Under consideration for publication in Formal Aspects of Computing

Verifying Anonymity in Voting Systems Using CSP

Murat Moran, James Heather and Steve Schneider
University of Surrey, Guildford, UK

Abstract. We present formal definitions of anonymity for voting protocols using the process algebra CSP. We analyse a number of anonymity definitions, and give formal definitions for *strong* and *weak* anonymity by highlighting the difference between these definitions. We show that the strong anonymity definition is too strong, whereas the weak anonymity definition is shown to be a suitable property for analysing voting systems. We model and analyse a conventional voting system, and Prêt à Voter, a paper-based, voter-verifiable e-voting system, as two case studies to demonstrate the validity of our formal definitions by specification checks using the trace equivalence notion in CSP and the refinement checker Failures-Divergences Refinement (FDR2). We give a detailed discussion on the results from the analysis, emphasizing the assumptions that we made in our model as well as the challenges in modelling electronic voting systems using CSP.

Keywords: Anonymity; Voting Systems; CSP; Formal Verification; Prêt à Voter; Conventional Voting System

1. Introduction

While elections offer voters a chance to exercise their democratic rights, voters must also be able to keep their vote secret. Some people may wish to publicize their votes freely although occasionally some might have to hide who they voted for. For instance, an employer can ask his employee how she has voted and take actions against her according to her political view. Thus, anonymity is one of the most critical requirements for voting systems. Many voting system protocols [Cha81, FOO92, Nef01, BG02, CRS05, Riv06, CEC$^+$08, CCM08, Adi08] have been proposed over the last decades which claim to provide anonymity, often without proof. There are a variety of definitions of anonymity in several formal languages and proof methods, such as pi calculus [FA02], modular approach [HS04], epistemic logic [GHPv05, BRS07, LJP10], probabilistic approaches [CPP06, DPP07] and both probabilistic and nondeterministic approach [BP05]. Recently, research has focused on giving precise formal definitions of desired properties of trustworthy voting systems, including anonymity [DKR06, BHM08, DKR09].

However, automated verifications of desired properties of these definitions have been been discussed or carried out. This is what we are aiming to achieve in this paper, i.e., automated formal verification of anonymity.

Correspondence and offprint requests to: Murat Moran, University of Surrey, Department of Computing, Guildford, Surrey GU2 7XH, UK. e-mail: m.moran@surrey.ac.uk, tel: +44(0)1483682263, fax: +44(0)1483686051

One of the early works in voting properties is due to Juels *et al.* [JCJ05], which describes anonymity as private elections using provable security. Further definitions of the desired properties of voting systems have been made with formal methods. For instance, Delaune *et al.* [DKR06] presents formal definitions of privacy type properties in terms of adaptive simulation in pi calculus. Delaune *et al.* [DKR09] discovers that their previous work has undesirable properties, and proposes definitions based on observational equivalence, which is not automated. Backs *et al.* [BHM08] proposes a new formalization of coercion-resistance for remote voting protocols in terms of observational equivalence, which implies vote privacy. The verification is automated, but some human effort is still required when transforming equivalences in their definition into a biprocess. The particular anonymity definition is given by Blanchet *et al.* in [BAF08]. Chothia *et al.* [COPD06] presents a framework for automatically checking anonymity using bisimilarity in the process algebraic language μCRL, analysing the FOO92 voting scheme [FOO92]. Similar to our analysis, the framework uses a passive intruder model: the observer.

We illustrate our formalism with two different voting systems. First, we verify the conventional voting system (CVS), which is the simple ballot-based voting system in which voters have to go to polling stations to vote using a booth, and put their votes in a ballot box. Although intuitively we accept that the voting system provides anonymity, we find this a chance to validate that our formal definitions of anonymity are given correctly. Second, we would like to verify a trustworthy voting system proposed more recently, which is claimed to provide anonymity based on mixnets and cryptography. Thus, we use Prêt à Voter [RS06], a paper-based voter verifiable trustworthy voting system, whose automated verification has not been done.

1.1. Contribution

We formalise two concise and generic definitions of anonymity, namely; *strong* and *weak* anonymity using the process algebra CSP. We give the first formal model of Prêt à Voter and conventional voting system. Although our approach has some constraints regarding FDR, we demonstrate that our formalization can be fully automated with it, analysing anonymity with respect to Prêt à Voter and CVS voting systems, and show they provide anonymity as expected.

1.2. Outline

In the next section, we give an overview of various formal anonymity definitions regarding voting systems. In Section 3, we introduce the CSP syntax and semantics. In Section 4, we summarize the Schneider *et al.* definition of strong anonymity in CSP with a referendum scenario as an example protocol, and we then define weak anonymity in terms of CSP for voting systems. In Section 5, we formalise our conventional voting system model, and analyse it according to the anonymity definitions that we have given. In Section 6, we give an overview of Prêt à Voter then we model and formally analyse it using CSP. Finally, in Section 7, we conclude with a discussion on formal definitions of anonymity for voting systems.

2. Anonymity Definitions in the Literature

We first consider several approaches to anonymity from the literature.

Pfitzmann *et al.* introduced anonymity as terminology to the literature [PK00]. Their definition is that in a message sender-receiver setting, anonymity is the state of being not identifiable within a set of subjects, the anonymity set, i.e. the set of all possible subjects who might cause an action. If we consider sender anonymity as the voter anonymity with respect to voter ID, and the messages being sent as the votes; then anonymity would mean that a specific vote is not linkable to any voter ID, and that no vote can be associated to any voter ID. In addition, *indistinguishability* is expressed as the state of being indistinguishable from the other elements in a set. In voting, it would mean the inability of distinguishing a vote from a bunch of votes. Furthermore, *unobservability* is also described as holding when an intruder cannot observe that a particular voter has voted, who is among the unobservability set, the set of all voters. Finally, the term, *pseudonymity*, is described as the use of pseudonyms as IDs, identifiers of subjects. Thus, we can consider a pseudonym as identifiers of the ballot papers, e.g., serial numbers, which can then be linked back to the voter IDs.

Fournet and Abadi [FA02] give a general privacy definition in pi calculus with respect to private authentication protocols. In the description, an observational equivalence notion [1] is used to formalise properties. They define anonymity as "two process behaviours should have the same interpretation on the model as long as they are indistinguishable by observation in all contexts." That is, two user processes U_1 and U_2 are identical in any context from the environment's point of view; we call this kind of anonymity definition *weak anonymity.*

Mauw *et al.* [MVd04] gives a general description of anonymity based on Pfitzmann *et al.* in [PK00]. Their definition states anonymity to be in such a way that a coercer should not be able to distinguish a user u and another user u', which is in u's anonymity group. That is, for every behaviour of the system that can be attributed to user u; there is another system behaviour that can be attributed to u'.

Shmatikov and Hughes [HS04] give a specification framework for anonymity and privacy based upon a function view in which system behaviour is described as a set of functions. In addition, the specifications of the desired properties are defined with observational equivalence using a modular approach. In the paper, several forms of anonymity in terms of a sender-receiver relation are described. We adopt some of those definitions, which are applicable to the voting scenario:

1. *Absolute voter anonymity (strong anonymity)*: an attacker cannot tell anything about the voter's identity, as every voter is plausible for every observed vote. In this model, an attacker should not be able to link a pseudonym with a sender id, e.g., voter ID and a serial number on ballot forms.
2. *Type-anonymity*: an attacker may learn the type of the voter. That is, in the case of a postal voting, if there are few voters, who registered, and cast their votes by postal voting, an attacker may reduce the number of the possible voters for a particular vote to a small group of voters, such as the voters using postal voters or the voters going polling station to vote. As another example, if there are only two voters who signed up for the remote voting, then observing an instance of a remote voting is enough for the attacker to reduce the number of possible voters for this vote to 2.
3. *Session-level*: an attacker may know the entire set of voters and the votes, but unable to link the votes to the voters' identities during an election, the session in their definition. For instance, if an attacker is observing a polling station where only one vote has been cast, he may, then deduce the voter's identity.
4. *Unobservability*: an attacker should not be able to identify that a particular voter has cast a vote; that is, a voting act should be unobservable.
5. *Untraceability*: an attacker or an observer should not be able to determine whether two votes cast in different locations have been cast by the same voter.

Juels *et al.* [JCJ05] describes anonymity as: the coercer/adversary cannot guess how a voter voted better than an adversarial algorithm whose only access is the final tally.

Kremer and Ryan [KR05] and Delaune *et al.* [DKR06, DKR09] define privacy of the election adopting Fournet and Abadi's general privacy definition [FA02] in pi calculus to voting system protocols. Delaune *et al.* uses the term "vote privacy" as a synonym for anonymity, and states that nobody has enough information to identify whether two voters swapped their votes. Therefore, if an observer cannot tell whether two arbitrary honest voters swapped their votes, then he cannot deduce information about how these voters cast their votes.

3. Communicating Sequential Processes

CSP is a formal language, designed to describe concurrent systems in terms of components that interact by means of message passing. CSP is a member within the process calculus family and introduced by Hoare in 1978 [Hoa78]. Since then it has been improved in terms of modelling concurrent systems as well as analysing security protocols [Low96, Ros97, RSG$^+$00, Ros10].

CSP allows us to model systems in terms of *processes*, which can synchronize and interact with the environment. Besides, it provides several semantic models to analyse the behaviour of processes and systems.

[1] The observational equivalence notion in this context is the same as the trace equivalence notion in CSP that we use in our definitions of anonymity in the next sections.

3.1. Syntax

Processes are defined in terms of a collection of *events* that the process can perform. In CSP, an event can happen when all processes agree on executing it, and the occurrence of an event should be regarded as an atomic action without time. That is, it happens when it is inevitable. Thus, the set of all events that are visible is called Σ, the set of all internal events is written τ and the set of all events that the process P can perform is called its *alphabet* and denoted αP. The simplest process is $STOP$, which fundamentally does nothing. $SKIP$ is another named process, which terminates immediately. However, it is not a deadlock as in $STOP$, but a successful termination. In addition, $RUN(A)$ is the process, which can always perform any member from the given set of events $A \subseteq \Sigma$. The process, RUN is defined as $RUN(A) = ?x : A \rightarrow RUN(A)$.

We can describe the CSP grammar for the processes, P, and Q, the set of events, A, variable, x, channel, c, events, a, and b, and a data, v, from data-type, T, as follows:

$P, Q :=$	processes
$STOP$	stop (deadlock)
$SKIP$	successful termination
$a \rightarrow P$	prefixing
$?x : A \rightarrow P$	prefix choice
$(a \rightarrow P \mid b \rightarrow Q)$	guarded alternative
$P \Box Q$	external choice
$\underset{x \in A}{\Box} P(x)$	indexed external choice
$P \sqcap Q$	nondeterministic choice
$\underset{x \in A}{\bigsqcap} P(x)$	indexed nondeterministic choice
$P \lhd b \rhd Q$	conditional choice
$P \parallel Q$	synchronous parallel composition
$\underset{x \in A}{\parallel} P(x)$	indexed parallel composition
$P \ {}_{\alpha P}\|_{\alpha Q} \ Q$	alphabetised parallel composition
$\underset{x \in A}{\parallel} (P(x), \alpha P(x))$	indexed alphabetised parallel composition
$P \underset{A}{\parallel} Q$	generalized parallel composition
$P \mid\mid\mid Q$	interleaving
$\underset{x \in A}{\mid\mid\mid} P(x)$	indexed interleaving
$P \setminus A$	hiding
$P[\![{}^{a}/_{b}]\!]$	renaming
$c?v$	data input
$c!v$	data output

Given a process P and an event a in Σ, the process $a \rightarrow P$ is called *prefixing*, which is initially willing to perform an event a. Therefore, it waits until the event, a, is performed then behaves like the process P. For instance, the process, $P_1 = a \rightarrow b \rightarrow STOP$ will perform the events, a and b, then it will terminate.

We can also describe recursive processes in CSP, which can perform an indefinite number of events. Thus, the processes, $P_2 = a \rightarrow b \rightarrow P_2$, is a recursively defined process, which alternates between the events, a and b. Moreover, instead of defining a recursive process with one equation, we can also use *mutual* recursion for the purpose. For instance, the process, $P_3 = a \rightarrow P_4$, behaves the same as P_2, where $P_4 = a \rightarrow P_3$.

3.2. Choice Operators

CSP provides a number of choice operators to describe processes, which offer a choice to their environments. One of them is named *guarded alternative* and denoted $' \mid '$ where distinct prefix events give options to the environment to choose any initial event and the behaviour of the corresponding process. A generalization of guarded alternative can be written as the process $?x : A \rightarrow P$, which accepts any event a from the set A and behaves like the matching process $P(a)$ if A is finite and $A \subseteq \Sigma$.

CSP also offers choice operations for the processes, which are called *external* and *nondeterministic* choice operators denoted as $\square$ and $\sqcap$ respectively. The process, $P \square Q$ can act like P or Q depending on the choice of the initial event chosen by the environment. For instance, for the process, $(a \rightarrow P) \square (b \rightarrow Q)$, if the first event, chosen is a then the process will behave as the process, P, after performing the event, a. Similarly, if the first event chosen is the event, b, subsequently the process will act as the process Q. While the external choice operator leaves the choice to its environment, in a nondeterministic process, the choice is made internally. Thus, the process, $(a \rightarrow P) \sqcap (b \rightarrow Q)$, can act as either $a \rightarrow P$ or $b \rightarrow Q$. Indexed versions of external and nondeterministic choices can be defined as $\underset{x \in A}{\square} P(x)$ and $\underset{x \in A}{\sqcap} P(x)$ respectively. In these indexed versions the choices are made among a number of processes.

In addition to these, an alternative syntax for the traditional conditional choice operator if $-$ then $-$ else is the $P \lhd b \rhd Q$ choice operator, which means that if the Boolean condition, b, is true, then it behaves as P else Q.

3.3. Parallel Operators

Systems can be a collection of processes that run in parallel and synchronise on the events that they agree to perform. Thus, *synchronous parallel composition* of the processes, P and Q is the process, $P \parallel Q$ in which case in order to perform an event, both P and Q have to be ready to perform it. We can describe it as follows:

$$?x : A \rightarrow P \parallel ?x : B \rightarrow Q = ?x : A \cap B \rightarrow (P \parallel Q)$$

A more general version of the parallel operators can be written $P \;_{\alpha P}\|_{\alpha Q}\; Q$, *alphabetised parallel.* It helps the complexed parallel combinations of processes to synchronise only the events that are common in their alphabets. Thus, they must only agree on the events in the intersection $\alpha P \cap \alpha Q$. Indexed version of alphabetised parallel is written:

$$\|_{i=1}^{n} (P_i, X_i) = P_1 \;_{X_1}\|_{X_2 \cup \cdots \cup X_n}\; (\ldots (P_{n-1} \;_{X_{n-1}}\|_{X_n}\; P_n) \ldots)$$

Where the alphabets X_i are clear, they will be elided and written $P_1 \parallel P_2 \parallel P_3 \ldots$ for readability.

The most effective parallel operator is the *generalised parallel*, commonly used with FDR [Ros10], as we can simply choose easily which events to synchronise by just giving an interface. Thus, the process $P \underset{X}{\parallel} Q$ must synchronise on the events in X and P and Q can run the events outside X independently.

Alternatively, we may wish to run any two processes independently each other, i.e., they do not synchronise on any events that they share. Interleaving operator is written "$|||$". Similarly, we can define the indexed interleaving operator as $\underset{x \in A}{|||} P(x) \mathrel{\widehat{=}} (\ldots (P_1) \;|||\; (P_2) \;|||\; \cdots \;|||\; P_n)$. All parallel operators, including interleaving are symmetric, associative and distributive over external and nondeterministic choice.

