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 properties for voting protocols using the process algebra CSP. We analyse a number of anonymity definitions, and give formal definitions for *strong* and *weak* anonymity, highlighting the difference between these definitions. We show that the strong anonymity definition is too strong for practical purposes; the weak anonymity definition, however, turns out to be ideal for analysing voting systems.

Two case studies are presented to demonstrate the usefulness of the formal definitions: a conventional voting system, and Prêt à Voter, a paper-based, voter-verifiable scheme. In each case, we give a CSP model of the system, and analyse it against our anonymity definitions by specification checks using the Failures-Divergences Refinement (FDR2) model checker. 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; FDR2

1. Introduction

Anonymity of one's vote lies at the heart of the democratic process. If the link between voter and vote is uncovered, then not only the secrecy but also the integrity of the election is threatened, because votes may be bought, or voters may be coerced into supporting particular candidates.

Many voting protocols [Cha81, FOO92, Nef01, BG02, CRS05, Riv06, CEC$^+$08, CCM08, Adi08] have been proposed over the last few decades that claim to provide anonymity, often without proof. There are a variety of definitions of anonymity in several paradigms, such as the pi calculus [FA02], a modular approach [HS04], epistemic logic [GHPv05, BRS07, LJP10], and probabilistic and non-deterministic approaches [CPP06, DPP07, BP05]. Recently, research has focused on giving precise formal definitions of security properties of trustworthy voting systems, including anonymity [DKR06, BHM08, DKR09].

However, little work has been done to provide a foundation for automated verification of such properties. Juels *et al.* [JCJ05] describe anonymity for elections using provable security. Further definitions of the desired

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

properties of voting systems have made use of formal methods; for instance, Delaune *et al.* [DKR06] give formal definitions of privacy properties in terms of adaptive simulation in the pi calculus. These authors then discovered in [DKR09] that their previous work had undesirable properties, and proposed definitions based on observational equivalence, but did not provide any means of automatic verification. Backes *et al.* [BHM08] propose a new formalization of coercion resistance for remote voting protocols in terms of observational equivalence, which implies vote privacy; their verification is automated, but some human effort is still required when transforming equivalences in their definition into a pair of processes, which have the same structure but differ only by terms. Chothia *et al.* [COPD06] present a framework for automatically checking anonymity using bisimilarity in the process algebraic language μCRL, analysing the voting scheme first given in [FOO92]. Their model, like ours, uses a passive intruder; however, their use of bisimilarity can produce false positives (that is, false attacks) because it effectively allows the intruder to see not just what actions are taken but where internal choices are resolved. Bisimilarity is more efficient to check, but seems too strong a notion of process equivalence for this application.

Apart from the works concerning voting systems, we also investigate in other anonymity definitions in the literature. Specifically, the strong anonymity definition given in [SS96] by Schneider and Sidiropoulos. They formally define strong anonymity for the security protocols using observational equivalence.

We illustrate our formalism with two different case studies. First, we verify a conventional voting system (CVS), which is a simple voting system in which voters have to go to polling stations to fill in a ballot paper in a private booth, and put them into a ballot box. Although intuitively we accept that this voting system provides anonymity, subject to appropriate assumptions on the various components, this system provides a useful mechanism for validating our formal definitions of anonymity. Secondly, we verify a simplified model of Prêt à Voter [RS06], a trustworthy voting system that aims to provide anonymity based on mixnets and cryptography. Prêt à Voter has not previously been subjected to automated verification.

1.1. Contribution

Having formalised the weak anonymity definition using the process algebra CSP, we compare two concise and generic definitions of anonymity, namely *strong* and *weak* anonymity with respect to voting systems. The comparison is made through formal automated analysis of two case studies: Prêt à Voter and conventional voting system. We present formal simplified CSP models of these two paper-based voting systems, and automatically analyse them against the *strong* and *weak* anonymity definitions. With the experimental results, we illustrate the main differences between these two anonymity definitions, i.e. the strong anonymity definition given in [SS96] is not appropriate to voting systems, the weak anonymity definition, however, is ideal for analysing voting systems.

1.2. Outline

In the next section, we give an overview of various formal anonymity definitions in the literature. In Section 3, we introduce CSP syntax and semantics. Section 4 then summarises the Schneider and Sidiropoulos definition of strong anonymity in CSP using a referendum protocol as an example; we then give a CSP definition of weak anonymity 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. Then in Section 6, we give an overview of Prêt à Voter, and formally model and 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. Schneider and Sidiropoulos [SS96] state that anonymity is a property of agents rather than the messages carried on the channels (the latter is stated to be a case of confidentiality). They give the strong anonymity definition, which our paper is partly based on. The definition is expressed informally as “a message that could have originated from one agent could equally have originated from any other”. If the message x originated by the user i is considered in the

form of $i.x$, then it could equally have been in the form of $j.x$, where j is a user from the set of all users (see Section 4 for the formal definition).

Pfitzmann *et al.* [PK00] define anonymity in a message sender-receiver setting, where it is specified as the state of not being identifiable within a given *anonymity set* of subjects—that is, we specify a set of all possible subjects who might cause an action. In a voting context, this would mean that no specific vote is linkable to any particular voter ID. In addition, an element possesses *indistinguishability* with respect to a given set if it is indistinguishable from all other elements in the set. In voting, this would naturally mean the inability to distinguish a particular vote from within a set of votes. *Unobservability* describes when an intruder cannot observe that a particular event has occurred—for example, that a particular voter has voted. Finally, the term *pseudonymity* describes the use of pseudonyms as identifiers of subjects. For instance, we can consider ballot serial numbers as pseudonyms that link voters to ballot papers and ballot papers to votes.

Fournet and Abadi [FA02] give a general privacy definition in the pi calculus with respect to private authentication protocols. In their description, an observational equivalence notion[1] is used to formalise properties. They define anonymity as the case where "two process behaviours 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; in what follows, we will call this kind of anonymity definition *weak anonymity*.

Mauw *et al.* [MVd04] define anonymity based on the work in [PK00] described above. Their definition specifies anonymity in such a way that a coercer should not be able to distinguish a user u from another user u' in the anonymity group of u. That is, for every behaviour of the system that can be attributed to user u, there is another indistinguishable system behaviour that can be attributed to u'.

Shmatikov and Hughes [HS04] give a specification framework for anonymity and privacy based upon a view in which system behaviour is described as a set of functions. The specifications of the desired properties are defined with observational equivalence using a modular approach. In their paper, several forms of anonymity in terms of a sender-receiver relation are described. We adopt some of those definitions that 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 (for example, a ballot serial number) with a sender ID (voter).
2. *Type-anonymity*: an attacker may learn the type of the voter. That is, in the case of a postal voting, if there are relatively few voters who registered and cast their votes by post, an attacker may in some cases be able to reduce the number of the possible voters for a particular vote to a proper subset of the set of voters (either the set of postal voters or its complement).
3. *Session-level*: an attacker may know the entire set of voters and the votes, but is 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, and each polling station constitutes a separate session, he may be able to 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 case where the coercer or 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.* use the term "vote privacy" as a synonym for anonymity, and look for cases where nobody has enough information to identify whether two voters swapped their votes. 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.

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

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 was 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.

3.1. Syntax

Processes are defined in terms of a collection of *events* that the process can perform. In CSP the occurrence of an event should be regarded as an atomic action without time. A synchronised event can happen when all processes agree on executing it; it happens when it is inevitable. The set of events that are visible is called Σ, and the internal events are written τ. Processes are associated with an interface or *alphabet*, denoted αP. If no alphabet is explicitly defined then it will be the set of events that the process can perform. 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 $RUN(A) \mathrel{\widehat{=}} \Box_{x \in A}\, x \to RUN(A)$.

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. Here we will introduce the elements of the language we use in this paper. See [Ros10, Sch99] for a fuller account of the language.

$P, Q :=$	processes
$STOP$	stop (deadlock)
$SKIP$	successful termination
$a \to P$	prefixing
$c?v \to P(v)$	data input
$c!v \to P$	data output
$P \Box Q$	external choice
$\Box_{x \in A} P(x)$	indexed external choice
$P \sqcap Q$	nondeterministic choice
$\sqcap_{x \in A} P(x)$	indexed nondeterministic choice
if b then P else Q	conditional choice
$P \parallel Q$	alphabetised parallel composition
$\parallel_{x \in A} (P(x), \alpha P(x))$	indexed alphabetised parallel composition
$P \mid\mid\mid Q$	interleaving
$\mid\mid\mid_{x \in A} P(x)$	indexed interleaving
$P \setminus A$	hiding
$P[\![R]\!]$	relational renaming

Given a process P and an event a in Σ, the prefix process $a \to P$ 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, $P_1 \mathrel{\widehat{=}} a \to b \to STOP$ will perform the events a and b, then it will terminate.

Events can also be structured into any number of parts. For example, an event of the form $c.v$ can represent a channel c passing value v. The set of values T that can pass along c is the *type* of c, so the set

of events associated with channel c of type T is $\{c.v \mid v \in T\}$. This is also written $\{|c|\}$. If C is a set of channels, then $\{|C|\} = \bigcup_{c \in C}\{|c|\}$.

