A Formal Analysis of the RIES Internet Voting Protocol

Cynthia Maasbommel

February 23, 2007

A thesis submitted in partial fulfillment of
the requirements for the degree

Master of Science
in
Computer Science

Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081 A
1081 HV Amsterdam
The Netherlands

vrije Universiteit amsterdam

Supervisor:

prof. dr. W. Fokkink
Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081A
1081 HV Amsterdam

Second Reader:

dr. F. van Raamsdonk
Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081A
1081 HV Amsterdam

Abstract

This paper describes the RIES protocol, which specifies a voting mechanism for use on the Internet. It was last used in the Second Chambre Election 2006. The paper offers a formal specification of the protocol. Here the muCRL language is used to formalize the protocol. The specification is proven correct concerning several properties in different situations using the finite state model checker from the CADP toolset. The properties to be verified are expressed in regular alternation-free mu-calculus and concern anonymity, verifiability and some feature validations.

Preface

Democracy. Something taken for granted today. The issue of Internet voting makes the discussion alive again. Is it really safe to trust Internet with one of the most critical anchors of our society?

Some think it is. Careful steps are taken all over the world to introduce Internet voting. The Netherlands are among the countries pioneering this form of democracy. As recent as November 2006 an experiment has taken place. Most discussion centers around security issues.

Formal Methods and Software Verification being my Master's specialization I could formally analyse this new way of voting. This is appealing to me as it offers the opportunity to practice things I have learned to such a significant subject.

Peter Quatfass, a friend of mine that I know from scuba diving, led me to this subject of electronic voting. For his job he occupies himself with everything that concerns elections, including elections on the Internet. He brought the following experiments on e-voting under my attention.

In the Netherlands two experiments have taken place to make Internet voting available to Dutch people abroad. The first concerns a protocol devised by LogicaCMG for the European Parliament election in June 2004. The second regards the so called RIES protocol designed by Piet Maclaine Pont and used during the Second Chambre election in November 2006. The fact that documentation of the latter was publicly available simplifies the analysis as being a higher level specification than the Java code offered by LogicaCMG. This makes the RIES protocol a suitable and interesting subject for my thesis. This feature has contributed in choosing the RIES protocol to verify. Moreover, this protocol does not only exist on paper; it has been used several times already. It is also very attractive due to some of its unique features as its transparency and above all, its verifiability. As it is the first protocol to offer this last feature it is all the more interesting to study it.

This thesis provides a description of the RIES protocol, which is subsequently formally specified. This specification is used to show that the protocol satisfies certain properties.

I would like to thank Wan Fokkink for being an enthusiastic supervisor that helped me reviewing both textual and formal products. Textual input has also been given by Piet Maclaine Pont and Engelbert Hubbers. The first has provided me with additional information on the protocol and the latter has shared his experience in testing and knowledge of RIES. I would also like to thank Hugo Jonker for lively discussions about this protocol that helped in keeping a critical focus. A warm word of thanks also goes to Radu Mateescu, designer of the CADP toolset, for helping me utilize the possibilities of the tool more fully. Finally, I would like to acknowledge Femke van Raamsdonk as she agreed to be second reader of my thesis.

Contents

Preface **3**

1 Introduction **5**

2 Remote Voting in the Netherlands **6**

3 Problem statement **9**
3.1 Verifiability . . . 9
3.2 Validity . . . 10
3.3 Anonymity . . . 10

4 The RIES protocol **10**
4.1 In general . . . 11
4.2 Design Requirements for RIES . . . 12
4.3 Experience with actual elections . . . 15
4.3.1 Local water board elections 2004 and 2005 . . . 15
4.3.2 Second Chambre Election 2006 . . . 15
4.3.3 Future applications . . . 18
4.4 In detail . . . 19
4.4.1 Organisation . . . 19
4.4.2 RIES in phases . . . 22

5 Informal review **35**
5.1 Strengths . . . 35
5.2 Weaknesses . . . 38

6 Formal verification **44**
6.1 Approach . . . 44
6.2 Model checking . . . 45
6.2.1 Model checking defined . . . 46
6.2.2 The value of . . . 47
6.2.3 The problem of . . . 47
6.2.4 A compromise . . . 49
6.3 Software and Other Machinery . . . 49
6.3.1 Formal specification . . . 51
6.3.2 Formal properties . . . 52

7 Algebraic Specification **53**
7.1 Data types . . . 54
7.2 Process behaviour . . . 55
7.2.1 General scenario . . . 55
7.2.2 Attacker Scenarios . . . 73

8 Properties and Verification results **88**
8.1 Macros . . . 90
8.2 Results . . . 93
8.2.1 All scenarios . . . 94
8.2.2 noFraud scenario S_0 . . . 99
8.2.3 sendVote scenario S_1 . . . 102
8.2.4 removeReceived scenario S_2 . . . 104
8.2.5 removeValid scenario S_3 . . . 106
8.2.6 addValid scenario S_4 . . . 107
8.2.7 makeInvalid scenario S_5 . . . 108
8.2.8 changeCount $scenarioS_6$ . . . 109
8.2.9 addCount scenario S_7 . . . 110
8.2.10 changeRef scenario S_8 . . . 110
8.2.11 Anonymity scenarios . . . 111

9 Conclusion **115**

References **117**

A μCRL specification of the RIES protocol **I**

1 Introduction

The future of Internet voting is near. Internet voting is an election system that uses encryption to allow a voter to transmit his secure and secret ballot over the Internet. It seems like the introduction of it in the political field is inevitable. Experiments with voting using the Internet are already taking place worldwide: the Netherlands, Belgium, France, England and the United States are only a few. In Switzerland it is even possible to cast votes via SMS text message since 2005. (*"First e-ballot..."*, 2005) The Netherlands are a little more reserved using these new techniques in democracy, although automated voting and voting in a voting office of choice has been possible for a while. The next step would be Internet voting. The advantages seem compelling. Internet voting promises the possibility of a convenient, efficient and secure facility for recording and tallying votes in an election. Voters can decide where and when to vote. It should result in a higher turn-out and lower costs for ballot distribution. (*"VVD pleit..."*, 2005) Moreover, voting over Internet seems a natural step since using the Internet becomes common practice for the Dutch population (Esch-Bussemakers, Geers, Maclaine Pont & Vink, 2002)

Enough reason for the VVD to suggest introducing Internet voting nationally for the Second Chambre elections in 2007. (*"VVD pleit..."*, 2005) Instead, the government decided to offer this possibility only to Dutch people abroad in the early national election of November, 2006. (*"Eerste internetstemmen..."*, 2006)

The protocol used was RIES (Rijnland Internet Election System), the subject of this thesis. It differs from other election systems as it is the first to offer a possibility to verify what happened to the votes. Something lacking in the traditionally held Italian Parliament election April 2006, during which alleged fraud by Berlusconi's coalition was committed. The sharp drop in the number of blank ballots caused suspicion and led to suggest that a computer program has been used to manipulate the electoral results by altering a large number of blank ballot papers to votes for Berlusconi's center-right coalition. (*"BBC NEWS..."*, 2006) This stresses the significance of verified voting, something advocated strongly by professor Dill, founder of Verified Voting Foundation and Verified Voting Organization. These organisations strive for reliable and publicly verifiable elections in the United States. (*"Verified Voting..."*, n.d.) This is exactly what the RIES protocol tries to achieve, although not by a voter verified paper trail as suggested by Prof. Dill in using voting machines. Verifiability not only increases the trust in and acceptance of the system but also enables detecting fraud. In this regard RIES claims to be safe: fraud can always be detected. Indeed, one of the properties examined here is detection of internal fraud. External fraud like hack attacks are not considered as the most vulnerable aspect of any cryptosystem are the people involved. This thesis will not worry about the precise advantages and disadvantages of Internet election systems either. Instead it will analyse one promising protocol that might be applied in the future more extensively.

RIES has already been examined in informal ways. A security and usability analysis is performed, as well an external penetration test. The design of the cryptography and server and network used is also examined. Even a large-scale user test has been carried out and although very valuable, these informal approaches miss the advantages of a formal analysis. A formal analysis can automatically encompass every possible program execution, which the protocol should be able to face. Together with its mathematical basis this enables

verifying certain system properties and theoretically prove the absence of errors. (Wang, Hidvey, Bailey & Whinston, 2000) Such a formal analysis has not been carried out yet. In this thesis I will.

My attention will be focused on some particular properties that the protocol is designed to meet: anonymity, verifiability that makes detection possible and validity to a certain level. In this thesis I hope to verify these properties. To do so I need to carefully study the RIES protocol and formalize it together with these properties to make reasoning about them possible. This involves model checking which serves as a means to verify certain properties.

My knowledge of this protocol is based on publicly available information and on personal communication with both the designer Mr. Maclaine Pont and a member of the penetration team Mr. Hubbers. I have contacted them to complement information given in (Hubbers, Jacobs & Pieters, 2005) and (Hubbers and Jacobs, 2004). These papers form the basis of my description of RIES. According to Mr. Maclaine Pont (designer of RIES) these articles are not sufficiently detailed to account for all RIES' properties. As these articles are general introductions to the subject there is a chance that certain information or inferences from this information made by me do not resemble the actual workings of the protocol. This risk is to be accepted since more detailed documentation is not available to me at this moment.