3.4. Abstraction Methods

The abstraction methods that we frequently use in our analysis are: the *hiding* abstraction method used as $P \setminus A$, which is used to make internal communications invisible to an observer and the *renaming* method shown as $P[\![R]\!]$ for a relation R, so that the occurrences of an event, a are replaced by events b such that aRb. We can use the renaming method to express that an observer can see that an event is happening, but he is unable to detect which event it is. An example for the hiding operator is that for a given set of events $A \in \Sigma$, we have the following step law:

$$(a \rightarrow P) \setminus A = \begin{cases} P \setminus A & \text{if} \quad a \in A, \\ a \rightarrow (P \setminus A) & \text{if} \quad a \notin A, \end{cases}$$

In renaming if R is a relation on the alphabet of process P, then $P[\![R]\!]$ behaves like P except that it performs different events. Whenever P can perform the event a, $P[\![R]\!]$ can perform each event from its relational image, $R(\{Pa\})$. Some earlier accounts of CSP [Sch99] have used a function or its inverse in place of the relation R, to provide *alphabet renaming* and *inverse renaming*, but in this paper we will use the more general approach using relations.

The special relational renaming that we use throughout this paper are one-to-one renaming, $P[\![^{a,\,b}/_{b,\,a}]\!]$ (a maps to b and b maps to a, many-to-one renaming, $P[\![^{a,\,a}/_{b,\,c}]\!]$ (b and c both map to a) and one-to-many renaming $P[\![^{b,\,c}/_{a,\,a}]\!]$ (a maps to both b and c).

3.5. Traces and Other Semantic Models

CSP provides a wide range of semantic models, which helps us to describe a process behaviour. We use *traces model*, $\mathcal{T}$, in this paper, which is the finite sequences of events that a process can perform. Traces are denoted $\langle a_1, a_2, \ldots, a_n\rangle$. The set of all traces of the process, P, is written *traces(P)*, which is a non-empty set as every process has the empty trace, $\langle\rangle$, in their trace sets. For instance, the traces of the process, $a \to b \to STOP$, is $\{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$. Some of the definitions in terms of traces model as follows:

$$\begin{array}{rcl}
traces(STOP) & = & \{\langle\rangle\} \\
traces(SKIP) & = & \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(a \to P) & = & \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in traces(P)\} \\
traces(P \,\Box\, Q) & = & traces(P) \cup traces(Q) \\
traces(P \sqcap Q) & = & traces(P) \cup traces(Q) \\
traces(P \setminus X) & = & \{s \upharpoonright (\Sigma \setminus X) \mid s \in traces(P)\} \\
traces(P \parallel Q) & = & traces(P) \cap traces(Q) \\
traces(P_{\alpha P}\|_{\alpha Q} Q) & = & \{s \in (\alpha P \cup \alpha Q)^* \mid s \upharpoonright \alpha P \in traces(P) \wedge s \upharpoonright \alpha Q \in traces(Q)\} \\
traces(P[\![R]\!]) & = & R[traces(P)]
\end{array}$$

$\checkmark$ is a special event that shows the process has terminated successfully, thus the process *SKIP* can also be written $\checkmark \to STOP$. If $s \in \Sigma^*$ is a finite sequence and $A \subseteq \Sigma$ then $s \upharpoonright A$ means that the sequence s is restricted to A. In relational renaming, the relation R on events lifts pointwise to a relation on traces.

The Traces model gives us sufficient information about the behaviour of our model for our formal analysis of anonymity. In addition to traces, CSP offers *failures model*, $\mathcal{F}$, which tells us more about what a process may refuse to perform and *failures/divergence model*, $\mathcal{M}$, which gives us more information on whether a process ever reaches a state that it can *diverge*, in other words, the process continues performing τ's forever and refuses all visible events.

3.6. Traces Refinement and Model Checking

Traces refinement is offered in CSP to compare behaviour of processes. If every trace of Q is also a trace of P, then Q trace-refines P or P is refined by Q, denoted $P \sqsubseteq_{\mathrm{T}} Q$, which we use heavily in this paper. If P and Q refine each other then they are trace equivalent denoted $P \equiv_{\mathrm{T}} Q$.

Failures-Divergences Refinement (FDR2) [GGH+] is the model checking tool that we use for our analysis, which was designed by Formal Systems (Europe) Ltd to check formal models created with the CSP formal language. It allows us to check assertions of refinements of specification and implementation (model). That is, $MODEL$ meets the specification, $SPEC$ if $MODEL$ is a refinement of $SPEC$. FDR checks the refinement automatically whether the $MODEL$ meets with the $SPEC$, if it does not meet FDR produces counter-examples of the refinement, which are sequences of events and considered the evidence for the violation of the specification. Although FDR is automated and easy to check refinements, the disadvantage of this model checking tool is the limitation to the number of states that it can explore.

4. Anonymity for Voting Systems

In the CSP approach to anonymity, we consider identity of the agents performing events in the form of *channel.i.x*, where the *channel* is a channel, i is the identity of the agent and x is the content of the event. Hence, anonymity can be described as that a message could have been originated from any other agent. That is, an event, *channel.i.x*, cannot, in some sense, be distinguished from *channel.j.x*, where i and j are two agents within the group of *USERS*. Thus, the set of all the messages can be written as:

$A = \{channel.i.x \mid i \in USERS, x \text{ is a data}\}$

Hence, if an observer has only access to the content of the message, that is if the identity of the agent is hidden from an observer, then the content could equally have been generated by any other agent from *USERS*.

Definition 1 (Strong Anonymity [SS96]). A process P is strongly anonymous on the alphabet $A \subset \Sigma$ if:

$$P[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \equiv_{\mathrm{T}} P \quad \text{, where } \beta \notin \Sigma$$

The original definition was cast using function renaming, which is reflected here. The first relation maps all the events on the alphabet A to a single event β, and the second maps β to all the events in A. Thus, if we rename every event of P from the alphabet, A, to the event, β (many-to-one renaming), then whenever an event, $a \in A$, is possible for P to perform; $P[\![^{\beta}/_{A}]\!]$ can perform its image β. Conversely, $[\![^{A}/_{\beta}]\!]$ is the one-to-many renaming, which maps the event, β back to the alphabet A. Thus, when β is possible for the renamed process $P[\![^{\beta}/_{A}]\!]$; any event in A is also possible for $P[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!]$, which maps every event of P on the alphabet A back to the events on A.

$P[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \equiv_{\mathrm{T}} P$ means that the two processes, P, and the renamed process, $P[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!]$, are trace equivalent, so indistinguishable from an observer's point of view regarding the traces model. The two corollaries are:

1. If the abstracted system P is anonymous on the set A and A', then P is anonymous on $A \cup A'$ if $A \cap A' \neq \emptyset$.
2. If P is anonymous on the set A and $A' \subseteq A$ then P is anonymous on the set A'.

Second anonymity definition that we are interested in is weak anonymity:

Definition 2 (Weak Anonymity). The process P is anonymous if:

$$P[\![^{a.y,\, a.x}/_{a.x,\, a.y}]\!] \equiv_{\mathrm{T}} P \quad \text{, where } a \in \Sigma \text{ and } x, y \text{ are two data of the same type}$$

That is, if we swap two instances of the same channel of P and the result is indistinguishable from an observer point of view.

For instance, suppose two honest voters are v_a and v_b and their votes are respectively, $vote.v_a.c_x$ and $vote.v_b.c_x$, where c_x is the candidate voted for. Thus, the weak anonymity definition can be applied for a voting system SYS with all possible behaviours of the voters, i.e., for any a, b and x values:

$$SYS \equiv_{\mathrm{T}} SYS[\![^{vote.v_a.c_x,\, vote.v_b.c_x}/_{vote.v_b.c_x,\, vote.v_a.c_x}]\!]$$

Thus, if the above refinement check holds, then the voting system provides anonymity under this definition.

To clarity the strong anonymity definition and the abstraction methods used, we can give this simple referendum example, where there are only two possible voters v_1 and v_2, and only one of them votes for or against a referendum. However, they are certain on how they would vote. That is, the first voter v_1 always says *yes* and the other voter v_2 always says *no* to a referendum. Then we define the process *Ref* with the external choice operator as:

$$\begin{aligned} Ref \mathrel{\widehat{=}}\ & v_1.votes \rightarrow yes \rightarrow STOP \\ & \Box\ v_2.votes \rightarrow no \rightarrow STOP \end{aligned}$$

Thus, if we want to verify whether the process satisfies anonymity, we need to check the trace equivalence $Ref[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \equiv_{\mathrm{T}} Ref$ for the set $A = \{v_1.votes, v_2.votes\}$.

$$\begin{aligned} Ref[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \mathrel{\widehat{=}}\ & v_1.votes \rightarrow (yes \rightarrow STOP \ \Box\ no \rightarrow STOP) \\ & \Box\ v_2.votes \rightarrow (no \rightarrow STOP \ \Box\ yes \rightarrow STOP) \end{aligned}$$

The refinement check does not hold, implying that the traces of each side are different with a counter-example $\langle v_1.votes, no \rangle$. The trace is a trace of $Ref[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!]$, but not of *Ref*. As a result, the process *Ref* is not anonymous as the vote *no* can give enough information about the voter. However, if we hide the events in $H = \{yes, no\}$, from the referendum observer, which means the observer does not have any direct information about their occurrence, then the new process *Ref1* becomes:

$$\begin{aligned} Ref1 \mathrel{\widehat{=}} Ref \setminus H \mathrel{\widehat{=}}\ & v_1.votes \rightarrow STOP \\ & \Box\ v_2.votes \rightarrow STOP \end{aligned}$$

Similarly, when we apply the strong anonymity definition to *Ref1*, the process $\mathit{Ref1}[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!]$ will have the same trace as *Ref1* as *yes* and *no* events are hidden. As a result, *Ref1* provides anonymity.

We can also limit the observer in a way that he can see the occurrence of events, but he is unable to identify which event the process is performing. For example, we can imagine that the votes are cast in envelopes. Thus, using *renaming* operator we can abstract away the sensitive information. Thus, the new process *Ref2* can be written:

$$\begin{aligned} \mathit{Ref2} &\mathrel{\widehat{=}} \mathit{Ref}[\![^{envelope,\, envelope}/_{yes,\, no}]\!] \\ &\mathrel{\widehat{=}} v_1.votes \to envelope \to STOP \\ &\quad \Box\ v_2.votes \to envelope \to STOP \end{aligned}$$

We can now verify whether *Ref2* provides anonymity with the trace equivalence, $\mathit{Ref2}[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \equiv_{\mathrm{T}} \mathit{Ref2}$. The equality holds for the set $A = \{v_1.votes, v_2.votes\}$, proving that the process *Ref2* provides anonymity.

Another abstraction method we may use is *masking*. In this abstraction method, the sensitive information carried by the events can be masked during the protocol using all the same events as noise. For instance, the process *Ref* can be written as the parallel combination of *Ref* with $RUN(M)$ where M is the set of events to be abstracted, namely; *yes* and *no* events. Therefore, the protocol $\mathit{Ref3} \mathrel{\widehat{=}} \mathit{Ref} \mathbin{|||} RUN(M)$ can perform any event from the set M, and the observer cannot tell whether the occurrence of such events are from *Ref* or from $RUN(M)$. Thus, the process *Ref3* with such events masked provides anonymity as the equality $\mathit{Ref3}[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \equiv_{\mathrm{T}} \mathit{Ref3}$ holds.

5. Modelling and Analysis of Conventional Voting System

We have described the anonymity requirement of a trustworthy voting system, and anonymity has been defined with CSP. The next step is to start anonymity specification check for our voting system, which behaves as a conventional voting system, and which we model in CSP. In our model (see Figure 1 for event sharing), we describe the system with respect to the processes as shown at the top of Figure 1. Each process is defined by the events that an election system can perform. Thus, the processes and the events on the model, which interact, are described as follows[2]:

A *Voter* from the set of voters chooses a candidate to vote from the given candidate list before going to the polling station and authorizing herself to the electoral official. As the choice of a candidate is made by the voter, nondeterministic choice has to be used within the definition: this choice is not under the control of the system. She then receives a ballot form with the serial number on it. In this case, as the ballot form is given by the authority to the voter, an external choice operator should be used to show that the voter accepts any ballot form given by the authority. Finally, she goes into a booth, votes according to her preference with a pen or a stamp then leaves the booth, casts her vote by dropping the ballot form to the ballot box and leaves the polling station. Hence, the voter process is modelled as:

$$\begin{aligned} VOTER(id) \mathrel{\widehat{=}} \bigsqcap_{c \in candidates} & choose.id.c \to openElection \to auth.id \to \\ & \Box_{s \in serials}\ collectform.id.s \to enterBooth!id \to mark.id.s.c \to \\ & \qquad leaveBooth!id \to cast.id.s.c \to closeElection \to VOTER(id) \end{aligned}$$

In the model, all the voters are described as below, in which voters synchronise on the events *openElection* and *closeElection*.

$$VOTERS \mathrel{\widehat{=}} \|_{id} VOTER(id)$$

The alphabets of the processes in this paper include all the events they can perform. Thus we do not present the alphabets of each processes here. However, the details about each alphabet can be seen in Appendix B. The alphabet of the process *VOTERS*, for instance, is shown below. Likewise, the alphabets of the other processes in this paper include all the events that each process has in their process definitions.

[2] In our model, we use three voters, candidates, and serials to restrict the state space in FDR.

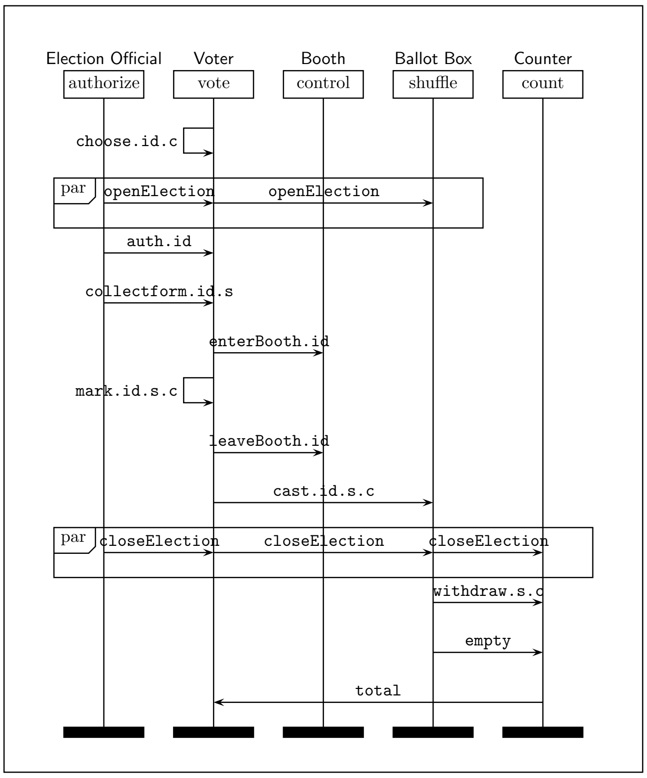


Figure 1. Conventional Voting System Design

$$\alpha\, VOTERS = \{\mid openElection,\ auth,\ collectform,\ enterBooth,\ leaveBooth,\ choose,\ mark,\ cast,\ closeElection \mid\}$$

An *Election Official* working in a polling station authenticates the eligible voters by their identification documents, and issues the ballot papers on which there are arbitrary and unique serial numbers. In order to model authentication external choice operator is used as that the election official does not choose the order of the voters who come for authentication. In our model, we have a set of pre-existing serial numbers that are assigned to the voters by the election official. Allocating a serial number to the voter should be performed nondeterministically as the official chooses them independently. Moreover, the election official does not allow two different ballot forms to have the same serial number. Therefore, two different voters cannot receive the same serial number (the same ballot form) to vote. The election official process can also open and close the election for a polling station in parallel with other processes that has *openElection* and *closeElection* in their alphabets.