The input process $c?x \rightarrow P(x)$ is initially prepared to accept a value that will be bound to the locally introduced variable x along channel c, and then behave as P having received input x. The output process $c!v \rightarrow P$ outputs value v along channel c. In this paper we will use structured events to describe events in voting systems, for example $vote.v.c$ can represent voter v casting a vote for candidate c.

We can also describe recursive processes in CSP, by means of recursive definitions of the form $N \mathrel{\widehat=} P$, where N is a process name that can appear in process P. N can also take parameters, giving definitions of the form $N(p) \mathrel{\widehat=} P(p)$. Thus, the processes, $P_2 \mathrel{\widehat=} a \rightarrow b \rightarrow P_2$, 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 definitions $P_3 \mathrel{\widehat=} c?x \rightarrow P_4(x)$ and $P_4(x) \mathrel{\widehat=} d!x \rightarrow P_3$ describe a process that repeatedly inputs and then outputs a value.

3.2. Choice Operators

CSP offers choice operations for processes, which are called *external* and *nondeterministic* choice operators denoted as $\Box$ and $\sqcap$ respectively. The process $P \Box Q$ can act like P or Q depending on the choice of the initial event chosen by the environment. For instance, for the process $(a \rightarrow P) \Box (b \rightarrow Q)$, 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 $b \rightarrow Q$ and the environment has no control over which. Indexed versions of external and nondeterministic choices allow the choices to be made among a number of processes.

In addition to these, there is also the traditional conditional choice if $-$ then $-$ else operator.

3.3. Parallel Operators

Systems can be made up of a collection of processes that run in parallel and synchronise on the events that they agree to perform. Alphabetised parallel $P \parallel Q$ executes P and Q in parallel, where they have to synchronise on those events that are in both of their alphabets, but they can perform other events independently. Thus, they must only agree on the events in the intersection $\alpha P \cap \alpha Q$. This operator is associative and commutative, so we can combine any number of processes in parallel in any order without ambiguity. Thus we may write $P \parallel Q \parallel R$ for the parallel combination of three processes.

Alternatively, we may wish to run any two processes independently of each other, i.e., they do not synchronise on any events, not even those that they share. The interleaving operator is written "$|||$". This also has an indexed form to describe the interleaving of a family of processes. 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 occurrences of events in A internal, and hence 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, $R[\{a\}]$. For instance, given the process $P \mathrel{\widehat{=}} a \to a \to STOP$, and the relations aRb and aRc, $P[\![R]\!]$ should be considered as:

$$P[\![R]\!] \mathrel{\widehat{=}} (b \to (b \to STOP \,\square\, c \to STOP)) \,\square\, (c \to (b \to STOP \,\square\, c \to STOP))$$

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 relation as described in [Ros10].

We often use a substitution-like notation for describing relations. We write $P[\![^a/_b]\!]$ to mean that the event or channel b is replaced by a in P, e.g., $(b \to STOP)[\![^a/_b]\!] \mathrel{\widehat{=}} a \to STOP$, and $(b?x \to STOP)[\![^a/_b]\!] \mathrel{\widehat{=}} a?x \to STOP$. More generally, we allow multiple substitutions $P[\![^{a,\,b}/_{b,\,a}]\!]$ (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). We will also overload notation and use $[\![^X/_Y]\!]$ to refer to the relation corresponding to the renaming.

A useful result on composing renamings is that renaming via relation R followed by renaming through R' is equivalent to renaming through the relational composition $R\,;R'$.

Lemma 1. $P[\![R]\!][\![R']\!] \mathrel{\widehat{=}} P[\![R\,;R']\!]$

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 *the traces model*, $\mathcal{T}$, in this paper, which is the finite sequences of events that a process can perform. Traces are sequences of events, denoted $\langle a_1, a_2, \ldots, a_n\rangle$. The empty trace is denoted $\langle\rangle$, and concatenation of two traces is denoted $tr_1\hat{}\,tr_2$. $tr \upharpoonright A$ is the projection of tr onto the set A (i.e. the sequence of events in tr that are in A), and $tr \setminus A$ is the projection of tr onto $\Sigma\backslash A$, i.e., the trace tr with events from A removed. Two traces tr_1 and tr_2 are related by R if they are pointwise related, i.e., they are the same length and the events at each position are related by R.

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 its trace set. For instance, the set of 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 the traces model are as follows:

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

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 *the 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 where 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 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 the *SPEC*. If the refinement does not hold, then FDR produces counter-examples of the refinement, which are sequences of events that demonstrate the violation of the specification. Although FDR is automated, and easily used to check refinements, it suffers from a problem that is generic for all model checking tools: state space explosion.

4. Anonymity for Voting Systems

In our CSP approach to anonymity, we will model agents' actions by events of the form $channel.i.x$, where the *channel* is a channel representing the type of event, i is the identity of the agent and x is the content of the event; for instance, $vote.a.c$ will represent voter a casting a vote for candidate c. Anonymity will concern the origin of these events: that is, it will deal with cases where 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*, and x is in the set $Data$.

Thus, the set of all the messages can be written as:

$$A = \{channel.i.x \mid i \in USERS, x \in Data\}$$

Intuitively, if an observer has access to only the content (x) of the message, and the identity of the agent (i) is hidden from the observer, then the content could equally have been generated by any other agent.

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

$$P[\![^{x}/_{y} \mid x, y \in A]\!] \equiv_{\mathrm{T}} P$$

The original definition in [SS96] expressed this definition using functional and inverse functional renaming, as follows (now cast in relational notation): $P[\![^{\beta}/_{A}]\!][\![^{A}/_{\beta}]\!] \equiv_{\mathrm{T}} P$ where $\beta \notin \alpha P$.

(We use $[\![^{\beta}/_{A}]\!]$ as shorthand for $[\![^{\beta}/_{x} \mid x \in A]\!]$, and $[\![^{A}/_{\beta}]\!]$ as shorthand for $[\![^{x}/_{\beta} \mid x \in A]\!]$. We also use $[\![^{A}/_{A}]\!]$ as shorthand for $[\![^{x}/_{y} \mid x, y \in A]\!]$.)

The definition we give here is equivalent, since the relational composition of the relations $[\![^{\beta}/_{A}]\!]$ and $[\![^{A}/_{\beta}]\!]$ is indeed $[\![^{A}/_{A}]\!]$.

$P[\![^{A}/_{A}]\!] \equiv_{\mathrm{T}} P$ means that the two processes, P, and the renamed process, $P[\![^{A}/_{A}]\!]$, are trace equivalent, so indistinguishable from the point of view of an observer who can see traces. The two corollaries are:

1. If the abstracted system P is anonymous on the sets 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'.

The second anonymity definition that we are interested in is weak anonymity.

Definition 2 (Weak Anonymity). The process P is weakly anonymous on a set of channels C of type T if:

$$P[\![^{c.x,\, d.x}/_{d.x,\, c.x} \mid x \in T]\!] \equiv_{\mathrm{T}} P$$

for any $c, d \in C$

This states that the process has the same behaviours if any two channels from C are swapped. That is, if we consistently swap $c.x$ and $d.x$ within P for all values of x, then the result is indistinguishable from the original from an observer's point of view. The ability to swap them without making any difference provides anonymity with respect to the channel that has been used. We will write $[\![^{c,\, d}/_{d,\, c}]\!]$ as shorthand for $[\![^{c.x,\, d.x}/_{d.x,\, c.x} \mid x \in T]\!]$.

It follows that strong anonymity on A implies weak anonymity on channels contained within A, as stated in the following lemma:

Lemma 2. If P is strongly anonymous on A and $\{|C|\} \subseteq A$, then P is weakly anonymous on channels C.

Proof Assume P is strongly anonymous. Then consider some arbitrary $c, d \in C$:

$$\begin{array}{rlll} P[\![^{c,\, d}/_{d,\, c}]\!] & \equiv_{\mathrm{T}} & P[\![^{A}/_{A}]\!][\![^{c,\, d}/_{d,\, c}]\!] & \text{by strong anonymity on } A \\ & \equiv_{\mathrm{T}} & P[\![^{A}/_{A}]\!] & \text{since } [\![^{A}/_{A}]\!] \,;\, [\![^{c,\, d}/_{d,\, c}]\!] = [\![^{A}/_{A}]\!] \\ & \equiv_{\mathrm{T}} & P & \text{by strong anonymity on } A \end{array}$$

For instance, in the context of voting suppose two honest voters v_a and v_b cast their votes for candidates c_x and c_y, modelled by the occurrence of events $vote.v_a.c_x$ and $vote.v_b.c_y$. The weak anonymity definition for a voting system SYS will be on the channels $\{vote.v \mid v \in VOTERS\}$. This specifies that, for any v_a, v_b values:

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

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

To clarify the difference between the strong anonymity definition and the weak anonymity definition, and the abstraction methods used, we give a 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, it is known that v_1 always says *yes* (if he votes at all), and similarly the other voter v_2 always says *no*. Then we define the process *Ref* by:

$$Ref \mathrel{\widehat{=}} vote.v_1 \rightarrow yes \rightarrow STOP \;\Box\; vote.v_2 \rightarrow no \rightarrow STOP$$

If we want to verify whether the process satisfies strong anonymity, we need to check the trace equivalence $Ref[\![^A/_A]\!] \equiv_{\mathrm{T}} Ref$ for the set $A = \{vote.v_1, vote.v_2\}$.