I will begin my thesis with briefly describing some of the projects in the Netherlands concerning electronic voting, the RIES protocol being one of them. It is followed by the problem statement of this thesis concerning the RIES protocol, which is studied in detail in the consecutive section. This protocol is formally analysed and the approach taken is described in the section Formal Verification, together with an explanation of the value of this approach and the software used. Subsequently, the algebraic specifications are given in a separate section, that also describes the scenarios designed to test certain properties. It is followed by a section describing the properties and the corresponding verification results. This section also includes some statistics about state space size and verification times. The thesis ends with a conclusion.

2 Remote Voting in the Netherlands

The Netherlands is one of the front runners concerning electronic voting. This does not only include Internet voting, but voting using voting machines instead of paper ballots as well. This automated form of voting has been introduced as early as 1982 (Niemoller, 2004), although not nationally. In fact, the capital Amsterdam was one of the last municipalities to replace traditional voting with this more modern one. However, this seems unnecessary and rather pointless when Internet voting will be introduced within a few years. Both forms of voting are topics of heavy discussions.

The action group 'Wij Vertrouwen Stemcomputers Niet' caused a lot of commotion by being able to buy two Nedap voting computers from a Dutch municipality. This enabled them to find out how the machine works, to rewrite software to modify results and even to find a way to compromise voter secrecy. (*"English -..."*, 2006)

Especially the latter resulted in the minister to decertify a different type of voting machine, a touch-screen based system produced by the former state press SDU as the

problem was much worse with this. The main reason was that it would endanger the order on election day. This affected about 10% of the voter population. (*"Electronic voting..."*, 2006)

Although experiments with Internet voting and a so called voter pass (kiezerspas) are certainly not encouraged by this opponents group either the government persists in studying these possibilities.

To make experiments possible some aspects from the constitution need to be revised. A special experiments law Kiezen op Afstand has been created, which can be inspected at www.kiezenuithetbuitenland.nl (*"Kiezen uit..."*, n.d.).

As early as 1997 an electronic voting specific change has already been made (*"Electronic voting..."*, 2006):

- Phased voting allowed on voting devices: first one chooses a party, then a candidate. Consequence: the complete list of candidates no longer needs to be visible on the devices.
- Voting devices may be used for two elections at the same time, in case of combined elections (e.g., parliament + referendum, or provincial + local elections).

This enabled the start of a project in 1999, designed to investigate whether the voting process could be made less location dependent. This project is called KOA, which stands for *Kiezen op Afstand* and has taken off under the supervision of minister Van Boxtel. (*"Internetverkiezing in..."*, 2004)

The project is divided in two types of experiments (*"Kiezen uit..."*, n.d.):

1. Voting at an arbitrary polling-station; voters are allowed to cast a vote in a polling-station of their choice in their municipality.
2. Internet Voting; eligible Dutch voters in foreign countries are enabled to cast a vote via the Internet.

Both experiments have taken place during the European Parliament election in June 2004.

Voting at a polling-station of the voter's choice was experimented with by the municipalities Assen, Deventer, Heerlen and Nieuwegein. The first impression is positive; from the voters' point of view, as well as from the municipalities' point of view. (*"Kiezen op Afstand"*, n.d.) Voting in a polling-station of the voter's choice has been repeated in March 2006 with the municipal election. (*"Kiezen uit..."*, n.d.)

The experiment in Internet voting in 2004 made use of a protocol designed by Logica CMG. More about this experiment will be said later.

As the evaluations have been positively evaluated, both experiments have been repeated during the Second Chambre 2006 Election. (*"Kiezen uit..."*, n.d.) A different protocol is used in Internet voting however. In 2006 an existing protocol has been adjusted for this purpose. Originally, the protocol was designed by parties that were asked to by 'Hoogheemraadschap van Rijnland', one of the Dutch local authorities on water management. (Hubbers, Jacobs & Pieters, 2005) This protocol has been used several times earlier for local elections via the Internet. As this protocol is the topic of this essay I will come back to this in detail later.

The difference between the Internet voting protocol used in 2004 and the protocol deployed in 2006 becomes apparent in the following three matters (*"Kiezen uit..."*, n.d.). In 2006 :

1. voters are able to check what happened to their vote;
2. voting with the telephone is not possible anymore;
3. the authentication procedure is more user friendly.

One last difference is procedural. Instead of allowing voting during ten days like in 2004, in 2006 this has been decreased to five.

At this moment a survey has started to evaluate the two last experiments in both voting from a polling-station of choice and using the Internet. The results are yet unknown.

From the two types Internet voting has caused the most discussion, although this medium offers a wealth of opportunities for making access to information, communication and offering service better and also more effective and efficient. (*"Kiezen uit..."*, n.d.) The Internet is already very important in offering information and services.

A lot of discussion about Internet voting is centered around privacy and security. The *Kieswet* says that elections should be secret (in particular, article 53 from the constitution (*"Wij vertrouwen..."*, n.d.) and that everyone should be able to vote freely, but is that possible through the Internet? Moreover, a big and critical project like practicing democracy via the Internet will attract hackers.

However, during the Internet voting project in June, 2004 no irregularities of this sort have been reported. The project was held in light of the European Parliament election. The Netherlands was the only country to offer the facility to vote via the Internet or phone during the European Parliament elections while staying in foreign countries. In this period 5351 voters have made use of this new facility to vote. (*"Ministerie van..."*, n.d.) This number is one third of the total amount of votes coming from people staying in foreign countries. They were obliged to state their wish to vote via the Internet or telephone several months before the start of the election. Over old-fashioned mail they were then sent a login, a password and unique codes. (*"Kiezen uit..."*, n.d.)

The source code used in this experiment was developed by Logica CMG and has been made public using the license GNU General Public License. This means that everyone interested can download, study and use this software. Especially, it makes it possible to check if this software really does what it should be doing. The software can be downloaded from the exchange platform www.OSOSS.nl (*"Open Standaarden..."*, n.d.), short for 'Open Standaarden en Open Source Software voor de overheid'. The most important motivation for making the code public was transparency. A government that offers the public a new facility to vote, stimulates the use of it by making the functioning of the software used public.

The integrity of the software for download can be verified using an MD5 checksum. A checksum is a unique number that is computed from the file. With this number you can verify that the integrity of the software file is intact. This file contains all Java classes written for online voting. Classes that are part of the general technology of Logica CMG are not open source. This means that it is only possible to inspect part of the code and not to compile and run it. In general, one can still understand the workings of the system.

It has been argued that one does not want to include the source code of the operating system either. (*"Open Standaarden..."*, n.d.) Although it does not affect the transparency of the system much, it does make detailed analysis impossible. As such it does not lend itself for a formal verification, in contrast to the RIES protocol.

3 Problem statement

A voting protocol is designed to meet particular properties. I have selected some that the RIES protocol should satisfy. The properties I have chosen to formalize and verify can roughly be grouped into three categories, being Verifiability, Validity and Anonymity. In this section I will further specify what is meant by these general concepts.

3.1 Verifiability

The term verifiability has been stretched to encompass several design features. First, everyone (a voter and a blank voter or abstainer) should be able to check what happened with his vote. In the RIES protocol this means that a voter can determine if his vote has been received and is valid. If it is not, the voter should be able to detect this as being a result of his own actions or as fraud. Second, a voter should also be able to determine which candidate his vote has been assigned to, which can also be a result of his own actions or fraud. Third, everyone interested should be able to do a recount given the received votes and the reference table. The result can be that the counts agree or that the recount differs which indicates fraud. Another important check involves SURFnet. SURFnet can check if its set of received votes matches the one published by TTPI.

The category Verifiability is quite broad and therefore classed into three parts; determinability, accurateness and detectability.

The first part concerns the guarantee that it can be verified what has happened to a vote, a set of votes or the count. Properties in this class are flagged with a D for determinability.

The second part guarantees the correctness of the protocol and these determinability properties, meaning that certain actions follow each other only for that particular item and not another. Every such property has been stated using two separate formulas, both indicated by Accurateness (A). The first is intended to verify that an OK action for a particular item (i.e. a vote) always follows a communication of that particular item. The second assures that an OK action for a particular item is only given for that item and not another as the former formula does not prevent that another OK action also follows the communication of that item. To clarify this distinction, I have borrowed some terminology from the statistics domain to further divide the category. The term hit (HIT) is used for every property that contains an eventually always constraint. With a correct rejection (CR) a property is meant that states that if something does not hold some statement is not made. The correct rejections are mainly intended to serve certain verifiability HIT properties. They guarantee that the verification is correct; false verifications do not occur. This will become more clear when presenting the properties in regular alternation-free μ-calculus formulas.

A special interest lies in showing that particular instances of fraud can be detected.

