_{sk}), CMtable, KeySt),
\]

   where

\[
CMtable = \left\{ \left\{ [h(1CC_i^{id}), \text{Enc}_s(skcC_i^{id}, CC_i^{id})] \right\}_{i=1}^{n}, [h(1VCC^{id}), \text{Enc}_s(skvcc, VCC^{id}|S_{VCC^{id}})] \right\}^{id}_S
\]

   • \( init_{DEV} := (EL_{pk}, pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(v)}) \) is distributed to the Voting Devices

Roughly speaking, the information stored in \( CMtable \) will allow the Voting Server to compute a set of short Choice Return Codes corresponding to any voter id choices \( \{J_i^{id}\} \) 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 \( \text{init}_A \), enters her verification card identifier \( VC_A \) and her voting choices \( \{J_i^A\} \subseteq \{J_i\} \) in her Voting Device, followed by her password \( SVK_A \). Alice’s voting choices are pairwise different and there are exactly \( v \) of them. Crucially, Alice keeps \( \{(J_i, CC_i^A)\}_{i=1}^{n} \) and \( VCC_A \) secret from her Voting Device, where \( CC_i^A \) stands for Alice’s short Choice Return Code for voting option \( J_i \).

3. Alice’s Voting Device, after establishing an authenticated connection with the Voting Server, sends Alice’s verification card ID \( VC_A \) to the Voting Server. The Voting Server on input \( VC_A \) retrieves Alice’s key store as \( VCC_{sk} \) and sends it to Alice’s Voting Device.

4. Alice’s Voting Device obtains Alice’s Verification Card private key \( k_A \) by running \( \text{GetKey}(SVK_A, VCC_{sk}) \).

   Next the Voting Device runs \( \text{CreateVote}(EL_{pk}, VC_A, \{J_i^A\}^{v}_i, k_A, kA, \{pk_{CCR}^{(1)}, \ldots, pk_{CCR}^{(v)}\}) \), obtaining a ballot

\[
b = (E1, E2, KA, P)
\]

that is received by the Voting Server. The ballot consists of several parts: \( E1 \) is an encryption of Alice’s voting choices \( \{J_i^A\}^{v}_i; E2 \) allows the Voting Server to compute the (short) Choice Return Codes corresponding to Alice’s vote; the remaining components are used to prove consistency between ciphertexts \( E1 \) and \( E2 \).

18
5. The Voting Server runs \texttt{ProcessVoteCheck(El}_{\text{pk}}, \text{VC}_{\text{id}}, b, \{\text{pk}_{\text{CCR}_i}^{(1)}\}_{i=1}^\psi ) 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 \text{VC}_A in the ballot box \bb, then the server adds an entry \((\text{VC}_A, b)\) to \bb and starts the following interaction with the CCR’s in order to compute a set of short Choice Return Codes \{\text{xCC}_i\}_{i=1}^\psi that are sent back to Alice’s Voting Device:

(a) Let \( b = (x_{E1}, x_{E2}, x_{K_{VC_A}}, x_P) \) and parse

\[
\begin{align*}
x_{E2} &= \left( \text{mEnc}_{c_1}(r), \text{mEnc}_{c_\psi}\left( (\text{pk}_{\text{CCR}_1}^{(1)}, \ldots, \text{pk}_{\text{CCR}_\psi}^{(\psi)}), (M_1, \ldots, M_\psi), x' \right) \right).
\end{align*}
\]

Each Choice Return Code Control Component CCR\(_i\) for \( i = 1, \ldots, 2 \) sends back to the Voting Server

\[
\begin{align*}
\text{E2Expo}_{i}^{(\psi)} &= \left( \text{mEnc}_{c_1} \left( \text{tildNat}(k_i^A, r) \right), \right. \\
&\text{mEnc}_{c_\psi} \left( (\text{pk}_{\text{CCR}_1}^{(1)}, \ldots, \text{pk}_{\text{CCR}_\psi}^{(\psi)}), (\text{tildNat}(k_i^A, M_1), \ldots, \text{tildNat}(k_i^A, M_\psi)), \text{tildNat}(k_i^A, x') \right) \right)
\end{align*}
\]

(b) The Voting Server computes

\[
\text{RAll} := \text{mergemExpo} \left( \text{mEnc}_{c_1} \left( \text{tildNat}(k_1^A, r) \right), \text{mEnc}_{c_1} \left( \text{tildNat}(k_2^A, r) \right) \right)
\]

and

\[
\text{E2ExpoAll} := \text{mergemExpo}(\text{E2Expo}_{1}^{(\psi)}, \text{E2Expo}_{2}^{(\psi)})
\]

(c) Each Choice Return Code Control Component CCR\(_i\) for \( i = 1, \ldots, 2 \) locally computes and sends to the Voting Server \( \text{tildNat}(sk_{\text{CCR}_i}^{(1)}, \text{RAll}), \ldots, \text{tildNat}(sk_{\text{CCR}_i}^{(\psi)}, \text{RAll}) \)

(d) Voting Server computes partial decryptors

\[
\text{pdec}_j := \text{randomMult} \left( \text{tildNat}(sk_{\text{CCR}_1}^{(j)}, \text{RAll}), \text{tildNat}(sk_{\text{CCR}_2}^{(j)}, \text{RAll}) \right)
\]

for \( j = 1, \ldots, \psi \)

(e) Voting Server computes pre-Choice Return Codes as \( \text{xpC}_j := \text{PartialDec} \left( \text{pdec}_j, (\text{RAll}, \text{E2ExpoAll}) \right) \)

for \( j = 1, \ldots, \psi \)

(f) Voting Server computes Choice Return Codes \((\text{xCC}_1, \ldots, \text{xCC}_\psi)\) by running \text{CreateRC}(\text{xpC}_1, \ldots, \text{xpC}_\psi, \text{VC}_A, c_{ak}, \text{CMtable})

6. Alice’s Voting Device displays to her the set of Choice Return Codes \{\text{xCC}_i\}_{i=1}^\psi received from the Voting Server. Alice then runs the human computable algorithm \text{AuditCodes}((\text{xCC}_1, \ldots, \text{xCC}_\psi), (\text{CC}_1^A, \ldots, \text{CC}_\psi^A)), that checks Alice’s expected return codes \{(\text{J}_i^A, \text{CC}_i^A)\}_{i=1}^\psi against the codes \{\text{xCC}_i\}_{i=1}^\psi displayed by the Voting Device

7. If \text{AuditCodes}((\text{xCC}_1)_{i=1}^\psi, (\text{CC}_1^A)_{i=1}^\psi) outputs true then Alice enters her Ballot Casting Key \text{BCK}^A in the Voting Device, which computes and forwards to the Voting Server the Confirmation Message \text{pCM}^A := (\text{BCK}_A)^{2\cdot x}.

8. After some checks, the Voting Server communicates with the CCR’s so that

\^Recall, that according to Protocol Specifications \( \mathcal{E} \), \text{pCM}^A should have been the following: \( \text{pCM}^A := (\text{BCK}_A)^{2\cdot x}. \) However, since modular squaring operation is not an atomic operation in Proverif, it has been omitted.
Each Choice Return Code Control Component CCR, for $i = 1, \ldots, 2$ sends to the Voting Server the exponentiation $H(pCM^A)^{kc_i^A} = H((BCK)^A)^{kc_i}$.

Voting Server computes $xpVCC^A := \text{merge}((pCM^A)^{kc_1^A}, (pCM^A)^{kc_2^A})$.

Let long vote cast return code $x1VCC^A = h(VC_A||xpVCC^A)$ and symmetric key $xskvcc^A = \delta(csk, x1VCC^A)$.

Voting Server sends to Voting Device $xVCC := \text{Dec}_s(xskvcc^A, \text{Enc}_s(skvcc^A, VCC^A))$ if the entry $\{h(x1VCC), \text{Enc}_s(skvcc^A, VCC^A)\}$ exists in $CMtable$.

else Voting Server outputs void.

9. The Voting Device displays $xVCC$ to Alice, who then checks if $xVCC = VCC^A$.

10. Once the election is closed, the Voting Server proceeds to do the cleansing of the ballot box by running $\text{Cleansing}(EL_{pk}^{(1)}, pk_{CCR}^{(1)}; \ldots; pk_{CCR}^{(n)}, VCC_{pk}, bb, cb)$. Let $L = \{(c_1, c_2)\}$ be the output list of encrypted voting options obtained as a result.

11. On input $L = \{(c_1, c_2)\}$, Mixing Control Component CCM$_1$ runs the mixing and partial decryption algorithm $\text{MixDec}_1(EL_{sk}^{(1)}, L = \{(c_1, c_2)\})$ and outputs

$$L = \{(c_1, c_2)\}, L_1 = \{(c_{1}^{(1)}, c_{2}^{(1)})\}_{mix}, p_n,$$

consisting of the original ciphertexts list $L$, a list $L_1$ of shuffled and partially decrypted ciphertexts together with $p_n$ a zero-knowledge proof of correct mixing and partial decryption.

12. On input $(L = \{(c_1, c_2)\}, L_1 = \{(c_{1}^{(1)}, c_{2}^{(1)})\}_{mix}, p_n)$, Mixing Control Component CCM$_2$, which is run by the Electoral Board, computes the final mixing and partial decryption algorithm

$$\text{MixDec}_2(EL_{sk}^{(2)}, L = \{(c_1, c_2)\}, L_1 = \{(c_{1}^{(1)}, c_{2}^{(1)})\}_{mix}, p_n)$$

and outputs

$$L = \{(c_1, c_2)\}, L_1 = \{(c_{1}^{(1)}, c_{2}^{(1)})\}_{mix}, L_v = \{\overrightarrow{v}\}_{mix}, p_n, p_a,$$

consisting of the original ciphertexts list $L$, the list $L_1$ of shuffled and partially decrypted ciphertexts together with $p_a$ a zero-knowledge proof of correct mixing and partial decryption by CCM$_1$, a list $L_v$ of voting choices $\overrightarrow{v} = (v_1, \ldots, v_\psi)$ cast by voters, and $p_a$ a zero-knowledge proof of correct mixing and partial decryption by CCM$_2$.

Events

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

- $\text{InsertBB}(VC_{id}, \text{bitstring})$ is reached when the Voting Server registers an entry bitstring for voter id in the Ballot Box $bb$.
- $\text{InsertBBccr}(VC_{id}, \text{bitstring})$ is reached when the Control Component CCR$_1$ registers an entry bitstring for voter id in the Control Component Ballot Box $bbccr$.
- $\text{Confirmed}(init_A, J_1, \ldots, J_\psi)$ is reached (or holds) iff Alice has checked the short Choice Return 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 $VC_A$. However, Alice has not yet finalised her vote.

\footnote{Recall, that according to Protocol Specifications, CCR’s response should have been the following: $H(pCM^A)^2 \cdot k_i$, where the squaring is used to map into a set of quadratic residues. In our model this has been captured by using a special hash function $H()$.}

\footnote{In $pCM^A$ is called CM$^A$}
Figure 4.1: Vote Cast Workflow of sVote Protocol

- **HasVoted**(init, VC, b, VCC, S) is reached when the Voting Server has processed Alice’s ballot and Ballot Casting Key and thus should have accepted it if behaving honestly (however the Voting Server is untrustworthy).

- **HappyUser**(init, J, \( J_1, \ldots, J_n \)) is reached when Confirmed(init, J) = true and Alice has checked and accepted the short Vote Cast Return Code displayed by her Voting Device. Thus, from Alice’s point of view she has finalised her vote.

- **BallotConfirmationProcessed**(VCid, b) is reached when InsertBBccr1(VCid, bitstring) is reached and the CCR1 processes the vote confirmation request with respect to ballot b on behalf of VCid.

- **HappyAuditor**(bb, cb, L, L1, Ln) is reached when auditor verifies that the cleansing was done correctly and that the zero-knowledge proofs are correct and that Ln is the result that was published.
Chapter 5

Cast-as-Intended Verifiability in the Presence of Dishonest Parties for sVote

Our Cast-as-Intended verifiability definition needs to take the form of a relaxed version of the Cast-as-Intended verifiability property analysed and proven in [6, 7], where the Voting Server was considered trustworthy.

Roughly speaking, we deem an adversary successful against the Cast-as-Intended verifiability property in the case of dishonest Voting Server, Choice Return Code Component CCR\(_2\) and Control Mixing Component CCM\(_2\) and honest Choice Return Code Component CCR\(_1\) and Control Mixing Component CCM\(_1\) only if for each voter \(i_d\) at most one entry \((V_{id}, b)\) appearing in CCR\(_1\)’s internal ballot box \(bb_{ccr_1}\). Thus we impose that a successful adversary only sends one ballot \(b_{id}\) per voter \(i_d\) to CCR\(_1\) to be processed.

Attacker model

In the dishonest parties scenario, an attacker is given the following capabilities:

- It can schedule an unbounded number of dishonest voters
- It can schedule several honest voters
- It can choose the choices \(J_1, \ldots, J_\psi\) for every voter
- It controls all corrupted Voting Devices and corrupted voters
- It controls the Voting Server
- It controls one of Choice Return Code Control Component, say CCR\(_2\)
- It controls one of Control Mixing Component, say CCM\(_2\)
- It has access to CCR\(_2\), CCM\(_2\)’s private data

An attacker does not:

- Control the Secure Data Manager nor the uncorrupted Voting Devices
- Have access to honest voters private audit data
- Control the trustworthy Choice Return Code Control component, say CCR\(_1\)
• Control the trustworthy Control Mixing Component, say CCM2
• Have access to CCR1, CCM2’s private data
• Importantly, cannot prevent CCR1 from logging every request to compute Choice Return Codes/Vote Cast Return Code they receive from the Voting Server, resulting in bbccr1 a log of entries (VCid, b)

We now consider two honest voters, and their 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. We set the unique honest voter id choices for the election to be publicly known constants Jid1, ..., Jidψ.

Our (relaxed) Cast-as-Intended verifiability property captures now the fact that the attacker that can only store one ballot per honest voter in the secure log bbccr1 may permute the votes of the honest voter when creating the ballot. Each possible reordering of the votes will be captured by a InsertBBCCR(·) event in the formulation of the property. In particular, if the honest control component CCR1 did not misbehave, then all InsertBBCCR(·) events must be equal, since only one ballot could be cast for each id. It follows that the ballot stored in secure log bbccr1 contains all the honest voter’s choices.

Verifiability properties

Property Cast-as-Intended

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

\[
\text{HappyUser}(\text{init}_i, J_1, \ldots, J_\psi) \text{ is reached } \Rightarrow \forall i \in 1..\psi, (\text{id}_i, b_i) \in \text{bbccr}_1 \\
\text{AND } \forall i \in 1..\psi, b_i = (E1_i, E2_i, K_{id}_i, P_i) \\
\text{AND } \forall i \in 1..\psi, E1_i = \text{Enc}(EL_{pk}, V_i, r_i) \\
\text{AND } \forall i \in 1..\psi, V_i = (v_i^1, \ldots, v_i^\psi) = (v(J_1^i), \ldots, v(J_\psi^i)) \\
\text{AND } \forall i \in 1..\psi, \{J_1^i, \ldots, J_\psi^i\} = \{J_1, \ldots, J_\psi\} \text{ as multisets}
\]  

Notice that the Cast-as-Intended property guarantees that if HappyUser(initid, J1id, ..., Jψid) is reached for an honest voter id, then there can be up to ψ entries (VCid, bi) ∈ bbccr1 for ∀i ∈ 1..ψ, such that for each intended voting option J ∈ {J1id, ..., Jψid} cast by the voter, there exists one ballot in bbccr1 that contains J, for this voter id. Finally, if it can be guaranteed that only one entry (VCid, b) exists in bbccr, then that ballot contains the voter’s intended voting options J1id, ..., Jψid, modulo ordering. The Proverif code corresponding to this is given in Figure 5.1 and can be found in the file /sources/cast-as-intended-2ccrs-Proverifv3.1.pve

Main process

Given that our analysis 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 and start its sub-process A with the initialization data from AliceData(·) and its pre-established voting choices J1, ..., Jψ
3. define and start concurrently an unbounded number of dishonest voters by giving their initialization data to the intruder
4. give the intruder control of the Voting Server and Control Component CCR2

The corresponding extended Proverif code (i.e. dependant on the value of ψ) is given in Figure 5.2
(* Cast-as-Intended Property : The Voting Server registers a vote for the honest voter if and only if the voter confirmed his vote to him. *)

(* Not achievable in the presence of a dishonest Voting Server *)

(* Cast-as-Intended Property (Relaxed) : The honest CCR registers a vote for the honest voter if and only if the voter confirmed his vote to him. *)

query Id:agent_id, Csk:symmetric_key, E2:bitstring, ELpk:public_ekey, VCCssk:private_skey, P:bitstring, R:nat,
VCCid:bitstring, E1:bitstring, S_VCC:bitstring, {J$i: nat |$i =1..$psi},
{E2$j: bitstring, R$j: nat, P$j: bitstring |$j =1..$psi | ,
},
VCid:agent_id;

event HappyUser(AliceData(honest(Id), Csk), {J$i: nat |$i =1..$psi}) ==>
VCid = deltaId(honest(Id)),

(* & & E1 = (Enc_c1(R),Enc_c2(ELpk,((v(J$i ) |$i =1..$psi)),R))
(* Linking J1..Jpsi to j1..jpsi : There may exist k ballots containing every j in j1..jpsi ; Equal since Single-vote *)

(* Consequently, if the voting choices in HappyUser(…) do not match the voter’s intentions, then the Server misbehaved. *)

{ & & event(InsertBBCCR(VCid, ((Enc_c1(R$j),Enc_c2(ELpk,((v(J$i |$i =1..$psi)),R$j)),
E2$j, pube(ske(VCid)),P$j)) |$j =1..$psi | \n }

& & {{(J$i |$j = j$s |$i =1..$psi | | |)} |$j =1..$psi | & & }

Figure 5.1: Process for Cast-as-Intended Relaxed from file /sources/cast-as-intended-2ccrs-Proverifv3.1.pve with comments inline

Results

Our Proverif model of the modified sVote verifies the (Relaxed) Cast-as-Intended Verifiability property. For the threat model considered in this section, ProVerif has automatically verified that the property Cast-as-Intended defined in Figure 5.1 holds for elections of type $(\psi, n)$, for small values of $\psi, n$ where $1 \leq \psi < n \leq 5$, against an attacker controlling every voter except for two honest voters, and controlling control components CCR$_2$, CCM$_1$ (alternatively CCR$_2$, CCM$_4$), the Voting Server and all Voting Devices.