$$ELECOFFICIAL(ids, serials) \mathrel{\widehat{=}} openElection \rightarrow OFFICIAL(ids, serials)$$
$$OFFICIAL(ids, serials) \mathrel{\widehat{=}} (\mathop{\Box}_{id \in ids} auth.id \rightarrow \Big(\mathop{\sqcap}_{s \in serials} collectform.id.s \rightarrow OFFICIAL(ids \setminus \{id\}), serials \setminus \{s\}) \Big)) \ \Box\ closeElection \rightarrow STOP$$

A *Booth* is a private environment for the voters to vote without distraction. Thus, in the model, the booth process can only allow one voter to go in, to vote and to leave at a time. Hence:

$$BOOTH \mathrel{\widehat{=}} \mathop{\Box}_{id \in voters} enterBooth.id \rightarrow leaveBooth!id \rightarrow BOOTH$$

A *Ballot Box* is a box where all cast votes are collected under the control of the election official. We assume that there is a private untappable channel between a voter and a ballot box (or, in other words, the voter fills in the ballot paper and casts the ballot unobserved). In the voting system model, a ballot box accepts the ballots from the voters and gathers them for the collection. Once the election is closed, the box can be opened, and all the ballot papers can be withdrawn for the tallying. Hence:

$$BOX \mathrel{\widehat{=}} openElection \rightarrow BOX1(\emptyset)$$

$$BOX1(Votes) \mathrel{\widehat{=}} \left(\mathop{\Box}_{\substack{id \in voters \\ s \in serials \\ c \in candidates}} cast.id.s.c \rightarrow BOX1(Votes \cup \{(s,c)\}) \right) \; \Box \; closeElection \rightarrow BOX2(Votes))$$

$$BOX2(\emptyset) \mathrel{\widehat{=}} empty \rightarrow STOP$$
$$BOX2(Votes) \mathrel{\widehat{=}} \mathop{\sqcap}_{(s,c) \in Votes} withdraw.s.c \rightarrow BOX2(Votes \setminus \{(s,c)\})$$

A *Counter* is the official who withdraws all the ballots in the ballot box and counts them according to the chosen candidates. Once the ballot box is empty, he announces the total votes that each candidate has. Hence:

$$COUNTER \mathrel{\widehat{=}} closeElection \rightarrow COUNTER1(0,0,0)$$
$$COUNTER1(i,j,k) \mathrel{\widehat{=}}$$
$$\left(\begin{array}{l} \left(\mathop{\Box}_{\substack{s \in serials \\ c \in candidates}} withdraw.s.c \rightarrow COUNTER1(i+1,j,k) \lessdot c = c1 \gtrdot STOP \right) \\ \Box \; withdraw.s.c \rightarrow COUNTER1(i,j+1,k) \lessdot c = c2 \gtrdot STOP \\ \Box \; withdraw.s.c \rightarrow COUNTER1(i,j,k+1) \lessdot c = c3 \gtrdot STOP \end{array} \right)$$
$$\Box$$
$$empty \rightarrow total!c1!i \rightarrow total!c2!j \rightarrow total!c3!k \rightarrow SKIP$$

Hence, the system for conventional voting is defined as a parallel composition of all the five processes defined above. As the alphabets of the processes are clear from their process definitions, we consider the parallel operator below as the alphabetised parallel operator with these obvious alphabets. Thus, starting from the most inner bracket each process is synchronised on the common events in their alphabets (for the combined processes the alphabet is the union of each alphabet).

$$SYSTEM \mathrel{\widehat{=}} VOTERS \parallel ELECOFFICIAL \parallel BOOTH \parallel BOX \parallel COUNTER$$

5.1. Sanity Checks

Before we perform the formal analysis on the model, the best practice is to check if the voting system is modelled correctly. Thus, we use a number of sanity checks on some special cases. For instance, a correctly modelled system should not allow a voter to vote after the election is closed. That is, we should not observe any cast events happening after the election is closed. Hence, if we hide all events except *closeEleciton* and the *cast* events, the *SYSTEM* process should not do a *cast* event after a *closeElection* event. The sanity specification and the assertion to be checked can be expressed as follows:

$$SNTY\_SPEC1 \mathrel{\widehat{=}} closeElection \rightarrow CLOSED \; \Box \; \left(\mathop{\Box}_{\substack{x \in voters \\ y \in serials \\ z \in candidates}} cast.x.y.z \rightarrow SNTY\_SPEC1 \right)$$
$$CLOSED \mathrel{\widehat{=}} closeElection \rightarrow CLOSED$$

$$SNTY\_SPEC1 \sqsubseteq_{T} SYSTEM \setminus \Sigma \setminus \{| \, closeElection, cast \, |\}$$

Similarly, we can also check whether the number of votes tallied in an election corresponds to the number of cast votes during the election. Thus, the specification *SNTY_SPEC2* and the assertion for this test can be defined as follows:

$$
\begin{array}{l}
\mathit{SNTY\_SPEC2} \mathrel{\widehat{=}} \mathit{COUNTTHIS(0,card(voters))}\\
\mathit{COUNTTHIS(n,t)} \mathrel{\widehat{=}} \Big(\mathop{\Box}\limits_{\substack{x\in voters\\ y\in serials\\ z\in candidates}} cast.x.y.z \rightarrow \text{if } n \leq \mathit{NumOfMaxPossVotes} \text{ then}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{COUNTTHIS(n+1,\ t+1)} \text{ else } \mathit{STOP}\\
\qquad\qquad\quad \Big)\\
\qquad\qquad\quad \Box\\
\qquad\qquad\quad \left(\mathop{\Box}\limits_{i\in numOfVotes} total.c1.i \rightarrow \mathit{COUNTTHIS1((n\text{-}i),t)} \right)\\
\mathit{COUNTTHIS1(s,z)} \mathrel{\widehat{=}} \mathop{\Box}\limits_{j\in numOfVotes} total.c2.j \rightarrow \text{if}\, s == j \text{ then } total.c3.0 \rightarrow \mathit{SKIP}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else}(\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{if}\, 0 \leq s - j \& s - j \leq \mathit{NumOfMaxPossVotes}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{then}\, total.c3.(s-j) \rightarrow \mathit{STOP}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } \mathit{STOP}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad)
\end{array}
$$

$\mathit{SNTY\_SPEC2} \sqsubseteq_{\mathrm{T}} \mathit{SYSTEM} \setminus \Sigma \setminus \{\mid cast, total \mid\}$

As expected the sanity checks are satisfied showing that the model does not let any voter vote after the election is closed, and it also does not let any lost votes or multiple cast votes happen in an election, which is a desired property for a fair election undoubtedly. Therefore, our conventional voting system model is considered correct.

5.2. Observer

In our analysis, we assume there is an observer, acting as a passive intruder and capable of seeing all the public information over the election protocol. Due to our scope, we do not consider a more powerful active intruder model, such as Dolev-Yao [DY83]. Thus, the capability of the observer is more limited compared to an active intruder. The information that the observer can see:

- elections opening and closing,
- the identity of voters and whether they have voted,
- voters getting in and out the polling station and the booth,
- voters casting a vote,
- taking ballot forms out of the ballot box and counting them,
- and the total votes that each candidate has after the final tally.

What the observer cannot see is:

- what ballot form a particular voter has been given, i.e., serial numbers. Thus, the observer cannot identify, which serial number is used by a particular voter. Moreover, no other information on the ballot form should be allowed that can prove whom the ballot form belongs to.
- how a voter marked and cast her ballot form.

Therefore, the system that the observer can see can be described as follows:

$\mathit{SYSTEM1} \mathrel{\widehat{=}} \mathit{SYSTEM} \setminus \{\mid mark, collectform \mid\}$

The system above is the CVS model, in which we hide secret information from the observer's point of view. Thus, the observer cannot see *mark* and *collectform* events happening. Similarly, the observer can see the voter is casting a vote, but not for whom. Hence, we rename *cast* events as *envelope* events. Therefore, the observer can only see an envelope, when a voter casts her vote, and it is defined as:

$\mathit{ABS\_SYSTEM} \mathrel{\widehat{=}} \mathit{SYSTEM1}[\![{}^{envelope}/_{cast.id.s.c}]\!]$

5.3. Strong Anonymity Analysis

We aim to understand the discussion of the anonymity definition and requirements made in [SS96]. Schneider and Sidiropoulos state that different definitions are required for different situations depending on the requirements of the situation. For instance, in a voting system where the anonymity of the voter identity is required, the anonymity definition that we described previously is too strong. That is, different votes are required to be generated by different voters, which means the two votes are not entirely independent. As a result, the anonymity check with their definition would fail in such systems, which require each vote might independently have been generated from the same voter.

As we defined the system that the observer can see, now we apply the strong anonymity definition of CVS model in CSP. Knowing what the observer can see in the protocol, anonymity is defined in terms of voters' choice, which is denoted as *choose* events. Therefore, the strong anonymity for the *choose* event is checked with the trace refinement, *ABS_SYSTEM* and *SPEC_STRONG*.

$$SPEC\_STRONG \mathrel{\widehat{=}} ABS\_SYSTEM[\![^{dummy}/_{choose.id.c1}]\!][\![^{choose.id.c1}/_{dummy}]\!]$$

The verification shows that the assertion does not hold with a counter example trace taken from FDR2, $\langle choose.v1.c1, choose.v1.c1\rangle$. That is, the protocol is not strongly anonymous from the observer's point of view as the CVS model does not let voters vote multiple times.

We have seen above that CVS does not meet the strong anonymity definition in [SS96], which is too strong for the voting systems. Hence, the next is to check if the weak anonymity definition is suitable specification for voting systems. For the machine-readable CSP (CSP_M) codes of the model presented here and the anonymity analysis, see Appendix B.

5.4. Weak Anonymity Analysis

In our second analysis, we use the weak anonymity definition formalised in Section 4. We analyse the system comparing these two situations, the first, in which the voters, $v1$ and $v2$, vote anyway they like, and the second that the voters swap their votes. Thus, the conventional voting system model from observer's point of view, *ABS_SYSTEM* and the system where we swap two votes, *SPEC_WEAK*, should be indistinguishable. We define the system where two votes are swapped as follows:

$$SPEC\_WEAK \mathrel{\widehat{=}} ABS\_SYSTEM[\![^{choose.v1.c,\, choose.v2.c}/_{choose.v2.c,\, choose.v1.c}]\!]$$

The system *SPEC_WEAK* is the system that we swap the votes *choose.v1.c* and *choose.v1.c*, where $c \in$ *candidates* is any candidate. Thus, anonymity verification of CVS can be shown as the trace equivalence of these two systems below. This shows us that the two systems are identical from the observer's point of view.

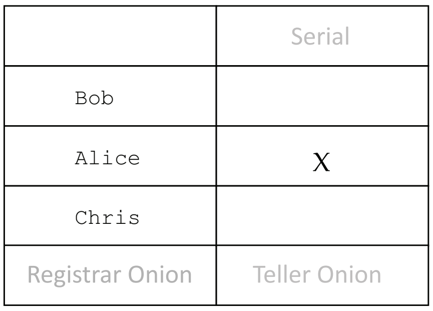
$$SPEC\_WEAK \equiv_{\mathrm{T}} ABS\_SYSTEM$$

The assertion holds, which means that the two systems refine each other, which concludes that the two systems are trace equivalent from the observer's point of view. Therefore, the conventional voting system model satisfies the anonymity requirement with the weak anonymity definition.

6. Modelling and Analysis of Prêt à Voter

Our second case study is to model Prêt à Voter and do the specification verification, which also shows that our anonymity definition is valid for the voting systems. Due to the complexity of these voting systems and the lack of formal analysis of these voting systems they can be vulnerable to several attacks. However, modelling such systems is a challenging issue as their building blocks using cryptographic systems such as homomorphic encryption schemes, ElGamal [ElG84] and Paillier [Pai99], and decryption and re-encryption mix such as in [Rya05, RS06]. In this section, we model the Prêt à Voter voting system in CSP and verify if our model does indeed satisfy anonymity property.

In this section, we first give an outline of the Prêt à Voter voting system, and informally describe what Prêt à Voter voting system parts are responsible for. Then, we formally build the processes, the events and the sets, used in the model. Moreover, we analyse the model against anonymity property in the last section.



	Serial
Bob	
Alice	X
Chris	
Registrar Onion	Teller Onion

Figure 2. Prêt à Voter Ballot Form

6.1. Outline of Prêt à Voter

Prêt à Voter is a paper based, voter-verifiable e-voting system, which uses cryptographic primitives to provide security to the system, and introduced by Peter Ryan in 2005 [Rya05] as an improvement of Chaum's scheme proposed in 2004 [Cha04]. Since then it has been enhanced by many papers [CRS05, Hea07, RBH$^+$09, RS06, Rya08, Rya06, RP05, RP10, XCH$^+$10]. In our modelling and analysis in CSP, we focus on the re-encryption mixes version of Prêt à Voter, proposed by Ryan *et al.* in 2006 [RS06].

We briefly explain how the re-encryption version of Prêt à Voter works in the following sections.

6.1.1. Voting with Prêt à Voter

Voting with Prêt à Voter and its ballot form are quite similar to the ones in the conventional voting system. Figure 2 illustrates a simple Prêt à Voter ballot form. On the left-hand column of the ballot form, a random permutation of candidates and on the right-hand column, the grid where the voter can mark her choice is presented with a perforation between them, which makes tearing a ballot form apart easier. The cryptographic value at the bottom of the right-hand column is called a *teller onion*, which embeds the exact candidate ordering on the left-hand side, which is encrypted under the teller's public key. The *registrar onion* value is used for the on-demand ballot generation in the booth, which we do not use in our model as we only model preprinted ballot forms by the authority.

An eligible voter goes to a polling station and authenticates herself to the election official and takes a random ballot form in an envelope, which is produced by an authority before the election day and needs to be protected until the voting is done. Once the voter goes into the booth, she can then mark her choice with a cross on the right-hand column, tear the ballot form apart, using the perforation, and destroy the left-hand column. Finally, she scans her ballot and takes the right-hand column of the ballot paper as her receipt, which she can later use to verify whether her ballot form appears on the web bulletin board (WBB) where the cast ballots are sent and displayed publicly. As the candidate name order on the left-hand column is random, the right-hand column does not reveal anything about her vote, keeping her vote secret. After all receipts have been sent to WBB by the election authorities, a shuffling process is executed by the mixnets using one of the appropriate public-key encryption algorithms suggested, such as ElGamal [ElG84] and Paillier [Pai99].

To provide ballot secrecy, transparency and trustworthiness, Prêt à Voter uses a combination of some cryptographic primitives such as public-key infrastructure and anonymising mixnets. The so-called *onion* embeds the candidate order, each layer in the onion is constructed by a single authority or by a set of tellers who have the distributed public secret key pairs. Thus, the onion value can only be decrypted by the trusted Tellers and with the corresponding secret keys.

A simplified version of system design of Prêt à Voter, with the messages sent between the agents is illustrated in Figure 3. The overall protocol run is:

- The authority in the voting system chooses a random value r from a seed space, computes the candidate list permutation, π, using a publicly agreed function f, $f(r) = \pi$ and finally encrypts the random value r using tellers public key, $\{r\}_{PK_T}$ and sends the tuple $(\pi, \{r\}_{PK_T})$ to the voter.
- The voter discards the permutation, π, chooses a candidate c, marks the ballot form finding the corresponding index value, ind, and sends the tuple $(ind, \{r\}_{PK_T})$ to WBB.
- Election official signs the receipt $(ind, \{r\}_{PK_T})$ and sends it back to the voter.