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

The refinement check does not hold, because $\langle vote.v_1, no\rangle$ is a trace of $Ref[\![^A/_A]\!]$, but not of *Ref*. This has happened because the 'no' vote is sufficient to identify the voter. However, if we hide the events in $H = \{yes, no\}$ from the observer, then the new process *Ref1* becomes:

$$Ref1 \mathrel{\widehat{=}} Ref \setminus H \mathrel{\widehat{=}} vote.v_1 \rightarrow STOP \;\Box\; vote.v_2 \rightarrow STOP$$

Now, when we apply the strong anonymity definition to *Ref1*, the process $Ref1[\![^A/_A]\!]$ will have the same trace as *Ref1*, and the specification is met.

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

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

We can now verify that *Ref2* provides anonymity by confirming the trace equivalence $Ref2[\![^A/_A]\!] \equiv_{\mathrm{T}} Ref2$. The equality holds for the set $A = \{vote.v_1, vote.v_2\}$, demonstrating that the process *Ref2* provides strong 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. The process $Ref3 \mathrel{\widehat{=}} Ref \;|||\; RUN(M)$ can perform any event from the set M, and the observer cannot tell whether the occurrence of such events is from *Ref* or from $RUN(M)$. The equality $Ref3[\![^A/_A]\!] \equiv_{\mathrm{T}} Ref3$ again holds, and so the process *Ref3* with such events masked provides strong anonymity.

However, let us assume that both voters are attending to the referendum, and each can vote only once. The new referendum process can be described as follows:

$$\begin{aligned} newRef \mathrel{\widehat{=}} \;& vote.v_1 \rightarrow yes \rightarrow vote.v_2 \rightarrow no \rightarrow STOP \\ & \Box vote.v_1 \rightarrow no \rightarrow vote.v_2 \rightarrow yes \rightarrow STOP \end{aligned}$$

When we apply the strong anonymity definition to the new referendum process *newRef*, the refinement does not hold although we hide *yes* and *no* events from the observer (either with the hiding or renaming methods). The counter-example $\langle vote.v_1, vote.v_1\rangle$ states that voter v_1 can vote twice, whereas *newRef* does not let this trace happen. To sum up, the strong anonymity definition states that any voter is plausible for any vote; it does not matter whether the voter has voted before.

On the other hand, the weak anonymity definition swaps only the occurrences of $vote.v_1$ and $vote.v_2$. Thus, the problem created by the strong anonymity definition is solved with the weak anonymity definition.

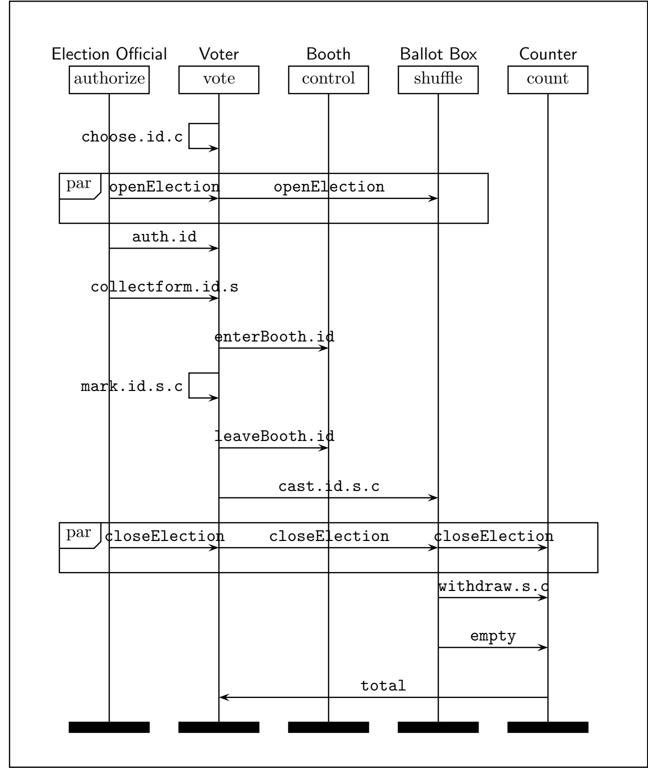

Figure 1. Conventional Voting System Design

However, we still need to abstract away the sensitive information *yes* and *no* from the observer's point of view by using one of the techniques provided above.

5. Modelling and Analysis of a Conventional Voting System

We have described anonymity properties of trustworthy voting systems, and defined anonymity in CSP. The next step is to model a conventional voting system in CSP, and to check it against appropriate anonymity specifications. In our model, we describe the system by means of the processes shown at the top of Figure 1.

The processes and the events used in our model are described as follows[2].

A *Voter* from the set of voters chooses a candidate to vote for from the given candidate list before going to the polling station and identifying herself to the electoral official. As the choice of candidate is made by the voter, nondeterministic choice is the appropriate CSP operator, since the choice is not under the control of the system.

She then receives a ballot form with a serial number on it; as the ballot form is given by the authority to the voter, an external choice operator is 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, and then leaves the booth, casts her vote by dropping the ballot form in the ballot box, and leaves the polling station. The voter process is modelled as:

$$VOTER(id) \mathrel{\widehat=} \bigsqcap_{c \in candidates} choose.id.c \rightarrow openElection \rightarrow auth.id \rightarrow$$

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

$$\Box_{s \in serials} \; collectform.id.s \rightarrow enterBooth!id \rightarrow mark.id.s.c \rightarrow$$
$$leaveBooth!id \rightarrow cast.id.s.c \rightarrow closeElection \rightarrow VOTER(id)$$

In the model, all the voters are as described 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 are not explicitly stated in the main body of the paper, but are taken to be the set of all the events they can perform. However, the details about each alphabet can be seen in Appendix B.

An *Election Official* working in a polling station authenticates eligible voters by their identification documents, and issues the ballot papers on which there are arbitrary and unique serial numbers. 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. The election official never gives the same serial number twice, so two different voters cannot receive the same serial number (the same ballot form) to vote. The election official process also opens and closes the election for a polling station; other processes synchronise on *openElection* and *closeElection* to keep track of the state of the election.

$$ELECOFFICIAL \mathrel{\widehat{=}} openElection \rightarrow OFFICIAL(voters, serials)$$
$$OFFICIAL(ids, serials) \mathrel{\widehat{=}} \Big(\Box_{id \in ids} \; auth.id \rightarrow \Big(\sqcap_{s \in serials} \; collectform.id.s \rightarrow OFFICIAL(ids \setminus \{id\}), serials \setminus \{s\}) \Big) \Big) \;\Box\; closeElection \rightarrow STOP$$

A *Booth* is a private environment for the voters to vote without being observed. Thus, in the model, the booth process allows one voter to go in, to vote and to leave before it allows the next voter to enter:

$$BOOTH \mathrel{\widehat{=}} \Box_{id \in voters} \; enterBooth.id \rightarrow leaveBooth!id \rightarrow BOOTH$$

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 collection. Once the election is closed, the box can be opened, and all the ballot papers can be withdrawn for the tallying:

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

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

A *Counter* is the official who removes all the completed ballots from the ballot box and tallies them. Once the ballot box is empty, he announces the total votes that each candidate has received:

$$COUNTER \mathrel{\widehat{=}} closeElection \rightarrow COUNTER1(0,0,0)$$
$$COUNTER1(i,j,k) \mathrel{\widehat{=}}$$

$$\left(\mathop{\Box}_{\substack{s \in serials \\ c \in candidates}} \left(\begin{array}{l} \text{if } c == c1 \text{ then } withdraw.s.c \rightarrow COUNTER1(i+1, j, k) \text{ else } STOP \\ \Box \text{ if } c == c2 \text{ then } withdraw.s.c \rightarrow COUNTER1(i, j+1, k) \text{ else } STOP \\ \Box \text{ if } c == c3 \text{ then } withdraw.s.c \rightarrow COUNTER1(i, j, k+1) \text{ else } STOP \end{array} \right) \right)$$
$$\Box$$
$$empty \rightarrow total!c1!i \rightarrow total!c2!j \rightarrow total!c3!k \rightarrow SKIP$$

The system for the conventional voting is defined as a parallel composition of all the five processes defined above:

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

5.1. Sanity Checks

Before we perform formal analysis on the model, it is wise to check that the voting system preserves some desired properties, by means of appropriate sanity checks. For instance, the 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. 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 votes cast during the election. The specification *SNTY_SPEC2* and the assertion for this test can be defined as follows:

$$SNTY_SPEC2 \mathrel{\widehat=} COUNTTHIS(0, card(voters))$$
$$COUNTTHIS(n,t) \mathrel{\widehat=} \left(\mathop{\Box}_{\substack{x \in voters \\ y \in serials \\ z \in candidates}} cast.x.y.z \rightarrow \text{if } n \leq NumOfMaxPossVotes \text{ then } COUNTTHIS(n+1,\ t+1) \text{ else } STOP \right)$$
$$\Box$$
$$\left(\mathop{\Box}_{i \in numOfVotes} total.c1.i \rightarrow COUNTTHIS1((n-i),t) \right)$$
$$COUNTTHIS1(s,z) \mathrel{\widehat=} \mathop{\Box}_{j \in numOfVotes} total.c2.j \rightarrow \text{if } s == j \text{ then } total.c3.0 \rightarrow SKIP \text{ else } (\text{if } 0 \leq s - j \,\&\, s - j \leq NumOfMaxPossVotes \text{ then } total.c3.(s-j) \rightarrow STOP \text{ else } STOP)$$

$$SNTY_SPEC2 \sqsubseteq_T SYSTEM \setminus \Sigma \setminus \{| cast, total |\}$$

As expected the sanity checks are satisfied showing that the model does not allow votes after the election is closed, and nor does it miscount the total number of votes. There are other sanity checks that it is wise to perform on the model, but for brevity we have discussed just these two.