Examples of Proverif files containing the source code for the automatic verification of the relaxed Cast-as-Intended verifiability property can be found at analysis/. For instance the files

```
analysis/cast-as-intended-2ccrs-Proverifv3.1.pv.CCR=2.n=4.psi=2.pv
analysis/cast-as-intended-2ccrs-Proverifv3.1.pv.CCR=2.n=2.psi=2.pv
```

contain the Proverif source code for verifying the relaxed Cast-as-Intended verifiability property in our model for $n = 4, \psi = 2$ and $n = 2, \psi = 2$. The first file is obtained by running the command

```
analysis$ ../expand/expand cast-as-intended-2ccrs-Proverifv3.1.pve CCR=2 n=4 psi=2
```

The file sources/cast-as-intended-2ccrs-Proverifv3.1.pve serves as a ’root’ model file from which the Cast-as-Intended verifiability model can be generated.

On the other hand, the file

```
analysis/cast-as-intended-2ccrs-psi1-Proverifv3.1.pv.CCR=2.n=2.psi=1.pv
```

contains the Proverif source code for verifying the relaxed Cast-as-Intended verifiability property in our model for $n = 2, \psi = 1$. This file is obtained by running the command

```
analysis$ ../expand/expand cast-as-intended-2ccrs-psi1-Proverifv3.1.pve CCR=2 n=2 psi=1
```

The files sources/cast-as-intended-2ccrs-psi1-Proverifv3.1.pve serves as a ’root’ model file from which the cast-as-intended verifiability model for $\psi = 1$ can be generated.
8. Main process -- initiates the election

process
(* Public output from Setup(...) -- Gives the election's parameters to the Intruder. *)
let ELpk = mergepk(pube(ELSk1), pube(ELSk2)) in
out(c, ELpk); out(c, vccspk);

(* Gives the honest voter's public data to the Intruder *)
out(c, deltaId(honest(idA)));
out(c, deltaId(honest(idB)));
out(c, pube(skSDM));

(* setup keys for CCRS *)
out(c, {{pube(skCCR$i$j) |$i =1..2} |$j =1..$ psi}});
out (c, {{pube(kCCR$i$)|$i =1..2}});

(* Roles for honest voter(s) -- one for Individual Verifiability. *)
Alice( Ch1, AliceData(honest(idA),csk),{ja$i|$i =1..$ psi} ) |
out(c, ske(deltaId(honest(idA)))))

(* Dishonest voter(s) : As many as possible for verifiability. *)
| !(new id: agent _id; out(c,id); out(c, AliceData(id,csk)))

(* Bulletin Board (ie. Voting Server) : is Dishonest *)
| out(c, ServData(ELpk,csk,vccssk))

(* SDM is honest *)
| !SDM(SDM_CCRChannel, {ja$i|$i =1..$n}, csk, vccssk)

(* CCR1 is honest *)
| !CCR(CCRId1, kCCR1, SDM_CCRChannel) | comm

(* Cleansing is computed honestly; secret permutation is implicitly applied here *)
| !VoteServer(deltaId(honest(idA)), deltaId(honest(idB)),ELpk)

(* CCM1 is dishonest; CCM2 is honest; Auditor is honest *)
| !CCM2 | !Auditor

Figure 5.2: Main process in /sources/cast-as-intended-2ccrs-Proverifv3.1.pve for Cast-as-Intended (Relaxed)
Chapter 6

Counted-as-Recorded Verifiability in the Presence of Dishonest Parties for sVote

Let an election be of type \((\psi, n)\) if the total number of options (candidates) 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 \(\psi\) of those choices different pairwise.

We consider two different types of voters:

- Honest voters with compromised Voting Devices
- Corrupted (or dishonest) 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 \(\psi\) pairwise different valid voting options \(J_{id}^1, \ldots, J_{id}^\psi\);
- 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 \(BCK_{id}\) as displayed on her private audit data after confirming her ballot.

Voter \(id\)'s Voting Device \(VD_{id}\) is honest (uncompromised, non-corrupted) iff its behaviour corresponds to the behaviour described in the protocol specification. \(VD_{id}\) is a compromised Voting Device or dishonest or corrupted if it behaves otherwise.

Attacker model

In the dishonest Voting Server scenario, an attacker is given the following capabilities:

- It can schedule an unbounded number of dishonest voters
- It can schedule several honest voters
- It can choose the choices \(J_1, \ldots, J_\psi\) for every voter
- It controls all corrupted Voting Devices and corrupted voters
- It controls the Voting Server
- It controls one of Choice Return Code Control Components, say \(CCR_2\)
• It controls one of Mixing Control Components, say CCM₁ (alternatively CCM₂)
• It has access to CCM₁, CCR₂’s private data (alternatively CCM₂, CCR₂’s private data)

An attacker does not:
• Control the Secure Data Manager nor the uncorrupted Voting Devices
• Have access to honest voters private audit data
• Control the trustworthy Choice Return Code Control component, say CCR₁
• Control the trustworthy Mixing Control component, say CCM₁ (alternatively CCM₂)
• Have access to CCR₁, CCM₁’s private data (alternatively CCR₁, CCM₂’s private data)
• Importantly, cannot prevent CCR₁ from logging every request to compute Choice Return Codes/Vote Cast Return Code they receive from the Voting Server, resulting in \( \text{bbccr}_1 \) a log of entries \((\text{VC}_i, b)\)

Our verifiability properties consider a number \(n\) finite of voting options. Our verifiability properties are dependent on \(\psi\), i.e. the number of voting choices that a voter casts in an election of type \((\psi, n)\).

To ease the analysis, we now consider only two honest voters, and their voting choices are constant and not chosen by the adversary. The adversary controls an unbounded number of other dishonest voters and the honest voters’ Voting Devices. We set the unique honest voters’ ids \(i \in \text{id}_A, \text{id}_B\) choices for the election to be publicly known constants \(J_{\text{id}_A}^1, \ldots, J_{\text{id}_A}^{\psi}\) and \(J_{\text{id}_B}^1, \ldots, J_{\text{id}_B}^{\psi}\).

---

Our counted-as-recorded verifiability property assumes the existence of a trustworthy party called Auditor that we leverage to limit the malicious actions that a dishonest Voting Server can perform. To be deemed successful, an attacker in our model must produce a protocol run in which the event \(\text{HappyAuditor}(\text{bb}, \text{cb}, L, L_1, L_v)\) is reached. This event is reached when auditor verifies that the cleansing was done correctly and that the zero-knowledge proofs are correct and that \(L_v\) is the result that has been published.

The above forces the attacker to produce a ballot box \(\text{bb}\) such that:
• There are no multiple entries \((\text{VC}_i, b)\) for a given \(\text{VC}_i\)
• Every entry \((\text{VC}_i, b)\) satisfies \(\text{ProcessVoteCheck}(..., \text{pk}_{CCM_i}^{(\psi)}; \text{VC}_i, b) = \text{true}\)
The attacker in our Proverif model is forced to generate a ballot box containing one ballot per each one of the honest voters $id_A, id_B$ and one ballot for a dishonest voter $id$. The adversary is also forced to apply the Cleansing faithfully in order to be considered successful in attacking the counted-as-recorded verifiability property. The corresponding Proverif code can be seen in Figure 6.1.

The process for the actions of CCM\textsubscript{1} follows:

\begin{verbatim}
let CCM1 =
in(CCM1Channel, (E1:bitstring, (E11:bitstring, E12:bitstring)));
let Mix1E1 = PDec1(ELSk1, E1) in
let Mix1E11 = PDec1(ELSk1, E11) in
let Mix1E12 = PDec1(ELSk1, E12) in
out(CCM2Channel, ((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), PMix1(ELSk1, (E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12)))
.
\end{verbatim}

where
1. $L := (E1,E11,E12)$, i.e. the Cleansed Ballot Box
2. $L_1 := (Mix1E1, Mix1E11, Mix1E12)$, i.e. the output ciphertext list from mixing and partial decryption by CCM\textsubscript{1}
3. $P_m := PMix1(ELSk1,(E1,E11,E12),(Mix1E1, Mix1E11, Mix1E12))$, e.g. the proof of correct mixing and decryption

The process for the actions of CCM\textsubscript{2} follows:

\begin{verbatim}
let CCM2 =
in(CCM2Channel, ((E1:bitstring, E11:bitstring, E12:bitstring), (Mix1E1:bitstring, Mix1E11: bitstring, Mix1E12:bitstring), MixProof1:bitstring));
if VerifP1(mergepk(pube(ELSk1), pube(ELSk2)), pube(ELSk1), (E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), MixProof1) = true then
let Mix2E1 = PDec2(ELSk2, Mix1E1) in
let Mix2E11 = PDec2(ELSk2, Mix1E11) in
let Mix2E12 = PDec2(ELSk2, Mix1E12) in
out(c, ((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12), PMix2(pube(ELSk1),ELSk2, (E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12)))
;
event HappyAuditor((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12))
.
\end{verbatim}

where
1. $L_o := (Mix2E1, Mix2E11, Mix2E12)$, i.e. the output ciphertext list from mixing and final decryption by CCM\textsubscript{2}
2. $P_m := PMix2(ELPk1,ELSk2,(E1,E11,E12),(Mix1E1,Mix1E11,Mix1E12),(Mix2E1,Mix2E11,Mix2E12)))$, e.g. the proof of correct mixing and decryption

To be aligned with [9], our model forces the attacker to produce a result and intermediate proofs such that in each (intermediate) step of mixing and decryption the corresponding zero-knowledge proofs are valid (e.g. pass the corresponding verification tests). In the following the Counted-as-Recorded verifiability property is formally stated.

**Counted-as-Recorded Verifiability Property**

Formally, *Counted-as-Recorded* is captured by asserting the following property for every trace at any moment and for every set of honest voters $id_A, id_B$ and a corrupted voter $id$:
Figure 6.2: Proverif query from file /sources/UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve for Counted-as-Recorded property with comments inline

HappyAuditor(bb, cb, L := \{E1, E11, E12\}, L_1, L_2 := \{V1, V11, V12\}) is reached \implies (6.1)

E1 = Enc(EL_{pk}, V1, x) \AND E11 = Enc(EL_{pk}, V11, r_1) \AND E12 = Enc(EL_{pk}, V12, r_2)

AND V1 = (v(J_1^{d_A}), ... , v(J_1^{id_B})) \AND V11 = (v(J_1^{d_A}), ... , v(J_1^{id_B})) \AND V12 = (v(J_1^{id_B}), ... , v(J_1^{id_B}))

AND (id, b_1) \in bb \AND (id_A, b_{11}) \in bb \AND (id_B, b_{12}) \in bb

AND b_1 = (E1, E2, K_{id}, P_1) \AND b_{11} = (E1, E2', K_{id}, P_{11}) \AND b_{12} = (E1, E2'', K_{id}, P_{12}) \implies (6.2)

The first part is read as: if for a legitimately cleansed Ballot Box bb made up of the ballots submitted by any two honest voters \(id_A, id_B\), whose choices for the election are \(J_1^{d_A}, ... , J_1^{id_A}\) and \(J_1^{id_B}, ... , J_1^{id_B}\) respectively, and an additional ballot from a corrupted voter id, and the event \(\text{HappyAuditor}(bb, cb, L_1, L_2)\) is reached, then there exist ciphertexts \(\text{Enc}(EL_{pk}, (v(J_1^{d_A}), ... , v(J_1^{id_A})), r_1)\) and \(\text{Enc}(EL_{pk}, (v(J_1^{id_B}), ... , v(J_1^{id_B})), r_2)\) in the Cleansed Ballot Box on behalf of voters \(id_A, id_B\) respectively, containing the corresponding voting choices. Furthermore, those voting choices will be part of the final count of the election.

The ProVerif code corresponding to this is given in Figure 6.2 and can be found in the files

sources/UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve
sources/UniversalVerifiability2CCMs-DishonestCCM2v3.1.pve

Results

For the threat model considered in this section, ProVerif has automatically verified that the property \(\text{Recorded-as-Cast}\) defined in Figure 6.2 holds for elections of type \((\psi, n)\), for small values of \(\psi \geq 2\) and \(\psi \leq n \leq 5\), against an attacker controlling every voter except for two honest voters, and controlling control components \(\text{CCR}_2, \text{CCR}_1\) (alternatively \(\text{CCR}_2, \text{CCR}_2\)), the Voting Server and all Voting Devices.

Examples of 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/UniversalVerifiability2CCMs-DishonestCCM1v3.1.CCR=2.n=5.psi=3.pv

contains the ProVerif source code for verifying the security properties of our model for \(n = 5, \psi = 3\), and it is obtained by running the command

analysis$ ../expand/expand UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve CCR=2 n=5 psi=3

(* 8. Main process -- initiates the election *)

process
  (* Public output from Setup(..) -- Gives the election's parameters to the Intruder. *)
  let ELpk = mergepk (pube(ELSk1), pube(ELSk2)) in
  out(c, ELpk); out(c, vccspk);

  (* Gives the honest voter's public data to the Intruder *)
  out(c, deltaId(honest(idA)));
  out(c, deltaId(honest(idB)));
  out(c, pube(skSDM));

  (* setup keys for CCRS *)
  out(c, ({ { pube(skCCR$i$)} |$i =1..2} |$j =1..$psi)});
  out (c, ({pube(kCCR$i$)}|$i =1..2)));

  (* setup public keys for CCMs *)
  out(c, (pube(ELSk$i$))|$i =1..2)));

  (* Roles for honest voter(s) -- one for Individual Verifiability. *)
  Alice( Ch1, AliceData(honest(idA),csk),{ja$i|$i =1..$psi} ) |
  Alice( Ch1, AliceData(honest(idB),csk),{jb$i|$i =1..$psi} ) |
  out(c, ske(deltaId(honest(idA))))

  (* Dishonest voter(s) : As many as possible for verifiability. *)
  | !(new id: agent _id; out(c,id); out(c,AliceData(id,csk)))

  (* Bulletin Board (ie. Voting Server) : is Dishonest *)
  (* | !Serv(Ch2,CompServerChannel, ServData(ELpk,csk,vccssk),Ch 2) *)
  | out(c, ServData(ELpk,csk,vccssk))

  (* SDM is honest *)
  | !SDM(SDM_CCRChannel, {ja$i|i=1..$n}, csk, vccssk)

  (* CCR1 is honest *)
  | !CCR(CCRId1, kCCR1, SDM_CCRChannel) | comm
  (* Cleansing is computed honestly; secret permutation is implicitly applied here *)
  | !VoteServer(deltaId(honest(idA)), deltaId(honest(idB)),ELpk)
  (* CCM1 is dishonest; CCM2 is honest; Auditor is honest *)
  | !CCM2 | !Auditor

Figure 6.3: Main process in /sources/UniversalVerifiability2CCMs-DishonestCCMv3.1.pve for Counted-as-Recorded property
The file sources/UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve serves as a 'root' model file from which the Counted-as-Recorded model for an untrustworthy CCM₁ can be generated (alternatively for an untrustworthy CCM₂ the file is sources/UniversalVerifiability2CCMs-DishonestCCM2v3.1.pve).
Chapter 7

Conclusions

VEleS security objectives

We believe that our interpretation of the security objectives fits with our model and defined properties. Informally, these objectives state that, under the trust assumptions of Section 2, an attacker is unable to change, misappropriate, or cast a vote with a (possibly) compromised voting device, voting server and all but one control components (of each group). Furthermore, this should hold independently of whether or not the vote has been registered by a trusted component of the system. A careful look at the definitions of the events HappyUser and HappyAuditor shows that all the mentioned cases are covered by these two events.

Limitations of Our Analysis

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

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

- In our abstraction we have modelled 2 CCRs instead of 4 CCRs, as well as 2 CCMs instead of 4 CCMs. This has helped enormously our work with the model and significantly improves the readability of the model and this report.

- We have not made use in our model of a series of non-interactive zero-knowledge proofs present in the sVote specification that are aimed at limiting the adversarial capabilities. We have proven that these proofs are not needed to prove any of the verifiability properties. However they can be very useful to audit whether the Return Codes Control Components deviated from their expected behaviour.

- In sVote the Choice Return Codes are computed interactively between trustworthy and untrustworthy components. This has prevented us from reusing the previous modelisation for an unbounded number of voting options n.
Bibliography


Appendix A

ProVerif source file /sources/cast-as-intended-2ccrs-Proverifv3.1.pve before expansion

1 (* ProVerif specification for Scytl’s sVote Voting Protocol V3.1 - September 2018 *)
2 (* file: Scytl "sVote Protocols Specifications v5.1" *)
3 (* Page numbers in comments below refer to file "sVote Protocols Specifications v5.1" *)
4 (* Sections in comments below refer to crypto paper "sVote with Control Components Voting
Protocol - Computational Proof of Complete Verifiability" *)
5 (* Properties evaluated in this file: (Relaxed) Cast-as-Intended Verifiability property.
*)
6 (* This specification models 2 CCR’s (to be tested only with $CCR=2$), 2CCM’s, an universe of
$n$ voting options and voter choosing $\psi > 1$ pairwise different voting options *)
7 (* Trust assumptions: *)
8 (* CCR1, CCM2 and Alice are trustworthy *)
9 (* CCR2, CCM1 and Voting Server and Voting Device are untrustworthy *)
10 (* RESULT *)
11 (* (Relaxed) Cast-as-Intended Verifiability property IS verified *)
12
13 set ignoreTypes = true .
14 set verboseClauses = short .
15 set verboseCompleted = true .
16 (* set debugOutput = true . *)
17 (* 1. Objects and Types *)
18
19 type agent_id. fun t_agent_id(agent_id) : bitstring [data,typeConverter].
20 (* The type for any IDs like e.g. voting card. *)
21 type password . fun t_password(password) : bitstring [data,typeConverter].
22 (* The type for user passwords. *)
23 type nat . fun t_nat(nat) : bitstring [data,typeConverter].
24 (* The type for natural numbers. *)
25 free c : channel . (* Public Channel for communications with the
intruder # public *)
26 free Ch1 : channel . (* Channel between Alice and her Voting
Device # public *)
27 free SDM_CCRChannel : channel [private] . (* Communication between SDM
and honest CCR # private *)
28 free CompServerChannel : channel . (* Channel between Server
and Voting Device # public *)
29 free AliceSDMChannel : channel [private] . (* Channel in which SDM
receives Alice’s pseudonymous identity idA # private *)
30 free UserChannel : channel [private] .
31 free VoteServerChannel : channel [private] .
32 fun v(nat) : bitstring [data] . (* The function from voting
choices to voting options # public , invertible *)
table bb(agent_id, bitstring) . (* The Ballot Box, storing the ballots for each agent # private *)

free idA, idB : agent_id . (* Defines two SVK for two honest voters Alice and Bob # public *)

free {ja$i,jb$i|i=1..$n} : nat . (* Voting choices for the Honest voters -- REA or OBS # public *)

free {a$i|i=1..$psi} : nat . (* MODELING : an indexed set of nat atoms, for disquality tests -- only in Test. *)

free {CCRId$i|i=1..1+1}: nat . (* 2 CCRs *)

free Ch2 : channel [private] .

free CCM1Channel, CCM2Channel : channel . (* Channels for inputs/outputs of CCM1, CCM2 respectively *)

fun hash(bitstring): bitstring .