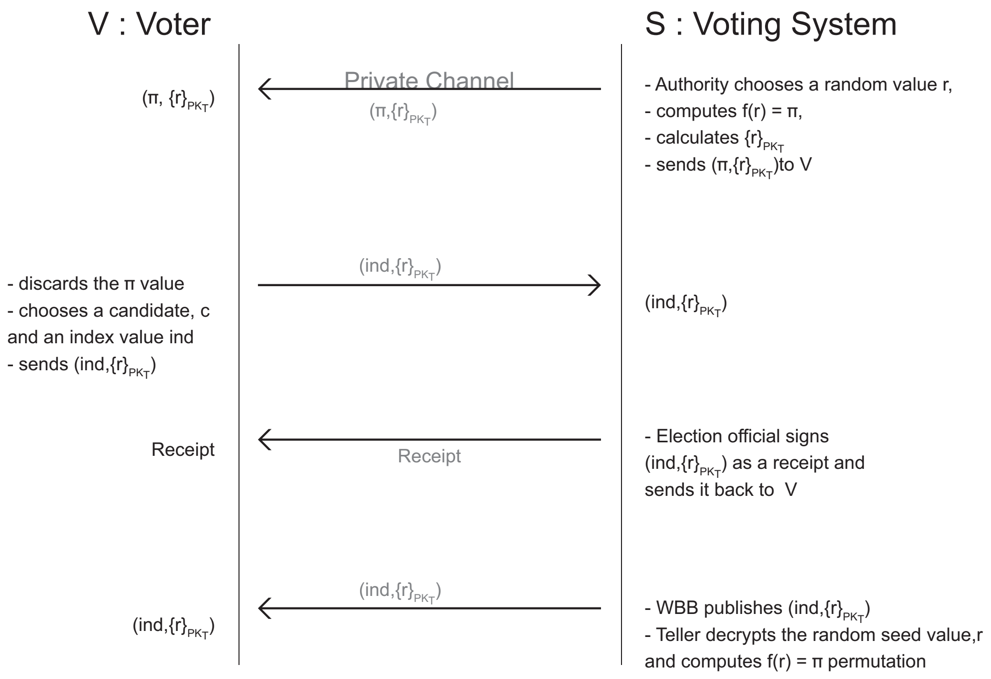

V : Voter
S : Voting System
Private Channel
$(\pi, \{r\}_{PK_T})$
$(\pi,\{r\}_{PK_T})$
- Authority chooses a random value r,
- computes f(r) = π,
- calculates $\{r\}_{PK_T}$
- sends $(\pi,\{r\}_{PK_T})$to V
$(ind,\{r\}_{PK_T})$
- discards the π value
- chooses a candidate, c and an index value ind
- sends $(ind,\{r\}_{PK_T})$
$(ind,\{r\}_{PK_T})$
Receipt
Receipt
- Election official signs $(ind,\{r\}_{PK_T})$ as a receipt and sends it back to V
$(ind,\{r\}_{PK_T})$
$(ind,\{r\}_{PK_T})$
- WBB publishes $(ind,\{r\}_{PK_T})$
- Teller decrypts the random seed value,r and computes f(r) = π permutation

Figure 3. Prêt à Voter Protocol

- WBB publishes $\{r\}_{PK_T}$ value, and the teller reveals the random value r using its secret key, SK_T, then the teller calculates π candidate permutation as $f(r) = \pi$.

Auditing: As Prêt à Voter intends to be a transparent trustworthy voting system; auditing is an important counter measure against incorrectly constructed ballots, incorrect recording of the votes and corrupt tellers. To audit ballot construction, the seed values of a number of ballot forms are revealed by stripping off the onion, hence the integrity of the ballots can be checked by recomputing the offset and onion value. On the other hand, the voter is also able to audit ballot papers during the voting phase with an empty ballot form. In this case, the voter scans the right-hand side of an empty ballot form, and the tellers reconstruct the left-hand side. The voter can check that the reconstructed left-hand side matches the printed ballot paper; the tellers also publish enough information to the WBB to enable anyone to perform appropriate cryptographic checks on the construction of the ballot paper.

One of the most important parts of the system is the tellers where decryption and anonymising mix are handled. Thus, the mechanism for auditing tellers is also introduced to be sure that decryption is done correctly by the tellers. To do this, randomized partial checking (RPC) proposed by [JJR02] is used. Tellers reveal half of their input and output links in a way that there is no complete route across so that no ballot receipt can be traced, and the remaining links are hidden. Hence, a modification of a value on one of the shuffles has a 50/50 chance of being caught. A teller has only a $\frac{1}{2^n}$ chance of getting away with modifying n votes.

Properties:

- enables a voter to check if her vote has been recorded accurately
- ensuring ballot secrecy
- ballot paper integrity
- integrity check with a single dummy vote
- the intermediate machines do not learn anything about voter's choice, which prevents information leakage.
- coercion resistant and prevents vote selling

6.2. Prêt à Voter System Components

The Prêt à Voter voting system model components can be described informally as follows:

Voter: Registered and eligible voter starts by deciding the candidate she wants to vote for. Afterwards, she authenticates herself in the polling station to the election authority. Once she is in, she then gets an empty ballot form from the authority and goes into the booth to place a mark for the candidate of her choice. The voter should shred the left-hand side (LHS) of the ballot form in the booth. Once she is outside the booth, she scans the right-hand side (RHS) of the ballot form she was given. She then gets a digitally signed receipt from the machine and leaves the polling station.

Election authority: In our model we assume that Prêt à Voter ballots are correctly created and pre-printed before the election phase, thus we do not model the authority who constructs the cryptographic seed values and generates the ballots. However, the election authority is responsible for distributing ballot forms to the voters during the authentication phase.

Machine in the booth: We call the system consisting of printer and scanner as a machine, which the voter interacts to cast her ballot form and receive the receipt. Machine is located in the polling station and outside the booth. The machine scans RHSs of the ballot forms, stores the encrypted ballot forms, and prints the receipts for the voter. The machine also passes the receipts to the WBB.

Web Bulletin Board (WBB): The WBB can store and publish all digitally signed receipts so that voters can check their serial numbers or receipts against the WBB. Moreover, the WBB can send a batch of encrypted votes to the mixnet for mixing, and it can, in addition, request a decryption for the shuffled votes from the decryption tellers to extract the real votes.

Mixnet: The mixnet receives all the encrypted votes from the WBB and re-encrypts each of them, then posts the resulting terms in a random order to the WBB. As there is no decryption in the mixnet, it needs only the tellers' public key for the re-encryption.

Decryption Tellers: As pre-printed ballots are created with the tellers' public key, they can jointly decrypt the votes shuffled and re-encrypted by the mixnet. Thus, after re-encryption mix phase, the tellers take the mixed encrypted votes, and decrypt these and perform the final tally with the plaintext values. (We have modelled the tellers as a single process; in practice, they would be a set of independently operated tellers running a threshold decryption protocol. We speak of a single teller from now on, but this is to be understood as the set of joint tellers acting in concert.)

Booth: The booth is a private environment for the voters to be able to vote freely. Only one voter should be present at a time, and no recording device should be allowed in the booth.

6.3. Definition of Functions, Data-types and Sets

Our Prêt à Voter voting system model is defined by a number of processes (see Figure 4). The figure illustrates the individual processes, shared events and the actions taken by the processes in our model. In the following paragraphs, we describe the functions, data-types and sets used to construct the events and the processes.

As we build a CSP model, we use abstract data-types. For instance, we consider encryption as a formal symbolic operation. The encryption function $enc(pkt, m)$ is the public-key encryption of the message m under the public key pkt, and it uses a constructor $Enc.(fact, fact)$. Thus, an encryption of a message, m, under a public key pkt is modelled as $enc(pkt, m) = Enc.(pkt, m)$. In public key infrastructure only the one who has the corresponding secret key (skt) can decrypt the message using the decryption function $dec(skt, enc(pkt, m))$. As a result, the inverse functions are $inverse(pkt) = skt$ and $inverse(skt) = pkt$, which make sure that the key pairs match. The key pairs, pkt and skt, are the teller's public key pairs, which are used to construct the ballot forms by the authority before the election. Teller's public key pkt is also used in re-encryption phase by the mixnet, who knows too what the public key is. As the teller knows the secret key, skt, he can extract the re-encrypted shuffled values and the onion values, which embeds the actual vote,

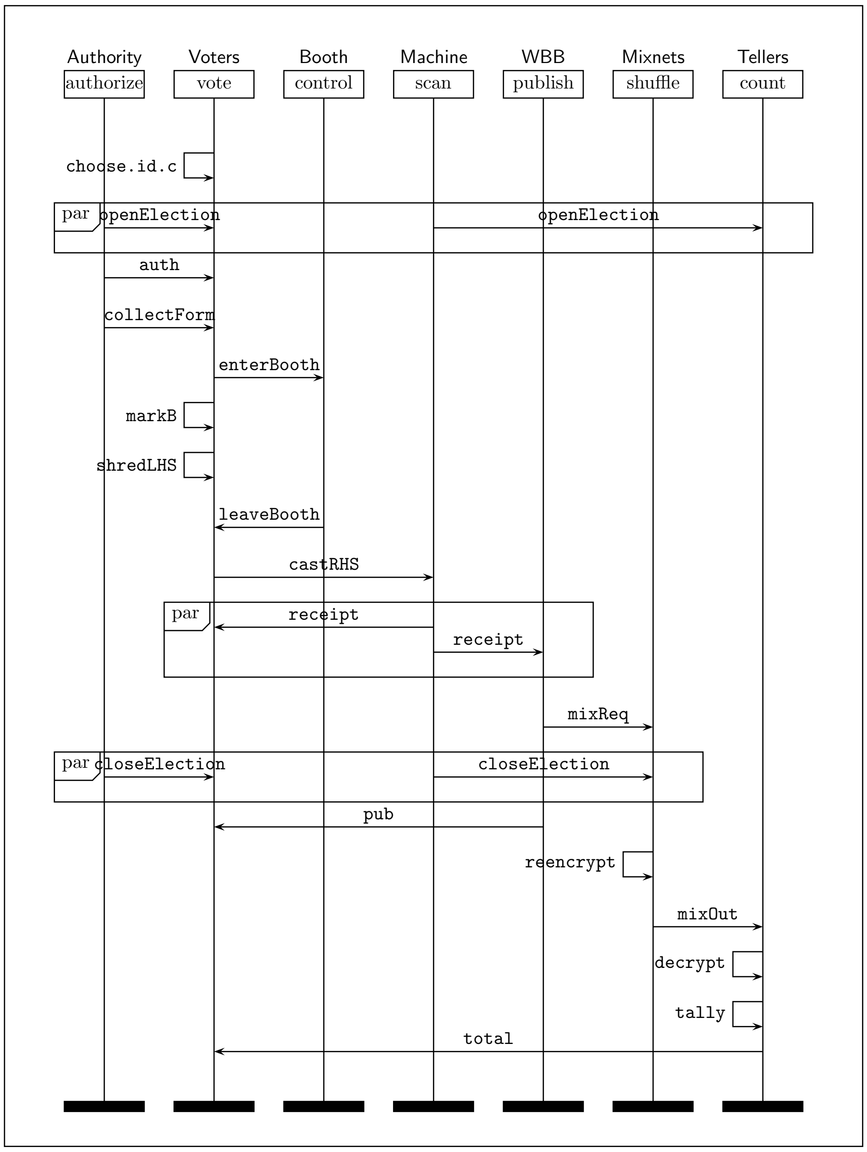


Figure 4. Prêt à Voter System Model

in the tallying phase. Similarly, the agent can sign a message, m, with his secret key, and anyone with the public key of the agent can check whether the message is coming from where it is supposed to come, it is shown as $\{\{m\}_{sk}\}_{pkt}$.

Our model has several data-types including sets and sequences and the events defined by data-types. Due to state exploration problem in FDR2, we limit the number of agents in our scenario. Thus, we consider two voters, $\{v1, v2\}$, voting for two candidates, $\{c1, c2\}$, using the two serial numbers, $\{s1, s2\}$.

A Prêt à Voter ballot form (see Figure 5) consists of a LHS and a RHS, so is our ballot form in the model. That is, on the LHS, there is a candidate list, the list of candidates, and on the RHS, there is a serial number, a grid that the voter places her mark in. Marking a ballot form is made by choosing a natural number starting from the number 1 and ending with the number of candidates in the particular election run. In our model, as there are two candidates, the voter chooses 1 for the first candidate or 2 for the second candidate. An empty ballot form is shown as the data *emptylist*. Moreover, the RHS also has an onion value at the bottom right, which embeds the candidate list of the particular ballot form encrypted by the election

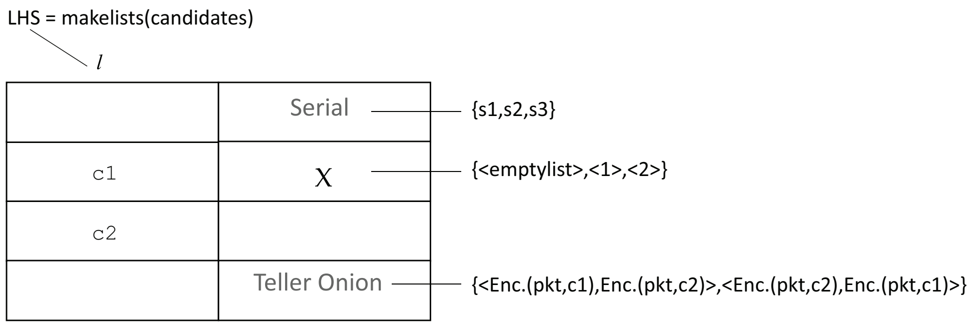

LHS = makelists(candidates)
l
Serial
{s1,s2,s3}
c1
X
{<emptylist>,<1>,<2>}
c2
Teller Onion
{<Enc.(pkt,c1),Enc.(pkt,c2)>,<Enc.(pkt,c2),Enc.(pkt,c1)>}

Figure 5. Prêt à Voter Ballot Model

authorities using teller's public key, *pkt*, before the election day. Some other data-types are:

$$\begin{aligned} fact \quad := \quad & pkt \mid skt \mid emptylist \mid ciphertext \\ & \mid Enc.(fact, fact) \end{aligned}$$

where (pkt, skt) is the public key pair of the teller.

We also use sets and sequences in the model so we can express the events. Some of them are explained below:

- The set of all possible candidate lists on the left-hand side of a ballot form:

$$LHSs = makelists(candidates) = \{\langle c\rangle\hat{\ }a \mid c \in candidates, a \in makelists(candidates \setminus \{c\})\}$$

- Empty ballot forms

$$\begin{aligned} emptyforms = \{&\langle clist, \langle ser, emptylist, \langle Enc.(pkt, c1), Enc.(pkt, c2)\rangle\rangle\rangle \\ & \mid clist \in LHSs, ser \in serials, c1, c2 \in candidates\} \end{aligned}$$

- The onions are defined as follows:

$$onions = \{\langle Enc.(pkt, c1), Enc.(pkt, c2)\rangle \mid c1, c2 \in LHSs\}$$

- Marked ballot forms

$$\begin{aligned} markedforms = \{&\langle clist, \langle ser, m, \langle Enc.(pkt, c1), Enc.(pkt, c2)\rangle\rangle\rangle \\ & \mid clist \in LHSs, ser \in serials, m \in \{1, 2\}\} \end{aligned}$$

Two special functions called *find* and *nth* are used to describe the actions taken by the agents. *find* is the function used by the voter process to see the corresponding grid for the candidate she has chosen. The function is defined with *head* and *tail* functions and returns either the number, 1 or 2. *nth* is also described by *head* and *tail* functions to find *n*th element of a sequence.

$$find(c, clist) = \begin{cases} 1 & \text{if} \quad c = head(clist) \\ 1 + find(c, tail(clist)) & \text{if} \quad c \neq head(clist) \end{cases}$$

$$nth(i, msg) = \begin{cases} head(msg) & \text{if} \quad i = 1 \\ nth(i-1, tail(msg)) & \text{if} \quad i \neq 1 \end{cases}$$

6.4. Processes and Events

As the systems are defined by the processes and the processes by the events, we now give how the processes in our Prêt à Voter model work with given data-types and events. We explain what events the processes in the model perform in the relevant process description.