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. (For the purposes of this paper, we do not consider active

intruders, such as those proposed by the Dolev-Yao model [DY83]; this is left as future work.) 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:

- which ballot form a particular voter has been given (that is, a link between serial numbers and voters);
- how a voter marked and cast her ballot form.

A first pass at describing the system that the observer can see is:

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

The system above is the model of the conventional voting system, in which we hide secret information from the observer, who thus cannot see *mark* and *collectform* events. However, for the *cast* events, we need to allow the event to be visible but to hide the content, so that the observer can see that the voter is casting a vote, but not for whom. Hence, we rename *cast* events as *envelope* events, and remove the data:

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

5.3. Strong Anonymity Analysis

As noted by Schneider and Sidiropoulos in [SS96], different definitions of anonymity are required for different situations. For instance, in a voting system where the anonymity of the voter identity is required, the strong anonymity definition that we gave previously is too strong. Two different votes must be generated by two different voters, which means that the two votes are not entirely independent. As a result, a strong anonymity check with their definition will fail, because it will require the possibility that two votes were cast by the same voter.

To see this, let us define strong anonymity for our voting system as follows, by masking the *choose* events:

$$SPEC_STRONG \mathrel{\widehat{=}} ABS_SYSTEM[\![{}^{dummy}/_{choose.id.c1}]\!][\![{}^{choose.id.c1}/_{dummy}]\!]$$

It can be seen, either by inspection or by using FDR, that the system does not satisfy this specification. FDR gives a counterexample trace of $\langle choose.v1.c1, choose.v1.c1\rangle$. In other words, the protocol does not provide strong anonymity from the observer's point of view as the CVS model does not let voters vote multiple times.

So our CVS does not meet the strong anonymity definition in [SS96]. Our next task is to check if the weak anonymity definition is a suitable specification for voting systems. For annotated machine-readable CSP (CSP_M) code of the model presented here, see Appendix B.

5.4. Weak Anonymity Analysis

For our second analysis, we will use the weak anonymity definition formalised in Section 4. We analyse the system, comparing two situations: the first, in which the voters $v1$ and $v2$ vote any way they like; and the second, in which the voters swap their votes. From the observer's point of view, *ABS_SYSTEM* (the first situation) and *SPEC_WEAK* (the second) should be indistinguishable. We define the system where two votes are swapped as follows:

$$SPEC_WEAK \mathrel{\widehat{=}} ABS_SYSTEM[\![{}^{choose.v1.c,\, choose.v2.c}/_{choose.v2.c,\, choose.v1.c}]\!]$$

SPEC_WEAK is the process where events *choose.v1.c* and *choose.v2.c* are swapped over, where $c \in candidates$ is any candidate. The claim that the CVS satisfies the weak anonymity definition is embodied in the trace equivalence of these two systems as shown below.

	Serial
Bob	
Alice	X
Chris	
	Teller Onion

Figure 2. Prêt à Voter Ballot Form

$SPEC_WEAK \equiv_{\mathrm{T}} ABS_SYSTEM$

The assertion holds, which means that the two systems are equivalent from the observer's point of view. Therefore, our model of a conventional voting system satisfies this definition of weak anonymity.

6. Modelling and Analysis of Prêt à Voter

Our second case study involves modelling and analysing Prêt à Voter using our anonymity definitions. Voting systems like Prêt à Voter are complex, and rarely subjected to formal analysis; in consequence there is the possibility that they are vulnerable to attacks. However, modelling such systems is a challenging problem. In this section, we model the Prêt à Voter voting system in CSP and verify that our model does indeed satisfy the weak anonymity property.

First we give an outline of the Prêt à Voter voting system, and informally describe its components. Then, we formally build the processes, the events and the sets used in the model. Finally, we analyse the model against the anonymity property given in Section 4.

6.1. Outline of Prêt à Voter

Prêt à Voter is a paper-based, voter-verifiable cryptographic e-voting system, introduced by Peter Ryan in 2005 [Rya05] as an improvement of Chaum's scheme proposed in 2004 [Cha04]. Since then, it has been improved and enhanced in many different ways [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 is (by design) quite similar to voting in the conventional voting system. Figure 2 illustrates a simple Prêt à Voter ballot form. On the left-hand column of the ballot form is printed a random permutation of the candidate names, and on the right-hand column are the boxes in which the voter can mark her choice. There is a perforation line separating the two halves. The cryptographic value at the bottom of the right-hand column is called a *teller onion*, which embeds the candidate ordering on the left-hand side, and is encrypted under the tellers' public key. This public key is a threshold key for an appropriate homomorphic encryption algorithm, such as ElGamal [ElG84] or Paillier [Pai99]; the precise algorithm is not important for the purposes of this paper.

An eligible voter goes to a polling station, authenticates herself to the election official and takes a random ballot form in an envelope. (These ballot forms have been produced by the election authorities before the election day, and kept sealed.) Once the voter goes into the booth, she then marks her choice with a cross on the right-hand column, tears the ballot form down the perforation line, and shreds the candidate list. Finally, she scans her ballot and takes the right-hand column of the ballot paper as her receipt; later, all the scanned right-hand sides will be published on a web bulletin board (WBB), and she will be able to use

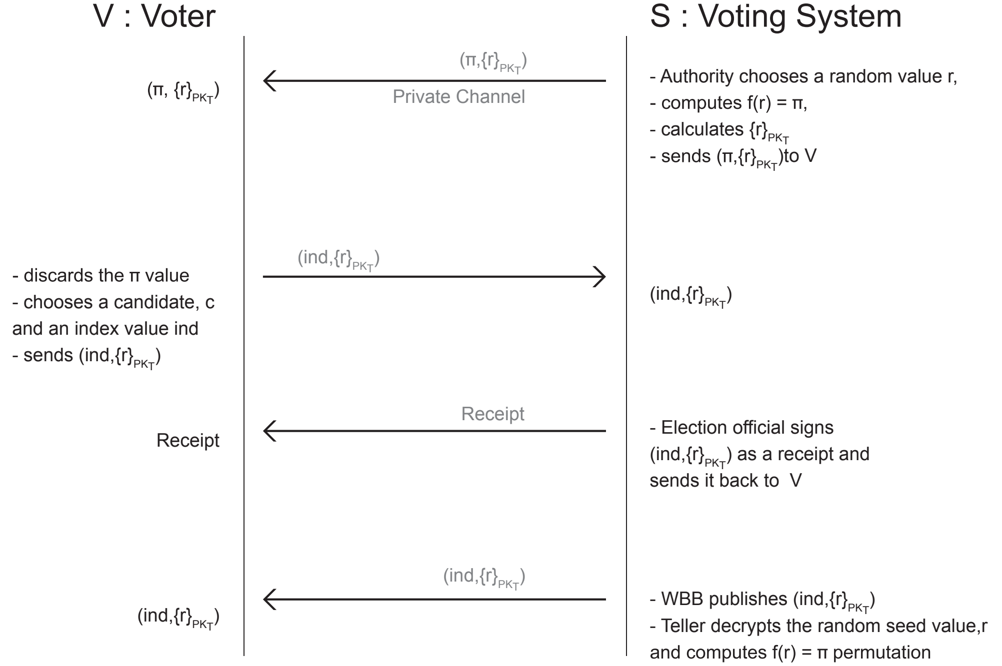

Figure 3. Prêt à Voter Protocol

her receipt to check that her vote has not been changed or deleted. As the candidate name order on the left-hand column is random, the right-hand column does not reveal anything about her vote.

After all the votes have been cast, the votes are passed through a mixnet; the mixed votes are then decrypted jointly by the tellers to reveal the anonymised votes.

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

- 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, (so $f(r) = \pi$), 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 chooses a candidate c, marks the ballot form finding the corresponding index value, ind, and sends the tuple $(ind, \{r\}_{PK_T})$ to WBB, discarding the permutation, π.
- The election official signs the receipt $(ind, \{r\}_{PK_T})$ and sends it back to the voter.
- The WBB publishes $\{r\}_{PK_T}$, 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 is intended to be a transparent and 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, and audit authorities check the integrity of the ballots by recomputing the seed values and teller onion value. Additionally, the voter is also able to audit the ballot paper handed to her: once she requests an audit, the ballot form is scanned 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. After auditing, the voter then receives a new ballot form; she may audit as many ballot forms as she wishes to convince herself that all is well.

One of the most important parts of the system is the tellers, who handle the mixing and decryption. There is also a mechanism for auditing the tellers to ensure that they do not manipulate the votes. This is typically done by means of randomized partial checking (RPC), as proposed in [JJR02]: the tellers reveal

a randomly chosen half of their input and output links in such a way that there is no complete route from input to output, so that no ballot receipt can be traced, and the remaining links are hidden. A modification of a single value has a 50% chance of being caught; a teller therefore has only a $\frac{1}{2^n}$ chance of getting away with modifying n votes.