(* Encryption scheme -- ElGamal/ElGamal with multiple encryptions -- Key pairs generated through Gen_e, which is implicit. *)

type private_ekey . fun t_private_ekey(private_ekey): bitstring [data, typeConverter]. (* The type for private decryption keys + type converter. *)

type public_ekey . fun t_public_ekey(public_ekey): bitstring [data, typeConverter]. (* The type for public encryption keys + type converter. *)

fun ske(agent_id) : private_ekey [private]. (* The private key associated to an agent_id # private *)

fun VCske(agent_id) : private_ekey [private]. (* The verification card secret key kid *)

fun pube(private_ekey) : public_ekey . (* The function to rebuild a public key from the private # public, noninvertible *)

letfun pke(Id: agent_id) = pube(ske(Id)) . (* The public enc. key associated to an agent_id # public, invertible *)

fun CCRReturnKey(nat, nat): private_ekey . (* The Choice Return Codes Encryption private key *)

free { { skCCR$i$j|i=1..1} |$j =1..$ psi } : private_ekey [private] . (* Choice Return Codes Encryption secret keys for honest CCR -- page 65 *)

free {skCCR$2}|$j=1..$psi} : private_ekey . (* Choice Return Codes Encryption secret keys for dishonest CCR *)

free {kCCR$i| i=1..1} : private_ekey [private] . (* Secret key for KDF computations honest CCR *)

free kCCR2: private_ekey . (* Secret key for KDF computations dishonest CCR *)

free {kCCR$i'|i=1..1} : private_ekey [private] . (* Secret key for KDF computations honest CCR *)

free kCCR2': private_ekey . (* Secret key for KDF computations dishonest CCR *)

free skSDM: private_ekey [private] .

free ELSk1: private_ekey . (* Election decryption key share held by CCM1 # public *)

free ELSk2: private_ekey [private] . (* Election decryption key share held by CCM2 # private *)

fun Enc_c1( nat) : bitstring . (* The c1 part of the asymmetric encryption function with explicit random number. *)

fun Enc_c2(public_ekey, bitstring, nat) : bitstring . (* The c2 part of the asymmetric encryption function with explicit random number. *)

letfun Enc(Pk: public_ekey, M: bitstring, R: nat) = (Enc_c1(R), Enc_c2(Pk, M, R)) . fun mEnc_c1( nat) : bitstring [data] . (* The c1 part of the asymmetric encryption function with explicit random number. *)
fun mEnc_cPsi (bitstring, bitstring, nat) : bitstring [data]. (* The c2 part of the asymmetric encryption function with explicit random number. *)

letfun mEnc (Pk: bitstring, M: bitstring, R: nat) = (mEnc_c1(R), mEnc_cPsi (Pk, M, R)).

(* Encryption *)

reduc forall Sk: private_ekey, M: bitstring, R: nat; Dec (Sk, (Enc_c1 (R), Enc_c2 (pube (Sk), M, R))) = M. (* Decryption *)

reduc forall Pk: public_ekey, M: bitstring, R: nat; VerifE (Pk, (Enc_c1 (R), Enc_c2 (Pk, M, R))) = true. (* Checks key *)

reduc forall Id: agent_id; Get_id (pube (ske (Id))) = Id. (* Extract Id *)

(* Signature scheme -- from sVote specification page 143 -- expected to be the RSA Probabilistic Signature Scheme (PSS) -- Gen_s implicit. *)

type private_skey. fun t_private_skey (private_skey): bitstring [data, typeConverter]. (* The type for private signing keys + type converter. *)

type public_skey. fun t_public_skey (public_skey): bitstring [data, typeConverter]. (* The type for public signing keys + type converter. *)

fun sks (agent_id): private_skey [private]. (* The private sig. key associated to an agent_id # private *)

fun pubs (private_skey): public_skey. (* The function to rebuild a public signing key from the private # public, noninvertible *)

letfun pk (id: agent_id) = pubs (sks (id)). (* The public sig. key associated to an agent_id # public, invertible *)

fun Sign (private_skey, bitstring): bitstring. (* The digital signature function with explicit random number. (Signature) *)

reduc forall Sk: private_skey, M: bitstring; Verify (pubs (Sk), M, Sign (Sk, M)) = true. (* Signature Verification *)

reduc forall Sk: private_skey, M: bitstring; Checks (pubs (Sk), Sign (Sk, M)) = M. (* More expressive than Verify - not used *)

reduc forall Sk: private_skey, M: bitstring; Get_Message (Sign (Sk, M)) = M. (* Worst case for modeling only - not used *)

(* Modeling of the Non-Interactive Zero-Knowledge Proofs of Knowledge when voting -- from sVote specification, page 89, item 4. -- Names changed for abstractions. *)

(* ' Schnorr Proof ' and 'Knowledge of encryption exponent ' and ' Plaintext Equality Proof ' are abstracted inside the *)

(* ZKP and VerifP operators. Consequently, the c1 and c2, which are intermediate values, are not needed anymore and thus abstracted away. *)

fun pCC (private_ekey, bitstring): bitstring. (* The built of partial Choice Return Codes (kID,vI) = vI<kID # public, noninvertible *)

fun tild (private_ekey, bitstring): bitstring. (* To exponentiate ciphertexts generated by function Enc # public, noninvertible *)

fun tildPCC (private_ekey, bitstring): bitstring. (* To exponentiate ciphertexts generated by functions Enc_c1 and mEnc_c1 # public, noninvertible *)

fun tildNat (private_ekey, nat): nat. (* To exponentiate ciphertexts generated by functions Enc_c1 and mEnc_c1 # public, noninvertible *)

fun ZKP (public_ekey, public_ekey, (public_ekey)[i=1..<psi], bitstring, bitstring, bitstring, nat): private_ekey. (* Modeling of the proof generation *)

reducforall ELpk: public_ekey, KVCId: public_ekey, (pk_CCR[i]:public_ekey [i=1..<psi]), (J[i]: nat [i=1..<psi]), R: nat, R': nat, vcid: agent_id; VerifP (ELpk, pubk (ske (vcid))), vcid, (pk_CCR[i] [i=1..<psi]), (Enc_c1 (R'), mEnc_cPsi ((pk_CCR[i] [i=1..<psi]), (pcC (ske (vcid), v(J[i])) [i=1..<psi])), R')), ZKP (ELpk, pubk (ske (vcid))), (pk_CCR[i] [i=1..<psi]), (Enc_c1 (R'), Enc_c2 (ELpk, ((v(J[i]) [i=1..<psi]), R'))), (mEnc_c1 (R'), mEnc_cPsi (((pk_CCR[i] [i=1..<psi]), (pcC (ske (vcid), v(J[i])) [i=1..<psi])), R'))), R', Ske (vcid)) = true.

(* Next follows a series of functions that model modular multiplications that take place at several parts of sVote protocol. *)

(* Different functions names imply they are applied to different input Types. *)

fun randomMult (nat, nat): nat.

fun mergepk ((public_ekey)[i=1..<2]): public_ekey.
fun mergemExpo (bitstring, bitstring): bitstring.
fun mergeExpo (bitstring, bitstring): bitstring.

reduc forall sk1: private_ekey, sk2: private_ekey, M: bitstring, R: nat; DecMerge (sk1, sk2, (Enc_c1(R), Enc_c2(mergepk (pube (sk1), pube (sk2)), M, R))) = M.

reduc forall kId1: private_ekey, kId2: private_ekey, M: bitstring, R: nat, {sk1$i: private_ekey |$i = 1..$psi}, {sk2$i: private_ekey |$i = 1..$psi}, {J$i: nat |$i = 1..$psi}, kVCId: private_ekey;

(* Rule A - rule to compute pCId_i's in the CreateCC phase (5.4.3 CreateCC algorithm, page 94) *)
partialDec ({{ randomMult (tildNat (sk1$i, mergemExpoNat (tildNat (kId 1, R), tildNat (kId 2, R)))), tildNat (sk2$i, mergemExpoNat (tildNat (kId 1, R), tildNat (kId 2, R)))) |$i = 1..$psi}),
(mEnc_c1(mergemExpoNat(
  tildNat (kId 1, R),
  tildNat (kId 2, R))))

({mergeExpo (tildPCC (kId 1, (pCC (kVCId, v(J$i))))), tildPCC (kId 2, (pCC (kVCId, v(J$i)))))} |$i = 1..$psi).
reduc forall sk: private_ekey, kVCId: private_ekey, kId1: private_ekey, kId2: private_ekey, {R$i : nat |$i = 1..$n}, M: bitstring, {J$i: nat |$i = 1..$n};

(* Rule B - rule to compute pCId_i's in the Setup phase (page 67) *)
(* Rule A + Rule B imply that pCId_i's coincide in both Setup and CreateCC phases *)
DecPCC (sk,
tildPCC (kVCId,
mergeExpo (tild (kId 1,
(Enc_c1 (R$i), Enc_c2 (pube (sk), v(J$i)), R$i)) |$i = 1..$n)),
tild (kId 2,
(Enc_c1 (R$i), Enc_c2 (pube (sk), v(J$i)), R$i)) |$i = 1..$n)))

(* Key and IDs derivation scheme *)
(...)

Let fun ELpk = pke\(\text{election}\) . (* The Electoral Board public key in pair (ELpk,ELsk) # public *)

Change - ELpk is built in a two-party way, ELsk is not used *)

(* The Codes secret key 'Csk' (for f(...) function) # private *)

Let fun vccskpk = pks\(\text{signature}\) . (* The Vote Cast Code Signer public key in (VCCspk,VCCskpk) # public *)

Let fun vccssk = sks\(\text{signature}\) . (* The Vote Cast Code Signer private key in (VSSpk,VCCspk) # private *)

(* Register(...,id,csk,vccssk) ie. the registration step for any new voter -- done off-line through functions -- from crypto paper section 4.3.2 *)

Let fun bck(agent_id) : bitstring \[private\] . (* The Ballot Casting Key 'BCK^id' of an agent id # private *)

Fun \(\text{CC(symmetric_key,agent_id,bitstring)}\) : bitstring \[private\] . (* The short choice code 'CC^id' of an agent id + voting option # private *)

Fun \(\text{VCC(symmetric_key,agent_id)}\) : bitstring \[private\] . (* The short vote cast code 'VCC^id' of an agent id # private *)

Fun \(\text{CM\_table(bitstring)}\) : bitstring \[unbounded\] -- opened with readCC\(\text{VCC}()\) . (*)
fun VCM_table(bitstring) : bitstring (* The Codes Mapping Table ('CM_id', unbounded) -- opened with readCC/VCC(). *)
reduce forall Csk:symmetric_key, VCid:agent_id, J:nat, pC:bitstring;
readCC(H1((VCid, pC))), CCM_table(Enc_s(delta((Csk,H1((VCid, pC))))) , CC(Csk, VCid,v(J)))
reduce forall Csk:symmetric_key, VCid:agent_id, CMId:bitstring, vCCId:bitstring, VCCSk:private_skey;
(* Enc_s(VCC, VCC) *)
readVCC(H1(H1((VCid, CMId))), VCM_table(Enc_s(delta((Csk, H1((VCid, CMId))))) , (vCCId, Sign(VCCSk, VCC(Csk, VCid)))))) = Enc_s(delta((Csk, H1((VCid, CMId)))), (vCCId, Sign(VCCSk, VCC(Csk, VCid))))

(* For any voter id, generates VCIdpk, VCIdsk ....... : pke(id) and ske(id) and GetVCks(id) -- distributed in encrypted private data. *) (* For any voter id, chooses a ballot casting key ... : bck(id) as defined above; -- distributed in AliceData and Mapping. *)
(* 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.*)
(* For any voter id, computes the Vote Cast Code VCC(id) ................ : with f(..) and pCC (..) and bck(id) -- distributed in AliceData and Mapping. *)
(* For any voter id, chooses the short CC and short VCC .............. : with CC(..) and VCC(..) -- distributed in AliceData and Mapping. *)
(* For any voter id, computes the signature of the Vote Cast Code .... : Sign(vccssk, VCC(id)) -- distributed in ServData. *)
(* For any voter id, stores a list of hashed choice codes {H(lCC(id,i))}_i=1..inf ...... : through MakeRFList -- distributed in ServData. *)

(* Registration data for any voter id -- AliceData(svk, csk) produced by the SDM for Alice -- opened with a reduction *)
fun AliceData(agent_id,symmetric_key) : bitstring [private].
reduce forall Csk:symmetric_key, id_Alice:agent_id, {J$i:nat|i=1..$psi};
GetAliceData(AliceData(id_Alice,Csk),{J$i|$i=1..$psi}) = (id_Alice, deltaId(id_Alice), deltaPassword(id_Alice), bck(id_Alice), VCC(Csk, deltaId(id_Alice)), {CC(Csk, deltaId(id_Alice), v(J$l))|$l=1..$psi}).
(* Note: The set of CC(id,v(J)) is a selection of CC corresponding to the {J$i|i=1..$psi}, decided at Alice's initialisation. *)

(* Registration data for the Voting Server (Bulletin Board) -- ServData(pke, csk, sks) produced by the SDM -- opened with a reduction *)
fun ServData(public_ekey,symmetric_key,private_skey) : bitstring [private].
reduce forall ELpk:public_ekey, Csk:symmetric_key, VCCssk:private_skey, id_Voter:agent_id;
GetServData(ServData(ELpk,Csk,VCCssk),id_Voter) = (ELpk,Csk, pubs(VCCssk), Enc_s(deltaKey(deltaPassword(id_Voter)), t_private_ekey(ske((id_Voter))))), Sign(VCCssk,VCC(Csk, id_Voter))

(* 4. Algebraic properties and List of Events *)
event Confirmed(bitstring, {nat|$i=1..$psi}). (* Issued by the voter when he/she confirms his/her vote. *)
event HappyUser(bitstring, {nat|$i=1..$psi}). (* Issued by the voter when he/she terminates successfully. *)
event AliceVoted(agent_id). event PKsReceived(agent_id). event StageReached(bitstring). event SetupCompleted(agent_id). event SDMCompleted(agent_id). event VoteReceived( agent_id, bitstring). (* Issued by the server when vote has been received *)
event InsertBB( agent_id, bitstring). (* Issued by the server when it adds something in BB. *)
event InsertBBCCR( agent_id, bitstring). (* Issued by the CCR when it adds something in BB. *)
event BallotConfirmationProcessed(agent_id, bitstring).
event HasVoted (bitstring, agent_id, bitstring, bitstring, bitstring). (* Issued by the server when it gets a voter's confirmation. *)

event NeverTrue. (* An event that is never activated, and thus, never true. *)

event Results (nat|$i=1..$psi|, nat|$i=1..$psi|, nat|$i=1..$psi|). (* Issued by the Tally when it publishes the results. *)

(* 5. Methods and Agents processes *)

(* GetKey(SVKid,VCksid) -- Computer retrieves the Verification Card private key from the keystore -- step 5.3 in page 87 of sVote specification *)

letfun GetKey (SVKid: password, VCksid: bitstring) =
    let KSpwd = deltaKey (SVKid) in
    let t_private_ekey (VCidsk: private_ekey) = Dec_s (deltaKey (SVKid), VCksid) in
    VCidsk.

(* CreateVote (ELpk, VCid, Vopt, VCidpk, VCidsk, pkCCRs, R, R') -- Computer creates the ballot -- from specification starting in page 87 *)

letfun CreateVote (ELpk: public_ekey, VCid: agent_id, {J$i: nat |$i =1..$ psi}, KVCId: public_ekey, kVCId: private_ekey, { pkCCR $i: public_ekey |$i =1..$ psi}, R: nat, R': nat) =
    let V = ({v(J$i)@$i=1..$psi} in let E1 = Enc (ELpk, V, R) in
    let E2 = mEnc ({pkCCR $i@$i=1..$psi}, {pCC (kVCId, v(J$i))@$i=1..$psi}, R') in
    let P = ZKP (ELpk, KVCId, {pkCCR$i@$i=1..$psi}, E1, E2, R, R', kVCId) in
    (E1, E2, KVCId, P).

(* ProcessVoteCheck (ELpk, VCid, B, pkCCRs) -- checks the ballot *)

letfun ProcessVoteCheck (ELpk: public_ekey, VCid: agent_id, B: bitstring, {pk_ CCR $i@$i=1..$ psi}) =
    let (xE1: bitstring, xE2: bitstring, xKVCId: public_ekey, xP: bitstring) = B in
    let Ok1 = VerifP (ELpk, xKVCId, VCid, {pk_ CCR $i@$i=1..$ psi}, xE1, xE2, xP) in
    true.

(* ConfirmVCid, VCidsk, BCKid) -- Computer generates a Confirmation Message -- Done directly inside 'Cmp' and 'Alice_Cmp'. *)

(* AuditBallotProof ({CC_Received$i@$i=1..$psi}), {CCid$i@$i=1..$psi}) -- Alice checks if all expected CC were indeed received. *)

(* ProcessConfirm (bb, VCid, pVCCid, Csk, VCCssk, S_VCCid) -- Server checks the retrieved short Vote Cast Code. *)

(* Typing of the messages -- between Voter and his Computer only *)

(* Alice -- The client process *)

let Alice (Ch1: channel, InitData: bitstring, {J$i: nat |$i=1..$psi}) =
    (* Checks that the voting choices are all different *)
    (* Retrieves registration data obtained from the SDM -- Set of initial data given to Alice by the SDM. *)
    (* No honest voter can use twice the same option *)
    (* Typing of the messages -- between Voter and his Computer only *)
    free mAC1, mAC2, mCA1, mCA2: bitstring.
    free decryptors, CodeMapping, EncVotingOptions, ExpoVotingOptions, EncBCK, ExpoBCK, pkCCRTag, pkall, VCIdTag, E2tag, id_VoterTag, VCKidTag, CMTag, CodeTags, VCCidTag, CCR_CMTag, vCCidTag, BallotTag, RAllTag, EZExpoTag: bitstring.