The last sub-category is thus classified under detectability (DC). This term is meant to cover the properties that are used to verify that it is possible to detect fraud in checking votes and the count. It should eventually be noticed when fraud is committed. The fraudulous actions concerned are ones that actively manipulate the results, others are not addressed. An example of the latter would be publishing incorrect information to spread confusion and force a re-election. These are all HIT formulas that are complemented by Correct Rejections. For every general form of fraud a Correct Rejection property is defined to assure that fraud is not signalled if it is not committed either. These properties are also classed under Accurateness (A).

3.2 Validity

I use the term validity here to include several concepts that are needed to demonstrate that the protocol implements certain functionalities correctly. The properties authentication and count a vote only once belong to this category. Furthermore, it is obvious that the protocol should make it impossible to present a vote in the received votes table that has not been cast. In the same sense such a vote cannot be marked valid either. In fact, it is not possible to validate a vote if it does not occur in the received votes table. In total this category contains nine properties.

3.3 Anonymity

Anonymity is undoubtedly the most crucial feature. It is also one of the most controversial properties to check as there are continuing debates about the question if this can be achieved on the Internet.

Anonymity is the state of being not identifiable within a set of subjects, the anonymity set. (Pfitzmann and Hansen, 2006) Here, this definition is used to mean that a potential voter cannot be uniquely characterized. To enable anonymity of a potential voter, there always has to be a set of voters with potentially the same attributes. (Pfitzmann and Hansen, 2006) The attributes concerned here are the sending and receiving of particular information. In the protocol all voters receive a ballot, claim information from a website and are capable of sending (casting) a vote. Most important of which is the casting of a vote. Anonymity ensures that a user may use a resource or service without disclosing the users identity. The requirements for anonymity provide protection of the user identity. Anonymity is not intended to protect the subject identity. (*"International Organization..."*, 1999) The difference here lies in guaranteeing the secrecy of the identity of the user as a natural person opposed to guaranteeing the secrecy of the service request. Here, this means that anonymity does not claim anything about secrecy of anything else than the user's identity, like communication. Anonymity requires that other users are unable to determine the identity of a user bound to a subject or operation, in this case a vote. (*"International Organization..."*, 1999)

4 The RIES protocol

After having discussed the features that will be proven to be satisfied by RIES, this protocol is elaborated on in more detail in this section.

4.1 In general

RIES stands for Rijnland Internet Election System. It is an online voting system that was developed by the 'Hoogheemraadschap van Rijnland', one of the Dutch local authorities on water management, in order to increase the number of people actually casting their vote and simultaneously to decrease the cost of such an election. The system is based upon the ideas from the master's thesis of Herman Robers, written in 1998 at TU Delft, as a former student of the designer of the RIES system Piet Maclaine Pont. Named CHOOSE, it was first used in 2000 during elections for the Delft Student Board. The basics of this system has been used to design RIES, but differs from it on one important point. CHOOSE uses chips to do cryptographic calculations. A smart card for each participant was needed to take care of performing the critical operations. For Rijnland and De Dommel the use of smart cards was not desirable at all, due to the high costs for the introduction of such a technology to their voters. Furthermore, the constraint was posed to not having voters install new hardware. Therefore, RIES was adjusted to use a different system for key management, authentication and cryptography. The cryptographic computations that the smart card would have performed are now done by the client's computer using JavaScript in the voter's browser. The JavaScript file can be automatically downloaded to each voter as part of his welcome screen to the elections. The script ensures that the voter can enter secret codes without disclosing them to any other system or party, including the election server. Within these scripts there are routines available to convert his choice for a candidate into the proper cryptographic vote. This choice is the only personal and sensitive part, which will only be transmitted in an encrypted form by the voter's browser to the election server on the Internet. (Hubbers, Jacobs & Pieters, 2005) Moreover, without disclosing their proof of vote the voters can check if their vote has been counted, with which this protocol distinguishes itself from many other voting protocols. This transparency makes the election recountable and verifiable. The latter is an important and praised feature: On June 23rd, 2006, it has been assigned the Public Service Award by the European Commission and the United Nations. The initiative should be an example to other authorities. The jury states that the system enhances the responsibility duty of the public sector and the transparency offered. (*"Het Waterschapshuis..."*, 2006)

Use of hashes and encryptions makes it possible for each voter to actually check afterwards if his vote has contributed appropriately to the final outcome. This check can be made by hand using information to be downloaded from the website (www.rijnlandkiest.nl for the water boards election and www.kiezenuithetbuitenland.nl for the Second Chambre Election) or using a special website that leads the voter through the process. For both elections it concerns a sub-site 'stemcontrole'.

Rijnland started their development by asking a third party to check the security risks involved with setting up an Internet election voting system. The Dutch company TNO carried out this preparatory research and came to the following conclusions: (Hubbers, Jacobs & Pieters, 2005)

- Many risks involved in voting by Internet are not higher than in voting by ordinary mail.
- There are some risks typical to Internet settings like DDOS attacks and Trojan horses on client machines. However, there exist procedural measures for the specific

situation of Internet voting.

- None of the currently available systems can be applied to Rijnland's election.

TNO was not the only institution that did research on the RIES protocol. Another is Cryptomathic (Denmark), that has examined the cryptographical design. TNO Human Factors looked at the usability for the voter. Madison Gurka checked the server and network design and security. Radboud University Nijmegen performed external penetration tests, led by Bart Jacobs and finally, Burger@Overheid (ICTU in Den Haag) carried out a large-scale user test. (*"Herverkiezing Rijnland..."*, 2005)

4.2 Design Requirements for RIES

RIES has been designed while keeping a few particular issues in mind. Some of them have surely contributed to the positive evaluations and are discussed briefly in reviewing the protocol in section 5. The design requirements include the following (*"Rijnland kiest..."*, n.d.):

Authentication Only legitimate voters can participate in the election. In this sense the goal is not to verify someone's identity, but to determine someone's eligibility and choice (*"Simpel,..."*, 2004);

Convenience and simplicity Legitimate voters should be able to cast a vote with minimal demands on devices, experience and knowledge;

Vote secrecy No one should be able to find out what someone has voted, in other words, the anonymity of a voter should be guaranteed;

Non-coercibility Voters should not be able to prove whom they voted for;

Count a vote only once A voter's choice should be counted only once;

Integrity One should not be able to change votes without being detected;

Accurateness All votes should be correctly saved and determined;

Reliability The system should work correctly at all times, even during several disturbances;

Verifiability It should be possible to ascertain if the votes have been correctly counted;

Possibilities to check afterwards All data concerning the election should be saved in a reliable and provable authenticate way;

Flexibility The system should be usable in other contexts as well;

Certifiability The systems have to be testable against essential criteria;

Transparency Eligible voters should be able to generally understand the voting process;

Cost effectiveness Election systems should be economically interesting in purchase, use and efficiency, both for eligible voters and organisators of elections.

The first feature, authentication, is an important concept in elections. As such, it is one of the properties I have chosen to verify.

The demand for convenience and simplicity, the second feature, is hard to prove in a formal way. However, in examining the RIES protocol informally it will follow that it has been designed to fulfill this need.

One of the most fundamental concepts in a democracy is vote secrecy. In general, vote secrecy comes in two levels. One is absolute anonymity, meaning that the contents of an observed vote cannot be determined and the other is somewhat weaker: inability to link the identity of the voter to a certain vote. The first depends on the strength of the hash, something that goes to far for this analysis. The second is as important as the first and will be verified in a formal way when considering anonymity.

In the RIES protocol both are striven for by using cryptography. This will be discussed in examining the protocol, without proving algorithms which is not possible. Secrecy is proven to be undecidable (Meadows, 2003). This fact is only emphasized by the realization that in the domain of cryptography, the race will never end between the code makers and code breakers (Denning, 1999).

Moreover, family voting (casting a vote under group pressure) is always a present danger. For these reasons vote secrecy can never be guaranteed 100 percent and is therefore not verified here.

Concerning family voting modern authentication methods do not help much either. The government is working on a PKI (Public Key Infrastructure) for authentication and in 2007 a new identity card is to be used, which could be employed for elections too. (*"Ministerie van..."*, n.d.) Even better would be an identification system using biometrics like iris scans or fingerprints. This would make it impossible to acquire a personal key illegally as it is embedded in the body. It could however promote horrifying scenes as seen in movies.

Besides family voting opponents of e-voting often mention voter coercion in this context: (forced) provable vote sale. Resistance to the latter is provided if voters are not able to prove whom they voted for. This enforces the next feature on the list, non-coercibility.

The RIES protocol has been designed in such a way to diminish this provability. The best proof would be something the voter could not have offered without casting a vote. The technical vote would thus not suffice as it could be computed manually. The acknowledgement returned by the server upon receival of the vote does however. It cannot be computed by the voter as a secret key is needed. However, a buyer would have no means of deciphering this acknowledgement and cannot be sure that the acknowledgement concerns a vote for the intended candidate. Moreover, the coerced voter could undo his actions by re voting; something allowed by the protocol. If it concerns a vote for

another candidate both votes will be judged invalid and hence not counted. This prevents both proving a vote and successfully forcing off a vote.