6.4.1. Election Authority Process

In Prêt à Voter re-encryption version, the ballots can be pre-printed or can be printed on-demand in the booth machine. We consider the former case, in which the election authority creates the ballot forms using teller's public key and forwards pre-printed ballot forms in an envelope to the voters so authority cannot see which ballot form is used by the voter. In our model, election authority behaves as an electoral official, who creates ballot forms and issues them to the voters directly. Therefore, the information, which should not be seen by an observer, is considered to flow over a private channel, such as an envelope. In our model, an empty ballot form is denoted as $\langle \mathit{clist}, \langle \mathit{ser}, \mathit{emptylist}, \mathit{Enc.}(\mathit{pkt}, \mathit{clist})\rangle\rangle$, in which *clist* is a permutation of the candidates list on the LHS, *ser* is the serial number taken from the set of serials, *emptylist* is the grid where the voter places her mark for her candidate and the last value $\langle \mathit{Enc.}(\mathit{pkt}, c1), \mathit{Enc.}(\mathit{pkt}, c2)\rangle$ presents the onion. As already seen, the onion value is the encryption of the list, *clist*, under the teller's public key, *pkt*. Consequently, the onion values can only be revealed by the decryption teller with the corresponding secret key.

Election authority synchronises on the channel, *openElection*, at first. Then upon request from the voters, he authenticates voters with their identification and issues them an empty ballot form with nondeterministically chosen serial number and candidate list with the channel *collectform*. Authority can perform these actions as long as there are eligible voters and serial numbers. Finally, he can close the election.

$$
\begin{aligned}
&AUTHORITY(ids, serials, lhs) \mathrel{\widehat{=}} openElection \rightarrow AUTHORITY1(ids, serials, lhs)\\
&AUTHORITY1(ids, serials, lhs) \mathrel{\widehat{=}} (\\
&\qquad \mathop{\Box}_{id \in ids} auth.id \rightarrow\\
&\qquad\qquad (\mathop{\sqcap}_{\substack{seri \in serials\\ ls \in lhs}} collectform.id.\langle ls, seri, emptylist, onionlist(pkt, ls)\rangle \rightarrow\\
&\qquad\qquad\qquad AUTHORITY1(ids \setminus \{id\}, serials \setminus \{seri\}, lhs)\\
&\qquad\qquad)\\
&\qquad)\\
&\qquad \Box\\
&\qquad closeElection \rightarrow STOP
\end{aligned}
$$

As we did in the conventional voting system model, the alphabet of each process is the set of all events that a process can perform. Thus, we present here the alphabet of $AUTHORITY$ below, and the rest of the alphabets for each process in the system can be defined similarly. For each alphabet, you can see the details in Apendix A.

$$\alpha AUTHORITY = \{| openElection, auth, collectform, closeElection |\}$$

6.4.2. Voter Process

Having chosen a candidate to vote for, the voter authenticates herself and accepts any ballot form given by the election authority. Then, she goes into the booth to select a candidate with the channel, *mark*, and after destroying the left-hand side, she leaves the booth. Afterwards, she casts her vote under the supervision of election authority using the machine supplied. Having cast the ballot form, the voter is provided a receipt of her vote, which is the right-hand side of the ballot form, including serial number. Once the voter gets her receipt and goes out the polling station, the voting finishes for her. The serial numbers on the receipts are used by the voter verification on the web bulletin board. Once all the voters have finished voting, they synchronise on the event *closeElection.*

$$
\begin{aligned}
&VOTER(id) \mathrel{\widehat{=}} \mathop{\sqcap}_{c \in candidates} choose!id.c \rightarrow openElection \rightarrow auth.id \rightarrow\\
&\qquad \mathop{\Box}_{\langle l,s,emptylist,o\rangle \in emptyforms} collectform.id.\langle l, s, emptylist, o\rangle \rightarrow enterBooth.id \rightarrow\\
&\qquad\qquad mark.\langle l, s, find(c,l), onionlist(pkt, l)\rangle \rightarrow\\
&\qquad\qquad\quad shredLHS.\langle s, find(c,l), onionlist(pkt, l)\rangle \rightarrow\\
&\qquad\qquad\qquad leaveBooth.id \rightarrow cast.\langle s, find(c,l), onionlist(pkt, l)\rangle \rightarrow\\
&\qquad\qquad\qquad\quad receipt.\langle s, find(c,l), onionlist(pkt, l)\rangle \rightarrow
\end{aligned}
$$

$$closeElection \rightarrow STOP$$

Thus the process for all voters is described by the parallel composition of the voters as;

$$VOTERS \mathrel{\widehat=} \underset{\{|openElection, closeElection|\}}{\|_{id \in voters}} VOTER(id)$$

6.4.3. Machine Process

MACHINE process is the medium by which the voters cast their vote and receive their receipts. It synchronises on the event *openElection* with the authority and the voters, then starts receiving the cast RHSs of the ballot forms and printing out the receipts for the voters before the election is closed. Because the machine accepts any RHSs, cast by the voters, we use external choice operator here.

$$\begin{aligned}
&MACHINE \mathrel{\widehat=} openElection \rightarrow MACHINE1\\
&MACHINE1 \mathrel{\widehat=} (\\
&\qquad \mathop{\Box}_{rhs \in markedRHSs} cast.rhs \rightarrow receipt.rhs \rightarrow MACHINE1\\
&\qquad)\\
&\qquad \Box\\
&\qquad closeElection \rightarrow MACHINE
\end{aligned}$$

6.4.4. Web Bulletin Board Process

Once the processes running in parallel agreed on *openElection*, the process *WBB* starts receiving the digital copies of cast RHSs by the machine process on the channel *receipt*. In addition, it keeps the receipts in a set called *Receipts*, which is initially an empty set. The WBB process can also request shuffling for each vote individually[3] from the mixnet process on the channel *mixReq*. However, serial numbers are stripped off before shuffling request sent by the WBB process. Moreover, the WBB requests mixing for only the chosen candidate, not for the whole onion value. Once the election is closed and there is no vote left to request a shuffle for, the process publishes all the receipts kept in the set, *Receipts*. As a remark, the published receipts consist of a serial number, an index indicating where the mark is and an onion value, which is the encryption of the candidate list on the LHS of the ballot form.

$$\begin{aligned}
&WBB \mathrel{\widehat=} openElection \rightarrow WBB1(\emptyset)\\
&WBB1(Receipts) \mathrel{\widehat=} (\mathop{\Box}_{\langle s,i,o\rangle \in markedRHSs} receipt.\langle s,i,o\rangle \rightarrow mixReq.\langle nth(i,o)\rangle \rightarrow\\
&\qquad WBB1(Receipts \cup \{\langle s,i,o\rangle\})\\
&\qquad)\\
&\qquad \Box\\
&\qquad closeElection \rightarrow WBB2(Receipts)
\end{aligned}$$

$$\begin{aligned}
&WBB2(\emptyset) \mathrel{\widehat=} STOP\\
&WBB2(Receipts) \mathrel{\widehat=} \mathop{\sqcap}_{rcp \in Receipts} pub.rcp \rightarrow WBB2(Receipts \setminus \{rcp\})
\end{aligned}$$

6.4.5. Mixnet Process

The *MIX* process behaves as a typical mixnet, which does a mix for the digital copies of receipts. However, as we explained previously, the receipts arrive in the mixnet one by one, and the process saves them in a set called *Batch*, which is initially empty. Before shuffling the votes, the process *MIX* re-encrypts each encrypted vote with teller's public key, *pkt*. The process considers each encrypted onion value as a different encryption. As soon as no votes left for re-encryption, defined by the channel, *bagempty*, the mixnet begins giving the re-encrypted votes out to the WBB nondeterministically and one by one. In addition, the mixnet forwards a different re-encrypted value to the WBB every time.

[3] In the Prêt à Voter voting system, the mix requests can also be sent as a batch of votes at once other than one by one.

In our abstraction modelling level, we assume that the mixnet is authentic and does not reveal any information about the votes. Furthermore, as the mixnet does not do any decryption, it does not possess any secret key. As having more than one mixnet would not change anything due to nondeterministic construction of mixnet, we use only one mixnet to re-encrypt and shuffle the votes for our modelling purpose.

$$
\begin{aligned}
&MIX \mathrel{\widehat=} openElection \rightarrow MIX1(0, \emptyset)\\
&MIX1(i, Batch) \mathrel{\widehat=} (\\
&\qquad \mathop{\Box}_{\langle Enc.(pkt,c)\rangle \in chosenCand} mixReq.\langle Enc.(pkt, c)\rangle \rightarrow \\
&\qquad\qquad reencrypt.reEnc(pkt, \langle Enc.(pkt, c)\rangle) \rightarrow \\
&\qquad\qquad\quad MIX1(i + 1, Batch \cup \{i, reenc(pkt, \langle Enc.(pkt, c)\rangle)\}) \\
&\qquad) \\
&\qquad \Box \\
&\qquad closeElection \rightarrow MIX2(Batch)
\end{aligned}
$$

$$
\begin{aligned}
&MIX2(\emptyset) \mathrel{\widehat=} bagempty \rightarrow STOP\\
&MIX2(Bag) \mathrel{\widehat=} \mathop{\sqcap}_{(i,\langle Enc.(pkt,c)\rangle)\in Bag} mixOut.\langle Enc.(pkt, c)\rangle \rightarrow \\
&\qquad\qquad MIX2(Bag \setminus \{(i, \langle Enc.(pkt, c)\rangle)\})
\end{aligned}
$$

6.4.6. Decryption Teller Process

The *TELLER* process receives the shuffled re-encrypted onion values from the mixnet, transferred on the channel *mixOut*. Because re-encryption and the ballot generation is performed under the teller's public key *pkt*, the teller now can decrypt each of them, and tally the plaintext values, with the event *tally*. Upon counting all the votes; teller publishes the election results for each candidate on the channel *total*.

$$
\begin{aligned}
&TELLER \mathrel{\widehat=} openElection \rightarrow TELLER1(0, 0)\\
&TELLER1(i, j) \mathrel{\widehat=} (\mathop{\Box}_{(i,\langle Enc.(pkt,a)\rangle)\in encList} mixOut.\langle Enc.(pkt, a)\rangle \rightarrow \\
&\qquad decrypt.dec(skt, \langle Enc.(pkt, a)\rangle) \rightarrow tally. < a > \rightarrow \\
&\qquad\quad (\\
&\qquad\quad \text{if } a = c1 \text{ then } TELLER1(i + 1, j) \text{ else} \\
&\qquad\qquad (\\
&\qquad\qquad \text{if } a = c2 \text{ then } TELLER1(i, j + 1) \text{ else } STOP \\
&\qquad\qquad) \\
&\qquad\quad) \\
&\qquad) \\
&\qquad \Box \\
&\qquad bagempty \rightarrow total.c1.i \rightarrow total.c2.j \rightarrow SKIP
\end{aligned}
$$

6.4.7. System Process

The Prêt à Voter voting system model is the parallel composition of the processes defined previously with the alphabets of each process (refer to Appendix A for the CSP$_M$ codes of the model and the alphabets of each process). Thus, the composition can be defined with an alphabetised parallel operator with the corresponding alphabets as follows:

$$
\begin{aligned}
&SYSTEM \mathrel{\widehat=} \\
&\quad (((((VOTERS \| AUTHORITY(voters, serials, LHSs)) \| MACHINE) \| BOOTH) \| WBB) \| MIX) \| TELLER
\end{aligned}
$$

6.5. Sanity Checks

Having modelled Prêt à Voter, the sanity of the model has also been checked throughout four sanity specification to verify whether the model is built correctly.

i. The first one of them is that *no one can be authenticated twice*. Thus, the specification can be modelled as:

$$
\begin{aligned}
&AUTH(v) \mathrel{\widehat{=}} auth.v \rightarrow STOP\\
&SNTY\_SPEC1(voters) \mathrel{\widehat{=}} \mathop{|||}_{id \in voters} AUTH(id)
\end{aligned}
$$

Hence, the refinement below, in which the events in Σ other than the event *auth* are hidden, should be satisfied by the voting system model, whose behaviour is expected to be limited by the specification. If *SYSTEM* does not perform any unexpected events, e.g., authenticating a voter twice, then assertion should hold otherwise FDR2 could produce an example trace that violates the refinement.

$$SNTY\_SPEC1(voters) \sqsubseteq_{\mathrm{T}} SYSTEM \setminus (\Sigma \setminus \{| \, auth \, |\})$$

ii. The second sanity check is that *no one can mark a ballot form before authenticating themselves.* As we are interested in only the authentication and marking actions, we hide all the other events from the system. Hence, once a voter is authenticated, she can then mark a ballot form, or any other authentication is also allowed (for this sanity check a voter's voting multiple times is not a matter). Thus, the specification process *SNTY_SPEC2(voters)* can be written as:

$$
\begin{aligned}
&ECHECK(v) \mathrel{\widehat{=}} auth.v \rightarrow AUTHED(v)\\
&AUTHED(v) \mathrel{\widehat{=}} \left(\mathop{\Box}_{x \in markedforms} mark.v.x \rightarrow AUTHED(v) \right)\\
&\qquad \Box\\
&\qquad \left(\mathop{\Box}_{id \in voters} auth.id \rightarrow AUTHED(v) \right)\\
&SNTY\_SPEC2(voters) \mathrel{\widehat{=}} \mathop{|||}_{id \in voters} ECHECK(id)
\end{aligned}
$$

$$SNTY\_SPEC2(voters) \sqsubseteq_{\mathrm{T}} SYSTEM \setminus (\Sigma \setminus \{| \, auth, mark \, |\})$$

Hence, the specification allows the system to let only the authenticated voters mark a ballot form. Therefore, no voters can vote unless they are authenticated before, if the refinement for this sanity check holds.

iii. The third sanity check is that *no one can vote after election closed.* That is, we should not observe any cast events happening after the election is closed. Hence, if we hide all events except *closeEleciton* and the *cast* events, the *SYSTEM* process should not do a *cast* event after a *closeElection* event. The sanity specification and the refinement can be expressed as follows:

$$
\begin{aligned}
&SNTY\_SPEC3 \mathrel{\widehat{=}} closeElection \rightarrow CLOSED\\
&\qquad \Box\\
&\qquad \left(\mathop{\Box}_{x \in markedRHSs} cast.x \rightarrow SNTY\_SPEC3 \right)\\
&CLOSED \mathrel{\widehat{=}} closeElection \rightarrow CLOSED
\end{aligned}
$$

$$SNTY\_SPEC3 \sqsubseteq_{\mathrm{T}} SYSTEM \setminus \Sigma \setminus \{| \, closeElection, cast \, |\}$$

iv. The last sanity check is that *the number of votes tallied corresponds to the number of cast votes.* What we check is whether the total number of cast votes is the same as the votes tallied. Because the events that we are interested in are the *cast* and *total* events, we hide the rest of the events in Σ from the system. Hence, the specification *SNTY_SPEC4* and the refinement can be defined as follows:

$$
\begin{aligned}
&SNTY\_SPEC4 \mathrel{\widehat{=}} COUNT(0)\\
&COUNT(n) \mathrel{\widehat{=}} \left(\mathop{\Box}_{x \in markedRHSs} cast.x \rightarrow COUNT(n{+}1) \right)\\
&\qquad \Box\\
&\qquad \left(\mathop{\Box}_{i \in numOfVotes} total.c1.i \rightarrow COUNT1(n{-}i) \right)\\
&COUNT1(j) \mathrel{\widehat{=}} total.c2.j \rightarrow STOP
\end{aligned}
$$

$$SNTY\_SPEC4 \sqsubseteq_{\mathrm{T}} SYSTEM \setminus \Sigma \setminus \{| \, cast, total \, |\}$$

All the sanity checks defined and explained above are satisfied by the Prêt à Voter voting system CSP model. As a result, it is shown that there is no unexpected action taken by the model.