(* Alice -- The client process *)

let Alice (Ch1: channel, InitData: bitstring, {J$i: nat |$i=1..$psi}) =
    (* Checks that the voting choices are all different *)
    { if {J$i = J$j & a$i <> a$j}@$j=1..$psi | | } then 0 else @$i=1..$psi\n
    (* Retrieves registration data obtained from the SDM -- Set of initial data given to Alice by the SDM. *)

    (* Voting part -- The voting process followed by agent Alice *)
    out (Ch1, (mAC1, VCid, SVKid, {J$i|@$i=1..$psi}));
    in (Ch1, (mCA1, {CC_Received$i|@$i=1..$psi}));
    (* event StageReached ({CCid$i@$i=1..$psi}); *)
if \(((CC\_Received{\text{i}}=CCid{\text{j}})\|{\text{i}=1..\psi}\|\|{\text{j}=1..\psi})\&\&\) then (* Compares the short Choice Codes. *)

\text{event Confirmed(InitData, \{J{i}|{\text{i}=1..\psi}\};)

out(Ch1, (mCA2,BCKid));

in( Ch1, (=mCA2,=VCCid)); (* Alice checks the Vote Cast Code's value. *)

\text{event HappyUser(InitData, \{J{i}|{\text{i}=1..\psi}\})

(* Computer -- The Alice's computer *)

let Cmp(Ch1: channel, ServChannel: channel, ELpk: public_ekey) =

new R: bitstring;

new R': bitstring;

in( Ch1, (=mAC1,Vcid: agent_id, SVKid: password,\{J{i}|{\text{i}=1..\psi}\})); (* The Start Voting Key plus the Voting options. *)

out(ServChannel, (VcidTag, Vcid));

(* Send the Voting Card ID to the Bulletin Board *)

in(ServChannel, (=pkall, (pk\_CCR{i}|{\text{i}=1..\psi})));

(* event PKsReceived (Vcid); *)

in( ServChannel, (=VCskidTag, VCksid: bitstring)); (* Receives the asso. Verification Card keystore *)

let kVCid : private_ekey = ske(Vcid) in

out(c, kVCid);

let B: bitstring = CreateVote(ELpk, Vcid,\{J{i}|{\text{i}=1..\psi}\}, pube(kVCid), kVCid, \{pk\_CCR{i}|{\text{i}=1..\psi}\}, rand2Nat(R, Vcid), rand2Nat(R', Vcid)) in

event AliceVoted(Vcid);

out(ServChannel, (BallotTag, B));

(* Sends the ballot (ie. 'Vote') to the server. *)

in( ServChannel, (=CodeTags, CC\_Set: bitstring)); (* Transmits the Choice Codes to the voter. *)

out(Ch1, (mCA1,CC\_Set));

in( Ch1, (=mAC2,BCKid:bitstring));

out(ServChannel, (CMTag, pCC(kVCid,BCKid))); (* The Confirm(Vcid,_,VCidsk ,BCK) voter's method *)

in( ServChannel, (=VCCidTag, VCCid: bitstring)); (* Transmits the Vote Cast Code to the voter. *)

out(Ch1, (mCA2,VCCid))

(* MODELING -- Alice plus her Computer together (if both honest, to avoid useless secure communications). *)

let Alice\_Cmp(InitData: bitstring,\{J{i}|{\text{i}=1..\psi}\},Ch2: channel,ELpk: public_ekey,\{pkCCR{i}: public\_ekey|{\text{i}=1..\psi}\}) =

new R: bitstring;

new R': bitstring;

(* Checks that the voting choices are all different *)

\{ if \{(J{i} = J{j} \&\& a{i} <> a{j})\|{\text{j}=1..\psi}\|\|\} then 0 else \{J{i}=1..\psi\}\n\}

(* No honest voter can use twice the same option *)

(* Retrieves registration data obtained from the SDM -- Set of initial data given to Alice by the SDM. *)

let (Vcid: agent_id, SVKid:password, BCKid:bitstring, VCCid:bitstring, \{CCid{i}:bitstring |{\text{i}=1..\psi}\}) = GetAliceData(InitData,\{J{i}|{\text{i}=1..\psi}\}) in

(* Voting part -- The voting process followed by agent Alice *)

out(Ch2, Vcid); (* Send the Voting Card ID to the Bulletin Board *)

in( Ch2, VCKsid:bitstring);

(* Receives the asso. Verification Card keystore *)

let kVCid = GetKey(SVKid, VCKsid) in

(* Recover the asso. the Voting Card private key *)
out(Ch2, CreateVote(ELpk,VCid,{J$i|$i=1..$psi}),pube(kVCId),kVCId,(pkCCR$i|$i=1..$psi),
rand2Nat(R,VCid),rand2Nat(R',VCid))); (* Sends the ballot (ie. 'Vote') to the
server. *)

in( Ch2, ({{CC_ Received$i: bitstring|$i=1..$psi}})); (* Receives the short Choice
Codes from server. *)
if ((({{CC_ Received$i= CCid$j}|$i=1..$psi }|| }|$j=1..$psi }&& ) then (* Compares
the short Choice Codes. *)
{
  event Confirmed(InitData, {J$i|i=1..$psi});
  out(Ch2, pCC(kVCId,BCKid)); (* The Confirm(VCId,_,VCidsk,BCK) voter's
method *)
  in( Ch2, =VCCid); (* Alice checks the Vote Cast Code's value. *)
  event HappyUser(InitData, {J$i|i=1..$psi})}
)

(* compute the exponentiation to this key - page 69 sVote specification *)
letfun CCRExpoVotingOptions( ctxtvId : bitstring , VCId : agent _id) =
{(tild ( deltaKeyAgent ( kCCR $i, VCId ), ctxtvId )|$i =1..1}) .

(* compute the exponentiation to this key - page 70 sVote specification *)
letfun CCRExpoBCK( ctxtbId : bitstring , VCId : agent _id) =
{(tild(deltaKeyAgent(kCCR$1',VCId),ctxtbId)|$i=1..1}) .

(* exponentiate E2 - Step 1 of 5.4.3 in sVote specification *)
letfun CCRExpoE2(E2: bitstring , VCId : agent _id) =
let ( mEnc _c1(R), mEnc _cPsi (pk , ({M$i: bitstring |$i =1..$ psi }) , R)) = E2 in
{(tild(deltaKeyAgent(kCCR$1',VCId),ctxtbId)|$i=1..1}) .

(* CCR - generate partial decryptors - Step 5 of 5.4.3 in page 95 of sVote specification *)
letfun CCRDecryptors ( RAll : nat ) = ({ ( { tildNat ( skCCR $j$i, RAll ) |$i =1..$ psi }) |$j =1..2}) .

(* CCR - Confirm - step 10 of 5.6.1 in page 103 of sVote specification *)
letfun CCRConfirm ( vcid : agent _id , pCM : bitstring ) =
{ tildPCC ( deltaKeyAgent ( kCCR$1', vcid ), hash ( pCM ) )|$i=1..2}) .

(* The Choice Return Codes Control Components - CCR_1, CCR_2, CCR_3 and CCR_4 *)
let CCR(i:nat,k:private_ekey, k':private_ekey, SDM_CCRChannel:channel) =
new sessionID:bitstring;
(* setup phase - setupCCR *)

(* generate the private/public key pairs *)
in(SDM_CCRChannel, (=VCIdTag, vcid : agent _id));

let kId = deltaKeyAgent(k,vcid) in
out(c, pube(kId));

let kId' = deltaKeyAgent(k',vcid) in
out(c, pube(kId'));

(* createCC *)

(* exponentiate E2 *)
let ( (pk_CCR$i:public_ekey |$i=1..$psi)) = ((mergepk((pube(skCCR$j$i) |$j=1..1+1)) |$i =1..$psi)) in

let ELpk = mergepk(pube(ELSk1), pube(ELSk2)) in
in(c, B:bitstring);
let (E1:bitstring,E2:bitstring, KVCId:public_ekey, P:bitstring) = B in
if ProcessVoteCheck(ELpk, vcid, B, \{pk_{CCR$i |$i=1..$psi}\}) = true then
get bb(=vcid, B2) in 0 else

(* vcid = deltaId(honest(idA)) then
lock(AliceCell);
read AliceCell as AliceHasVoted;
if AliceHasVoted = unused then
let V = DecMerge(ELSk1, ELSk2, E1) in*
event InsertBBCCR(vcid, B);
(*AliceCell := (V, sessionID);
unlock(AliceCell);*)
insert bb(vcid, B);

(* Honest CCR stores in its internal log that it has proceeded to process the pair (vcid,B) *)
)

let (mEnc_c1(R), mEnc_cPsi(pk, \{(M$i:bitstring|$i=1..$psi)\}, R)) = E2 in
out(c, (E2ExpoTag, i, (mEnc_c1(tildNat(kId,R)), mEnc_cPsi(pk, \{(tild(kId,M$i)|$i=1..$psi }
}}, tildNat(kId,R))))
else insert bb(vcid, B);

(* Honest CCR stores in its internal log that it has proceeded to processed the pair (vcid,B) *)
event InsertBBCCR(vcid, B);

let (mEnc_c1(R), mEnc_cPsi(pk, \{(M$i:bitstring|$i=1..$psi)\}, R)) = E2 in
out(c, (E2ExpoTag, i, (mEnc_c1(tildNat(kId,R)), mEnc_cPsi(pk, \{(tild(kId,M$i)|$i=1..$psi }
}}, tildNat(kId,R))))

in(c, RAAll:nat);
out(c, \{(tildNat(skCCR$i1,RAll) |$i=1..$psi)\});

(* Process Confirm *)
in(c, (=CMTag, pCM:bitstring));
event BallotConfirmationProcessed(vcid, B); (* Honest CCR stores in its internal log that it has contributed to the confirmation of the pair (vcid,B) *)
out(c, tildPCC(kId’, hash(pCM)))) .

let comm =
! ((in(AliceSDMChannel, Id:agent_id); out(UserChannel, Id))
)

(* The Secure Data Manager - trustworthy *)
let SDM(IDM_CCRChannel:channel, \{J$i:nat|$i=1..$n\}, Csk:symmetric_key, vccsk:private_skey) =

(* setup phase *)
new r: bitstring;
new r’: bitstring;
new R2: bitstring;
in(UserChannel, id_Voter:agent_id); (* SDM receives voter's pseudonym id over a private channel *)
let VCId:agent_id = deltaId(id_Voter) in
out(SDM_CCRChannel, (VCIdTag, VCId));
let svk:password = deltaPassword(VCId) in

(* generate the keys *)
let kVCId:private_skey = ske(VCId) in
out(c, pube(kVCId));
(* encrypt set of voting options *)
let ((r$i:nat|$i=1..$n)) = ((rand(r, v(J$i)) |$i=1..$n)) in
let ctxtvID:bitstring = ((Enc(pube(skSDM), v(J$i), r$i) |$i=1..$n)) in

(* compute encrypted ballot casting key *)
let BCKId = bck(id_Voter) in
let R = rand2Nat(r’, id_Voter) in
let ctxtbId = Enc(pube(skSDM), hash( pCC(kVCId,BCKId) ), R) in
466  out(c, (EncBCK, ctxtxtbId));
467  event SetupCompleted(VCId);
468
469  (* calculate the Choice return codes encryption public key - page 65 sVote specification *)
470  let (pk_CCR$i:public_ekey |$i=1..$psi) = ((mergepk(({pube(skCCR$j$i) |$j=1..i+1}) |$i=1..$psi)) in
471
472  (* wait for exponentiations from CCRs *)
473  out(c, (EncVotingOptions, ctxtxtvId));
474  (* in( SDM _ CCRChannel, (= ExpoVotingOptions, = CCRId1, ctxtxtvIDExpo1: bitstring )); *)
475  let ((ctxtxtvIDExpo1|i=1..1)) = CCRExpoVotingOptions(ctxtxtvID, VCId) in
476  out(c, (= ExpoVotingOptions, = CCRId2, ctxtxtvIDExpo2: bitstring ));
477
478  (* the following computes the quantity appearing after sentence "Computes the exponentiation of each element above to kID" in the sVote specification, page 71 *)
479  let ctxtpcId: bitstring = tildPCC(kVCId, mergeExpo(ctxtxtvIDExpo1, ctxtxtvIDExpo2)) in
480  let pCId = DecPCC(skSDM, ctxtpcId) in
481
482  (* the following computes the quantity appearing after sentence "Multiply the values received from the Control Components and obtain the encrypted pre-Vote Cast Return Code": in the sVote specification, page 72 *)
483  in(c, (= ExpoBCK, = CCRId2, tildctxbID2: bitstring));
484  let ((tildctxbID2|$i=1..1)) = CCRExpoBCK(tildctxbId, VCId) in
485  (* let ctxbckId = tildPCC(kVCId, mergeExpo(tildctxbID1, tildctxbID2)) in *)
486  let ctxbckId = mergeExpo(tildctxbID1, tildctxbID2) in
487
488  (* compute the remaining return codes and the mapping*)
489  let CMId = DecPCCOneHash(skSDM, ctxbckId) in
490  let (pCId|$i:bitstring |$i=1..$n) = pCId in
491  let (lCCId|$i:symmetric_ekey |$i=1..$n) = ((delta((Csk,H1((VCId, pCId|i)))) |$i=1..$n)) in
492  let lvCCId:symmetric_ekey = delta((Csk,H1((VCId, CMId))) in
493  let vCCId = VCC(Csk,VCId) in
494  let CCMtable = ((CCMtable(Enc_s(lCCId$1, CC(Csk,VCId,v(J$i)))) |$i=1..$n)) in
495  let VCMtable = VCM_table(Enc_s(lvCCId, (vCCId, Sign(vccsk, VCC(Csk,VCId))))) in
496  (* event StageReached(CCMtable); *)
497  out(c, (CodeMapping, (CCMtable, VCMtable))); event SDMCompleted(VCId).
498
499  let Serv(Ch2: channel, CompChannel: channel, InitData: bitstring, CTally: channel) =
500  in(CompChannel, (= VCIdTag, VCid: agent_id));
501  out(Ch2, (VCIdTag, VCid));
502  (* Retrieves Registration data for this Voting Card Id *)
503  let (ELpk:public_ekey, Csk:symmetric_key, VCCspk:public_skey, VCksid:bitstring, S_VCC: bitstring) = GetServData(InitData, VCId) in
504  (* Provides VCKsid to the voting device, and asks for the ballot (alias 'vote') *)
505  in(Ch2, (=pkall, ((pk_CCR$i:public_ekey |$i=1..$psi))));
506  out(CompChannel, (pkall, ((pk_CCR$i |$i=1..$psi))));
507  out(CompChannel, (VCCskidTag, VCCskid));
508  in(CompChannel, (=BallotTag, B:bitstring));
509  let (x:bitstring, E2:bitstring, y:public_ekey, z:bitstring) = B in
510  event VoteReceived(VCId, B);
511
512  (* Voting part -- the Ballot processing and confirmation followed by the Server *)
513  let Ok1 = ProcessVoteCheck(ELpk, VCId, B, (pk_CCR$i |$i=1..$psi)) in
514
515  (* event PKsReceived(VCid); *)
516  (get bb(=VCid,B2) in 0 else
517  (event InsertBB(VCid,B); insert bb(VCid,B);
518  (* out(Ch2, (E2tag, E2)); *)
519  (* create CC *))
(* obtain the encrypted partial choice return codes - Step 1 in 5.4.3 in page 94 sVote specification *)
in(c, E2Exp0 _2: bitstring);
let ({E2Exp0 _$j: bitstring |$j =1..1}) = CCRExpoE2(E2, VCid ) in
let (mEnc_c1(RAll1), E2Exp0AllCtxt _1: bitstring) = E2Exp0 _1 in
let (mEnc_c1(RAll2), E2Exp0AllCtxt _2: bitstring) = E2Exp0 _2 in
(* multiply the encrypted partial choice return codes - Step 3 in 5.4.3 in page 95 sVote specification *)
let RAll : nat = mergemExpoNat ( RAll 1, RAll 2) in
let E2Exp0AllCtxt : bitstring = mergemExpo (E2Exp0AllCtxt _1 , E2Exp0AllCtxt _2) in

(* obtain the partial decryptors - Step 4 in 5.4.3 in page 95 sVote specification *)
in(c, ({ randomExpoSK 2$i: nat |$i =1..$ psi }));
let ( ({ ({ randomExpoSK $j$i: nat |$i =1..$ psi }) |$j =1..1}) = CCRDecryptors ( RAll ) in
let ({ pDec $i: nat |$i =1..$ psi }) = ({ randomMult ({ randomExpoSK $j$i |$j =1..1+1}) |$i =1..$ psi }) in

(* obtain the pre Choice Return Codes through partial decryption - Step 8 in 5.4.3 in page 96 sVote specification *)
in(c, ((randomExpoSK2$i: nat |$i=1..$psi)));
let ( {{((randomExpoSK$j$i: nat |$i=1..$psi)) |$j=1..1}) = CCRDecryptors(RAll) in
let (({randomDec$i: nat |$i=1..$psi}) = ((randomMult((randomExpoSK$j$i |$j=1..1+1)) |$i=1..$ psi)) in

(* sent them to Alice's computer *)
(out (CompChannel , (CodeTags , CC_ Set ));
!( (* vote confirmation *)
in(Ch2, (= CodeMapping , (CCMTable : bitstring , VCMTable : bitstring )));
let ( ((pC$i:bitstring |$i=1..$psi)) = PC in
let ( {{CCMTable$i:bitstring |$i=1..$psi},(x$i:bitstring |$i=1..$n-$psi}) = CCMTable in
let (({lCCId$i: symmetric_ekey |$i=1..$psi}) = (delta ( (Csk ,H1((VCid , pC$i))) ) |$i=1..$ psi)) in
let ( {{EncCC$i: bitstring |$i=1..$psi}} = ((readCC(H(lCCId$i), CCMTable$i)|$i=1..$psi)) in
let CC_ Set:bitstring = ((Dec_s(lCCId$i, EncCC$i) |$i=1..$psi)) in (* CC$i *)

(* computing pVCCId - step 10 5.6.2 in page 103 of sVote specification *)
in(c, pCM2:bitstring);
let ( {pCM$i:bitstring |$i=1..1}) = CCRConfirm(VCid , cm) in
let pCMA11:bitstring = mergeExpo({pCM$i|$i=1..1+1}) in

(* retrieving lVCCId - step 11 5.6.2 in page 103-104 of sVote specification *)
let lvCCId:symmetric_ekey = delta ( (Csk ,H1((VCid , pCM2))) ) in
let tmpVCC = readVCC(H(lvCCId), VCMTable) in
let (vCCId:bitstring,xsign:bitstring) = Dec_s(lvCCId, tmpVCC) in
let Ok2 = Verify ( VCCspk , vCCId , xsign ) in
(* event PKsReceived(idA); *)
(insert cb(VCid,vCCId,S_VCC);
  event HasVoted(InitData,VCid,B,vCCId,S_VCC); (* Add confirmation to the Confirmation Box 'cb' *)
  out(CompChannel, (VCCIdTag, vCCId)); out(CTally, (VCid,B,vCCId,S_VCC)) (*Phase 1 -- Sends the ballot to the Tally *)
)))))

(******************************************************************)
(* New functions, processes and rules for Universal Verifiability *)
(******************************************************************)

fun PDec1(private_ekey , bitstring):bitstring . (* Partial decryption applied by CCM1 *)
fun PMix1(private_ekey , bitstring , bitstring):bitstring. (* Mixing applied by CCM1 *)
fun PMix2(public_ekey, private_ekey, bitstring, bitstring, bitstring,bitstring):bitstring. (* Mixing applied by CCM2 *)
event HappyAuditor (bitstring, bitstring, bitstring). (* Reached when the Auditor is satisfied with the proofs by CCM1 and CCM2 *)

(* reduction rule for verifying the output of partial decryption and mixing of CCM1 - shuffle 1 embedded *)

fun VerifP11(public_ekey, bitstring, bitstring, bitstring): bool
VerifP11(pube(sk1), ((Enc_c1(R), Enc_c2(ELpk, \{(v(J$i ) |$i=1..$psi)\}, R)),
(Enc_c1(R1), Enc_c2(ELpk, \{(v(K1$i ) |$i=1..$psi)\}, R1)),
(Enc_c1(R2), Enc_c2(ELpk, \{(v(K1$i ) |$i=1..$psi)\}, R2))),
(PDeci(sk1, (Enc_c1(R1), Enc_c2(ELpk, \{(v(K1$i ) |$i=1..$psi)\}, R1))),
PDeci(sk1, (Enc_c1(R2), Enc_c2(ELpk, \{(v(K2$i ) |$i=1..$psi)\}, R2)))) = true
otherwise forall pk: public_ekey, E: bitstring, MixiE: bitstring; VerifP11(pk, E, MixiE, Mixi2E) = false.