6.1.2. Prêt à Voter System Properties

Anonymity/Voter privacy: The system is stated to provide voter anonymity; no one learns anything about the voter's choice (also known as ballot secrecy).

Verifiability: The voters can verify that their receipts appear on the bulletin board after the election.

Ballot paper integrity: Ballot paper integrity is checked in the polling station by the voters by feeding right-hand side of a ballot form into the scanner. Once the candidate list on the left-hand side of the ballot form has been reproduced, then the voters can compare those two: original left-hand side and reproduced one. This process can last as long as the voters wish.

Coercion-resistance: The system is coercion-resistant, meaning that a voter cannot prove to an adversary how she has voted even if the voter cooperates with the adversary during the election. The system is also resistant to vote-selling.

6.2. Prêt à Voter System Components

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

Voter: A registered voter starts by choosing a candidate to vote for. Afterwards, she authenticates herself in the polling station to the election authority. She then gets an empty ballot form from the authority and goes into the booth to place a mark for the candidate she has chosen. She shreds 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 then receives a 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; 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: This is the printing and scanning machine that the voter interacts with to cast her ballot and receive the receipt. The machine is located in the polling station, but outside the booth. The machine scans the RHS of a ballot form, stores the encrypted ballot form, and prints the receipt for the voter. The machine also passes the receipt to the WBB.

Web Bulletin Board (WBB): The WBB stores and publishes all signed receipts so that voters can check their receipts against the WBB. The WBB also sends a batch of encrypted votes to the mixnet for mixing, and it can, in addition, request 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. It 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 after they have been shuffled and re-encrypted by the mixnet. Thus, after the re-encryption mix phase, the tellers take the mixed and encrypted votes, decrypt them, and perform the final tally with the plaintext values. (For simplicity, 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.)

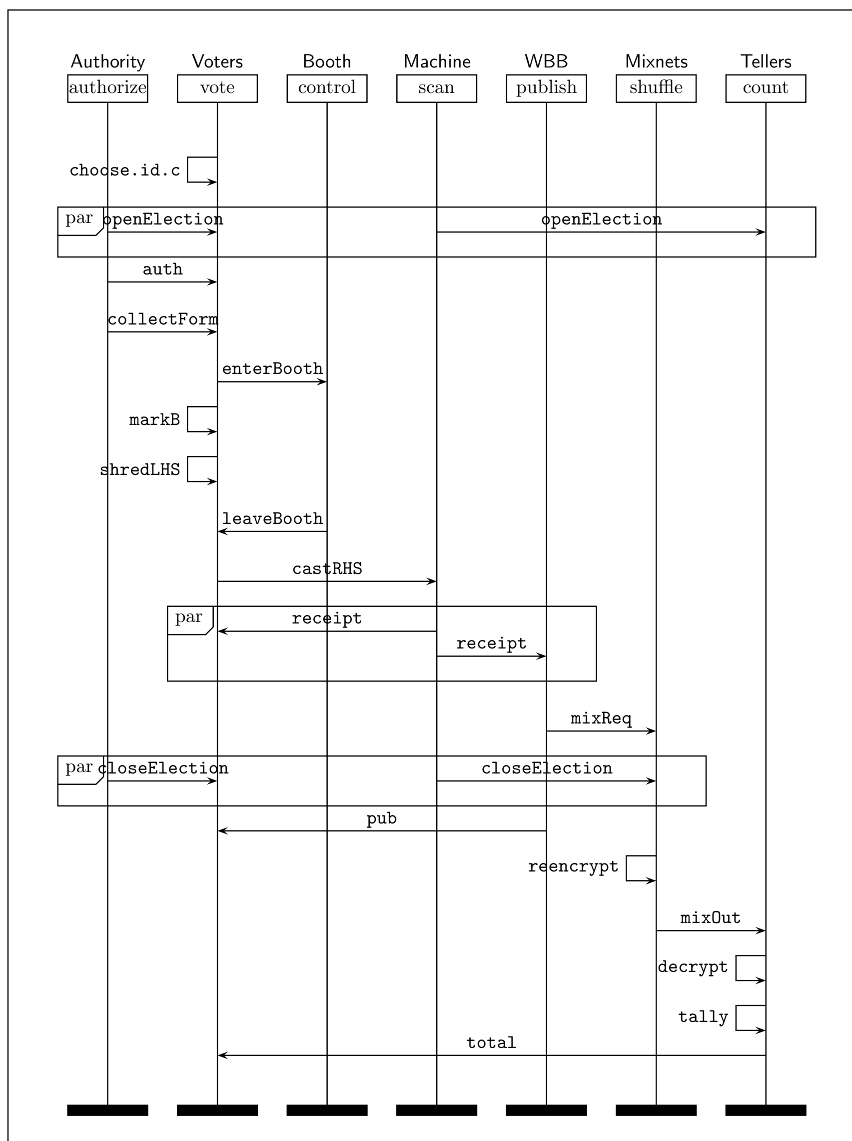

Figure 4. Prêt à Voter System Model

Booth: The booth is a private environment to enable the voters to vote unobserved. 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, the 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.

We use abstract data-types as appropriate in a CSP model. For instance, we consider encryption as a formal symbolic operation. The encryption function $enc(pkt, m)$ is the public-key encryption of the message

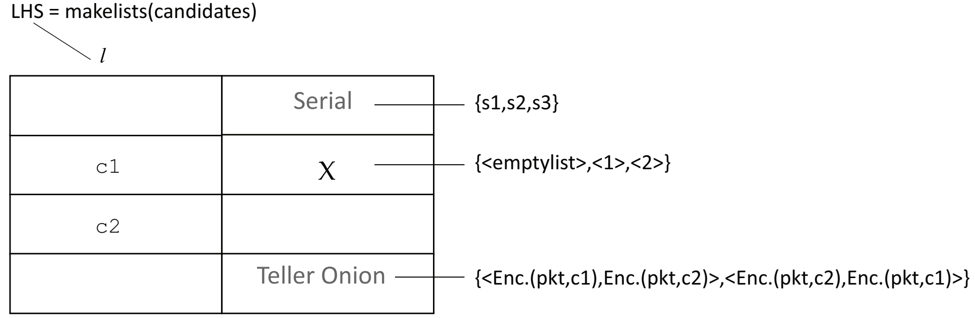

Figure 5. Prêt à Voter Ballot Model

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 $enc(pkt, m) = Enc.(pkt, m)$. Only the one who has the corresponding secret key $inverse(pkt) = skt$ can decrypt the message using the decryption function $dec(skt, enc(pkt, m))$. The key pair formed by pkt and skt is the teller's key pair; the public part has been used to construct the ballot forms by the authority before the election. It is also used in the re-encryption phase by the mixnet. As the teller knows the secret key, he can extract the re-encrypted shuffled values and the onion values, which embed the actual vote, in the tallying phase. In order to avoid state explosion in the model checker, we limit the number of agents in our model. 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. On the LHS, there is a candidate list, 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 representing an index into the candidate list as it appears on the ballot paper; as there are two candidates in our setup, the voter chooses 1 for the first candidate or 2 for the second candidate. An empty ballot form is shown as the data *emptylist*.

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

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

Some further expressions and notation 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. The *find* function is used by the voter process to see the corresponding grid for the candidate she has chosen. The function is defined by means of the *head* and *tail* functions, and returns either 1 or 2. The *nth* function is

also defined using *head* and *tail* to extract the nth element of a sequence.

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

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

6.4. Processes and Events

We now describe how the processes in our Prêt à Voter model work with these data-types and events. We explain what events each process in the model performs in the relevant process description.

6.4.1. Election Authority Process

In the re-encryption version of Prêt à Voter, 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 the teller's public key and forwards pre-printed ballot forms in an envelope to the voters so that the authority cannot see which ballot form is used by which voter. (In practice, the ballot generation code is run on a diskless workstation, which generates the ballots, prints them, and then shuts down, keeping no record of its actions. The candidate lists are kept in only two places: printed on the ballot paper, and on the WBB encrypted under the threshold public key.) In our model, the election authority behaves as an electoral official who creates ballot forms and issues them to the voters directly. Therefore, information about the candidate list is considered to flow over a private channel. In the model, an empty ballot form is denoted as $\langle clist, \langle ser, emptylist, Enc.(pkt, clist)\rangle\rangle$, in which *clist* is a permutation of the candidate 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 $\langle Enc.(pkt, c1), Enc.(pkt, c2)\rangle$ represents the onion, which is the encryption of the list, *clist*, under the teller's public key, *pkt*. Consequently, the onion values can be revealed only by the decryption teller using the corresponding secret key.

The election authority first opens the election (event *openElection*). Then, upon request from the voters, he authenticates voters with their identification and issues them an empty ballot form with a non-deterministically chosen serial number and candidate list over the channel *collectform*. The authority can perform these actions for as long as there are eligible voters and serial numbers. Finally, he closes the election (*closeElection*).