Of course, a coercer would succeed in his intention if he would accompany the voter until the election is closed. This resembles the before mentioned concept of family voting. In discussing strengths and weaknesses I will come back to this.

The RIES protocol has processed the property of non-coercibility in a nice way while keeping verification intact. In most cases a clear trade off is observed between non-coercibility and verifiability (Mote, 2001). The price paid here is a not so user-friendly way of verifying.

Contrasting voter secrecy, the property to count a voter's intent only once can be formalized and is one of the features I have selected to verify. It is grouped under Validity. One vote should clearly be counted only once but the RIES protocol complicates things by making it possible to cast several times. It has been agreed that in such a situation the vote via Internet takes precedence over the postal vote. As the specification does not take the postal votes into account this part is not addressed. However, it can also occur that the voter uses Internet to cast more than once. In the case of identical votes the vote is reduced to one, a procedure imitated in the specification. If it concerns votes for different candidates all are judged invalid.

The feature integrity is classed under the verifiability concept and is verified by formalizing the ability to detect fraud.

The demand to save the cast votes and determine its meaning correctly (termed accurateness here) is also part of the set of properties I will verify. This feature is expressed in the fact that the voter can affirm if his vote has been received, if it is valid and for which candidate it is counted. Moreover, voters are able to detect fraud with saving and determining the vote.

Reliability is a very general concept and as such cannot be easily formalized. For some conditions under which reliability should prevail testing is preferred, like a server shut down. However, I will cover part of this feature by showing that some properties will hold under several fraudulous actions and that others will fail, but nevertheless be detected.

The next feature in the list of design constraints concerns verifiability in the sense that it should be possible to determine if the votes have been counted correctly. The RIES protocol makes this possible for everyone that is interested in doing so. This makes the protocol different from most other online voting protocols and an interesting feature to verify, classed under verifiability.

The same goes for the possibility to check afterwards. All data needed should be saved in a reliable and provable authenticate way. In the RIES protocol this not only regards leaving the voter keys, MDC key and reference tables deposited at a notary before the election starts, but also guaranteeing the authenticity of the reference table and the table containing the received votes. The latter involves hashes and it goes too far to prove its correctness.

The tally result can be verified using public information in the website; something that is formalized and verified under the concept of verifiability.

The server entity compares its received votes against the table published by TTPI, the party that operates on them. Another part of the verifiability constraint is that every voter should be capable of checking if and how his vote has been counted in the result.

The RIES protocol is designed in such a way that a voter can check if his vote has been received, if it has been marked valid and if his vote has been assigned to the correct candidate. This too is formalized and verified.

Flexibility and cost effectiveness are other concepts that do not lend themselves for formalizing. The first only plays a role in water board elections and not in the Second Chambre Election 2006 as only one way of voting is to be supported by the RIES system.

Making the system as transparent as possible helps in enlarging its certifiability, although it can never be satisfied due to undecidability of secrecy for example. The same mechanism should make it possible for eligible voters to understand the general idea of the voting process. This is assumed to be sufficiently captured by publishing explanations and instructions.

4.3 Experience with actual elections

The protocol has been applied in local water board elections in 2004 and during the Second Chambre elections in 2006. In the waterboard election paper votes were transformed into a form equal to the Internet votes to make it possible for the RIES protocol to process them in the same way as the Internet votes. It was not necessary to state the preferred way of voting beforehand, in 2006 this was obliged. Paper votes were no longer processed by the RIES protocol, but by a vote office assigned to this job (*"Kiezen uit..."*, n.d.).

4.3.1 Local water board elections 2004 and 2005

At the time of writing this, the system has been used several times. *Hoogheemraadschap Rijnland* used it in the period of September 25 until October 6 2004 for election of the waterboard. It was the first time that voters were offered the opportunity to vote via the Internet. More than 70,000 people made use of this facility. A re-election took place in the period from April 12 until April 21 2005. Another water management authority *De Dommel* deployed it from November 6 until November 19 2004. Over 800,000 eligible voters could cast their vote via the Internet. These public water board elections have been the largest formal Internet elections on the planet so far. (*"RIES for..."*, 2004)

Some figures of these elections are listed in table 1, taken from (Meijer, 2005).

	Rijnland 2004	Dommel 2004	Rijnland 2005
Eligible voters	1,363,787	878,118	127,778
Number of seats	36	37	1
Postal votes	160,647	120,201	13,390
Internet votes	72,235	50,196	6,490

Table 1: Tally statistics for RIES-based Waterboard elections

All times the system functioned without meaningful problems.

4.3.2 Second Chambre Election 2006

The most recent experiment in Internet Voting took place during the Second Chambre Election 2006 using RIES. A few modifications were needed for this new application. For

example, the list system is now supported (the water boards use a person system), the user interface has been modified and postal votes will no longer be processed by the RIES system as in 2004, but by appointed voting offices. (*"Kiezen uit..."*, n.d.)

Using the Internet, eligible voters abroad could cast a vote starting Saturday, November 18th 2006 from 7.30 until 21.00 at Wednesday November 22nd 2006. (*"Kiezen uit..."*, n.d.)

It is argued that time is a crucial factor for voters abroad and can therefore be a bottle-neck that would result in abstaining from voters. The experiments with Internet voting should reveal if this form of casting a vote would reduce this.

Joost Beukers, divisional head of KOA at the ministry of Home Affairs, expected that 63% of the eligible voters in foreign countries would cast a vote via the Internet using RIES (*"Online verkiezingen"*, n.d.). It turned out to be about 57,76 %. This number does not indisputably enable drawing conclusions about the successfulness of the experiment. I will try to explain this here.

The expectation that introducing Internet voting would increase turn-out concerns eligible voters that normally abstain from voting. The main reason that this experiment cannot form an answer to this question is that the eligible voters abroad do not receive a ballot automatically. In contrast with the procedure followed in the Netherlands, eligible voters abroad have to signal that they are willing to participate in elections held in the Netherlands (*"Kiezen uit..."*, n.d.) This implicates that it concerns people that are willing to vote and register themselves as such. The turn-out is expressed in a percentage of these registered willing-to-vote people that actually cast a vote in some way. Naturally, these numbers are therefore always high.

The total number of eligible voters living abroad in different years is unknown, but needed to draw fair conclusions about the fluctuation in turn-out that may or may not be caused by Internet voting. The ones that do not register themselves as eligible voters do not receive a request as this practice has proven to fail as their true location is not always known. It did not yield much response either and together with high costs it was chosen to replace this approach by demanding active registration by eligible voters in foreign countries. (*"Kiezen uit..."*, n.d.)

Therefore, the percentages of those people that actually cast a vote will not fluctuate much between elections held in a traditional way and by modern ones, using telephones and Internet. The following facts and figures will make this clear.

The possibility for people with a Dutch nationality to vote from abroad has been offered since 1983. (*"Parlement en..."*, n.d.) In earlier elections voting from foreign countries was made possible by postal mail. For example, in the Second Chambre Election 2003 a total of 25.644 eligible voters in foreign countries registered to vote (*"Kiezen uit..."*, n.d.). A percentage of 88,2 % actually cast a vote during this re-election needed after the Balkenende cabinet ended prematurely. (*"Documenten -..."*, n.d.) This represents 22.618 voters casting a vote by postal mail. The expectation was that this number would show a significant increase when the possibility to vote via Internet is added in 2006. It turned out to be 29.818, equalling 86,92 %. Although the absolute number of 29.818 voters is higher than 22.618 voters, the percentages obviously show something different. The difference between 88,2 % and 86,92 % is not very large and even shows a decline instead of an increase. When considering the turn-out for the previous years 2002 and 1998 being 86,8 % and 82,8 % respectively (*"Documenten -..."*, n.d.), it seems fair to say that based on the

percentages in 2006, 2002 and 1998 the turn-out remains relatively steady and Internet Voting did not cause a bigger turn-out abroad in 2006.

A total of 34.305 eligible voters that live or work (either permanently or temporary) in foreign countries had registered themselves as eligible voters for the Parliament Elections 2006 in the Netherlands. From this amount a group of 14.377 voters (41,91 %) preferred the possibility to vote by postal mail, authorize someone else to vote or using a *stempas* while staying in the Netherlands. The remaining 19.929 voters signed up for Internet voting, which is 58,09 % of the total. (*"Kiezen uit..."*, n.d.)

In the end, it turned out that from those two groups the percentage that did actually cast a vote was highest among the Internet voters. From the 14.377 registered for mail voting 10.003 voters cast a vote, equaling 69,58 %.

The turn-out among the registered Internet voters was quite higher: 92 % of these voters actually made use of the facility (*"Ministerie van..."*, n.d.), which is the number what was made public as to prove its success. This equals 19.815 voters, which is 57,76 % of all eligible voters in foreign countries.

Concerning these Internet voters the corresponding statistics show that some voters have cast a vote more than once, resulting in 430 invalid votes. An invalid vote is defined here as a blank vote, a duplicate vote, a vote of which the key has been used to cast a vote for two or more different candidates (a false duplicate), a vote that identifies the voter, a vote using an unknown voter identifier or candidate identifier, an expired vote and a vote coming from a non-existent ballot. In the elections for the second Chambre, 31 blank votes have been received, as well as 316 duplicate and false duplicates and 83 expired votes. (*"RIES KOA ..."*, n.d.)