6.6. Anonymity Analysis of Prêt à Voter

We have already defined and tested the anonymity definition with the conventional voting system in Section 5. Having modelled Prêt à Voter in process algebra CSP in previous section, we can now conduct further anonymity analysis. In our model, we check if the Prêt à Voter voting system provides anonymity property with respect to the same anonymity definitions as used previously. Throughout the anonymity analysis of the model, we use the specification checks against an observer defined earlier. To recall, breaking anonymity would mean that an observer can link a voter to her vote. In the first analysis, no machine misbehaves as we only consider the attacker as an observer, but we also perform a formal verification of the model with a corrupted election official. We assume that public-key infrastructure is secure, and the observer does not have enough computational power to break these key pairs. As we symbolise encryption and decryption, we are abstracting away any cryptographic vulnerabilities and attacks on them. As any recording device in the booth would violate the privacy, we must also assume that there is a private environment to vote.

6.6.1. Observer

As in CVS verification, we have an observer who can see all the public information and some sensitive data that is listed below so the observer is limited in terms of the events that he has access to. Thus, the capabilities of the observer are:

- election's opening and closing,
- the identity of the voters and if they have voted in the authentication process,
- who gets in and out the booth,
- voters shredding the left-hand side of a ballot form, but not able to see what the left-hand side is
- voters casting a vote outside the booth,
- voters collecting a receipt from the machine,
- the receipts published by the WBB,
- the WBBs requesting a mix from the mixnet and received re-encrypted shuffled votes,
- plaintext votes after the teller's decryption process,
- tallying of each vote and the total votes that each candidate has after tallying finishes.

What the observer cannot see is:

- what ballot form a voter has been given, namely; the event *collectform*. Thus, the observer cannot identify which serial number is used by a particular voter. Moreover, no other information on the ballot form should be allowed that can prove to whom the ballot form belongs.
- voters marking her vote in the booth, which is modelled with the event, *mark*.
- obviously, the observer should not see which candidate the voter has chosen to vote.

Therefore, the system that the observer can see can be described as follows:

$$SYSTEM1 \mathrel{\widehat{=}} SYSTEM \setminus \{| \, mark, collectform \, |\}$$

We use renaming abstraction method to hide the sensitive information from the observer. As a result, the observer cannot distinguish the difference among encrypted onion values. Thus, all the encryption values look the same to the observer. Here *ABS_SYSTEM* is the system that is seen by the observer, which is composed by the renaming operator and the special function, *mask()*. It converts all encrypted data to one single type, *ciphertext*.

$$\begin{aligned} ABS\_SYSTEM \mathrel{\widehat{=}} SYSTEM1 & [\![^{shred.\langle s,x,mask(o)\rangle} / _{shred.\langle s,x,o\rangle}]\!] \\ & [\![^{cast.\langle s,x,mask(o)\rangle} / _{cast.\langle s,x,o\rangle}]\!] \\ & [\![^{receipt.\langle s,x,mask(o)\rangle} / _{receipt.\langle s,x,o\rangle}]\!] \\ & [\![^{pub.\langle s,x,mask(o)\rangle} / _{pub.\langle s,x,o\rangle}]\!] \\ & [\![^{mixReq.mask(enc)} / _{mixReq.enc}]\!] \\ & [\![^{mixOut.mask(enc)} / _{mixOut.enc}]\!] \\ & [\![^{reencrypt.mask(enc)} / _{reencrypt.enc}]\!] \end{aligned}$$

6.6.2. Strong Anonymity Analysis

From our anonymity analysis over the conventional voting system, we have shown that strong anonymity is not a valid specification check for the voting systems. We also expect this to be the same for the Prêt à Voter voting system.

In order to apply the strong anonymity definition, we abstract the *choose* events to another event called, *dummy* event, which is not in Σ (*SYSTEM*'s alphabet). Therefore, we expect that once the event *choose* is renamed back to the Σ, all the events in this set should be possible, which means a *choose* event could have been equally generated by some other voter. However, from the fact that the voting systems require that each vote should be cast by a different voter, the voting system does not satisfy the strong anonymity. Hence, if we apply the strong anonymity definition, the specification process is defined as:

$SPEC\_STRONG \mathrel{\widehat{=}} ABS\_SYSTEM[\![^{dummy}/_{choose.id.c1}]\!]\,[\![^{choose.id.c1}/_{dummy}]\!]$

As we expected, the refinement, $ABS\_SYSTEM \sqsubseteq_{\mathrm{T}} SPEC\_STRONG$ does not hold, proving that the strong anonymity definition is too strong for the Prêt à Voter voting system as well. The counterexample trace is:$\langle choose.v1.c1, choose.v1.c1\rangle$. The trace tells us that the voter $v1$ can vote twice for the same candidate, which is a violation to the protocol. Therefore, Prêt à Voter is not strongly anonymous either due to strong anonymity definition.

6.6.3. Weak Anonymity Analysis

We now give the trace refinement results of the weak anonymity analysis. We expect to happen that the two systems, the normal *ABS_SYSTEM* and the system, *SPEC_WEAK*, where we swap two votes are trace equivalent from the observer's point of view.

Suppose two votes are *choose.v2.c* and *choose.v1.c*, where *v1* and *v2* are candidates and *c* is any candidate from the candidates list. Hence, to apply the weak anonymity definition, we use renaming operator to swap the votes as shown below. This means that we swap the voters,*v1* and *v2* over, who voted for the same candidate *c*.

$SPEC\_WEAK \mathrel{\widehat{=}} ABS\_SYSTEM[\![^{choose.v1.c,\, choose.v2.c}/_{choose.v2.c,\, choose.v1.c}]\!]$

Hence, the assertions, $ABS\_SYSTEM \sqsubseteq_{\mathrm{T}}$ and $\sqsupseteq_{\mathrm{T}}$ $SPEC\_WEAK$ are both satisfied by the Prêt à Voter voting system model, showing that the two systems are trace equivalent. As a result, the Prêt à Voter voting system provides anonymity in terms of weak anonymity definition.

6.6.4. Misbehaving Agents

In the previous section, we analyse Prêt à Voter with respect to an observer whose capabilities are the public information on the channels that he is allowed to see. In this section, we give more power to an observer by giving away the information kept by the trusted agents to demonstrate in which cases the anonymity is not satisfied.

A trusted election official holds critical information about the privacy of the election and anonymity of the voters. For instance, the *auth* event, which illustrates the authentication process between the authority and the voters. An observer has already access to this information in the previous scenarios. However, the event *collectform* is hidden from the observer's point of view. This event carries the information about the ballot papers and the voters they are assigned. We demonstrate that the observer who can see *collectform* events can break voter's anonymity.

As we hide *collectform* from the observer in the process *SYSTEM1*, we need a new process *SYSTEM2* in which only the *mark* events are hidden. Hence, the process *ABS_SYSTEM2* should be defined in terms of *SYSTEM2* exactly the same way as described above.

$SYSTEM2 \mathrel{\widehat{=}} SYSTEM \setminus \{|\, mark \,|\}$

The specification *SPEC_WEAK2* is then can be written:

$SPEC\_WEAK2 \mathrel{\widehat{=}} ABS\_SYSTEM2[\![^{choose.v1.c,\, choose.v2.c}/_{choose.v2.c,\, choose.v1.c}]\!]$

At this point, to verify the anonymity, we check whether the refinement, $ABS\_SYSTEM2 \equiv_{\mathrm{T}} SPEC\_WEAK2$ holds. As one can expect, the specification check fails as the observer knows the left-hand side of the ballot form via *collectform*, which shows the candidate order. In addition using *shredLHS* events he can deduce in which grid the voter placed a mark on the ballot form. Therefore, anonymity is not satisfied if the authority gives away the vulnerable information on the channel *collectform* to the observer.

7. Results and Discussion

In this paper, we have investigated formal anonymity definitions, in particular *strong* and *weak* anonymity for the voting systems. We have used conventional voting system and Prêt à Voter as two case studies to validate the weak anonymity definition using process algebra CSP and FDR2 refinement checker. First, we present our experimental results in the following section, then we conclude with a discussion.

7.1. Results

The first motivation of the paper was to explore why the strong anonymity is too strong for voting protocols. Our analysis has revealed that using the strong anonymity definition, we could not verify whether the voting system models provide anonymity, which does not let multiple votes to be cast. The main reason is that intuitively a voting process should not allow a voter to vote twice. Thus, while strong anonymity specification requires two voters should be independent, the voting systems mandate that two arbitrary voters must be different, meaning the two instances of voters are not entirely independent. Therefore, we used the weak anonymity as our specification, and showed that the anonymous voting systems are indeed anonymous, and they can be verified with this specification automatically. Therefore, we conclude that the weak anonymity definition is a generic specification for all voting systems

In particular, we verified that CVS and Prêt à Voter did not provide anonymity with the strong anonymity definition as we expected. We, however, could prove that these voting systems actually satisfy the requirement with the weak anonymity definition.

Our anonymity definition covers the voting systems in which the final tally is published. Even if all the voters vote for the same candidate, although how the voter has voted is obvious; an observer still cannot identify whether two voters have swapped their votes. Hence the anonymity definition, however, is satisfied by the system. Contrarily, in the corner case of one voter voting in an election, the definition is still applicable, showing that even in this case the voting systems provide anonymity.

We also investigated the case in which the authority can assign the same serial number to two different voters for Prêt à Voter. One of the trace examples produced from the strong anonymity analysis is similar to the previous ones, $\langle \mathit{choose.v2.c1}, \mathit{choose.v2.c1} \rangle$. This means that $v2$ can vote for the same candidate twice. But, the weak anonymity is still satisfied, though two different voters use the same serial number. In the same way, we also demonstrated that the CVS model satisfies the weak anonymity definition, even if there is no serial number at all, but not the strong anonymity as the trace, $\langle \mathit{choose.v3.c2}, \mathit{choose.v3.c1} \rangle$, violates the specification.

7.2. Discussion

Although, the strong anonymity definition is too strict for the voting systems, we can still use it as an anonymity specification in such systems that allow a voter to vote multiple times in an election where only the winner of the election is announced not the full tally.

Earlier, we mentioned the state exploration problem in FDR (a general problem for all model checking tools such as ProVerif [BS12]). Thus, we could only verify the voting systems we modelled on relatively small models with only a few voters and candidates. To generalize the verification to models of arbitrary size, there are several techniques in the literature, such as *structural* and *data-independent* induction[Ros97, Laz99, Ros10]. However, data-independence techniques do not easily apply to the models that we have developed as the established results require rather strict conditions on the models, which ours do not satisfy and where it is not currently clear whether it is possible to manipulate them into the appropriate form. For instance, using functions (such as *card*) on data-types is not allowed, and the replicated parallel operator is banned, which we use heavily in defining our models. The most important limitation is on the specifications; there should be no hiding or renaming operators used in the specification, which our anonymity specifications are based on. Therefore, the data-independent induction technique cannot be applied to our current model. Similarly, the structural induction technique also appears to be a promising approach. However, there are a few limitations on this technique too. One has to be creative to find a finite-state description of the behaviour we want to use in the inductive step [Ros10], and it is not clear that this is possible, as larger models will have more states unless we can find a way of abstracting them. Although we believe that our results are generally applicable to larger models, it is not straightforward to apply either data independence or induction to our current models, and we leave these techniques as future work.

The observer in our models acts as a passive intruder, who can only see the public channels. The observer is limited to private channels such as the channel between the voters and the authority, and the channel on which the voters mark their ballot. If we had a more powerful intruder model like Dolev-Yao, he could *learn*, *send*, *receive*, *block* as well as *see* the messages sent over the agents. Thus, the current models of the voting systems would not provide anonymity as the intruder could block all the votes sent to mixnet but the one cast by a particular voter, violating her anonymity. We aim to investigate further on the anonymity and the voting systems with an active intruder model.

Acknowledgements

This work is a much expanded and improved version of a previous paper presented at the CryptoForma Worksop (Limerick, June 2011). We are grateful to the anonymous referees on that paper, and to those who participated in the discussion following the presentation, for their helpful comments and suggestions. We would also like to thank to David M. Williams for his technical discussion and priceless contribution. We also thank the members of the project, Trustworthy Voting Systems (TVS), for their contribution.

This work was carried out under the project Trustworthy Voting Systems (TVS) with support from Engineering and Physical Sciences Research Council (EPSRC). The first-named author is funded by Republic of Turkey Ministry of National Education. The research was conducted whilst the second-named author was a Royal Academy of Engineering/Leverhulme Trust Senior Research Fellow.

References

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium*, pages 335–348, 2008.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3 – 51, 2008.

[BG02] Dan Boneh and Philippe Golle. Almost entirely correct mixing with applications to voting. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 68–77, New York, NY, USA, 2002. ACM.

[BHM08] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *CSF*, pages 195–209, 2008.

[BP05] Mohit Bhargava and Catuscia Palamidessi. Probabilistic anonymity. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin / Heidelberg, 2005.

[BRS07] A. Baskar, R. Ramanujam, and S. P. Suresh. Knowledge-based modelling of voting protocols. In *TARK*, pages 62–71, 2007.

[BS12] Bruno Blanchet and Ben Smyth. *ProVerif: Automatic Cryptographic Protocol Verifier User Manual and Tutorial*, 2012. http://www.proverif.ens.fr/.

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

[CEC$^+$08] David Chaum, Aleksander Essex, Richard Carback, Jeremy Clark, Stefan Popoveniuc, Alan T. Sherman, and Poorvi L. Vora. Scantegrity: End-to-end voter-verifiable optical-scan voting. *IEEE Security & Privacy*, 6(3):40–46, 2008.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24:84–90, February 1981.

[Cha04] David Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2(1):38–47, 2004.

[COPD06] Tom Chothia, Simona Orzan, Jun Pang, and Mohammad Torabi Dashti. A framework for automatically checking anonymity with MuCRL. In *TGC*, pages 301–318, 2006.

[CPP06] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. In *Information and Computation*. Springer, 2006.

[CRS05] David Chaum, Peter Y. A. Ryan, and Steve A. Schneider. A practical voter-verifiable election scheme. In *ESORICS*, pages 118–139, 2005.

[DKR06] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *CSFW*, pages 28–42, 2006.

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

[DPP07] Yuxin Deng, Catuscia Palamidessi, and Jun Pang. Weak probabilistic anonymity. *Electronic Notes in Theoretical Computer Science*, 180(1):55–76, June 2007.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198 – 208, mar 1983.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, 1984.

[FA02] Cédric Fournet and Martín Abadi. Hiding names: Private authentication in the applied pi calculus. In *ISSS*, pages 317–338, 2002.

[FOO92] A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. In *AUSCRYPT*, pages 244–251, 1992.

[GGH$^+$] Paul Gardiner, Michael Goldsmith, Jason Hulance, David Jackson, Bill Roscoe, Brian Scattergood, and Bryan Armstrong. FDR2 user manual.

[GHPv05] Flavio D. Garcia, Ichiro Hasuo, Wolter Pieters, and Peter van Rossum. Provable anonymity. In *Proceedings of the 2005 ACM workshop on Formal methods in security engineering*, FMSE '05, pages 63–72, New York, NY, USA, 2005. ACM.