(* reduction rule for verifying the output of partial decryption and mixing of CCM2 - shuffle 2 embedded *)

VerifP2(pube(sk1), pube(sk2),
(PDeci(sk1, (Enc_c1(R), Enc_c2(mergepk(pube(sk1), pube(sk2)), \{(v(J$i ) |$i=1..$psi)\}, R))))) = true
otherwise;
((Enc_c1(R2), Enc_c2(mergepk(pube(sk1), pube(sk2))), \((\{v(K2$i) | i = 1..\psi\} \cup R2))\)),
(PDec1(sk1, (Enc_c1(R), Enc_c2(mergepk(pube(sk1), pube(sk2))), \((\{v(J$i) | i = 1..\psi\}) \cup R))\)),
PDec1(sk1, (Enc_c1(R1), Enc_c2(mergepk(pube(sk1), pube(sk2))), \((\{v(K1$i) | i = 1..\psi\}) \cup R1))\)),
(PDec1(sk1, (Enc_c1(R2), Enc_c2(mergepk(pube(sk1), pube(sk2))), \((\{v(K2$i) | i = 1..\psi\}) \cup R2))\)))).
((\((v(J$i) | i = 1..\psi))\)),
((\((v(K1$i) | i = 1..\psi))\)),
((\((v(K2$i) | i = 1..\psi))\))) = true.

reduc forall sk1: private_ekey, sk2: private_ekey, M: bitstring, R: nat;
PDec2(sk2, PDec1(sk1, (Enc_c1(R), Enc_c2(mergepk(pube(sk1), pube(sk2))), M, R)))) = M.

letfun Cleansing (VCId: agent_id, ELpk: public_ekey, \{pk_CCR$i: public_ekey | i = 1..\psi\}, B: bitstring, vCCId: bitstring, S_VCC: bitstring) =
  if ProcessVoteCheck(ELpk, VCId, B, \{pk_CCR$i | i = 1..\psi\}) = true then
    if Verify(vccspk, vCCId, S_VCC) = true then
      let (E1: bitstring, E2: bitstring, KVCID: public_ekey, P: bitstring) = B in
        E1.
      free mixTag: bitstring.

(* The Voting Server applies the Cleansing to the Ballot Box that the Voting Server built
during the Election *)
let VoteServer(VCId1: agent_id, VCId2: agent_id, ELpk: public_ekey) =
  let \{pk_CCR$i: public_ekey | i = 1..\psi\} = (mergepk(\{pube(skCCR$j | j = 1..1+i) | i = 1..\psi\}) in
    (* Cleansing -for Alice (VCID1) and Bob (VCID2) and a corrupted voter VCId *)
    get bb(VCId, B1) in
      (* get cb(VCId, vCCId, S_VCC) in *)
      in(c, (vCCId: bitstring, S_VCC: bitstring));

    (* two different outputs depending on the order where Alice and Bob's encrypted votes are
output after cleaning *)
    let E1: bitstring = Cleansing(VCId, ELpk, \{pk_CCR$i: public_ekey | i = 1..\psi\}, B1, vCCId, S_VCC) in
      get bb(VCId1, B1) in
        (* get cb(VCId1, vCCId1, S_VCC) in *)
        in(c, (vCCId1: bitstring, S_VCC: bitstring));

      let E1: bitstring = Cleansing(VCId1, ELpk, \{pk_CCR$i: public_ekey | i = 1..\psi\}, B1, vCCId1, S_VCC) in
        get bb(VCId2, B12) in
          (* get cb(VCId2, vCCId2, S_VCC) in *)
          in(c, (vCCId2: bitstring, S_VCC: bitstring));

      let E1: bitstring = Cleansing(VCId2, ELpk, \{pk_CCR$i: public_ekey | i = 1..\psi\}, B12, vCCId1, S_VCC) in
        let E2: bitstring = Cleansing(VCId1, ELpk, \{pk_CCR$i: public_ekey | i = 1..\psi\}, B, vCCId1, S_VCC) in
          (out(CCM1Channel, (E1, (E11, E12)))) | out(CCM1Channel, (E1, (E12, E11)));

      (* Dishonest CCM1 outputs shuffled partially decrypted ciphertexts and proof of correct
partial decryption and mixing *)
      let CCM1 =
        in(CCM1Channel, (E1: bitstring, (E1: bitstring, E12: bitstring)));
      let Mix1E1 = PDec1(ELSk1, E1) in
      let Mix1E11 = PDec1(ELSk1, E11) in
      let Mix1E12 = PDec1(ELSk1, E12) in
      out(CCM2Channel, ((E1, (E11, E12)), (Mix1E1, Mix1E11, Mix1E12), PMix1(ELSk1, (E1, E11, E12))
                   , (Mix1E1, Mix1E11, Mix1E12)));

let CCM2 =
in(CCM2Channel, ((E1: bitstring, E11: bitstring, E12: bitstring)), (MixIE1: bitstring, MixIE11: bitstring, MixIE12: bitstring), MixIE1: bitstring, MixIE11: bitstring, MixIE12: bitstring));
in(VoteServerChannel, (E1orig: bitstring, E11orig: bitstring, E12orig: bitstring));
if (VerifP11(pube(ELSk1), (E1orig, E11orig, E12orig)), (MixIE1, MixIE11, MixIE12, MixIE1: bitstring, MixIE11: bitstring, MixIE12: bitstring)) = true then let Mix2E1 = PDec2(ELSk2, MixIE1) in
let Mix2E11 = PDec2(ELSk2, MixIE11) in
let Mix2E12 = PDec2(ELSk2, MixIE12) in
out(c, ((E1, E11, E12), (MixIE1, MixIE11, MixIE12), (Mix2E1, Mix2E11, Mix2E12)));
(* query VCid: agent_id, B1: bitstring, B2: bitstring; event(InsertBB(VCid, B1)) & event(InsertBB(VCid, B2)) ==> B1 = B2. *)

(* Cast-as-Intended Property (Relaxed) : The honest CCR registers a vote for the *honest* voter if and only if the voter confirmed his vote to him. *)

(* This property is achieved in ProVerif *)


event (HappyUser(AliceData(honest(Id), Csk), {J$i |$i =1..$ psi })) => VCid = deltaId (honest(Id))

(* Linking J1...psi to j1..psi: There may exist k ballots containing every j in j1..psi ; Equal since Single-vote *)

(* Consequently, if the voting choices in HasVoted(...) do not match the voter’s intentions , then the Server misbehaved. *)

{&& event (InsertBBCCR(VCid, ((Enc_c1(R)$j),Enc_c2(ELpk,((v(J$j) )|$i =1..$ psi )},R$j)),E2$j,pube(ske(VCid)),P$j)) |$j =1..$ psi |&

(* REMARK: If the honest CCR did not misbehave, then all these InsertBBCCR(...) must be equal all together, since only one could be cast for *)

(* each Id. It follows that the ballot shown by HasVoted(...) contains all the honest voter’s choices j1...jk, and thus no more *)

(* that these since the j1...jk differ two-by-two and each ballot contains exactly k voting choices. *)

(* Universal Verifiability property: query for correct final election result *)

(* query E1:bitstring, E11:bitstring, Mix1E1:bitstring, Mix1E11:bitstring, Mix1E12:bitstring, Mix2E1:bitstring, Mix2E11:bitstring, Mix2E1:nat, R:nat, R1:nat , R2:nat, (J$i: nat|$i =1..$ psi ), {K1$i: nat|$i =1..$ psi }, {K2$i: nat|$i =1..$ psi }, ELpk: public_ekey;

event (HappyAuditor(((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12)) =>

(* E1 = (Enc_c1(R),Enc_c2(ELpk,((v(J$j) )|$i =1..$ psi )},R))

&& E1 = (Enc_c1(R1),Enc_c2(ELpk,((v(K1$i) )|$i =1..$ psi )},R1))

&& E2 = (Enc_c1(R2),Enc_c2(ELpk,((v(K2$i) )|$i =1..$ psi )},R2))

&& Mix2E1 = (v(J$j) )|$i =1..$ psi ))

&& ((Mix2E1 = (v(K1$i) )|$i =1..$ psi )) & Mix2E12 = ((v(K2$i) )|$i =1..$ psi ))) ||(Mix

2E11 = (v(K2$i) )|$i =1..$ psi )))& Mix2E12 = ((v(K1$i) )|$i =1..$ psi )))

.*)

(* 8. Main process -- initiates the election *)

process

(* Public output from Setup(...) -- Gives the election’s parameters to the Intruder. *)

let ELpk = mergepk(pube(ELSk1), pube(ELSk2)) in

out(c, ELpk); out(c, vccspk);

(*out(c, vccspk); *)

(* Gives the honest voter’s public data to the Intruder *)

out(c, deltaId(honest(idA))); out(c, deltaId(honest(idB))); out(c, pube(skSDM));

(* setup keys for CCRs *)

out(c, ( {pube(skCCR$i$j) |$i =1..2} |$j =1..$ psi )

out(c, ( {pube(kCCR$i) |$i =1..2} )

(* setup public keys for CCMs *)
\texttt{out(c, \{pube(ELSk$\i$)$\i$=1..2}\);}

\texttt{(* Roles for honest voter(s) -- one for Individual Verifiability. *)}
\texttt{Alice( Ch1, AliceData(honest(idA), csk),\{ja$\i$|$\i$=1..$psi$\} |}
\texttt{Alice( Ch1, AliceData(honest(idB), csk),\{jb$\i$|$\i$=1..$psi$\} |}
\texttt{out(c, ske(deltaId(honest(idA))))}
\texttt{(*out(c, ske(deltaId(honest(idB)))) *)}

\texttt{(* Dishonest voter(s) : As many as possible for verifiability. *)}
\texttt{| !(new id: agent_id; out(c, id); out(c, AliceData(id, csk)))}

\texttt{(* Bulletin Board (ie. Voting Server) : is Dishonest *)}
\texttt{| out(c, ServData(ELpk, csk, vccsk))}

\texttt{(* SDM is honest *)}
\texttt{| !SDM(SDM:\textsubscript{CCRChannel}, \{ja$\i$|$\i$=1..$n$\}, csk, vccsk)}

\texttt{(* CCR1 is honest *)}
\texttt{| !CCR(CCRId1, kCCR1, kCCR1', SDM:\textsubscript{CCRChannel}) | comm}

\texttt{(* Cleansing is computed honestly; secret permutation is implicitly applied here *)}
\texttt{| !VoteServer(deltaId(honest(idA)), deltaId(honest(idB)),ELpk)}
\texttt{(* CCM1 is dishonest; CCM2 is honest; Auditor is honest *)}
\texttt{| !CCM2 | !Auditor}

\texttt{sources/cast–as–intended–2ccrs–Proverifv3.1.pve}
Appendix B

ProVerif source file
/sources/UniversalVerifiability2CCMs-DishonestCCM1v3.1.pve before expansion

(* ProVerif specification for Scytl's sVote Voting Protocol V3.1 - September 2018 *)
(* file: Scytl "sVote Protocols Specifications v5.1" *)
(* Sections in comments below refer to crypto paper "sVote with Control Components Voting Protocol - Computational Proof of Complete Verifiability" *)
(* This specification models 2 CCR's (to be tested only with #CCR=2), 2CCM's, an universe of $n$ voting options and voter choosing #psi > 1 pairwise different voting options *)
(* Trust assumptions: *)
(* CCR1, CCM1 and Alice are trustworthy *)
(* CCR2, CCM2 and Voting Server and Voting Device are untrustworthy *)
(* RESULT *)
(* Universal Verifiability property IS verified *)

set ignoreTypes = false .
set verboseClauses = short .
(* set debugOutput = true . *)
(* 1. Objects and Types *)
type agent_id . fun t_agent_id(agent_id) : bitstring [data , typeConverter ].
   (* The type for any IDs like e.g. voting card. *)
type password . fun t_password(password) : bitstring [data , typeConverter ].
   (* The type for user passwords. *)
type nat . fun t_nat(nat) : bitstring [data , typeConverter ].
   (* The type for natural numbers. *)
free c : channel .
   (* Public Channel for communications with the intruder # public *)
free Ch1 : channel .
   (* Channel between Alice and her Voting Device # public *)
free SDM_CCRChannel : channel [private] .
   (* Communication between SDM and honest CCR # private *)
free CompServerChannel : channel .
   (* Channel between Server and Voting Device # public *)
free AliceSDMChannel : channel [private] .
   (* Channel in which SDM receives Alice's psedonymous identity idA # private *)
free UserChannel : channel [private] .
free VoteServerChannel : channel [private] .
fun v(nat) : bitstring [data] .
   (* The function from voting choices to voting options # public, invertible *)
table bb(\text{agent\_id}, \text{bitstring}) . (* The Ballot Box, storing the ballots for each agent # private *)

table bbccr(\text{agent\_id}, \text{bitstring}) . (* The Ballot Box, storing the ballots for each agent # private *)

table cb(\text{agent\_id}, \text{bitstring}, \text{bitstring}) . (* The Confirm. Box, storing VCC/S-VCC for each agent id # private *)

free idA, idB : \text{agent\_id} . (* Defines two SVK for two honest voters Alice and Bob # public *)

free \{ja$i,jb$i|$i =1..n\} : \text{nat} . (* Voting choices for the Honest voters -- REA or OBS # public *)

free \{a$i|$i =1..\psi\} : \text{nat} . (* MODELING : an indexed set of nat atoms, for disquality tests -- only in Test. *)

free \{CCRId$i|$i =1..1+1\}: \text{nat} . (* 2 CCRs *)

free Ch2 : \text{channel [private]} .

free CCM1Channel, CCM2Channel : \text{channel} . (* Channels for inputs/outputs of CCM1, CCM2 respectively *)

fun hash(\text{bitstring}):\text{bitstring} .

(* 2. Intruder capabilities and functions *)

(* Encryption scheme -- ElGamal/ElGamal with multiple encryptions -- Key pairs generated through \text{Gen\_e}, which is implicit. *)

type \text{private\_ekey} .

fun t\_private\_ekey(private\_ekey): \text{bitstring [data, typeConverter]} . (* The type for private decryption keys + type converter. *)

type \text{public\_ekey} .

fun t\_public\_ekey(public\_ekey): \text{bitstring [data, typeConverter]} . (* The type for public encryption keys + type converter. *)

fun ske(\text{agent\_id}) : \text{private\_ekey [private]} . (* The private key associated to an id # private *)

fun VCske(\text{agent\_id}) : \text{private\_ekey [private]} . (* The verification card secret key *)

fun pube(private\_ekey) : public\_ekey . (* The function to rebuild a public key from the private # public, noninvertible *)

letfun pke(Id: \text{agent\_id}) = pube(ske(Id)) . (* The public enc. key associated to an id # public, invertible *)

fun CCRReturnKey(nat, nat): \text{private\_ekey} . (* The Choice Return Codes Encryption private key *)

free \{ skCCR$i$j|$i =1..1, \}$j =1..\psi\} : \text{private\_ekey [private]} . (* Choice Return Codes Encryption secret keys for honest CCR -- page 65 *)

free \{ skCCR$i|$i =1..1, $\psi\} : \text{private\_ekey} . (* Choice Return Codes Encryption secret keys for honest CCR *)

free \{kCCR$i|$i =1..1\} : \text{private\_ekey [private]} . (* Secret key for KDF computations honest CCR *)

free kCCR2: \text{private\_ekey} . (* Secret key for KDF computations dishonest CCR *)

free \{kCCR$i'|$i =1..1\} : \text{private\_ekey [private]} . (* Secret key for KDF computations honest CCR *)

free kCCR2': \text{private\_ekey} . (* Secret key for KDF computations dishonest CCR *)

free \text{skSDM} : \text{private\_ekey} . (* Election decryption key share held by CCM1 * # public *)

free \text{ELSk1: private\_ekey} . (* Election decryption key share held by CCM2 * # private *)

fun Enc\_c1( nat) : \text{bitstring} . (* The c1 part of the asymmetric encryption function with explicit random number. *)

fun Enc\_c2(public\_ekey, bitstring, nat) : \text{bitstring} . (* The c2 part of the asymmetric encryption function with explicit random number. *)

letfun Enc(Pk: public\_ekey, M: bitstring, R: nat) = (Enc\_c1(R), Enc\_c2(Pk, M, R)) . fun mEnc\_c1( nat) : \text{bitstring [data]} . (* The c1 part of the asymmetric encryption function with explicit random number. *)

fun mEnc\_cPsi(bitstring, bitstring, nat) : \text{bitstring [data]} . (* The c2 part of the asymmetric encryption function with explicit random number. *)

let fun mEnc(Pk: bitstring, M: bitstring, R: nat) = (mEnc\_c1(R), mEnc\_cPsi(Pk, M, R)) .