Table 2 presents these statistics found at (*"RIES KOA ..."*, n.d.).

Gross number used codes	19929
-Number eligible casters	19929
Gross number received votes	20245
-Number received Internet votes	20245
Gross number invalid votes	430
-Number blank votes	31
-Number duplicates and false duplicates	316
-Number votes identifying voter	0
-Number votes unknown encrypted voter id	0
-Number votes unknown encrypted candidate id	0
-Number expired votes	83
-Number non-existent ballot votes	0
Number valid votes	19815
Number voters casting invalid votes	114

Table 2: Tally statistics Internet voting Second Chambre 2006

After the election the Internet voters were given the opportunity to perform a check on the processing of the individual votes. Only 0,5 % actually did. As the strength from the RIES protocol comes from this possibility to check this result is rather disappointing. (Hubbers, 2007)

Using more traditional ways of voting by paper and voting machine the turn-out in the Netherlands during the same election was comparable to the ones achieved abroad: 80,35 % of the eligible voters had cast a vote. This represents 9.854.998 voters of the total of 12.264.503 eligible voters. This number has not been that high since 1989. (*"Uitslag van..."*, 2006) The turn-out in Second Chambre elections fluctuates over the years. The course of this turn-out can be inspected in table 3, in which the turn-out abroad using postal mail and Internet is also presented.

The first column until 2006 origins from the CBS (*Verkiezingsuitslag van...*, 2006).

Turn-out in %	At home	Abroad Postal mail	Internet
1967	94,9	-not in use-	
1971	79,1	-not in use-	
1972	83,5	-not in use-	
1977	88,0	-not in use-	
1981	87,0	-not in use-	
1982	81,0	-not in use-	
1986	85,8	-no data-	
1989	80,3	-no data-	
1994	78,8	-no data-	
1998	73,3	82,8	-not in use-
2002	79,1	86,8	-not in use-
2003	80,0	88,2	-not in use-
2006	80,35	86,92*	
		29,16	57,76

Table 3: Turnout in % for Second Chambre elections

The turn-out at home is expressed in percentages of the total amount of eligible voters. The turn-out abroad is expressed in percentages of the total amount of eligible voters that have registered themselves as such. Note that the results for Abroad of 2006* are additionally split up into Postal Mail and Internet votes as presented in the subsequent row.

The use of voting machines was introduced in 1982 (Niemoller, 2004) and the figures above do not indicate a clear increase in voting. Thus, fluctuation in turn-out seems to be natural and hardly due to Internet voting or other modern ways of voting.

4.3.3 Future applications

RIES has not only been developed for the two water boards Rijnland and De Dommel, but will be made available for all 26 Dutch water boards for their 2008 elections. This will concern about twelve million eligible voters (*"online verkiezingen"*, n.d.). Use of RIES for elections, referenda or opinion surveys at reasonable terms can also be discussed. (*"RIES for..."*, 2004) For now, the government has been granted to use the system three times.

4.4 In detail

The RIES protocol can be understood best by dividing it into three phases: the preparing actions needed before the election, the casting of votes during the election and the processing of all votes after the election. Here, it has been chosen to use the terms initial, election and tally part to denote these phases, as done by Piet Maclaine Pont (*"RIES facts..."*, 2004). In short, the phases are defined by:

1. The initial part, in which a reference file for every eligible voter is calculated containing all its possible votes. Parameters used are first generated, as explained later.
2. The election part, in which voters can cast their vote either by postal mail or by Internet. Using the latter medium the vote will be encrypted first before sending it over the Internet.
3. The tally part, in which the final results will be calculated and information is made available to allow all voters and independent parties to validate the results.

They are explained more fully in the following sections, but first a distinction is made between the several entities that exist in this protocol. It is meant as an introduction to the more detailed discussion of the phases.

4.4.1 Organisation

The organisation of the RIES protocol becomes apparent when discussing the several roles and tasks.

The roles that can be identified in the RIES protocol are:

- the central controller *TTPI*
- p voters $V_1, \ldots, V_p$
- JavaScript engine
- the server administrator *SURFnet*
- vote office

From this list, the main party in the actual elections is the central controller *TTPI*. Here, TTP stands for Trusted Third Party which in cryptography is an entity which facilitates interactions between two parties who both trust the third party; they use this trust to secure their own interactions. The designer Piet Maclaine Pont of Mullpon v.o.f. and the developer Arnout Hannink of Magic Choice b.v. together form TTPI for TTP Internetstemmen (Maclaine Pont, personal communication, September 2005).

The second entity in this list is the voter. All the voter needs is an Internet connection and an Internet browser that supports SSL and JavaScript. At the website www.internetstemmen.nl the voter is to follow the instructions there to choose a particular candidate that is translated to an encrypted vote by the JavaScript engine at the local client. Next, it is sent to SURFnet, that maintains the network servers needed to

collect all votes and computes the end result. An acknowledgement that serves as a proof of vote in case of irregularities is sent back to the voter. This acknowledgement is calculated by a server remotely maintained by TTPI, but physically present at SURFnet (personal communication, Maclaine Pont, September 2005). The end result is finally sent to the vote office, which publishes the end result and some other files needed to verify the outcome.

After this global understanding of the activity of each entity in this protocol it is convenient to list the successive steps taken by each, starting with TTPI.

For all roles the corresponding activities can be divided in those before the election and those after the election. It is responsible for taking care of the preparation and execution of the Internet election and is responsible for calculating the result afterwards.

TTPI
Before the election:

- Generate unique keys for each eligible voter;
- Generate one election identifier;
- Generate candidate identifiers;
- Generate synchronization key;
- Generate acknowledgement key;
- Compute the reference table that contains every possible vote for every voter;
- Publish this pre-election table together with an MD5 hash;
- Hand over the unique voter keys and reference file to the vote office;
- Destroy the keys.

After the election:

- Convert paper votes to technical votes;
- Receive technical vote file from SURFnet;
- Compute hash of technical vote file;
- Match vote with reference table:
  - Check validity;
  - Remove double votes but one;
  - Keep a log file indicating the reason for invalidity.
- Count votes per candidate;
- Send result to vote office.

- Publish halves of all acknowledgements.

Voter *i*
Before the election:

- Receive ballot containing personal and general codes needed to cast a vote.

During the election:

- Send key to the JavaScript engine by logging in;
- Choose candidate *j*;
- Receive acknowledgement.

After the election:

- Verify vote;
- Verify outcome.

JavaScript engine :
During the election:

- Calculate technical vote using the key and candidate received from voter *i*;
- Send technical vote to SURFnet via SSL.

SURFnet :
During the election:

- Receive technical vote from JavaScript engine via SSL;
- Strip technical vote from network address and time;
- Send back an acknowledgement [1];
- Save technical vote.

After the election:

- Send technical vote file to TTPI;
- Publish technical vote file.

[1]The acknowledgement is really being computed by the RIES application that runs at a server from SURFnet and is controlled by TTPI by a VPN connection. Here, it is classed under SURFnet as the acknowledgement is sent by means of one of their servers. However, the computation of the acknowledgement is not part of the knowledge of SURFnet.

Vote office
Before the election:

- Receive voter keys from TTPI;
- Assign voter keys to voters and distribute them.

After the election:

- Receive result from TTPI;
- Make the results public.

Most of these tasks are made visible in the figures presented in the next section. Together, the figures represent the RIES protocol in a slightly modified form to correspond to some of the assumptions made in formalizing the protocol. The formal specification will be discussed in the section Algebraic Specification. First, the RIES protocol is presented as it is, discussing the phases.

4.4.2 RIES in phases

Initial part In preparing the election several parties are active of which TTPI is the most important one. This party is concerned with generating several keys to ensure among other things that critical information is safely encrypted. It creates the following keys and attributes before the start of the election (*"Functionaliteiten RIES..."*, 2004):

- A unique voter key K_i for every voter V_i;
- One receipt acknowledgement key;
- One synchronisation key;
- One unique election identifier EI;
- A unique candidate identifier CI_j for every candidate C_j.

Details of the generation of keys and identities are not available. Therefore, it is assumed that the keys and identities are determined using a injective function, meaning every value generated this way is unique.

The third key generated is the synchronization key. How the RIES protocol uses this key to synchronize actions is not publicly known. The usage of the remaining attributes is discussed next.

A unique 3-DES (Data Encryption Standard) key is generated for each entitled voter, with the latter coming from a list of voters composed by the authorities. The total number of voters is published to be sure that no voters are added or deleted during the procedure. (Hubbers, Jacobs & Pieters, 2005)

Presumably, this key is made up of (*"Simpel,..."*, 2004)

- Non-confidential information:
  - participation group
    Every voter belongs to a certain district and category. This is coded into one string called the participation group.
- Confidential information:
  - vote code to enable 'anonymous 'identification
  - password

As the key K_i is based on DES (Data Encryption Standard) it can be used for both encryption and decryption (Hubbers and Jacobs, 2004). This is called symmetry.