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

As in the conventional voting system model, the alphabet of each process is the set of all events that a process can perform. Thus, the alphabet of $AUTHORITY$ is as below; the alphabets for the remaining processes in the system can be inferred from their definitions. The full details appear in Appendix 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 using the channel *mark*, and after destroying the left-hand side, she leaves the booth. Afterwards, she casts her vote under the supervision of the election authority using the machine supplied. Having cast the ballot form, the voter is provided with a receipt of her vote, which is the right-hand side of the ballot form, including the serial number. Once the voter gets her receipt and leaves the polling station, voting finishes for her. The serial numbers on the receipts are used by the voter for verification on the web bulletin board. Once all voters have finished voting, they synchronise on the event *closeElection*.

$$\begin{array}{l} \mathit{VOTER}(id) \mathrel{\widehat{=}} \displaystyle\prod_{c \in candidates} \mathit{choose!id.c} \rightarrow \mathit{openElection} \rightarrow \mathit{auth.id} \rightarrow \\ \quad \displaystyle\Box_{\langle l,s,emptylist,o\rangle \in emptyforms} \mathit{collectform.id}.\langle l, s, emptylist, o\rangle \rightarrow \mathit{enterBooth.id} \rightarrow \\ \quad\quad \mathit{mark}.\langle l, s, \mathit{find}(c,l), \mathit{onionlist}(pkt, l)\rangle \rightarrow \\ \quad\quad\quad \mathit{shredLHS}.\langle s, \mathit{find}(c,l), \mathit{onionlist}(pkt, l)\rangle \rightarrow \\ \quad\quad\quad\quad \mathit{leaveBooth.id} \rightarrow \mathit{cast}.\langle s, \mathit{find}(c,l), \mathit{onionlist}(pkt, l)\rangle \rightarrow \\ \quad\quad\quad\quad\quad \mathit{receipt}.\langle s, \mathit{find}(c,l), \mathit{onionlist}(pkt, l)\rangle \rightarrow \\ \quad\quad\quad\quad\quad\quad \mathit{closeElection} \rightarrow \mathit{STOP} \end{array}$$

Thus the process representing all voters is described by the parallel composition of the voters as:

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

6.4.3. Machine Process

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

$$\begin{array}{l} \mathit{MACHINE} \mathrel{\widehat{=}} \mathit{openElection} \rightarrow \mathit{MACHINE1} \\ \mathit{MACHINE1} \mathrel{\widehat{=}} (\\ \quad \displaystyle\Box_{rhs \in markedRHSs} \mathit{cast.rhs} \rightarrow \mathit{receipt.rhs} \rightarrow \mathit{MACHINE1} \\ \quad) \\ \quad \Box \\ \quad \mathit{closeElection} \rightarrow \mathit{MACHINE} \end{array}$$

6.4.4. Web Bulletin Board Process

Once the *openElection* event has occurred, the process *WBB* starts receiving the digital copies of cast right-hand sides returned by the *MACHINE* process on the channel *receipt*. It keeps track of the receipts in a set called *Receipts*, which is initially an empty set. The WBB process can also request shuffling for the votes by sending them one by one to the mixnet process[3] on the channel *mixReq*. However, serial numbers are stripped off beforehand. Once the election is closed and all votes have been sent for shuffling, the process publishes all the receipts kept in the set *Receipts*. 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.

$$\begin{array}{l} \mathit{WBB} \mathrel{\widehat{=}} \mathit{openElection} \rightarrow \mathit{WBB1}(\emptyset) \\ \mathit{WBB1}(\mathit{Receipts}) \mathrel{\widehat{=}} (\displaystyle\Box_{\langle s,i,o\rangle \in markedRHSs} \mathit{receipt}.\langle s, i, o\rangle \rightarrow \mathit{mixReq}.\langle nth(i, o)\rangle \rightarrow \\ \quad \mathit{WBB1}(\mathit{Receipts} \cup \{\langle s, i, o\rangle\}) \end{array}$$

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

$$
\begin{aligned}
&\quad)\\
&\quad \Box\\
&\quad closeElection \rightarrow WBB2(Receipts)\\
&WBB2(\emptyset) \mathrel{\widehat{=}} STOP\\
&WBB2(Receipts) \mathrel{\widehat{=}} \bigsqcap_{rcp \in Receipts} pub.rcp \rightarrow WBB2(Receipts \setminus \{rcp\})
\end{aligned}
$$

6.4.5. Mixnet Process

The *MIX* process behaves as a mixnet, which performs a mix for the digital copies of the 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. Once all votes have been re-encrypted, as indicated by the event *bagempty*, the mixnet begins giving the re-encrypted votes out to the WBB non-deterministically and one by one.

In our abstraction modelling level, we assume that the mixnet is honest and does not reveal any information about the mapping from input to output. As having more than one mixnet would not make any difference as a consequence of the non-deterministic construction of the mixnet, we use just one mixnet to re-encrypt and shuffle the votes in our model.

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

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 ballot generation are performed under the teller's public key *pkt*, the teller now can decrypt each of them, and tally the plaintext values, outputting the result for each candidate on channel *total*.

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

6.4.7. System Process

The Prêt à Voter voting system model is the parallel composition of the processes defined previously (refer to Appendix A for the CSP_M codes of the model and the alphabets of each process). The composition is defined as follows:

$$SYSTEM \mathrel{\widehat{=}} VOTERS \parallel AUTHORITY \parallel MACHINE \parallel BOOTH \parallel WBB \parallel MIX \parallel TELLER$$

6.5. Sanity Checks

The following four sanity checks help to give confidence that the model of Prêt à Voter is correct.

i. The first sanity check ensures that *no one can be authenticated twice*. The specification is:

$$AUTH(v) \mathrel{\widehat{=}} auth.v \rightarrow STOP$$
$$SNTY_SPEC1(voters) \mathrel{\widehat{=}} \mathop{|||}_{id \in voters} AUTH(id)$$

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. A violation of the specification would mean that the model allows a voter to be authenticated twice.

$$SNTY_SPEC1(voters) \sqsubseteq_T SYSTEM \setminus (\Sigma \setminus \{| auth |\})$$

ii. The second sanity check is that *no one can mark a ballot form before being authenticated*. As we are interested only in the authentication and marking actions, we hide all the other events from the system. Thus, the specification process *SNTY_SPEC2(voters)* can be written as:

$$ECHECK(v) \mathrel{\widehat{=}} auth.v \rightarrow AUTHED(v)$$
$$AUTHED(v) \mathrel{\widehat{=}} \left(\mathop{\Box}_{x \in markedforms} mark.v.x \rightarrow AUTHED(v) \right) \Box \left(\mathop{\Box}_{id \in voters} auth.id \rightarrow AUTHED(v) \right)$$
$$SNTY_SPEC2(voters) \mathrel{\widehat{=}} \mathop{|||}_{id \in voters} ECHECK(id)$$

$$SNTY_SPEC2(voters) \sqsubseteq_T SYSTEM \setminus (\Sigma \setminus \{| auth, mark |\})$$

The specification allows only authenticated voters to mark a ballot form.

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

$$SNTY_SPEC3 \mathrel{\widehat{=}} closeElection \rightarrow CLOSED \;\Box\; \left(\mathop{\Box}_{x \in markedRHSs} cast.x \rightarrow SNTY_SPEC3 \right)$$
$$CLOSED \mathrel{\widehat{=}} closeElection \rightarrow CLOSED$$

$$SNTY_SPEC3 \sqsubseteq_T SYSTEM \setminus \Sigma \setminus \{| closeElection, cast |\}$$

iv. The last sanity check ensures 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 number of 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. The specification *SNTY_SPEC4* and the refinement can be defined as follows:

$$SNTY_SPEC4 \mathrel{\widehat{=}} COUNT(0)$$
$$COUNT(n) \mathrel{\widehat{=}} \left(\mathop{\Box}_{x \in markedRHSs} cast.x \rightarrow COUNT(n+1) \right) \Box$$

$$\left(\mathop{\Box}_{i \in numOfVotes} total.c1.i \rightarrow COUNT1(n\text{-}i) \right)$$
$$COUNT1(j) \mathrel{\widehat{=}} total.c2.j \rightarrow STOP$$

$$SNTY_SPEC4 \sqsubseteq_{\mathrm{T}} SYSTEM \setminus \Sigma \setminus \{| \, cast, total \, |\}$$

FDR confirms that all the sanity checks defined above are satisfied by the Prêt à Voter voting system CSP model.

6.6. Anonymity Analysis of Prêt à Voter

We have already defined and tested the anonymity definition against the conventional voting system model in Section 5. Now that we have modelled Prêt à Voter in CSP, we can conduct further anonymity analysis. We check here if the Prêt à Voter voting system model provides anonymity with respect to the same anonymity definitions we used previously. Throughout the anonymity analysis of the model, we use the specification checks against the 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 consider only an observer as an attacker, but we also perform a formal verification of the model with a corrupted election official. We assume that the 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. The abstractions of which events are visible to the observer also effectively assume that the booth really is private.

6.6.1. Observer

As with the CVS verification, we have an observer who can see all the public information and some sensitive data that is listed below. The observer can see:

- the election's opening and closing,
- the identity of the voters and if they have voted in the authentication process,
- who goes in and out of the booth,
- voters shredding the left-hand side of a ballot form (but not 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:

- which ballot form a voter has been given, so the observer cannot identify which serial number is used by a particular voter,
- the marking of the vote in the booth,
- the choice of candidate.