(* Encryption *)
reduc forall Sk: private_ekey, M: bitstring, R: nat; Dec(Sk, (Enc_c1(R), Enc_c2(pube(Sk), M, R))) = M . (* Decryption *)
reduc forall Pk: public_ekey, M: bitstring, R: nat; Verif(Pk, (Enc_c1(R), Enc_c2(Pk, M, R))) = true . (* Checks key *)
reduc forall Id: agent_id; Get_Id(pube(ske(Id)))

(* Signature scheme -- from sVote specification page 143 -- expected to be the RSA Probabilistic Signature Scheme (PSS) -- Gen_s implicit. *)
type private_skey. fun t_private_skey(private_skey): bitstring [data, typeConverter]. (* The type for private signing keys + type converter. *)
type public_skey. fun t_public_skey(public_skey): bitstring [data, typeConverter]. (* The type for public signing keys + type converter. *)
fun sks(agent_id) : private_skey. (* The private sig. key associated to an agent_id # private *)
fun pubs(private_skey) : public_skey. (* The function to rebuild a public signing key from the private # public, noninvertible *)
letfun pks(Id: agent_id) = pubs(sks(Id)). (* The public sig. key associated to an agent id # public, invertible *)
fun Sign(private_skey, bitstring) : bitstring. (* The digital signature function with explicit random number. (Signature) *)
reduc forall Sk: private_skey, M: bitstring; Verify(pubs(Sk), M, Sign(Sk, M)) = true. (* Signature Verification *)
reduc forall Sk: private_skey, M: bitstring; Checks(pubs(Sk), Sign(Sk, M)) = M. (* More expressive than Verify - not used *)
reduc forall Sk: private_skey, M: bitstring; Get_Message(Sign(Sk, M)) = M. (* Worst case for modeling only - not used *)

(* Modeling of the Non-Interactive Zero-Knowledge Proofs of Knowledge when voting -- from sVote specification, page 89, item 4. -- Names changed for abstractions. *)
(* 'Schnorr Proof' and 'Knowledge of encryption exponent' and 'Plaintext Equality Proof' are abstracted inside the *)
(* ZKP and VerifP operators. Consequently, the c1^- and c2^- , which are intermediate values, are not needed anymore and thus abstracted away. *)

fun pCC(private_ekey, bitstring) : bitstring. (* The built of partial Choice Return Codes (kID, vi) = vi^kId # public, noninvertible *)
fun til(P:private_ekey, bitstring): bitstring. (* To exponentiate ciphertexts generated by function Enc # public, noninvertible *)
fun tildCC(private_ekey, bitstring): bitstring. (* Used to sequentially exponentiate ciphertexts - for computing ctxtpcId and ctxbckId # public, noninvertible *)
fun tildNat(private_ekey, nat): nat. (* To exponentiate ciphertexts generated by functions Enc_c1 and mEnc_c1 # public, noninvertible *)
fun tildPCC(private_ekey, bitstring): bitstring. (* To exponentiate ciphertexts generated by function mEnc_cPsi # public, noninvertible *)

fun ZKP(private_ekey, public_ekey, {public_ekey|$i=1..$ psi}, bitstring, bitstring, nat, nat, private_ekey): bitstring. (* Modeling of the proof generation *)
reduc forall ELpk: public_ekey, KVCid: public_ekey, {pk_CCR$i:public_ekey|$i=1..$ psi}, {J$i :nat|$i=1..$ psi}, R: nat, R': nat, vcid: agent_id;
VerifP(ELpk, pube(ske(vcid)), vcid, {pk_CCR$i|$i=1..$ psi}, (Enc_c1(R),Enc_c2(ELpk, {(v(J$i))|$i=1..$ psi})), R), (mEnc_c1(R'), mEnc_cPsis((pk_CCR$i|$i=1..$ psi))), (pCC(ske(vcid)), (v(J$i))|$i=1..$ psi)), ZKP(ELpk, pube(ske(vcid)), {pk_CCR$i|$i=1..$ psi}, (Enc_c1(R),Enc_c2(ELpk, {(v(J$i))|$i=1..$ psi})), R), (mEnc_c1(R'), mEnc_cPsis((pk_CCR$i|$i=1..$ psi))), (pCC(ske(vcid)), (v(J$i))|$i=1..$ psi)), R', R', ske(vcid))) = true.

(* Next follows a series of functions that model modular multiplications that take place at several parts of sVote protocol *)
(* Different functions names imply they are applied to different input Types *)
fun randomMult(nat,nat): nat.
fun mergepk({public_ekey|$i=1..$ psi}): public_ekey.
fun mergeExpo(bitstring, bitstring): bitstring.
reduc forall kId1: private_ekey, kId2: private_ekey, M: bitstring, R:nat, {sk1$i: private_ekey|$i=1..$psi}, {sk2$i: private_ekey|$i=1..$psi}, {J$i: nat|$i=1..$psi}, kVCId:private_ekey;

(* Rule A - rule to compute pCId_i's in the CreateCC phase (5.4.3 CreateCC algorithm, page 94) *)
partialDec(({ randomMult ( tildNat (sk 1$i, mergemExpoNat ( tildNat ( kId 1, R), tildNat ( kId 2, R))),
mergemExpoNat ( tildNat (sk 1$i, mergemExpoNat ( tildNat ( kId 1, R), tildNat (kId2, R)))))|$i=1..$psi}),
(mEnc_c1(mergemExpoNat(
tildNat(kId1, R),
tildNat(kId2, R)))))
= ({mergeExpo ( tildPCC ( kId 1, ( pCC ( kVCId , v(J$i))))), tildPCC(kId2, (pCC(kVCId, v(J$i)))))|$i=1..$psi} .

reduc forall sk: private_ekey, kVCId:private_ekey, kId1:private_ekey, kId2:private_ekey, {R$i:nat|$i=1..$n}, M: bitstring, {J$i:nat|$i=1..$n};

(* Rule B - rule to compute pCId_i's in the Setup phase (page 67) *)
(* Rule A + Rule B imply that pCId_i's coincide in both Setup and CreateCC phases *)
DecPCC (sk, tildPCC(kId1, (pCC(kVCId, v(J$i))))),
tildPCC(kId2, (pCC(kVCId, v(J$i)))))) =

mergeExpo(tildPCC(kId1, (pCC(kVCId, v(J$i)))),
tildPCC(kId2, (pCC(kVCId, v(J$i)))))) .

reduc forall sk:private_ekey, kVCId:private_ekey, kId1:private_ekey, kId2:private_ekey, R:nat, M:bitstring;

(* Rule C - states that pVVCId coincides in both Setup (page 72, specification) and ProcessConfirm (step 5.6.2 in page 103 of sVote specification) phases *)
DecPCCOneHash (sk, mergeExpo(tild(kId1, Enc_c1(R), Enc_c2(pube(sk), hash ( pCC ( kVCId ,M)))),
tild(kId2, Enc_c1(R), Enc_c2(pube(sk), hash ( pCC ( kVCId ,M)))))) =

mergeExpo(tildPCC(kId1, hash(pCC(kVCId, M))),
tildPCC(kId2, hash(pCC(kVCId, M)))) .

(* Symmetric encryption scheme -- expected to be based on AES-GCM *)
type symmetric_ekey. fun t_symmetric_ekey(symmetric_ekey): bitstring [data, typeConverter].
(* The type for symmetric encryption keys. *)
fun Enc_s(symmetric_ekey, bitstring) : bitstring .
(* Encryption (symmetric key) *)
reduc forall SKey:symmetric_ekey, M:bitstring; Dec_s(SKey,Enc_s(SKey,M)) = M.
(* Decryption (symmetric key) *)

(* Key and IDs derivation scheme *)
(* free IDseed : bitstring # public *)
(* Not needed: value fixed *)
(* free KEYseed : bitstring # public *)
(* Not needed: value fixed *)
fun deltaId(agent_id) : agent_id .
(* The delta function, for agents IDs *)
fun deltaKey (password) : symmetric_ekey . (* The delta function, for symmetric encryption keys for keystore # public *)
fun deltaPassword (agent_id): password [private]. (* The delta function to assign passwords SVKid to voters # private *)
fun deltaKeyAgent (private_ekey, agent_id) : private_ekey . (* Used by the CCRs to create kId1, kId2, ... # public *)
fun honest (agent_id) : agent_id [private] . (* MODELING : a function to separate honest and dishonest agents (only in Properties) *)

(* Keyed Pseudo-Random and hash functions -- from cryptoPaper, section 3.2 -- Note: Not the same keys as for symmetric encryption *)
type symmetric_key. fun t_symmetric_key (symmetric_key): bitstring [data, typeConverter]. (* The type for symmetric keys for KPR f(...) *)
fun H (symmetric_ekey) : bitstring . (* The hash function # public, noninvertible *)
fun f (symmetric_key, bitstring) : symmetric_ekey . (* The keyed pseudo-random function f # public, noninvertible *)
fun H1 (bitstring) : bitstring . (* The hash function # public, noninvertible *)
fun delta (bitstring) : symmetric_ekey . (* The mask generation function # public, noninvertible *)

fun rand (bitstring, bitstring): nat . (* to sample random nonces when SDM encrypts voting options in the Setup phase *)
fun rand2 (bitstring, agent_id): bitstring .
fun rand2Nat (bitstring, agent_id): nat . (* rand2 and rand2Nat are used to sample random nonces that are unique per agent_id *)
fun kdfCCR (bitstring, nat): private_ekey .
fun kdf (bitstring, agent_id): nat .

(* 3. Initialization sequence (done off-line in this modeling). *)
free election : agent_id . (* The election id - used to derive public/private keys # public (sk is private) *)
free signature : agent_id . (* The signature id - used to derive public/private keys # public (ak is private) *)
(* letfun ELpk = pke (election) . (* The Electoral Board public key in pair (ELpk, ELsk) # public *)
( * change - ELpk is built in a two-party way, ELsk is not used *)
(* letfun ELsk = ske (election) . (* The Electoral Board private key in pair (ELpk, ELsk) # private *)
free csk : symmetric_key [private] . (* The Codes secret key 'Csk' (for f(...)) function # private *)
letfun vccspk = pks (signature) . (* The Vote Cast Code Signer public key in (VCCspk, VCCsk) # public *)
letfun vccssk = sks (signature) . (* The Vote Cast Code Signer private key in (VSSspk, VCCsk) # private *)

(* Register(...,id,csk,vccssk) ie. the registration step for any new voter -- done off-line through functions *)
fun bck (agent_id) : bitstring [private]. (* The Ballot Casting Key 'BCK^-id' of an agent id # private *)
fun CC (symmetric_key, agent_id, bitstring) : bitstring [private]. (* The short choice code 'CC^-id' of an agent id + voting option # private *)
fun VCC (symmetric_key, agent_id) : bitstring [private]. (* The short vote cast code 'VCC^-id' of an agent id # private *)
fun CCD_table (bitstring) : bitstring . (* The Codes Mapping Table ('CM_id', unbounded) -- opened with readCC/VCC() . *)
fun VCM_table (bitstring) : bitstring . (* The Codes Mapping Table ('CM_id', unbounded) -- opened with readCC/VCC() . *)
reduce forall Csk:symmetric_key, VCid:agent_id, J:nat, pc:bitstring ;
readCC H1 ((VCid, pc)) , CCD_table (Enc_s ( delta(A (Csk,H1((VCid, pc)))) , CC(Csk, VCid,v(J)))) = Enc_s ( delta(A (Csk,H1((VCid, pc)))) , CC(Csk,VCid,v(J))) .
reduce forall Csk:symmetric_key, VCid:agent_id, CMid:bitstring, vCCid: bitstring, VCCsk: private_skey;
( * Enc_s (VCC, VCC) * )
readVCC( H1(H1((VCid, CMid)))) , VCM_table(Enc_s( delta( (Csk, H1((VCid, CMid)))) , (vCCid, Sign(VCCsk, VCC(Csk, VCCid))))))) = Enc_s( delta( (Csk, H1((VCid, CMid)))) , (vCCid, Sign(VCCsk, VCC(Csk, VCCid)))) .

(* For any voter id, generates VCidpk, VCCidsk ....... : pke(id) and ske(id) and GetVCks(id) -- distributed in encrypted private data. *)

(* For any voter id, chooses a ballot casting key ... : bck(id) as defined above; -- distributed in AliceData and Mapping. *)

(* 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. *)

(* For any voter id, computes the Vote Cast Code VCC(id) ............ : with f(..) and pCC(...) and bck(id) -- used to build the Mapping. *)

(* For any voter id, chooses the short CC and short VCC .............. : with CC(id) and VCC(...) -- distributed in AliceData and Mapping. *)

(* For any voter id, computes the signature of the Vote Cast Code .... : Sign(vccssk, VCC(id)) -- distributed in ServData. *)

(* For any voter id, stores a list of hashed choice codes {H(lCC(id,i))}_i=1..inf ...... : through MakeRFList -- distributed in ServData. *)

(* Registration data for any voter id -- AliceData(svk,csk) produced by the SDM for Alice opened with a reduction *)

fun AliceData ( agent_id , symmetric_key ) : bitstring [ private ].
reduc forall Csk : symmetric_key , id_Alice : agent_id , {J$i: nat |$i =1..$ psi};
GetAliceData (AliceData(id_Alice,Csk),{JS$i|$i =1..$psi}) = (id_Alice, deltaId(id_Alice),
deltaPassword(id_Alice), bck(id_Alice), VCC(Csk,deltaId(id_Alice)), {CC(Csk,deltaId(id_Alice),v(J$i))|$i =1..$ psi}.

(* Note: The set of CC(id,v(J)) is a selection of CC corresponding to the {J$i|$i =1..$ psi}, decided at Alice’s initialisation. *)

(* Registration data for the Voting Server (Bulletin Board) -- ServData(pke,Csk,sk) produced by the SDM with a reduction *)

fun ServData ( public_ekey , symmetric_key , private_skey ) : bitstring [ private ].
reduc forall ELpk : public_ekey , Csk : symmetric_key , VCCssk : private_skey , id_Voter : agent_id;
GetServData (ServData(ELpk,Csk, VCCssk),id_Voter) = (ELpk,Csk, pubs ( VCCssk ), Enc_s( deltaKey(deltaPassword(id_Voter)),t_private_ekey(ske ((id_Voter))))), Sign(VCCssk,VCC(Csk, id_Voter))) .

(* 4. Algebraic properties and List of Events *)

event Confirmed(bitstring, (nat|$i=1..$psi)). (* Issued by the voter when he/she confirms his/her vote. *)

event HappyUser(bitstring, (nat|$i=1..$psi)). (* Issued by the voter when he/she terminates successfully. *)

event AliceVoted(agent_id) .

event PKsReceived(agent_id) .

event StageReached(bitstring).

event SetupCompleted(agent_id) .

event SDMCompleted(agent_id) .

event VoteReceived(agent_id, bitstring). (* Issued by the server when vote has been received *)

event InsertBB(agent_id, bitstring). (* Issued by the server when it adds something in BB. *)

event InsertBBCCR(agent_id, bitstring). (* Issued by the CCR when it adds something in BB. *)

event BallotConfirmationProcessed(agent_id, bitstring) .

event HasVoted( bitstring, agent_id, bitstring, bitstring, bitstring). (* Issued by the server when it gets a voter’s confirmation. *)

event NeverTrue. (* An event that is never activated, and thus, never true. *)

event Results({nat|$i=1..$psi}, {nat|$i=1..$psi}, {nat|$i=1..$psi}). (* Issued by the Tally when it publishes the results. *)

(* 5. Methods and Agents processes *)
(* GetKey (SVKid, VCksid) -- Computer retrieves the Verification Card private key from the keystore -- step 5.3 in page 87 of sVote specification *)

letfun GetKey (SVKid : password, VCksid : bitstring) =
  let KSpwd = deltaKey (SVKid) in
  let t_private_ekey (VCidsk : private_ekey) = Dec_s (deltaKey (SVKid), VCksid) in
  VCidsk.

(* CreateVote (ELpk, VCid, Vopt, VCidpk, VCidsk, pkCCRs, R, R') -- Computer creates the ballot -- from specification page 87 *)

letfun CreateVote (ELpk : public_ekey, VCid : agent_id, {J$i: nat |$i =1..$ psi}, KVCId : public_ekey, kVCId : private_ekey, { pkCCR $i: public_ekey |$i =1..$ psi }, R : nat, R' : nat ) =
  let V = ({v(J$i)|$i =1..$ psi}) in 
  let E1 = Enc (ELpk, V, R) in
  let E2 = mEnc ( ({pkCCR$i|$i =1..$ psi}), ({pCC ( kVCId,v(J$i))|$i =1..$ psi})), R') in
  let P = ZKP (ELpk, KVCId, {pkCCR$i|$i =1..$ psi}, E1, E2, R, R', kVCId) in
  (E1, E2, KVCId, P).

(* ProcessVoteCheck (ELpk, VCid, B, pkCCRs) -- checks the ballot *)

letfun ProcessVoteCheck (ELpk : public_ekey, VCid : agent_id, B : bitstring, {pk_ CCR $i: public_ekey |$i =1..$ psi} ) =
  let (xE1: bitstring, xE2: bitstring, xKVCId : public_ekey, xP: bitstring) = B in
  let Ok1 = VerifP (ELpk, xKVCId, VCid, {pk_ CCR $i |$i =1..$ psi}, xE1, xE2, xP) in
  true.

(* Confirm (VCid, B, VCidsk, BCKid) -- Computer generates a Confirmation Message -- Done directly inside 'Cmp' and 'Alice_Cmp'. *)

(* AuditBallotProof (({ CC_ Received $i|$i =1..$ psi }) ,({ CCid $i|$i =1..$ psi })) -- Alice checks if all expected CC were indeed received. *)

(* ProcessConfirm (bb, VCid, pVCCid, Csk, VCCssk, S_ VCCid) -- Server checks the retrieved short Vote Cast Code. *)

(* Typing of the messages -- between Voter and his Computer only *)
free mAC 1, mAC 2, mCA 1, mCA 2 : bitstring.

(* Typing of messages *)
free decryptors, CodeMapping, EncVotingOptions, ExpoVotingOptions, EncBCK, ExpoBCK, pkCCRTag, pkall, VCidTag, E2tag, id_VoterTag, VCskidTag, CMTag, CodeTags, VCCidTag, CCR_CMTag, vcCidTag, BallotTag, RAllTag, E2ExpoTag: bitstring.