Here, the unique voter key is used to encrypt the vote to make authentication possible as well as to keep the contents secret. Therefore, a vote consists of two parts serving these two goals. The first part enables establishing the voter's authenticity without violating his anonymity. The second part contains the vote. (Hubbers, Jacobs & Pieters, 2005) The authentication part depends on two attributes: the personal key to ensure eligibility of the caster and an election identifier. An election identifier is assumed to be assigned to one particular election. Encrypting this election identifier with the key will result in a unique value that is associated with one particular voter. In the rest of the paper this result will be called the encrypted voter identity (EVI).

An election identifier should be generated for each election to avoid confusion and fraud with former elections. For an analogous reason a separate encrypted voter identity should be used in every election. (Salomonson, 2004) The same argument goes for candidate identifiers that make up the second part of the vote. Every real-life candidate listed is associated with a candidate identifier, that is to be generated by TTPI. The personal key is used again to encrypt the candidate identifier, which represents the candidate of the voter's choice. This encrypted part will be referred to as encrypted candidate identity (ECI) from here.

The encryption used here is called a MAC. MAC stands for Message Authentication Code, which is a special hash function that uses a secret key K_i for calculating the characteristic value. A MAC algorithm accepts as input this key and an arbitrary-length message to be authenticated, and outputs a MAC value. (*"Message Authentication..."*, n.d.) Without the key it is fairly impossible to calculate this value and to extract the original input from this value. (Hubbers and Jacobs, 2004) The value protects a message's integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. (*"Message Authentication..."*, n.d.)

Note that two values are generated here: one for the encrypted voter identity and one for the encrypted candidate identity. In the first case, the function takes the unique secret key K_i and the election identifier as arguments. In a somewhat abstract form this results in:

$$EVI = MAC_{K_i}(EI)$$

In the second case, the candidate identifier is used instead of the election identifier to represent the choice the voter has made. In a similar abstract form this comes down to:

$$ECI = MAC_{K_i}(CI_j)$$

Together an arbitrary vote for the candidate associated with the candidate identifier CI_j using key_i and the election identifier EI looks as follows:

$$TV_{ij} = MAC_{K_i}(EI), MAC_{K_i}(CI_j)$$

This is the so called Technical Vote, that is used in a later stage for authentication in a somewhat modified form. The authentication in RIES is quite simple. The idea is to authenticate the vote that is received instead of the voter. To make this possible, the unique key is used to generate all possible votes a particular voter can cast using this key before the start of the election. (Hubbers, Jacobs & Pieters, 2005) Every single vote coming from this collection is eligible and hence coming from an eligible voter, assuming that the key is kept secret. This ensures that every vote received can be judged authentic or unauthentic easily, simply by checking its appearance in the file.

In the same way its validity can be determined. To detect the casting of several differing votes using the same key, one should check if the received votes are all originated from this same collection. If they are, all votes from this set are not counted. Identical votes from this set appearing more than once are all judged invalid but one. (*"Kiezen uit..."*, n.d.)

Such a set is generated for all voters, resulting in a set of collections ordered by EVI. It will be made public before the start of the election. This file thus contains all possible votes for every voter in MAC encrypted form. (Hubbers, Jacobs & Pieters, 2005) To prevent the casting of these encrypted votes the two parts of every vote are protected by a one-way MDC-2 hash that can be seen as a fingerprint.

A random vote for the candidate associated with the candidate identifier CI_j using key_i and the election identifier EI has the following abstract structure:

$$MDC2(MAC_{K_i}(EI)), MDC2(MAC_{K_i}(CI_j))$$

It prevents exposing the votes in a form usable for voting, that is a form accepted by the server receiving all cast votes. The hash is needed to keep these technical votes secret before and during the elections. Without it, there would be a clear mapping from each technical vote to a certain candidate and everyone could then pick one and send it straight to the vote server via openSSL. (personal communication, Hubbers, June 2005) Because of this MDC-2 hash no one but the eligible voter can know the pure technical vote, which can only be computed using the unique secret key of a voter. On the contrary, everyone can compute the MDC-2 hash and therefore everyone can compute the result.

MDC stands for Modification Detection Code, which is a one-way hash. A one-way hash can be seen as a fingerprint. A hash function H is a transformation that takes a variable-size input M and returns a unique fixed-size string, which is called the hash value h (that is, $h = H(m)$). A hash function H is said to be one-way if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that $H(x) = h$. The important characteristic is that given M it is easy to compute $H(M) = h$, while given h it is hard to find an M satisfying $H(M) = h$ and given M it is hard to find another M' satisfying the relation $H(M) = H(M')$. (Hubbers

and Jacobs, 2004) MDC-2 is developed by IBM that outputs 128-bits, based on the DES algorithm. Nowadays it is implemented in the freely available *openSSL*. (Hubbers and Jacobs, 2004) This hash function is a keyless one and can be calculated by everyone as such. This makes it possible to recalculate the MDC values of the received votes that will be published after the election is closed. It is not necessary but it will make sure that the MDC values that were already published in the reference table are correct.

For every hashed vote a mapping to the corresponding candidate exists. This makes it possible to find out for which candidate a particular vote is intended. This information is also made public before the start of the election. Together, the information is called the pre-election or reference table and is used to calculate and verify the results. This would make a file containing 50 million values if 1 million voters and 50 candidates are concerned (so called potential votes). This table gets published on the Internet in zip format, together with a one way MD5 hash that is to guarantee that the file has not been manipulated. MD stands for Message Digest and is an algorithm designed by Rivest that should result in a unique value when computed over a given file. (Hubbers and Jacobs, 2004)

For a system with n voters and m candidates the reference table in table 4 is generated, sorted by EVI.

$MDC2(MAC_{K_1}(EI))$	$MDC2(MAC_{K_1}(CI_1))$	C_1
	$MDC2(MAC_{K_1}(CI_2))$	C_2
	$\vdots$	$\vdots$
	$MDC2(MAC_{K_1}(CI_m))$	C_m
$MDC2(MAC_{K_2}(EI))$	$MDC2(MAC_{K_2}(CI_1))$	C_1
	$MDC2(MAC_{K_2}(CI_2))$	C_2
	$\vdots$	$\vdots$
	$MDC2(MAC_{K_2}(CI_m))$	C_m
$\vdots$	$\vdots$	$\vdots$
$MDC2(MAC_{K_n}(EI))$	$MDC2(MAC_{K_n}(CI_1))$	C_1
	$MDC2(MAC_{K_n}(CI_2))$	C_2
	$\vdots$	$\vdots$
	$MDC2(MAC_{K_n}(CI_m))$	C_m

Table 4: Reference table for n voters and m candidates

The MAC and MDC hashes make it impossible to inverse the process of encryption. (*"Kiezen uit..."*, n.d.) Moreover, the algorithms used should ensure the generation of a unique hash values for every vote. (*"Kiezen uit..."*, n.d.)

After the election all received votes are made public in both its encrypted form and its hashed form. (Hubbers, Jacobs & Pieters, 2005) With the hashed form everyone interested can recalculate the results by looking up the corresponding candidate for every valid vote in the published file and every individual voter can verify if and how their vote has been included in the result. In particular, he can verify if his vote has been assigned to the correct candidate.

After generation of the keys and the reference table the keys are handed over to the vote office. (*"Herverkiezing Rijnland"*, 2005) Subsequently, the used keys K_i are expected to be destroyed by TTPI as they are no longer needed. Presumably, the vote office will assign the keys to all eligible voters. A few weeks before the start of the election the keys together with the candidates are to be distributed to the eligible voters. This distribution takes the form of old-fashioned postal mail and therefore is subject to old-fashioned interception of this mail.

To discuss the second key listed here, the receipt acknowledgement key is used to acknowledge that a vote has been received. It should serve as a proof for both the voting application and the voter in the case an external third party is needed to solve disputes.

From personal communication with Mr. Maclaine Pont (September 2005) it follows that this one key is used to compute again one MAC hash over the technical vote received. One part is sent back to the voter over SSL as the other will be saved at the server side and published in a file by TTPI after the election.

An example taken from a test stage performed in June 2004 by Hubbers shows how presumably a technical vote and its corresponding voter's half of the receipt acknowledgement is represented. (personal communication, Hubbers, September 2005)

Three votes and their corresponding acknowledgements are shown in table 5. Presumably the acknowledgement is split in two to ensure that the voter has cast his vote in time; during the election.