[Hea07] James Heather. Implementing STV securely in Prêt à Voter. In *CSF*, pages 157–169, 2007.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[HS04] Dominic Hughes and Vitaly Shmatikov. Information hiding, anonymity and privacy: a modular approach. *Journal of Computer Security*, 12(1):3–36, 2004.

[JCJ05] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Proceedings of the 2005 ACM workshop on Privacy in the Electronic Society*, WPES '05, pages 61–70, New York, NY, USA, 2005. ACM.

[JJR02] Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security Symposium*, pages 339–353, 2002.

[KR05] Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. pages 186–200. Springer, 2005.

[Laz99] Ranko S. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking*. D. phil. thesis, Oxford University Computing Laboratory, 1999.

[LJP10] Barbara Lucie Langer, Hugo Jonker, and Wolter Pieters. Anonymity and verifiability in voting: Understanding (un)linkability. In *ICICS*, pages 296–310, 2010.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[MVd04] S. Mauw, J. Verschuren, and E. P. de Vink. A formalization of anonymity and onion routing. In *ESORICS*, pages 109–124, 2004.

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, CCS '01, pages 116–125, New York, NY, USA, 2001. ACM.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[PK00] Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 1–9, 2000.

[RBH$^+$09] Peter Y. A. Ryan, David Bismark, James Heather, Steve A. Schneider, and Zhe Xia. Prêt à Voter: a voter-verifiable voting system. *IEEE Transactions on Information Theory*, 4(4):662–673, 2009.

[Riv06] Ronald L. Rivest. The threeballot voting system, 2006.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[RP05] Peter Y. A. Ryan and Thea Peacock. Prêt à Voter: a systems perspective. Technical report, University of Newcastle, 2005.

[RP10] Peter Y. A. Ryan and Thea Peacock. In David Chaum, Markus Jakobsson, Ronald Rivest, Peter Ryan, Josh Benaloh, Miroslaw Kutylowski, and Ben Adida, editors, *Towards Trustworthy Elections*, volume 6000 of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin / Heidelberg, 2010.

[RS06] Peter Y. A. Ryan and Steve A. Schneider. Prêt à Voter with re-encryption mixes. In *ESORICS*, pages 313–326, 2006.

[RSG$^+$00] Peter Y. A. Ryan, Steve A. Schneider, Michael H. Goldsmith, Gavin Lowe, and A. W. Roscoe. *The Modelling and Analysis of Security Protocols : the CSP Approach*. Addison-Wesley Professional, first edition, 2000.

[Rya05] Peter Y. A. Ryan. A variant of the chaum voter-verifiable scheme. In *Proc. 2005 Workshop on Issues in the Theory of Security*, pages 81–88, 2005.

[Rya06] Peter Y. A. Ryan. Putting the human back in voting protocols. In *Security Protocols Workshop*, pages 20–25, 2006.

[Rya08] Peter Y. A. Ryan. Prêt à Voter with paillier encryption. *Mathematical and Computer Modelling*, 48(1):1646–1662, 2008.

[Sch99] Steve A. Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[SS96] Steve A. Schneider and Abraham Sidiropoulos. CSP and anonymity. In *ESORICS*, pages 198–218, 1996.

[XCH+10] Zhe Xia, Chris Culnane, James Heather, Hugo Jonker, Peter Y. A. Ryan, Steve A. Schneider, and Sriramkrishnan Srinivasan. Versatile Prêt à Voter: Handling multiple election methods with a unified interface. In *INDOCRYPT*, pages 98–114, 2010.

A. Modelling Prêt à Voter

```
-------------------------------------------------------------------------------
-- PaV.csp
-- Modelling a cryptography based voting system, Pret A Voter in CSP, and
-- verifying anonymity.
-- Murat Moran, June 2011
-- Dept. of Computing University of Surrey
-------------------------------------------------------------------------------
-- Data types
datatype fact =  c1 | c2 | v1 | v2 | v3 | s1 | s2 | s3
                    | pkt | skt | emptylist | ciphertext
                    | Enc .(fact, fact)

-- 2 candidates, 3 voters, 3 serial number
candidates      = {c1, c2}
voters          = {v1, v2}
serials         = {s1, s2}

-- The number of possible total votes that a candidate can get
numOfVotes      = {0..card(voters)}
-- The number of possible candidates on a ballot form
numOfCand       = {1..card(candidates)}
-- The number of maximum votes
maxVotes        = card(voters)

-- The nth function is to find nth element of a message
nth(i,msg) = if i==1 then head(msg) else nth(i-1,tail(msg))

-- Encryption function
enc(pkt,m) = Enc.(pkt,m)

--Reencryption function
reEnc(pkt,x) = x

-- Decryption function
dec(skt, <Enc.(pkt,a)>) = <a>

-- Public and secret key pairs
inverse(pkt) = skt
inverse(skt) = pkt

-- find is the function that finds the index value for a given candidate and
-- candidate list.

find(c,clist) = if c == head(clist) then 1
                     else 1+find(c,tail(clist))

-- List of all possible candidate lists on the left hand side of the ballot
-- forms.
LHSs            = makelists(candidates)
makelists({s})  = {<s>}
makelists(S)    = {<s>^a|s<-S,a<-makelists(diff(S,{s}))}

-- The empty and marked right hand side of the ballot forms

markedlists = {find(cand,left) | cand <- candidates, left<-LHSs}
```

```
-- The set of all possible onions, which are the encryption of LHS of the
-- ballot forms
onions      = {onionlist(pkt,l)|l<-LHSs}

onionlist(k,L) = onionlisthelper(k,L,length(L))
onionlisthelper(k,L,i) = if i==1 then <Enc.(k,head(L))>
                        else <Enc.(k,head(L))>^onionlisthelper(k,tail(L),i-1)

-- The set of all marked RHSs of the ballot forms
markedRHSs  = {<ser,list,o>,<ser,list,<ciphertext,ciphertext>>
                        |ser<-serials,list<-markedlists,o<-onions}

-- The set of empty ballot forms
emptyforms  = {<left,ser,emptylist,onionlist(pkt,left)>,<left,ser,emptylist,
                    <ciphertext,ciphertext>> |left <-LHSs,ser<-serials}

-- The set of marked ballot forms
markedforms = {<left,ser,list,onionlist(pkt,left)>,
                    <left,ser,list,<ciphertext,ciphertext>>
                        |left<-LHSs,ser<-serials,list<-markedlists}

-------------------------------------------------------------------------------
-- Process Alphabets
-------------------------------------------------------------------------------
aVTR = {|openElection, auth, collectform, enterBooth, leaveBooth, choose,
        mark, shredLHS, cast, receipt, closeElection|}
aAUTH = {|openElection, auth, collectform, closeElection|}
aMAC = {|openElection, cast, receipt, closeElection|}
aBTH = {|enterBooth, leaveBooth|}
aWBB = {|openElection, receipt, closeElection, mixReq, pub, bagempty|}
aMIX = {|openElection, mixReq, reencrypt, closeElection, mixOut, bagempty|}
aTEL = {|openElection, mixOut, decrypt, total, bagempty|}

-------------------------------------------------------------------------------
-- The VOTER process accepts an empty ballot form and cast the ballot form.
-------------------------------------------------------------------------------
-- channel openElection and closeElection means open a run of an election
-- protocol and closing it.
channel openElection, closeElection
-- channel auth is authority's authenticating the voters.
channel auth            :voters
-- channel collecform is voter's collecting an empty ballot form.
channel collectform     :voters.emptyforms
-- channel mark is voter's marking a ballot form (choosing the candidate).
channel mark            :voters.markedforms
-- channel shredLHS is voter's shreding the LHS of her ballot form.
channel shredLHS        :markedRHSs
-- channel cast is voter's casting her vote using the machine supplied.
channel cast            :markedRHSs
-- channel receipt is voter's taking away their receipts provided from
-- MACHINE process.
channel receipt         :markedRHSs
-- channel choose is that the voter decides for whom to vote.
channel choose      :voters.candidates

VOTER(id) = |~|c:candidates@choose.id.c -> openElection ->
    auth.id -> [] <l,s,emptylist,o> : emptyforms @collectform!id.
        <l,s,emptylist,o> -> enterBooth!id ->
            mark!id.<l,s,find(c,l),onionlist(pkt,l)>->
                shredLHS.<s,find(c,l),onionlist(pkt,l)> ->
                    leaveBooth!id -> cast.<s,find(c,l),onionlist(pkt,l)> ->
                        receipt.<s,find(c,l),onionlist(pkt,l)> ->
                            closeElection-> STOP

VOTERS = ([|{|openElection, closeElection|}|] ids:voters @ VOTER(ids))

-------------------------------------------------------------------------------
```

```
-- The process BOOTH is an empty booth that controls voter entering and
-- leaving the booth. Thus, there will be no two voters in the booth at the
-- same time.
-----------------------------------------------------------------------------
-- channel leaveBooth and enterBooth is voter's getting in the booth and
-- getting out from the booth.
channel leaveBooth, enterBooth: voters

BOOTH =  enterBooth?id -> leaveBooth.id -> BOOTH
-----------------------------------------------------------------------------
-- The process AUTHORITY is the election authority who authenticates the
-- voters and distributes empty ballot forms to the them with
-- non-deterministically chosen left hand side of ballots and serial numbers,
-- which are different on each ballot form and a candidate list, finally
-- closes the election.
-----------------------------------------------------------------------------
AUTHORITY(ids,sns,lhs) = openElection -> AUTHORITY1(ids,sns,lhs)

AUTHORITY1(ids,sns,lhs) =
        (
            [] id:ids @auth.id ->
                (
                |~| ls:lhs, seri:sns@
                    collectform!id.<ls,seri,emptylist,onionlist(pkt,ls)> ->
                        AUTHORITY1(diff(ids,{id}),diff(sns,{seri}),lhs)
                )
        )
        [] closeElection -> STOP
-----------------------------------------------------------------------------
-- MACHINE process works as a scanner and a printer. MACHINE allows voters
-- to cast their votes and collect each receipt in the set of Votes.
-----------------------------------------------------------------------------
MACHINE = openElection -> MACHINE1

MACHINE1 =   (
                    [] rhs :markedRHSs @cast.rhs -> receipt.rhs ->
                        MACHINE1

                    )
                    []
                    closeElection -> MACHINE
-----------------------------------------------------------------------------
-- The process WBB opens the election with AUTHORITY. It collects each receipt
-- that is cast in a set, Receipts, and requests a shuffling from mixnet for
-- each votes. After election closed, it publishes the receipts with serial
-- numbers on them.
-----------------------------------------------------------------------------
-- The set of encrypted candidates
chosenCand = {<Enc.(pkt,c)>,<ciphertext> | c<-candidates}
-- channel mixReq is WBB's requesting mix for the receipts he received.
channel mixReq: chosenCand
-- channel pub is WBB's publishing the cast votes.
channel pub : markedRHSs

WBB = openElection -> WBB1({})

WBB1(Receipts) = ([] <s,i,o>:markedRHSs @receipt.<s,i,o> ->
                                mixReq.<nth(i,o)> ->
                                    WBB1(union(Receipts,{<s,i,o>}))
                )
                [] closeElection -> WBB2(Receipts)

WBB2({}) = bagempty -> STOP
WBB2(Receipts) = |~|rcp:Receipts@pub.rcp -> WBB2(diff(Receipts,{rcp}))

-----------------------------------------------------------------------------
-- The MIX process is the mixnet that re-encrypts each onion value and index
-- value and shuffle the batch of encrypted votes non-deterministically and
-- sends the encrypted values to the TELLER process for decryption and
-- tallying.
------------------------------------------------------------------------------
-- The set of encrypted receipts which include index values and onion values.
encList = {reEnc(pkt, <Enc.(pkt,c)>),<ciphertext> | c<-candidates}
-- channel encrypt is mixnet's encrypting the receipts.
channel reencrypt : encList
-- channel mixOut is mixnet's giving away the re-encrypted shuffled receipts.
channel mixOut  : encList
-- channel bagempty means there is no other receipt left to mix.
channel bagempty

MIX = openElection -> MIX1(0,{})

MIX1(i,Batch) = card(Batch) <= card(candidates) &
                (
                [] <Enc.(pkt,c)>:chosenCand @ mixReq.<Enc.(pkt,c)> ->
                                        reencrypt.reEnc(pkt,<Enc.(pkt,c)>) ->
                    MIX1(i+1, union(Batch,{(i, reEnc(pkt,<Enc.(pkt,c)>))}))
                )
                [] closeElection -> MIX2(Batch)

MIX2({}) = bagempty -> STOP
MIX2(Bag) =   |~|(i,<Enc.(pkt,a)>):Bag@
                    mixOut.<Enc.(pkt,a)> ->
                        MIX2(diff(Bag,{(i,<Enc.(pkt,a)>)}))
------------------------------------------------------------------------------
-- The TELLER processes receives encrypted shuffled votes from MIX process,
-- and decrypt each one of them using teller's and mixnet's secret keys, skt
-- and skt respectively.
------------------------------------------------------------------------------
channel decrypt: decryptedVotes
-- channel decrypt is Teller's decrypting onion values.
channel tally   : decryptedVotes
-- channel total is Teller's announcing final tally for each candidate.
channel total   : candidates.numOfVotes
-- The set of all decrypted votes
decryptedVotes = {<a> | a<-candidates}

TELLER = openElection -> TELLER1(0,0)

TELLER1(i,j) = i+j <= maxVotes &
        (
        [] <Enc.(pkt,a)>:encList@mixOut.<Enc.(pkt,a)> ->
                decrypt.dec(skt,<Enc.(pkt,a)>) ->

            (if a == c1 and i<=1 then TELLER1(i+1,j) else
                (
                    if a == c2 and j<=1 then
                        TELLER1(i,j+1) else STOP
                )
            )
        )
        [] bagempty -> total.c1.i -> total.c2.j -> SKIP

------------------------------------------------------------------------------
-- SYSTEM is the Pret a Voter Voting System with the alphabet Sigma
------------------------------------------------------------------------------
SYSTEM =  (((((VOTERS[aVTR||aAUTH]AUTHORITY(voters, serials, LHSs))
                        [Union({aVTR,aAUTH})||aMAC]MACHINE)
                        [Union({aVTR,aAUTH,aMAC})||aBTH]BOOTH)
                        [Union({aVTR,aAUTH,aMAC,aBTH})||aWBB]WBB)
                        [Union({aVTR,aAUTH,aMAC,aBTH,aWBB})||aMIX]MIX)
                        [Union({aVTR,aAUTH,aMAC,aBTH,aWBB,aMIX})||aTEL]TELLER

Sigma = Union({aVTR,aAUTH,aMAC,aBTH,aWBB,aMIX,aTEL})
```

```
-------------------------------------------------------------------------------
-- SANITY CHECK 1 -------------------------------------------------------------
-------------------------------------------------------------------------------
-- No one can be authenticated twice.
-------------------------------------
AUTH(v) = auth.v -> STOP

SPEC(voters) = [|{||}|]v:voters@AUTH(v)

assert  SPEC(voters) [T=  SYSTEM \ diff(Sigma,{|auth|})

-- Assertion holds meaning SYSTEM always authenticates a different voter than,
-- previuosly authenticated ones which also means a voter cannot be
-- authenticated twice.

-------------------------------------------------------------------------------
-- SANITY CHECK 2 -------------------------------------------------------------
-------------------------------------------------------------------------------
-- No one can mark a ballot form before authenticating themselves.
-------------------------------------------------------------------

ECHECK(v) = auth.v -> AUTHED(v)
AUTHED(v) = mark.v?_ -> AUTHED(v)
            []auth.v?_ -> AUTHED(v)

ECHECKALL(voters) = [|{||}|]v:voters@ECHECK(v)

assert  ECHECKALL(voters) [T=  SYSTEM \ diff(Sigma,{|auth, mark|})

-- Assertion holds meaning SYSTEM only allows the authenticated voters go in
-- the booth.

-------------------------------------------------------------------------------
-- SANITY CHECK 3 -------------------------------------------------------------
-------------------------------------------------------------------------------
-- No one can vote after election closed.
-----------------------------------------

SPEC2 = closeElection -> CLOSED
             [] cast?_ -> SPEC2
CLOSED = closeElection -> CLOSED

assert SPEC2 [T= SYSTEM \ diff(Sigma,{|closeElection,cast|})

-- Assertion holds, which means that SYSTEM does not let the voters perform
-- the events, which are in the alphabet S2 once the election is closed.

-------------------------------------------------------------------------------
-- SANITY CHECK 4 -------------------------------------------------------------
-------------------------------------------------------------------------------
-- The number of votes tallied corresponds the number of cast votes.
---------------------------------------------------------------------

COUNTCHECK = COUNT(0)

COUNT(n)  = (cast?_ -> if 0<=n and n<= maxVotes then COUNT(n+1) else STOP)
            [](total.c1?i ->  0<=n and n<= maxVotes & COUNT1(n-i))

COUNT1(j) = 0<=j and j<=maxVotes & total.c2.j -> STOP

assert  COUNTCHECK [T= SYSTEM \diff(Sigma,{|cast,total|})

-- Assertion holds, which means that the SYSTEM reflects the number of cast
-- votes to the final tally.

-------------------------------------------------------------------------------
```

```
-- SECURITY CHECKS-------------------------------------------------------------
-------------------------------------------------------------------------------
-- For the anonymity, the SYSTEM that an observer can see should be limited.
-- Thus, SYSTEM1 is the system that we hide the mark events and the SYSTEM2
-- is the events where we rename a number of events, which carry confidential
-- information that should be hidden from the observer.

-- The maskFact function is used to rename all the encryptions to the one
-- single data, ciphertext.
maskFact(Enc._) = ciphertext
maskFact(x) = x
-- The mask function is for the sequences of encryption data.
mask(<>) = <>
mask(<x>^xs) = <maskFact(x)>^mask(xs)

-- mark and collectform events are hidden from the observer.
SYSTEM1  = SYSTEM \ {|mark,collectform|}

ABS_SYSTEM = SYSTEM1
                [[shredLHS.<s,x,o> <- shredLHS.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[cast.<s,x,o> <- cast.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[receipt.<s,x,o> <- receipt.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[pub.<s,x,o> <- pub.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[mixReq.enc <- mixReq.mask(enc) | enc<-chosenCand]]
                [[mixOut.enc <- mixOut.mask(enc) | enc<-chosenCand]]
                [[reencrypt.enc <- reencrypt.mask(enc) | enc<-chosenCand]]
-------------------------------------------------------------------------------
-- Weak Anonymity --
--------------------
-- We did a specification check over the weak anonymity definition of Delaune
-- et al. using the hiding and renaming operators. An observer can only see
-- what is public and cannot distinguish if the two encrypted value are the
-- same,i.e., the encrypted values look all the same from the observer point
-- of view.

SPEC_WEAK = ABS_SYSTEM
                [[choose.v1.cand <- choose.v2.cand,
                  choose.v2.cand <- choose.v1.cand | cand<-candidates]]
assert SPEC_WEAK  [T= ABS_SYSTEM
assert ABS_SYSTEM [T= SPEC_WEAK

-- Both assertion hold proving that Pret a Voter preserves voter's anonymity.
-------------------------------------------------------------------------------
-- Strong Anonymity --
----------------------
-- We also checked the system against the strong anonymity definition of
-- Steve Schnieder et al. abstracting choose events to a dummy event and
-- abstracting them back. The system and the observer are the same as we used
-- for the weak anonymity verification.

channel dummy

SPEC_STRONG = ABS_SYSTEM
                [[choose.id.c1 <- dummy |  id<-voters]]
                [[dummy <- choose.id.c1 |  id<-voters]]

assert ABS_SYSTEM [T= SPEC_STRONG

-- The assertion does not hold as expected, and here is the one counter
-- example trace taken from FDR2 violating the strong anonymity definition.
-------------------------------------------------------------------------------
------- Counter Example -------------------------------------------------------
-- choose.v1.c1
-- choose.v1.c1
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-- Missbehaving Agents --
----------------------
-- Corrupted Authority
----------------------
SYSTEM2  = SYSTEM \ {|mark|}

ABS_SYSTEM2 = SYSTEM2
                [[shredLHS.<s,x,o> <- shredLHS.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[cast.<s,x,o> <- cast.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[receipt.<s,x,o> <- receipt.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[pub.<s,x,o> <- pub.<s,x,mask(o)>
                            | s<-serials, x <- numOfCand, o<-onions]]
                [[mixReq.enc <- mixReq.mask(enc) | enc<-chosenCand]]
                [[mixOut.enc <- mixOut.mask(enc) | enc<-chosenCand]]
                [[reencrypt.enc <- reencrypt.mask(enc) | enc<-chosenCand]]

SPEC_WEAK2 = ABS_SYSTEM2
                [[choose.v1.cand <- choose.v2.cand,
                  choose.v2.cand <- choose.v1.cand | cand<-candidates]]
assert SPEC_WEAK2  [T= ABS_SYSTEM2
-------------------------------------------------------------------------------
------- Counter Example 2 ---------------------------------------------------
--choose.v1.c2
--choose.v2.c1
--openElection
--auth.v1
--collectform.v1.<<c1,c2>,s1,emptylist,<Enc.(pkt,c1),Enc.(pkt,c2)>>
--enterBooth.v1
--shredLHS.<s1,2,<ciphertext,ciphertext>>
--------------------------------- END ---------------------------------------
-------------------------------------------------------------------------------
```