(* Alice -- the client process *)
let Alice (Ch1: channel, InitData : bitstring, {J$i: nat |$i =1..$ psi}) =
  (Checks that the voting choices are all different *)
  { if {((J$i = J$j && a$i <> a$j)|$j =1..$ psi | || } then 0 else |$i =1..$ psi|
  } (* No honest voter can use twice the same option *)
  (Retrieves registration data obtained from the SDM -- Set of initial data given to Alice by the S.*)
  let (id_Alice: agent_id, VCid: agent_id, SVKid: password, BCKid: bitstring, VCCid: bitstring, {CCid$1: bitstring|$1=1..$ psi}) = GetAliceData (InitData, {J$i|$i =1..$ psi}) in
  out (AliceSDMChannel, id_Alice);
  (Voting part -- The voting process followed by agent Alice *)
  out (Ch1, (mAC1,VCid, SVKid, {J$i|$i =1..$ psi}));
  in( Ch1, (=mCA1,((CC_Received$1: bitstring|$1=1..$ psi))));
  (* event StageReached(((CCid$1|$1=1..$ psi))); *)
  if {((CC_Received$1=CCid$|j|$1=1..$ psi) || )|$j=1..$ psi| && } then (* Compares the short Choice Codes. *)
  { (*event Confirmed(InitData, {J$i|$i =1..$ psi}));
    out (Ch1, (mAC2,BCKid));
    in( Ch1, (=mCA2, VCCid));
    if {((CC_Received$1=CCid$|j|$1=1..$ psi) || }|$j=1..$ psi| && } then (* Alice checks the Vote Cast Code's value. *)
    { event HappyUser(InitData, {J$i|$i =1..$ psi})
      (*Alice_Cmp (InitData, {J$i|$i =1..$ psi}));
    }
  }
let Cmp (Ch 1: channel, ServChannel: channel, ELpk: public_ekey) =
  new R: bitstring;
  new R': bitstring;
in(Ch 1, (mAC 1, Vcid: agent_id, SVKid: password, {{J$i: nat |$i =1..$psi}}));
  (* The Start Voting Key plus the Voting options. *)
out(ServChannel, (VcidTag, Vcid));
  (* Send the Voting Card ID to the Bulletin Board *)
in(ServChannel, (pkall, {{pk_ CCR$i: public_ekey |$i =1..$psi}}));
  (* event PKsReceived(Vcid); *)
in( ServChannel, (VCidTag, VCid ));
  (* Send the Voting Card ID to the Bulletin Board *)
in(ServChannel, (pkall, {{pk_ CCR$i: public_ekey |$i =1..$psi}}));
  (* event PKsReceived (Vcid); *)
in( ServChannel, (VCskidTag, VCksid: bitstring));
  (* Receives the asso. Verification Card keystore *)
let kVCid : private_ekey = ske(Vcid) in
out(C, kVCid);
let B: bitstring = CreateVote(ELpk, VCid, {{J$i|$i =1..$psi}, pube(kVCid), kVCid, {pk_ CCR$i |$i =1..$psi}, rand2Nat(R, VCid), rand2Nat(R', VCid)});
  (* Sends the ballot (ie. 'Vote') to the server. *)
in( ServChannel, (CodeTags, CC_Set: bitstring));
  (* Transmits the Choice Codes to the voter. *)
out(Ch, (mCA1, CC_Set));
in(Ch 1, (mAC2, BCKid: bitstring));
out( ServChannel, (CMTag, pCC(kVCid, BCKid)));
  (* The Confirm(Vcid,_, VCi dsk, BCK) voter's method *)
in( ServChannel, (VCidTag, VCcid: bitstring));
  (* Transmit the Vote Cast Code to the voter. *)
out(Ch 1, (mCA2, VCcid));
in(Ch 2, VCksid: bitstring);
  (* Receives the asso. Verification Card keystore *)
let kVCid = GetKey(SVKid, VCksid) in
out(Ch 2, CreateVote(ELpk, VCid, {{J$i|$i =1..$psi}, pube(kVCid), kVCid, {pk_ CCR$i |$i =1..$psi}, rand2Nat(R, VCid), rand2Nat(R', VCid)});
  (* Sends the ballot (ie. 'Vote') to the server. *)
in( Ch 2, VCcid);
  (* Receives the asso. Verification Card keystore *)
let kVCid = GetKey(SV Kid, VCksid) in
out(Ch 2, CreateVote(ELpk, VCid, {{J$i|$i =1..$psi}, pube(kVCid), kVCid, {pk_ CCR$i |$i =1..$psi}, rand2Nat(R, VCid), rand2Nat(R', VCid)});
  (* Sends the ballot (ie. 'Vote') to the server. *)
in( Ch2, CCid$1: bitstring);  (* Receives the short Choice Codes from server. *)
if {{J$s = J$j & a$i <> a$j} |$j =1..$psi | || } then 0 else {{s =1..$psi | \n}} \n)  (* No honest voter can use twice the same option *)
{if {{(J$i = J$j & a$i <> a$j) |$j =1..$psi | || } then 0 else {{s =1..$psi | \n}} (* No honest voter can use twice the same option *)
(* Retrieves registration data obtained from the SDM -- Set of initial data given to Alice by the SDM. *)
let (Vcid: agent_id, SVKid: password, BCKid: bitstring, VCCid: bitstring, {{CCid$1: bitstring |$i =1..$psi}}) = GetAliceData(InitData, {{J$i|$i =1..$psi}}) in
(* The voting process followed by agent Alice *)
out(Ch 2, VCCid);
  (* Send the Voting Card ID to the Bulletin Board *)
in( Ch 2, VCCid: bitstring);
  (* Receives the asso. Verification Card keystore *)
out(Ch 2, GetKey(SVKid, VCCid));
out(Ch 2, CreateVote(ELpk, VCid, {{J$i|$i =1..$psi}, pube(kVCid), kVCid, {pk_ CCR$i |$i =1..$psi}, rand2Nat(R, VCid), rand2Nat(R', VCid)});
  (* Sends the ballot (ie. 'Vote') to the server. *)
in( Ch 2, {{CC_Received$1: bitstring |$i =1..$psi}});
  (* Receives the short Choice Codes from server. *)
if {{(CC_Received$1 = CCid$1)$i =1..$psi | || } |$j =1..$psi | && } then  (* Compares the short Choice Codes. *)
event Confirmed(InitData, {{J$i|$i =1..$psi}});
out(Ch 2, pCC(kVCid, BCKid));
  (* The Confirm(Vcid,_, VCi dsk, BCK) voter's method *)
(* Alice checks the Vote Cast Code’s value. *)

in( Ch2, VCCid);

(* compute the exponentiation to this key - page 69 sVote specification *)
letfun CCRExpoVotingOptions(ctxtvId: bitstring, VCId: agent_id) =
  (\(i\) | \(i\) = 1..\(\psi\)) (\(\tilde{\delta}_{\text{KeyAgent\,}(\text{kCCR}_i, \text{VCId})}, \text{ctxtvId}\) | \(i\) = 1..1)).

(* compute the exponentiation to this key - page 70 sVote specification *)
letfun CCRExpoBCK(ctxtbId: bitstring, VCId: agent_id) =
  (\(i\) | \(i\) = 1..\(\psi\)) (\(\tilde{\delta}_{\text{KeyAgent\,}(\text{kCCR}_i', \text{VCId})}, \text{ctxtbId}\) | \(i\) = 1..1)).

(* exponentiate E2 - Step 4.4.3.2 in sVote specification *)
letfun CCRExpoE2(E2: bitstring, VCId: agent_id) =
  (\(i\) | \(i\) = 1..\(\psi\)) (\(\tilde{\delta}_{\text{KeyAgent\,}(\text{kCCR}_j, \text{VCId})}, \text{E2}\) | \(j\) = 1..2)).

(* CCR - generate partial decryptors - Step 5 of 5.4.3 in page 95 of sVote specification *)
letfun CCRDecryptors(RAll: nat) = (\(i\) | \(i\) = 1..\(\psi\)) (\(\text{skCCR}_i\text{, }R\) | \(i\) = 1..1)).

(* CCR - Confirm - step 10 of 5.6.1 in page 103 of sVote specification *)
letfun CCRConfirm(vcid: agent_id, pCM: bitstring) =
  (\(i\) | \(i\) = 1..\(\psi\)) (\(\text{PCC}_i\text{, }\text{hash\,(pCM)}\) | \(i\) = 1..2)).

(* The Choice Return Codes Control Components - CCR_1, CCR_2, CCR_3 and CCR_4 *)
let CCR(i: nat, k: private _ekey, k': private _ekey, SDM_CCRChannel: channel) =
  (* setup phase - setupCCR *)
in(SDM_CCRChannel, (= VCIdTag, vcid: agent_id));

let kId = \(\text{deltaKeyAgent\,(k, vcid)}\) in
out(c, pube(kId));

let kId' = \(\text{deltaKeyAgent\,(k', vcid)}\) in
out(c, pube(kId'));

(* createCC *)
(* exponentiate E2 *)
let (pk_CCR$i:\text{public\_ekey \mid }i=1..\psi) = (mergepk\,\{\text{pube\,(skCCR}\text{, }j)\mid j=1..1\}) | i=1..\psi) in
let ELpk = mergepk\,(\text{ELsk1})\, pube(ELsk2)) in
in(c, B: bitstring);
let (E1: bitstring, E2: bitstring, KVCId: public _ekey, P: bitstring) = B in

if ProcessVoteCheck(ELpk, vcid, B, pk_CCR$i | i=1..\psi) = true then
  get bb(= vcid, B2) in 0 else
  (* Honest CCR stores in its internal log that it has proceeded to processed the pair (vcid,B) *)
event InsertBBCCR(vcid, B);

let (mEnc_c1(R), mEnc_cPsi(pk, (M$i: bitstring | i=1..\psi)), R)) = E2 in
out(c, (E2ExpoTag, 1, (mEnc_c1(\text{tildNat\,(kId, R)}), mEnc_cPsi(pk, (\text{tild\,(kId, M$i)} | i=1..\psi))\, tildNat(kId, R)))));
in(c, RAll: nat);
out(c, ((tildNat(skCCR1$i, RAll)|$i=1..$psi)));

(* Process Confirm *)
in(c, (CMTag, pCM: bitstring));
event BallotConfirmationProcessed(vcid, B);  (* Honest CCR stores in its internal log that it has contributed to the confirmation of the pair (vcid,B) *)
out(c, tildPCC(kId', hash(pCM))));

let comm =
  ! ((in(AliceSDMChannel, Id: agent_id); out(UserChannel, Id))
  )

(* The Secure Data Manager - trustworthy *)
let SDM(SDM_CCRChannel: channel, {J$i: nat |$i =1..$ n}, Csk: symmetric_key, vccsk: private skey) =

(* setup phase *)
new r: bitstring;
new r': bitstring;
new R2: bitstring;

in(UserChannel, id_ Voter: agent _id); (* SDM receives voter 's pseudonym id over a private channel *)
let VCId: agent _id = deltaId(id_ Voter ) in
out (SDM _ CCRChannel , ( VCIdTag , VCId ));
let svk: password = deltaPassword( VCId ) in

(* generate the keys *)
let kVCId: private _ ekey = ske( VCId ) in
out(c, pube( kVCId ));

(* encrypt set of voting options *)
let ({r$i: nat |$i =1..$ n}) = ({ rand (r, v(J$i))|$i =1..$ n}) in
let ctxtvID : bitstring = ({ Enc( pube( skSDM ), v(J$i), r$i) |$i =1..$ n}) in

(* compute encrypted ballot casting key *)
let BCKId = bck(id_ Voter ) in
let R = rand 2 Nat (r', id_ Voter ) in
let ctxtbId = Enc( pube( skSDM ), hash( pCC( kVCId , BCKId ) ) , R) in
out(c, (EncBCK , ctxtbId ));
event SetupCompleted ( VCId );

(* calculate the Choice return codes encryption public key -page 65 sVote specification *)
let ({ pk_ CCR $i: public _ ekey |$i =1..$ psi }) = ({ mergepk({ pube ( skCCR $j$i) |$j =1..1+1}) |$i =1..$ psi }) in

(* wait for exponentiations from CCRs *)
out(c, (EncVotingOptions, ctxtvID));
(*)
in(SDM_CCRChannel, (=ExpoVotingOptions, =CCRId1, ctxtvIDEexpo1: bitstring)); *
let ((ctxtvIDEexpo1|$i=1..1|) = CCRExpoVotingOptions(ctxtvID, VCId) in
in(c, (=ExpoVotingOptions, =CCRId2, ctxtvIDEexpo2: bitstring));

(* the following computes the quantity appearing after sentence "Computes the exponentiation of each element above to kID" in the sVote specification, page 71 *)
let ctxtpcId : bitstring = tildPCC( kId , mergeExpo( ctxtvIDEexpo1, ctxtvIDEexpo2)) in
let pCID = DecPCC( skSDM, ctxtpcId )

(* the following computes the quantity appearing after sentence "Computes the exponentiation to kID and obtains the encrypted pre-Vote Cast Return Code:" in the sVote specification, page 72 *)
in(c, (=ExpoBCK, =CCRId2, tildctxbID_2: bitstring));
let ((tildctxbID_2|$i=1..1|) = CCRExpoBCK( tildctxbID , VCId ) in
(*
let ctxbckId = tildPCC( kVCId, mergeExpo(tildctxbID_1, tildctxbID_2)) in
let ctxbckId = mergeExpo(tildctxbID_1, tildctxbID_2) in
(* compute the remaining return codes and the mapping *)
let CMId = DecPCCOneHash (skSDM, ctxbckId) in
let (pCId$i: bitstring |$i = 1..$n) = pCId in
let (1VCId$i: symmetric_ekey |$i = 1..$n) = (delta (Csk, H1 ((VCId, pCId$i))) ) |$i = 1..$n) in
let lvCCId: symmetric_ekey = delta (Csk, H1 ((VCId, CMId))) in
let vCCId = VCC (Csk, VCId) in
let CCMtable = ({ CCM_table (Enc_s(lCCId$i, CC(Csk,VCId,v(J$i)))) |$i = 1..$n}) in
let VCMtable = VCM_table (Enc_s(lvCCId, (vCCId, Sign (vccsk, VCC (Csk, VCId)))) ) in
(* event StageReached (CCMtable); *)
out (c, (CodeMapping, (CCMtable, VCMtable)))

let Serv (Ch2: channel, CompChannel: channel, InitData: bitstring, CTally: channel) =
in(CompChannel, (= VCIdTag, VCid: agent_id));
out (Ch2, (VCIdTag, VCid));

(* Retrieves Registration data for this Voting Card Id *)
let (ELpk: public_ekey, Csk: symmetric_key, VCCspk: public_skey, VCksid: bitstring, S_VCC: bitstring) = GetServData (InitData, VCid) in
(* Provides VCksid to the voting device, and asks for the ballot (alias 'vote') *)
in(Ch2, (pkall, ({ pk_CCR$i: public_ekey |$i = 1..$psi})));
out (CompChannel, (pkall, ({ pk_CCR$i: |$i = 1..$psi})));
out (CompChannel, (VCskidTag, VCksid));
in (CompChannel, (= BallotTag, B: bitstring));
let (x: bitstring, E2: bitstring, y: public_ekey, z: bitstring) = B in
event VoteReceived (VCid, B);

(* Voting part -- the Ballot processing and confirmation followed by the Server *)
let Ok1 = ProcessVoteCheck (ELpk,VCid,B, {pk_CCR$i: |$i = 1..$psi}) in

(* create CC *)
(* obtain the encrypted partial choice return codes -Step 1 in 5.4.3 in page 94 sVote specification *)
in(c, E2Expo_2: bitstring);
let ((E2Expo$j: bitstring |$j = 1..1)) = CCRExpoe2 (E2, VCid) in
let (mEnc_c1(RAll11), E2ExpoAllCtxt_1: bitstring) = E2Expo_1 in
let (mEnc_c1(RAll12), E2ExpoAllCtxt_2: bitstring) = E2Expo_2 in
(* multiply the encrypted partial choice return codes - Step 3 in 5.4.3 in page 95 sVote specification *)
let RAll: nat = mergemExpoNat (RAll1, RAll2) in
let E2ExpoAllCtxt = mergemExpo (E2ExpoAllCtxt_1, E2ExpoAllCtxt_2) in

(* obtain the partial decryptors - Step 4 in 5.4.3 in page 95 sVote specification *)
in(c, (randomExpoSK$i: nat |$i = 1..$psi)));
let ( ( (randomExpoSK$j: nat |$j = 1..1)) = CCRDecryptors (RAll) ) in
let ((pDec$i: nat |$i = 1..$psi) = (randomMult ( (randomExpoSK$j: |$j = 1..1) |$i = 1..$psi)) ) in

(* obtain the partial decryptors through partial decryption - Step 5 in 5.4.3 in page 96 sVote specification *)
in (Ch2, (=CodeMapping, (CCMtable: bitstring, VCMtable: bitstring)));
let ((pC$i: bitstring |$i = 1..$psi) = pC in
let ((CCMtable$i: bitstring |$i = 1..$n), (x$i: bitstring |$i = 1..n)) = CCMtable in
let ((1CCId$i: symmetric_ekey |$i = 1..$n)) = (delta (Csk, H1 ((VCId, pC$i))) ) |$i = 1..$n) in

(* obtain the pre Choice Return Codes through partial decryption - Step 8 in 5.4.3 in page 96 sVote specification *)
let pC: bitstring = partialDec ((pDec$i |$i = 1..$psi)), (mEnc_c1 (RAll), E2ExpoAllCtxt) in

(* obtain the short choice return codes - 5.4.4 in page 97 of sVote specification *)
in (Ch2, (=CodeMapping, (CCMtable: bitstring, VCMtable: bitstring)));
let ((pC$i: bitstring |$i = 1..$psi) = pC in
let ((CCMtable$i: bitstring |$i = 1..$n), (x$i: bitstring |$i = 1..$n)) = CCMtable in
let ((1CCId$i: symmetric_ekey |$i = 1..$n)) = (delta (Csk, H1 ((VCId, pC$i))) ) |$i = 1..$n) in
let (\{ EncCC\$i: bitstring |\$i =1..\$psi \}) = (\{ readCC(H(lCCId\$i), CCMTable\$i) |\$i =1..\$psi \}) in

\(*\) sent them to Alice's computer \(*\)

let CC\_Set : bitstring = (\{ Dec\_s(lCCId\$i, EncCC\$i) |\$i =1..\$psi \}) in (* CC\$i *)

(* sent them to Alice's computer *)

(out(CompChannel, (CodeTags, CC\_Set)));

(* vote confirmation *)

!( (* vote confirmation *)

in(Ch2, (=CodeMapping, CMTable:bitstring));

in(CompChannel, (=CMTag, cm:bitstring));

(* computing pvCCId - step 10 5.6.2 in page 103 of sVote specification *)

in(c, pCM2:bitstring);

let ((pCM\$i:bitstring|\$i =1..1)) = CCRConfirm(VCid, cm) in

let pCMAll : bitstring = mergeExpo(\{ pCM\$i|\$i =1..1+1\}) in

(* retrieving lvCCId - step 11 5.6.2 in page 103 of sVote specification *)

let lvCCId: symmetric\_ekey = delta( (Csk, H1((VCid, pCMAll))) ) in

let tmpVCC = readVCC(H(lvCCId), VCMTable) in

let vCCId : bitstring, xsign : bitstring) = Dec\_s(lvCCId, tmpVCC) in

let Ok2 = Verify(VCCspk, vCCId, xsign) in

(* event PKsReceived ( idA ); *)

(insert cb(VCid, vCCId, S\_VCC);

event HasVoted(InitData, VCid, B, vCCId, S\_VCC); (* Add confirmation to the

Confirmation Box 'cb' *)

out(CompChannel, (vCCIdTag, vCCId)); out(CTally, (VCid, B, vCCId, S\_VCC)) (* Phase 1

-- Sends the ballot to the Tally *)

))))).