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

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

We use a renaming abstraction to hide the sensitive information from the observer. As a result, the observer cannot distinguish among encrypted onion values: all the encryptions look essentially the same to the observer. Here *ABS_SYSTEM* is the system that is seen by the observer, which is formed using the renaming operator and the special function *mask*(). This function converts all encrypted data to one single value, *ciphertext*.

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

6.6.2. Strong Anonymity Analysis

From our anonymity analysis with the conventional voting system, we have shown that strong anonymity is not an appropriate specification check. We also expect this to be the case for the Prêt à Voter voting system.

In order to apply the strong anonymity definition, we abstract the *choose* events away by renaming them to another event called *dummy*, which is not in *SYSTEM*'s alphabet. We then rename this event back to any *choose* event, which means that any *choose* event should be replaceable by any other *choose* event. This, of course, implies that any *choose* event could have been generated by any voter. However, from the fact that the voting systems require that each vote should be cast by a different voter, we would expect the model not to satisfy the specification. The specification is defined as:

$$SPEC_STRONG \mathrel{\widehat=} ABS_SYSTEM[\![^{dummy}/_{choose.id.c1}]\!][\![^{choose.id.c1}/_{dummy}]\!]$$

As expected, the refinement $ABS_SYSTEM \sqsubseteq_T SPEC_STRONG$ does not hold, demonstrating that the strong anonymity definition is too strong for the Prêt à Voter voting system as well. The counterexample trace FDR provides 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 of the protocol. Therefore, Prêt à Voter does not satisfy strong anonymity.

6.6.3. Weak Anonymity Analysis

We now give the results of the weak anonymity analysis. We expect to discover that the two systems, the normal *ABS_SYSTEM* and the system *SPEC_WEAK*, in which we swap two votes over, 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 candidate list. To apply the weak anonymity definition, we use the renaming operator to swap the votes as shown below. This means that we swap the voters *v1* and *v2* over.

$$SPEC_WEAK \mathrel{\widehat=} ABS_SYSTEM[\![^{choose.v1.c,\, choose.v2.c}/_{choose.v2.c,\, choose.v1.c}]\!]$$

FDR tells us that the assertions $ABS_SYSTEM \sqsubseteq_T$ and $\sqsupseteq_T$ $SPEC_WEAK$ 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 weak anonymity with respect to this definition.

6.6.4. Misbehaving Agents

In the previous section, we analysed Prêt à Voter with respect to an observer who can see the public information on the channels that are available to him. 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 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 event *collectform* carries information mapping voters to ballot forms. We demonstrate here that an observer who can see *collectform* events can break a 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. 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* can be written:

$SPEC_WEAK2 \widehat{=} ABS_SYSTEM2[\![^{choose.v1.c,\, choose.v2.c}/_{choose.v2.c,\, choose.v1.c}]\!]$

Now, to verify anonymity, we check whether the refinement, $ABS_SYSTEM2 \equiv_{\mathrm{T}} SPEC_WEAK2$ holds. As we would expect, the trace equivalence check fails because the observer knows the left-hand side of the ballot form via *collectform*, which shows the candidate order. This shows that this information must remain secret if the anonymity property is to hold. A similar result can be obtained for the *shredLHS* events, from which an observer could deduce in which grid the voter placed a mark on the ballot form.

7. Results and Discussion

In this paper, we have investigated formal anonymity definitions, and in particular *strong* and *weak* anonymity for voting systems. We have used a conventional voting system and Prêt à Voter as two case studies to validate these definitions, and have shown how to provide automated analysis with respect to these definitions using the process algebra CSP and the FDR2 model checker.

7.1. Results

Our analysis has shown that the strong anonymity definition is too strong for analysing anonymity in most voting systems, if voters are not allowed to cast multiple ballots. The strong anonymity specification requires two voters to be independent, but voting systems typically (and our two models in particular) mandate that voters may cast only one vote each, implying that two given votes may not come from the same voter.

We therefore used the weak anonymity as our specification, and showed that our conventional voting system and Prêt à Voter both provide anonymity, and both models can be verified with respect to this specification automatically.

Our anonymity definition covers voting systems in which the final tally is published. Even if all the voters vote for the same candidate, although it is clear how each voter voted, an observer still cannot identify whether two voters have swapped their votes (because this is a null operation), and so the anonymity definition is still satisfied. Similarly, the definition is still met in elections with an electorate consisting of a single voter.

We also investigated the case in which the authority can assign the same serial number to two different voters for Prêt à Voter, showing that weak anonymity is still satisfied in this case. In the same way, we also demonstrated that the CVS model satisfies the weak anonymity definition even if there are no serial numbers.

7.2. Discussion

Although the strong anonymity definition is too strict for most voting systems, we can still use it as an anonymity specification in systems that allow a voter to vote multiple times in an election, or where only the winner of the election is announced and not the full tally. For instance, strong anonymity may be an appropriate definition for television polls where votes are cast by sending an SMS to a particular number.

State space considerations meant that we could verify only relatively small models, with 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. It is not currently clear whether it is possible to manipulate them into the appropriate form, but it seems unlikely: for instance, using functions (such as *card*) on data-types is not allowed, and the replicated parallel operator is banned, which are key features in our models. Inequality tests are also forbidden, and these are implicit in determining a winner. The most important limitation is on the specifications: there should be no hiding or renaming operators used in the specification, and our anonymity specifications are based on these operators. 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 away. Certainly any model that publishes a final tally will

not be susceptible to such a technique, since the addition of an extra voter increases the number of possible resullts. Future work will concentrate on finding a technique that does allow us to infer results on a large system based on results on a small system.

The observer in our models acts as a passive intruder, who can see some channels but not others. A natural next step is to model a more powerful intruder along the lines of a Dolev-Yao intruder [DY83], who is in control of the network, and can mount active attacks. Modelling such a setup requires a great deal of care: for example, current models would not provide anonymity in the presence of a Dolev-Yao intruder, as he could block all the votes sent to the mixnet except the one cast by a particular voter. The tally would consist of a single vote, and this would violate the targeted voter's anonymity. It is a genuine attack as long as there exist no private channels between the WBB and the mixnet, and the mixnet and the voter. We intend to focus on solving these modelling problems as the next stage in this work.

Acknowledgements

This work is a much expanded and improved version of a previous paper presented at the CryptoForma Workshop (Limerick, June 2011). We are grateful to the anonymous referees of that paper, and to those who participated in the discussion following the presentation, for their helpful comments and suggestions; the anonymous referees of this version of the paper were also extremely helpful in suggesting many improvements. We would additionally like to thank to David M. Williams and the members of the Trustworthy Voting Systems (TVS) project for their helpful comments.

This work was carried out under the EPSRC-funded Trustworthy Voting Systems (TVS) project. The first-named author is funded by Republic of Turkey Ministry of National Education. The research was conducted in part while 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.

[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.

[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. In *ESOP*, pages 186–200, 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. A threat analysis of Prêt à Voter. 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, SETS AND FUNCTIONS ----------------------------

datatype fact =  c1 | c2 | v1 | v2 | v3 | s1 | s2 | s3
                  | pkt | skt | emptylist | ciphertext
                  | Enc .(fact, fact)

-- The sets of candidates, voters and serial numbers
candidates      = {c1, c2}
voters          = {v1, v2}
serials         = {s1, s2}

-- The set of all possible number of votes that a candidate can get
numOfVotes      = {0..card(voters)}

-- The set of all possible number of candidates on a single ballot form
numOfCand       = {1..card(candidates)}

-- The number of voters
maxVotes        = card(voters)

-- The nth function finds the n'th element of a message/sequence
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

-- The function find() 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))

-- LHSs is the set of all possible candidate lists on a ballot form
LHSs            = makelists(candidates)
makelists({s})  = {<s>}
makelists(S)    = {<s>^a|s<-S,a<-makelists(diff(S,{s}))}

-- The set of all possible index values on right hand sides of the ballot forms
markedlists = {find(cand,left) | cand <- candidates, left<-LHSs}

-- The set of all possible onions, which are the encryptions of the LHSs 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 possible marked RHSs of the ballot forms
```

```
markedRHSs  = {<ser,list,o>,<ser,list,<ciphertext,ciphertext>>
                        |ser<-serials,list<-markedlists,o<-onions}

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

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

-- The set of all possible encrypted candidate lists
chosenCand = {<Enc.(pkt,c)>,<ciphertext> | c<-candidates}

-- The set of all possible encrypted receipts, which include index values and onion values
encList = {reEnc(pkt, <Enc.(pkt,c)>),<ciphertext> | c<-candidates}

-- The set of all possible decrypted votes
decryptedVotes = {<a> | a<-candidates}

------------------------------------------------------------------------------------------
-- CHANNELS ------------------------------------------------------------------------------

-- The channels openElection and closeElection represents opening a run of an election
-- and closing it respectively.
channel openElection, closeElection
-- The channel auth is the authority's authenticating the voters.
channel auth        : voters
-- The channel collecform is the voter's collecting an empty ballot form.
channel collectform : voters.emptyforms
-- The channel choose is that the voter decides whom to vote for.
channel choose      : voters.candidates
-- The channel mark is the voter's marking a ballot form.
channel mark        : voters.markedforms
-- The channel shredLHS is the voter's shreding the LHS of a ballot form.
channel shredLHS    : markedRHSs
-- The channel cast is the voter's casting a vote using the machine supplied.
channel cast        : markedRHSs
-- The channel receipt is the voter's taking away their receipts provided from MACHINE.
channel receipt     : markedRHSs
-- The channel leaveBooth and enterBooth is the voters' getting in and out the booth.
channel leaveBooth, enterBooth: voters
-- The channel mixReq is the WBB's requesting a shuffle for the receipts he received.
channel mixReq      : chosenCand
-- The channel pub is the WBB's publishing the cast votes.
channel pub         : markedRHSs
-- The channel encrypt is the mixnet's encrypting the receipts.
channel reencrypt   : encList
-- The channel mixOut is the mixnet's giving away the re-encrypted shuffled receipts.
channel mixOut      : encList
-- The channel bagempty ensures that there is no receipt left to mix.
channel bagempty
-- The channel decrypt is the Teller's decrypting the onion values.
channel decrypt     : decryptedVotes
-- The channel total is the Teller's announcing final tally for each candidate.
channel total       : candidates.numOfVotes

------------------------------------------------------------------------------------------
-- PROCESSES -----------------------------------------------------------------------------

------------------------------------------------------------------------------------------
-- The VOTER process receives an empty ballot form and casts the ballot form.