B. Modelling Conventional Voting System

```
-------------------------------------------------------------------------------
-- conventional.csp
-- Modelling and conventional voting system and verifying anonymity.
-- Murat Moran, February 2010
-- Dept. of Computing University of Surrey
-------------------------------------------------------------------------------
-- Datatypes defines which type a channel can be for instance datatype VOTERID
-- are the objects which could be one of the voters shown below, thus a
-- channel such as collectForm is defined with an object of a VOTERID and an
-- object of a SERIALS. Similarly, datatype CANDIDATES defines the candidates,
-- which could only be c1, c2 or c3.
-------------------------------------------------------------------------------

datatype VOTERID    = v1 | v2 | v3
datatype CANDIDATES = c1 | c2 | c3
datatype SERIALS    = s1 | s2 | s3
-------------------------------------------------------------------------------
-- The sets below defines the set of candidate names, voter identities and
-- serial numbers on the ballot papers
-------------------------------------------------------------------------------
candidates = {c1, c2, c3}
voters     = {v1, v2, v3}
serials    = {s1, s2, s3}
-------------------------------------------------------------------------------
-- The channel choose represents the votes cast by voters.
-- The channel cast represents a voter casting a ballot by dropping it
-- into ballot box.
-- The channel collectForm represents an election official giving a ballot
-- to a voter.
-- The channel auth represents the election official authenticating
-- a voter.
-----------------------------------------------------------------------------
channel choose   :VOTERID.CANDIDATES
channel mark     :VOTERID.SERIALS.CANDIDATES
channel cast     :VOTERID.SERIALS.CANDIDATES
channel auth     :VOTERID
channel collectform:VOTERID.SERIALS
-----------------------------------------------------------------------------
-- The channel openElection and closeElection represent election official
-- opening and closing election, which is also synchronised by all voters.
-----------------------------------------------------------------------------
channel openElection, closeElection

-----------------------------------------------------------------------------
-- The sets below defines the alphabets of the corresponding processes
-- such as aVTR is the alphabet of VOTERS process.
-----------------------------------------------------------------------------
aVTR = {|choose, openElection, auth, collectform, enterBooth, mark,
         leaveBooth, cast, closeElection|}
aBTH = {|enterBooth, leaveBooth|}
aBOX = {|openElection, cast, closeElection, withdraw, empty|}
aCNT = {|withdraw,empty, closeElection, total|}
aOFF = {|openElection, closeElection, auth, collectform|}

-----------------------------------------------------------------------------
-- The process VOTERS are the voters who enter the polling station, authenti-
-- cate themselves with the official get a ballot form with their id and
-- serial number on it, enter the booth, vote for a chosen candidate, leave
-- the booth, drop the envelope to the ballot box.
-----------------------------------------------------------------------------
VOTER(id) =  |~| c:candidates@choose.id.c -> openElection -> auth.id ->
                    []s:SERIALS@ collectform.id.s -> enterBooth!id ->
                        mark.id.s.c -> leaveBooth!id -> cast.id.s.c ->
                            closeElection ->VOTER(id)
VOTERS = [|{|openElection,closeElection|}|] ids:voters @ VOTER(ids)

-----------------------------------------------------------------------------
-- The process BOOTH is an empty booth that controls voter entering the booth
-- and leaving the booth.
-----------------------------------------------------------------------------
channel leaveBooth, enterBooth: VOTERID

BOOTH = enterBooth?id -> leaveBooth!id -> BOOTH

-----------------------------------------------------------------------------
-- The process ELECOFFICIAL is the election official who opens and closes the
-- election, and authenticates the voters and distribute ballot papers to the
-- authenticated voters with serial numbers.
-----------------------------------------------------------------------------
ELECOFFICIAL(ids,sns) = openElection -> OFFICIAL(ids,sns)

OFFICIAL(ids,sns) = ([] id:ids @ auth.id ->
                        (|~|seri:sns@ collectform.id.seri ->
                                OFFICIAL(diff(ids,{id}),diff(sns,{seri}))
                        )
                    )
                    []
                    closeElection -> STOP
-----------------------------------------------------------------------------
-- The process BOX is the ballot box which allows voters to drop their votes
-- in and parse all the votes to the COUNTER with the event withdraw until the
-- box is empty.
-- The channel withdraw represents a ballot being withdrawn by the counter.
-- The channel empty represents an empty ballot box.
```

```
-----------------------------------------------------------------------------
channel withdraw:SERIALS.CANDIDATES
channel empty

BOX =  openElection -> BOX1({})

BOX1(Votes) =   ([]id:voters, s:serials, c:candidates@cast.id.s.c ->
                    BOX1(union(Votes,{(s,c)}))
                )
                []
                closeElection -> BOX2(Votes)

BOX2(Votes) =   if Votes=={} then empty -> STOP
                    else |~|(s,c):Votes@ withdraw.s.c ->
                        BOX2((diff(Votes,{(s,c)})))

-----------------------------------------------------------------------------
-- The process COUNTER is the counter which withdraws the votes from ballot
-- box and count them according to if the box is empty just give the total
-- votes for per candidate.
-- The channel total represents the number of votes that each voter possesses.
-----------------------------------------------------------------------------
NumOfMaxPossVotes = card(voters)
PossVotes = {0..NumOfMaxPossVotes}

channel total:CANDIDATES.PossVotes

COUNTER = closeElection  -> COUNT(0,0,0)

COUNT(i,j,k) =
    ([]s:serials,c:candidates@
        (
           if (c==c1) and i<=2 then withdraw.s.c -> COUNT(i+1,j,k) else STOP
        [] if (c==c2) and j<=2 then withdraw.s.c -> COUNT(i,j+1,k) else STOP
        [] if (c==c3) and k<=2 then withdraw.s.c -> COUNT(i,j,k+1) else STOP
        )
    )
    []
    empty -> total!c1!i -> total!c2!j -> total!c3!k -> SKIP
-----------------------------------------------------------------------------
-- SYSTEM represents the conventional voting system.
-----------------------------------------------------------------------------
SYSTEM  =   (
                (
                    (
                        VOTERS[ aVTR|| aOFF]ELECOFFICIAL(voters,serials)
                    )
                    [Union({aVTR, aOFF})||aBTH]BOOTH
                )
                [Union({aVTR, aOFF, aBTH})||aBOX]BOX
            )
            [Union({aVTR, aOFF, aBTH, aBOX})||aCNT]COUNTER

Sigma = Union({aVTR, aOFF, aBTH, aBOX, aCNT})

-----------------------------------------------------------------------------
-- SANITY CHECK 1 ------------------------------------------------------------
-----------------------------------------------------------------------------
-- No one can vote after election closed.
-----------------------------------------

SPEC1  = closeElection -> CLOSED
            [] cast?_ -> SPEC1
CLOSED = closeElection -> CLOSED

assert SPEC1 [T= SYSTEM \ diff(Sigma,{|closeElection,cast|})
```

```
-- Assertion holds, which means that SYSTEM does not let the voters perform
-- the events, which are in the alphabet S2 once the election is closed.

-----------------------------------------------------------------------------
-- SANITY CHECK 2 -----------------------------------------------------------
-----------------------------------------------------------------------------
-- The number of votes tallied corresponds the number of cast votes.
-------------------------------------------------------------------

SPEC2 = COUNTTHIS(0,card(voters))

COUNTTHIS(n,t)  =
            (cast?_ -> if 0<=n and n<= NumOfMaxPossVotes
                                then COUNTTHIS(n+1,t+1) else STOP
            )
            []
                        (
            total.c1?i ->
                    0<=n and n<= NumOfMaxPossVotes & COUNTTHIS1((n-i),t)
            )
COUNTTHIS1(s,z) = total.c2?j ->
                    if s==j then total.c3.0 -> SKIP
                        else 0<=s-j and s-j<=NumOfMaxPossVotes &
                            total.c3.(s-j) -> STOP

assert  SPEC2 [T= SYSTEM \diff(Sigma,{|cast,total|})

-- Assertion holds, which means that the SYSTEM reflects the number of cast
-- votes to the final tally.

-----------------------------------------------------------------------------
-- SECURITY CHECKS-----------------------------------------------------------
-----------------------------------------------------------------------------
-- For the anonymity, the SYSTEM that an observer can see should be limited.
-- Thus, SYSTEM1 is the system that we hide the mark events and the SYSTEM2
-- is the events where we rename a number of events, which carry confidential
-- information that should be hidden from the observer.

-- We hide mark and collecform events from the observer
SYSTEM1 = SYSTEM \ {|mark, collectform|}

-- cast events are renamed as envelope event, as the observer can see that a
-- voter casts a vote but not for whom the vote is.

channel envelope

ABS_SYSTEM = SYSTEM1
                [[cast.id.s.c <- envelope
                            | id<-voters, s<-serials, c<-candidates]]

-----------------------------------------------------------------------------
-- Weak Anonymity --
--------------------

-- We apply the weak anonymity definition of Delaune et al to the abstracted
-- process ABS_SYSTEM by swapping two votes.

SPEC_WEAK = ABS_SYSTEM[[choose.v1.c <- choose.v2.c, choose.v2.c <- choose.v1.c
                            |c<-candidates]]

assert ABS_SYSTEM [T= SPEC_WEAK
assert SPEC_WEAK  [T= ABS_SYSTEM

-- Both assertions hold proving that CVS provides anonymity.

-----------------------------------------------------------------------------
-- Strong Anonymity --
```

```
----------------------
-- We also checked the system against the strong anonymity definition of
-- Steve Schnieder et al. abstracting choose events to a dummy event and
-- abstracting back to the set of voters.

channel dummy

SPEC_STRONG = ABS_SYSTEM [[choose.id.c1 <- dummy | id<-voters]]
                         [[dummy <- choose.id.c1 | id<-voters]]

assert ABS_SYSTEM [T= SPEC_STRONG

-- The assertion does not hold proving that strong anonymity is not supported
-- by CVS, and here is the one example trace taken from FDR2 violating the
-- strong anonymity definition.

------------------------------------------------------------------------------
------- Counter Example ------------------------------------------------------
-- choose.v1.c1
-- choose.v1.c1
--------------------------------- END ----------------------------------------
------------------------------------------------------------------------------
```