(new functions, processes and rules for Universal Verifiability *)

(fun PDec\_1(private\_ekey, bitstring):bitstring. (* Partial decryption and

re-encryption applied by CCM1 *)

fun PMix\_1(private\_ekey, bitstring, bitstring):bitstring. (* Mixing proofs

computed by CCM1 *)

fun PMix\_2(public\_ekey, private\_ekey, bitstring, bitstring, bitstring):bitstring.

(* Mixing proofs computed by CCM2 *)

event HappyAuditor(bitstring, bitstring, bitstring). (* Reached

when the Auditor is satisfied with the proofs by CCM1 and CCM2 *)

(* reduction rule for verifying the output of partial decryption and mixing of CCM1 -

shuffle 1 embedded*)

fun VerifP\_11(public\_ekey, bitstring, bitstring, bitstring):bool

reduc forall E1:bitstring, E11:bitstring, E12:bitstring, Mix1E1:bitstring, Mix1E11:bitstring,

Mix1E12:bitstring, Mix2E1:bitstring, Mix2E11:bitstring, Mix2E12:bitstring, R:nat, R1:

nat, R2:nat, \{J\$i: nat |\$i =1..\$psi \}, \{K1\$i: nat |\$i =1..\$psi \}, \{K2\$i: nat |\$i =1..\$psi \}, sk1:

private\_ekey, ELpk : public\_ekey;

VerifP\_11( pube (sk1),

((Enc\_c\_1(R),Enc\_c\_2(ELpk,((v(J\$i) |\$i =1..\$psi \)),R)),

(Enc\_c\_1(R1),Enc\_c\_2(ELpk,((v(K1\$i) |\$i =1..\$psi \)),R1)),

(Enc\_c\_1(R2),Enc\_c\_2(ELpk,((v(K2\$i) |\$i =1..\$psi \)),R2)))),

(PDec\_1(sk1, (Enc\_c\_1(R),Enc\_c\_2(ELpk,((v(J\$i) |\$i =1..\$psi \)),R))),

PDec\_1(sk1, (Enc\_c\_1(R1),Enc\_c\_2(ELpk,((v(K1\$i) |\$i =1..\$psi \)),R1))),

PDec\_1(sk1, (Enc\_c\_1(R2),Enc\_c\_2(ELpk,((v(K2\$i) |\$i =1..\$psi \)),R2)))),

PMix\_1(sk1,

((Enc\_c\_1(R),Enc\_c\_2(ELpk,((v(J\$i) |\$i =1..\$psi \)),R)),

(Enc\_c\_1(R1),Enc\_c\_2(ELpk,((v(K1\$i) |\$i =1..\$psi \)),R1)),

(Enc\_c\_1(R2),Enc\_c\_2(ELpk,((v(K2\$i) |\$i =1..\$psi \)),R2)))),

(PDec\_1(sk1, (Enc\_c\_1(R),Enc\_c\_2(ELpk,((v(J\$i) |\$i =1..\$psi \)),R))),

PDec\_1(sk1, (Enc\_c\_1(R1),Enc\_c\_2(ELpk,((v(K1\$i) |\$i =1..\$psi \)),R1))),

PDec\_1(sk1, (Enc\_c\_1(R2),Enc\_c\_2(ELpk,((v(K2\$i) |\$i =1..\$psi \)),R2)))))) = true

otherwise forall pk:public\_ekey, E:bitstring, Mix1E:bitstring, Mix2E:bitstring; VerifP\_11(pk, E, Mix1E, Mix2E) = false .
(* reduction rule for verifying the output of partial decryption and mixing of CCM2 -
shuffle 2 embedded *)
VerifP12(pube(sk1),
((Enc_c1(R),Enc_c2(ELpk,(v(J$i )|$i =1..$ psi)),R)),
(Enc_c1(R2),Enc_c2(ELpk,(v(K2$i )|$i =1..$ psi)),R2)),
(Enc_c1(R1),Enc_c2(ELpk,(v(K1$i )|$i =1..$ psi)),R1))),
PDec1(sk1, (Enc_c1(R),Enc_c2(ELpk,(v(J$i )|$i =1..$ psi)),R)),
PDec1(sk1, (Enc_c1(R1),Enc_c2(ELpk,(v(K1$i )|$i =1..$ psi)),R1))),
PDec1(sk1, (Enc_c1(R2),Enc_c2(ELpk,(v(K2$i )|$i =1..$ psi)),R2))),
(Enc_c1(R),Enc_c2(ELpk,(v(J$i )|$i =1..$ psi)),R),
(Enc_c1(R1),Enc_c2(ELpk,(v(K1$i )|$i =1..$ psi)),R1)),
(Enc_c1(R2),Enc_c2(ELpk,(v(K2$i )|$i =1..$ psi)),R2))),
(PDec1(sk1, (Enc_c1(R),Enc_c2(ELpk,(v(J$i )|$i =1..$ psi)),R))),
PDec1(sk1, (Enc_c1(R1),Enc_c2(ELpk,(v(K1$i )|$i =1..$ psi)),R1))),
PDec1(sk1, (Enc_c1(R2),Enc_c2(ELpk,(v(K2$i )|$i =1..$ psi)),R2))),
(({ v(J$i )|$i =1..$ psi }),
({v(K1$i )|$i =1..$ psi }),
({v(K2$i )|$i =1..$ psi }))) = true.

(* reduction rule for verifying the output of partial decryption and mixing of CCM2 *)
VerifP2(pube(sk1), pube(sk2),
(PDec1(sk1, (Enc_c1(R),Enc_c2(mergepk(pube(sk1), pube(sk2)),(v(J$i )|$i =1..$ psi)),R))),
PDec1(sk1, (Enc_c1(R1),Enc_c2(mergepk(pube(sk1), pube(sk2)),(v(K1$i )|$i =1..$ psi)),R1))),
PDec1(sk1, (Enc_c1(R2),Enc_c2(mergepk(pube(sk1), pube(sk2)),(v(K2$i )|$i =1..$ psi)),R2))),
(({ v(J$i )|$i =1..$ psi }),
({v(K1$i )|$i =1..$ psi }),
({v(K2$i )|$i =1..$ psi })))) = true.

reduc forall sk1: private_ekey , sk2: private_ekey , M: bitstring , R: nat ;
PDec2(sk2, PDec1(sk1, (Enc_c1(R), Enc_c2(mergepk(pube(sk1), pube(sk2))), M, R)))) = M.

letfun Cleansing (VCId:agent_id , ELpk : public_ekey , { pk_ CCR $i: public_ekey |$i =1..$ psi } , B: bitstring , S_ VCC:bitstring) =
if ProcessVoteCheck(ELpk,VCId,B,(pk_ CCR|$i =1..$ psi)) = true then
if Verify(vccspk,vccd,S_ VCC) = true then
let (E1:bitstring , E2:bitstring , KCVID:public_ekey , P:bitstring) = B in
E1 .

free mixTag:bitstring .

(* The Voting Server applies the Cleansing to the Ballot Box that the Voting Server built
during the Election *)
let VoteServer (VCId1: agent_id, VCId2: agent_id, ELpk: public_ekey) =
let ({pk_CCR$i: public_ekey |$i = 1..n}) = ((mergepk({pub(e$kCCR$j) |$j = 1..i+1}) |$i = 1..n})
in
(* Cleansing -for Alice (VCId1) and Bob (VCId2) and a corrupted voter VCId *)
get bb(VCId, B1) in
(* get cb (= VCId, vCCId, S_ VCC ) in *)
in(c, (vCCId: bitstring, S_ VCC: bitstring));

(* two different outputs depending on the order where Alice and Bob's encrypted votes are output after cleansing *)
let E1: bitstring = Cleansing(VCId1, ELpk, {pk_CCR$i|$i = 1..n}, B1, vCCId, S_ VCC) in
get bb(VCId1, B11) in
(* get cb (= VCId1, vCCId, S_ VCC ) in *)
in(c, (vCCId: bitstring, S_ VCC: bitstring));
let E11: bitstring = Cleansing(VCId1, ELpk, {pk_CCR$i|$i = 1..n}, B11, vCCId, S_ VCC) in
get bb(VCId2, B12) in
(* get cb (= VCId2, vCCId, S_ VCC ) in *)
in(c, (vCCId: bitstring, S_ VCC: bitstring));
let E12: bitstring = Cleansing(VCId2, ELpk, {pk_CCR$i|$i = 1..n}, B12, vCCId, S_ VCC) in
out((VoteServerChannel, (E1, (E11, E12)) | out((CCM1Channel, (E1, (E11, E12)))))

(* Dishonest CCM1 outputs shuffled partially decrypted ciphertexts and proof of correct partial decryption and mixing *)
let CCM1 =
in((CCM1Channel, (E1:bitstring, (E11:bitstring, E12:bitstring))));
let Mix1E1 = PDec1(ELSk1, E1) in
let Mix1E11 = PDec1(ELSk1, E11) in
let Mix1E12 = PDec1(ELSk1, E12) in
out((CCM2Channel, ((E1, (E11, E12)), (Mix1E1, Mix1E11, Mix1E12)), PMix1(ELSk1, (E1,E11,E12) , (Mix1E1, Mix1E11, Mix1E12))));

let CCM2 =
in((CCM2Channel, ((E1:bitstring, (E11:bitstring, E12:bitstring)), (Mix1E1:bitstring, Mix1E11:bitstring, Mix1E12:bitstring)), PMix1(ELSk1, (E1,E11,E12) , (Mix1E1, Mix1E11, Mix1E12)));
in((VoteServerChannel, (E1orig:bitstring, (E11orig:bitstring, E12orig:bitstring))));
if (VerifP11(pub(ELSk1), (E1orig, E11orig, E12orig), (Mix1E1, Mix1E11, Mix1E12), MixProof1) = true) || (VerifP12(pub(ELSk1), (E1orig, E11orig, E12orig), (Mix1E1, Mix1E11, Mix1E12), MixProof1) = true) then
let Mix2E1 = PDec2(ELSk2, Mix1E1) in
let Mix2E11 = PDec2(ELSk2, Mix1E11) in
let Mix2E12 = PDec2(ELSk2, Mix1E12) in
out((c, ((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12)), (Mix2E1, Mix1E11, Mix1E12)), PMix2( pub(ELSk1), ELSk2, (E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix1E11, Mix1E12) ));
event HappyAuditor((E1orig, E11orig, E12orig), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12))

let Auditor =
in((CCM2Channel, ((E1:bitstring, (E11:bitstring, E12:bitstring)), (Mix1E1:bitstring, Mix1E11:bitstring, Mix1E12:bitstring)), PMix1(ELSk1, (E1,E11,E12) , (Mix1E1, Mix1E11, Mix1E12)));
in((c, (Mix2E1:bitstring, Mix2E11:bitstring, Mix2E12:bitstring)), MixProof2));
if VerifP11(pub(ELSk1), (E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), MixProof1) = true || VerifP12(pub(ELSk1), (E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), MixProof1) = true then
if VerifP2(pub(ELSk1), pub(ELSk2), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12), MixProof2) = true then
out((c, ((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12)), (Mix2E1, Mix2E11, Mix2E12), MixProof2));
event HappyAuditor((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12))
(6. Second Phase -- The Tally after all votes are collected.)

No Tally for individual verifiability properties, only for universal verifiability.

(7. Security properties)

Successfull-run Properties: Properties false iff the agents could drive the protocol to a successful state; Expected normal run.

query VCid: agent_id, B: bitstring;
   \( \text{event(PKsReceived(VCid))} \Rightarrow \text{event(NeverTrue)}. \)

Successfull-run Properties: Properties false iff the agents could drive the protocol to a successful state; Expected normal run.

query B: bitstring;
   \( \text{event(StageReached(B))} \Rightarrow \text{event(NeverTrue)}. \)

query idA: agent_id;
   \( \text{event(SetupCompleted(idA))} \Rightarrow \text{event(NeverTrue)}. \)

query idA: agent_id, Csk: symmetric_key, \( \{J$i| nat \mid 1 \leq i \leq \psi \}; \)
   \( \text{event(Confirmed(AliceData(honest(idA),Csk), \{J$i| nat \mid 1 \leq i \leq \psi \}) \Rightarrow \text{event(NeverTrue)}. \}) \)

query idA: agent_id, Csk: symmetric_key, ELpk: public_ekey, VCCssk: private_skey, B: bitstring, VCC: bitstring, S_ VCC: bitstring;
   \( \text{event(HasVoted(ServData(ELpk,Csk,VCCssk), deltaId(honest(Id)), B, VCC, S_ VCC)) \Rightarrow \text{event(NeverTrue)}. \)

query Id: agent_id, Csk: symmetric_key, E2: bitstring, ELpk: public_ekey, VCCssk: private_skey, P: bitstring, R:nat, VCCid : bitstring, E1: bitstring, S_VCC: bitstring, \( \{J$i| nat \mid 1 \leq i \leq \psi \}; \)
   \( \{\{J$iJ$j| nat \mid 1 \leq i \leq \psi \}; E2j: bitstring, R$j: nat, P$j: bitstring | 1 \leq j \leq \psi \}, VCid: agent_id; \)
   \( \text{event(InsertBBCCR(VCid, (E1,E2,pube(ske(VCid)),P))} \)
   \( \{ \text{event(InsertBBCCR(VCid, ((Enc_c1(R$j),Enc_c2(ELpk,(v(J$iJ$j)| nat \mid 1 \leq i \leq \psi ))},E2j,pube(ske(VCid)),P$j))) | 1 \leq j \leq \psi \} \}
   \( \text{& & \{J$iJ$j = ja$j| nat \mid 1 \leq i \leq \psi \} \} \}

(7. Security properties)

Universal Verifiability property: query for correct final election result.)
query E1: bitstring, E11: bitstring, E12: bitstring, Mix1E1: bitstring, Mix2E1: bitstring, Mix1E11: bitstring, Mix1E12: bitstring, Mix2E1: bitstring, Mix2E11: bitstring, Mix2E12: bitstring, R: nat, R1: nat, R2: nat, \( \{J_i: \text{nat} \mid i = 1..\psi \} \), \( \{K1_i: \text{nat} \mid i = 1..\psi \} \), \( \{K2_i: \text{nat} \mid i = 1..\psi \} \), ELpk: public_ekey;

event (HappyAuditor((E1, E11, E12), (Mix1E1, Mix1E11, Mix1E12), (Mix2E1, Mix2E11, Mix2E12))) ==>

\[ \begin{align*}
E1 &= (\text{Enc}_c1(R), \text{Enc}_c2(ELpk, (\langle v(J_i) \mid i = 1..\psi \rangle), R)) \\
& \quad \& E11 = (\text{Enc}_c1(R1), \text{Enc}_c2(ELpk, (\langle v(K1_i) \mid i = 1..\psi \rangle), R1)) \\
& \quad \& E12 = (\text{Enc}_c1(R2), \text{Enc}_c2(ELpk, (\langle v(K2_i) \mid i = 1..\psi \rangle), R2)) \\
& \quad \& \text{Mix2E1} = (\langle v(J_1) \mid i = 1..\psi \rangle) \\
& \quad \& \text{(Mix2E11} = (\langle v(K1_1) \mid i = 1..\psi \rangle) \quad \& \quad \text{Mix2E12} = (\langle v(K2_1) \mid i = 1..\psi \rangle)) \quad \| \quad (\text{Mix2E11} = (\langle v(K2_1) \mid i = 1..\psi \rangle) \quad \& \quad \text{Mix2E12} = (\langle v(K1_1) \mid i = 1..\psi \rangle))
\end{align*} \]

(* 8. Main process -- initiates the election *)

process

(* Public output from Setup(...) -- Gives the election's parameters to the Intruder. *)

let ELpk = mergepk(pube(ELSk1), pube(ELSk2)) in
out(c, ELpk); out(c, vccspk);

(*out(c, vccspk);
*)

(* Gives the honest voter's public data to the Intruder *)
out(c, deltaId(honest(idA)));
out(c, deltaId(honest(idB)));
out(c, pube(skSDM));

(* setup keys for CCRS *)
out(c, (\{ pube(kCCR$i$j) \mid i = 1..2 \} \{ j = 1..\psi \}));
out(c, (\{ pube(kCCR$i$1) \mid i = 1..2 \}));

(* setup public keys for CCMs *)
out(c, (\{ pube(ELSk$i$1) \mid i = 1..2 \}));

(* Roles for honest voter(s) -- one for Individual Verifiability. *)
Alice( Chi, AliceData(honest(idA), csk), \{ja$i\mid i = 1..\psi \} ) |
Alice( Chi, AliceData(honest(idB), csk), \{jb$i\mid i = 1..\psi \} ) |
out(c, ske(deltaId(honest(idA))));
out(c, ske(deltaId(honest(idB))));

(* Dishonest voter(s) : As many as possible for verifiability. *)
| !(new id: agent_id; out(c, id); out(c, AliceData(id, csk)))

(* Bulletin Board (ie. Voting Server) : is Dishonest *)
| !Serv(Ch2, CompServerChannel, ServData(ELpk, csk, vccssk), Ch2) *
| out(c, ServData(ELpk, csk, vccsk));

(* SDM is honest *)
| !SDM(SDM_CCRChannel, \{ja$i\mid i = 1..n\}, csk, vccssk)

(* CCR1 is honest *)
| !CCR(CCRId1, kCCR1, kCCR1', SDM_CCRChannel) \| comm

(* Cleansing is computed honestly; secret permutation is implicitly applied here *)
| !VoteServer(deltaId(honest(idA)), deltaId(honest(idB)), ELpk)

(* CCM1 is honest; CCM2 is dishonest; Auditor is honest *)
| !CCM1 | !Auditor

sources/UniversalVerifiability2CCMs–DishonestCCM1v3.1.pve
© Copyright notice:

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

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

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