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 the voters' entering and leaving the
-- booth. Hence, there will be no two voters in the booth at the same time.

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 LHSs,
-- unique serial numbers and a candidate list. He, then, closes the election.

AUTHORITY               = openElection -> AUTHORITY1(voters, serials, LHSs)

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
------------------------------------------------------------------------------------------
-- The process MACHINE operates as a scanner and printer. MACHINE allows voters to cast
-- their votes and to collect their receipts.

MACHINE = openElection -> MACHINE1

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

                    )
                    []
                    closeElection -> MACHINE
------------------------------------------------------------------------------------------
-- The process WBB opens the election with AUTHORITY. It, then collects each receipt that
-- is cast in the set Receipts, and requests a shuffling from the mixnet for each vote.
-- After the election is closed, it publishes the receipts along with the serial numbers.

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.


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 process TELLER receives the encrypted shuffled votes from the process MIX, and
-- decrypts each using the teller's secret key, skt.

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
------------------------------------------------------------------------------------------
-- The process SYSTEM is Pret a Voter voting system model.

SYSTEM =  (((((VOTERS[aVTR||aAUTH]AUTHORITY)
                          [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
------------------------------------------------------------------------------------------
-- Process Alphabets
------------------------------------------------------------------------------------------
-- The sets below defines the alphabets of the corresponding processes
-- such as aVTR is the alphabet of VOTERS process.

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|}
-- Sigma is the alphabet of the process SYSTEM.
Sigma = Union({aVTR,aAUTH,aMAC,aBTH,aWBB,aMIX,aTEL})

------------------------------------------------------------------------------------------
-- SANITY CHECKS ---------------------------------------------------------------------

------------------------------------------------------------------------------------------
-- 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. This 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 that SYSTEM only allows the authenticated voters to
-- get 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 cast
-- events, 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 ----------------------------------------------------------------------

-- The system, which the observer can see, should be limited. Hence, SYSTEM1 is the
-- abstracted system in which we hide mark and collectform events.

SYSTEM1  = SYSTEM \ {|mark,collectform|}

-- The function maskFact() is used to rename all the encrypted data to a single data,
-- ciphertext.
maskFact(Enc._) = ciphertext
maskFact(x) = x

-- The function mask()  is for the sequences of encryption data.
mask(<>) = <>
mask(<x>^xs) = <maskFact(x)>^mask(xs)

-- ABS_SYSTEM is the system where we rename a number of events, which carry confidential
-- information that should be hidden from the observer.
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. The observer can only see what is public, and
-- he cannot distinguish whether the two encrypted values are the same, i.e., the
-- encrypted values should look all the same to the observer.

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 assertions hold, showing that Pret a Voter preserves voter anonymity.
-----------------------------------------------------------------------------------------
-- Strong Anonymity --
----------------------
-- We also checked the system against the strong anonymity definition by abstracting
-- choose events to a dummy event and abstracting them back.

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 we expected, and here is a counter example trace taken
-- from FDR2, showing a violation of the strong anonymity.
-----------------------------------------------------------------------------------------
------- 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 A Conventional Voting System

```
----------------------------------------------------------------------------------------
-- conventional.csp
-- Modelling and conventional voting system and verifying anonymity.
-- Murat Moran, February 2010
-- Dept. of Computing University of Surrey

----------------------------------------------------------------------------------------
-- DATA-TYPES, SETS AND FUNCTIONS ----------------------------

-- The datatype VOTERID defines the voters.
datatype VOTERID    = v1 | v2 | v3

-- The datatype CANDIDATES defines the candidates.
datatype CANDIDATES = c1 | c2 | c3

-- The datatype SERIALS defines the serial numbers.
datatype SERIALS    = s1 | s2 | s3

-- The sets below define the set of all candidates, voter ids and serial numbers.
candidates = {c1, c2, c3}
voters     = {v1, v2, v3}
serials    = {s1, s2, s3}

-- The number of voters
NumOfMaxPossVotes = card(voters)

-- The set of all possible votes that a candidate can get
PossVotes = {0..NumOfMaxPossVotes}

----------------------------------------------------------------------------------------
-- CHANNELS ----------------------------------------------------------------------------

-- The channel openElection and closeElection represent the election official's opening
-- and closing the election, which is also synchronised by all voters.
channel openElection, closeElection
-- The channel choose is that the voter decides whom to vote for.
channel choose   : VOTERID.CANDIDATES
-- The channel mark represents a marking action for the chosen candidate.
channel mark     : VOTERID.SERIALS.CANDIDATES
-- The channel cast represents a voter casting a ballot form by dropping it into a ballot
-- box.
channel cast     : VOTERID.SERIALS.CANDIDATES
-- The channel collectform represents the election official giving a ballot to a voter.
channel collectform: VOTERID.SERIALS
-- The channel auth represents the election official authenticating a voter.
channel auth     : VOTERID
-- channel leaveBooth and enterBooth is the voters' getting in and out the booth.
channel leaveBooth, enterBooth: VOTERID
-- The channel withdraw represents a ballot being withdrawn by the counter.
channel withdraw: SERIALS.CANDIDATES
-- The channel empty represents an empty ballot box.
channel empty
-- The channel total represents the number of votes that each voter possesses.
channel total    : CANDIDATES.PossVotes
```

```
--------------------------------------------------------------------------------
-- PROCESSES -------------------------------------------------------------------

--------------------------------------------------------------------------------
-- The VOTER process receives an empty ballot form and casts the ballot form.

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 the voters' entering and leaving the
-- booth. Hence, there will be no two voters in the booth at the same time.

BOOTH = enterBooth?id -> leaveBooth!id -> BOOTH
--------------------------------------------------------------------------------
-- The process ELECOFFICIAL is the election official, who opens and closes the election.
-- He authenticates the voters and distributes the ballot forms to the authenticated
-- voters with unique serial numbers.

ELECOFFICIAL      = openElection -> OFFICIAL(voters,serials)

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 in which the voters drop their votes. Then, it
-- parses all the votes to the COUNTER with the event withdraw until the box is 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 operates as a counter, which withdraws the votes from the ballot
-- box and counts them. If the box is empty, it just returns the total votes for each
-- candidates.

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 (CVS).

SYSTEM  =   (
```

```
                (
                    (
                        VOTERS[ aVTR|| aOFF]ELECOFFICIAL
                    )
                    [Union({aVTR, aOFF})||aBTH]BOOTH
                )
                [Union({aVTR, aOFF, aBTH})||aBOX]BOX
            )
            [Union({aVTR, aOFF, aBTH, aBOX})||aCNT]COUNTER
-----------------------------------------------------------------------------------------
-- Process Alphabets
-----------------------------------------------------------------------------------------
-- The sets below defines the alphabets of the corresponding processes
-- such as aVTR is the alphabet of VOTERS.

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|}
-- Sigma is the alphabet of the process SYSTEM.
Sigma = Union({aVTR, aOFF, aBTH, aBOX, aCNT})

-----------------------------------------------------------------------------------------
-- SANITY CHECKS -------------------------------------------------------------------------

-----------------------------------------------------------------------------------------
-- 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 cast
-- events, 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 -----------------------------------------------------------------------

-- The system, which the observer can see, should be limited. Hence, SYSTEM1 is the
-- abstracted system in which we hide mark and collectform events.
```

```
SYSTEM1 = SYSTEM \ {|mark, collectform|}

-- ABS_SYSTEM is the system where we rename a number of events, which carry confidential
-- information that should be hidden from the observer. Hence, cast events are renamed as
-- envelope events, letting the observer 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 over.

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, showing that the CVS provides anonymity.
-----------------------------------------------------------------------------------------
-- Strong Anonymity --
----------------------
-- We also checked the system against the strong anonymity definition by abstracting
-- choose events to a dummy event and abstracting them back.

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 the strong anonymity is not provided by the
-- CVS. Here is an example trace taken from FDR2 violating the strong anonymity.
-----------------------------------------------------------------------------------------
------- Counter Example -----------------------------------------------------------------
-- choose.v1.c1
-- choose.v1.c1
------------------------------------------ END ------------------------------------------
-----------------------------------------------------------------------------------------
```