Technical vote	Receipt acknowledgement
3ae7b369646cd197 25267fa64258c8cc	FBAAE73C
830f08d28a5288bc 94f280d425c8749e	7036465B
96d2b67f1d314c73 494c4cd3b7493648	F15C3345

Table 5: Example of technical votes and receipt acknowledgements

The acknowledgement is computed using a single secret key known to TTPI and an external third party only. It is generated by TTPI and present in the RIES application that runs at a SURFnet server. As the acknowledgement can only be computed by the application in reaction to the receival of a vote it should serve a prove that a particular vote has been received, in time. The acknowledgement cannot be used as a proof of vote as it does not reveal the contents and is computed with a secret key known to TTPI only. (Personal communication, Maclaine Pont, September 2005)

During irregularities the voter could show his half and the server side could show theirs. It should be the case that both parts can be recalculated with the receipt acknowledgement key by the third party that serves as an independent control party. If the results correspond, everything is alright. If not, one party is discredited. (Personal communication, Maclaine Pont, September 2005)

This should prevent false claims of the voter about the sending of a particular vote and undetected adding and removing of votes at the server side. In the latter case the policy followed is to declare the election not valid, even if it concerns just one voter. (Personal communication, Maclaine Pont, September 2005)

The application is protected from outsiders as it is cryptographic hardware that computes the acknowledgement. (personal communication, Maclaine Pont, September 2005) Such hardware is specifically tasked with handling cryptographic processes. It stores a Local Master Key (LMK) inside a tamper-resistant container and the LMK never appears in the host computer. All other cryptographic keys used by a business application are encrypted by the LMK inside the hardware device and can then be safely stored inside the host computer. The advantage here is that third parties (in this case, SURFnet) cannot gain access to it. Software, on the other hand, can in theory be hacked. (*"Prime factors"*, n.d.)

As the application is designed by TTPI, this party could alter the RIES application during the election without the need for hacking. The application at SURFnet's server is controlled by TTPI by a VPN connection. However, TTPI cannot modify the software in reaction to the content of cast votes because the votes are not accessible to TTPI at that moment. Moreover, TTPI cannot declare a vote invalid by recalculating the acknowledgement using a different key and claiming that the acknowledgement held by the voter is not valid, because such fraud is eventually detected by a third party that is brought into action to solve disputes like this. (personal communication, Maclaine Pont, September 2005)

Based on this general information provided by Mr. Maclaine Pont it seems well designed. However, some critical remarks are made in section 5.2 concerning this use of acknowledgements.

This section ends with a general overview of several tasks and entities in the initial phase as described here. It is presented in figure 1. Some matters are simplified in correspondence to the algebraic specification given later. For instance, the vote office is omitted in the formal specification of the RIES protocol. Publishing information on the Internet is symbolised here by a cloud and a bin is used for expressing destroyal of the keys.

The election part In this stage the voter can choose to cast a vote via the Internet or postal mail. Here, only Internet voting is considered.

To cast a vote a voter should login at www.internetstemmen.nl (water board election) or at www.internetstembureau.nl (Second Chamber 2006) and needs to follow the instructions on his ballot that he received by postal mail. If he did not receive one, he can request for a duplicate ballot. This will make the original one invalid. The ballot also contains his personal codes. Using this code the voter gains access to the voting service and the same code will be used to encrypt the vote. (*"Kiezen uit..."*, n.d.)

As soon as the voter i inserts this information on his ballot into the webpage his personal key K_i becomes available to the JavaScript interpreter of his browser. This script is automatically downloaded when opening the webpage and is responsible for calculating the MAC values as well as keeping a status record of the voting process for this particular voter. The latter enables that a voter can interrupt and resume the voting procedure at any time. This information is saved only at the client side, which is made possible by the use of frames in combination with JavaScript (*"Kiezen uit..."*, n.d.).

The reason to use JavaScript instead of chips is that it is available in every web browser, even if it is an old version. In this way, a voter only needs a standard web browser to be

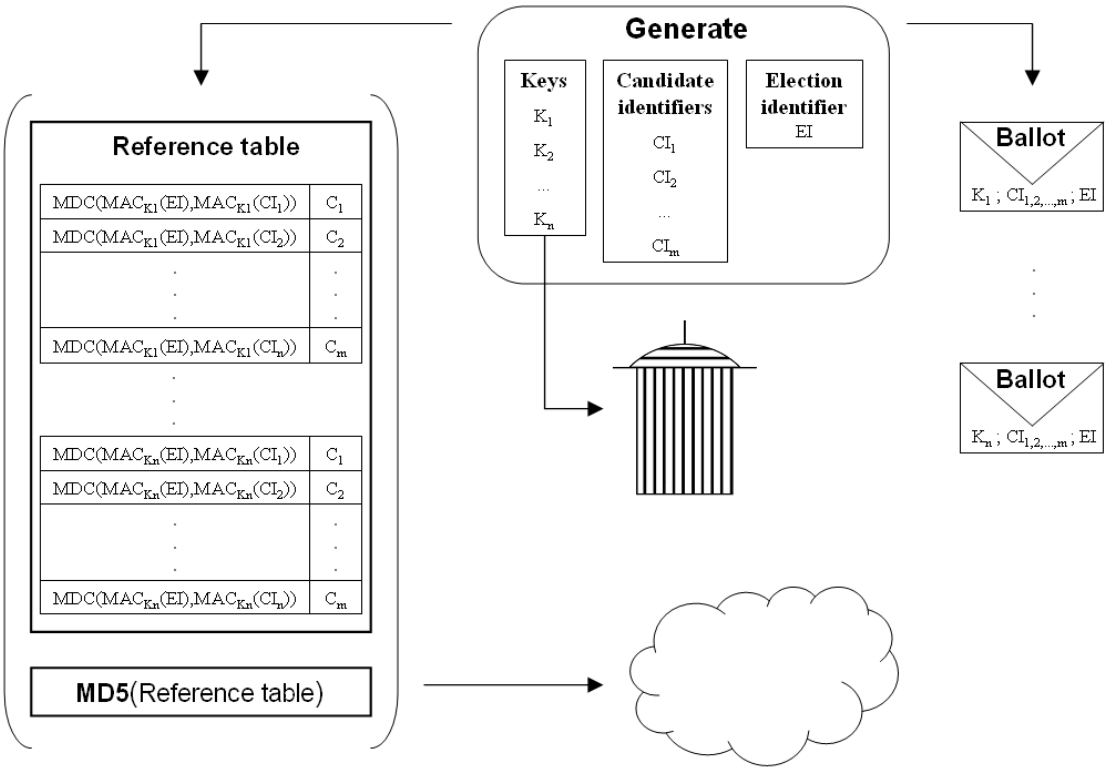

Figure 1: Activities in initial phase

able to vote. Moreover, the use of JavaScript makes it possible for every voter to verify the source code of the software as it is contained in the HTML pages. (*"Kiezen uit..."*, n.d.)

Of course, the web browser must have access to the Internet and must enable JavaScript. For security reasons scripts are often disabled, but for the purpose of Internet voting it should be activated. Moreover, the browser should support the 128-bit format that results from encrypting.

After entering his password the voter needs to select his favourite candidate j. The script will now calculate two values; $MAC_{K_i}(EI)$ and $MAC_{K_i}(C_j)$. These two together form the so-called technical vote. These two parts are then sent to the vote server via an SSL-connection. Important to note here is that K_i is not sent, but something is that can only be constructed using K_i.

SSL is short for Secure Sockets Layer, a cryptographic protocol for transmitting private documents via the Internet. SSL works by using a private key to encrypt data that's transferred over the SSL connection. Both Netscape Navigator and Internet Explorer support SSL, and many websites use the protocol to obtain confidential user information, such as credit card numbers. By convention, URLs that require an SSL connection start with https instead of http. Together with a symbol depicting a lock this would convince

the user of its authenticity. (*"What is..."*, n.d.)

SSL provides endpoint authentication and communications privacy over the Internet using cryptography. In typical use, only the server is authenticated (i.e. its identity is ensured) while the client remains unauthenticated. The protocols allow client/server applications to communicate in a way designed to prevent eavesdropping, tampering, and message forgery. (*"SSL"*, n.d.) Thus, SSL makes it hard to intercept and read messages and conversations.

All communication from and to every voter is handled by an independent task at the server side, administered by SURFnet. (personal communication, Maclaine Pont, September 2005) It receives all the votes cast and sends back a receipt acknowledgement to the voter. To compute this acknowledgement the software at the server first removes date, time and network addresses from the vote. The vote is then passed to the cryptographic hardware designed by TTPI that runs in this server. This will compute another MAC value of the technical vote with a single secret key known only to TTPI and an external third party. Subsequently, this value is divided in two pieces of which one is returned to SURFnet's software and sent to the voter and one is published by TTPI after the election is closed. (personal communication, Piet Maclaine Pont, September 2005) This division of tasks ensures that TTPI will not have the combination of a network address (IP) and a technical vote to its disposal. SURFnet, however, could save this information but this illegal action should be noticed when reviewing the server. This network address would not form a direct link to the voter as it is free to vote from an arbitrary location. (personal communication, Maclaine Pont, September 2005)

After this removal of potentially critical information the received vote is added to the so-called post-election table or received vote table as they are sent by the voter, meaning they are saved as MAC values. At this moment no verification of validity is performed. This will be performed in the third phase. (*"Kiezen uit..."*, n.d.)

The system also allows for the acceptance of multiple vote attempts from the same voter in the same election. These multiple vote attempts could happen in case of network disturbances or errors, PC or browser mistakes, user mistakes or by the fact that a specific voter casts his vote through different systems or the same system for the same election, to be sure it will arrive. Technically, there is no limit on the number of votes per voter. Of course, only one vote will be counted and in the case the voter decided to vote via Internet as well as via mail, only the Internet vote will be taken into account (*"RIES for..."*, 2004).

This feature makes it vulnerable to DDOS attacks, as will be explained later in section 5.2. When done voting, it is up to the voter to destroy his ballot to keep his personal key private.

For the voter the voting procedure simply comes down to the following steps: (*"Herverkiezing Rijnland..."*, 2005)

- Go to www.internetstemmen.nl
- A welcome screen appears and at the same time the JavaScript loads into the browser of the voter's computer
- Fill in the personal codes needed to authenticate the voter.

- The virtual ballot appears.
- During the water board election, more information about the candidates can be retrieved here. Also, a program to help the voter make a decision between various candidates is offered.
- Choose a candidate and confirm the choice.
- Send the vote by confirming the password.
- The voting office confirms the receipt of the vote by showing technical information about the cast vote: the MAC values and a hash value that was calculated over one part of the vote using the acknowledgement key.

The activities performed during the election are expressed in figure 2 together with the action of publishing the tally result. This figure is presented in the next section which discusses the tally part.

The tally part When the election is closed, it is no longer possible to cast votes. In this phase the RIES protocol makes it possible to check what has happened to ones' vote. A voter can check if his vote has been received, if it has been received correctly and if it has been found valid. In certain conditions, a falsum to these questions can signal fraud. The server administrator SURFnet can also check for fraud by comparing its own collection of received votes to the collection that is published by TTPI. Moreover, it is possible for everyone to do a recount using the information published by TTPI. Of course this count depends on the correctness of the public information. It is up to the voting office to make the end results public. Assuming its correctness a difference between the count of TTPI and such an individual recount signals fraud in the count process. Let's look at these issues in more detail.

First, SURFnet makes available its own version of the received votes and transfers a copy of them to TTPI to determine the outcome of the election. Then TTPI computes an MD5 hash from the file that contains all technical votes. This hash is needed to prove that TTPI has not changed the file received from the server. (Hubbers, Jacobs & Pieters, 2005)

The outcome is subsequently calculated by computing the MDC-2 hashes of the two parts of the technical vote. As the protocol accepts all incoming votes and explicitly allows to vote more than once, specific rules have been designed to determine which vote is to be counted. These rules can only be carried out after all votes have been received as the validity of a vote could depend on others; a vote can therefore not be verified one by one as they are received.(Hubbers, n.d.)

The counting rules have been made public in the experiments law *Kiezen op Afstand*. A vote is said to be invalid in the following cases (*"Kiezen uit..."*, n.d.) :

1. A blank vote is treated as an invalid one.
2. If identical votes are received, all but one are considered invalid.
   Identical votes are votes composed of the same key, election identifier and candidate identifier.

3. If there are two or more votes containing the same key but different candidates, all these votes are judged invalid.

Moreover, only votes that occur in the reference table are accepted. Here, they are referred to as possible votes or, if they fail to match an entry in the table, invalid votes. This adds a rule to the above:

4. A vote that does not occur in the reference table is not possible and hence invalid.

Considering the above stated rules, the following procedure is carried out. First a check is made if these votes are possible. All possible votes have been created in the reference table and the received votes are checked against it. Only the votes that appear in it are kept. If they do not the corresponding vote is marked invalid. Next, identical votes should be reduced to one vote by declaring all votes but one invalid. The last step in retrieving all valid received votes is to mark all votes invalid containing a voter identity that has been used to cast a vote for two or more different candidates. (Hubbers, n.d.) For all votes declared invalid, the reason for this decision is saved in a file.

The net result is a table containing only the valid votes received. To determine how many votes have been assigned to every candidate the MDC hash of every valid vote is matched against an entry in the reference table. These values are used as an index in the reference table to determine the corresponding candidate. Remember that the reference table only consists of MDC hashes and a mapping to the corresponding candidate. For every vote the count for the corresponding candidate is increased by one.

Since this MDC hash is a key-less hash anyone can compute it and hence anyone can recalculate the result of the elections. To do so, all received votes should be made public. This is done by the voting office. It will publish the results calculated by TTPI. The table containing all received technical votes together with the MDC2 value for each and extra information (i.e. a log file associated with an as invalid marked vote) will be published on the Internet.

Figure 2 shows the publication of all information needed together with the actions performed during the election part.

The count procedure that is performed when it concerns a valid vote is visualized in figure 3. This figure also applies to the check procedure as performed by the voter, which is discussed in the next paragraph.

SURFnet can check if this table coming from TTPI has a one-to-one correspondence to the SURFnet file. (Hubbers, Jacobs & Pieters, 2005) The SURFnet file is available for download as well, making this check accessible for everybody. To check if a particular vote has been processed correctly the same information is needed. Only the individual voter can perform this check. The next section discusses this in more detail.

The check phase The check phase is part of the tally part. The system is designed in such a way that every voter can check if and how his vote has been counted after the election and everyone interested can verify the outcome by a recount. There are offered several possibilities to enable voters and outsiders to verify the outcome of the election. (*"Rijnland kiest"*, n.d.) These can be divided in three categories:

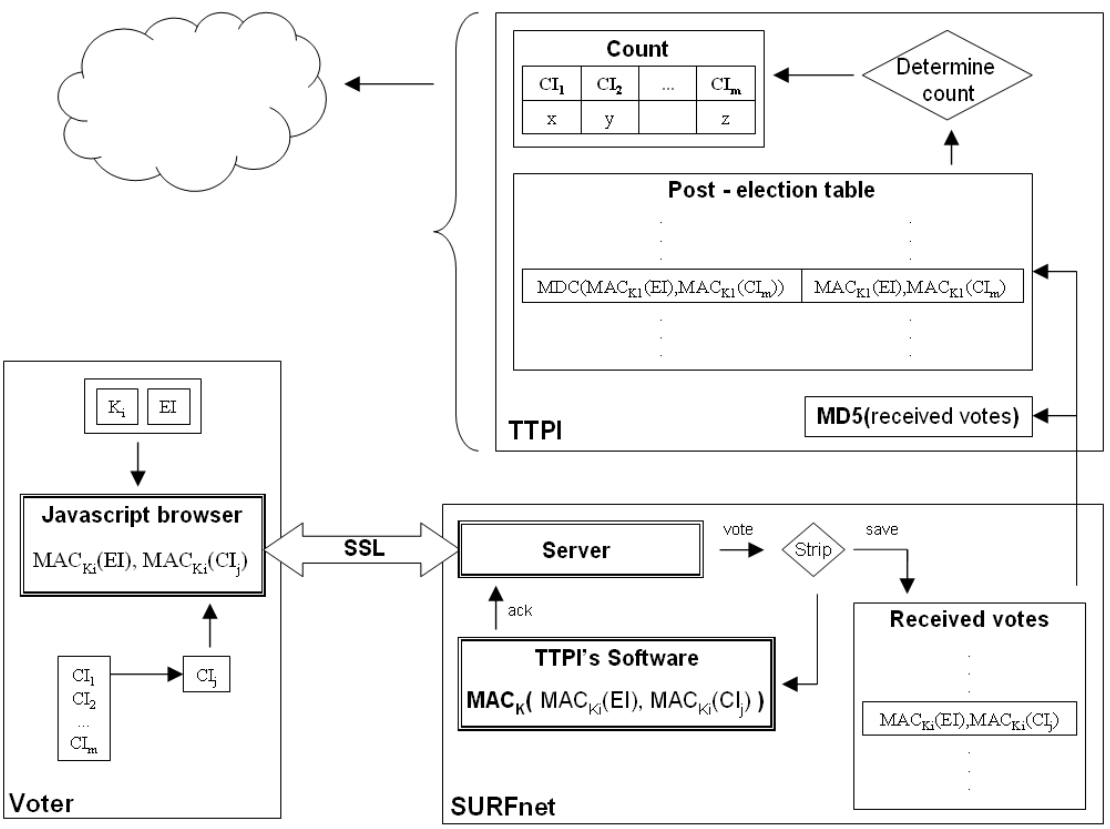

Figure 2: Activities during the election and tally part

- Public check
  All information needed for verification should be made public at www.rijnlandkiest.nl/controlebestanden (Water Board Election). Every file has the form of several zip files ordered by district/category and is published together with its MD5 value to guarantee integrity. These values are also published in advertisements in the newspaper. For the Second Chamber Election 2006 the needed files are published at the location www.kiezenuithetbuitenland.nl/stemcontrole. With this information the check can be done (*Functionaliteiten RIES...*, 2004):
  - by hand, combining all public information and possibly recalculating all hashes concerned. The user needs to download files, which together have a size of 150Gb; quite large for the average end user. (Hubbers, 2007)
  - online using a specific site that guides the user through the check procedure and uses pre-calculated values. The user is only required to enter his technical vote.

- Specific check
  Every individual voter can request personal assistance in a RIES consulting hour

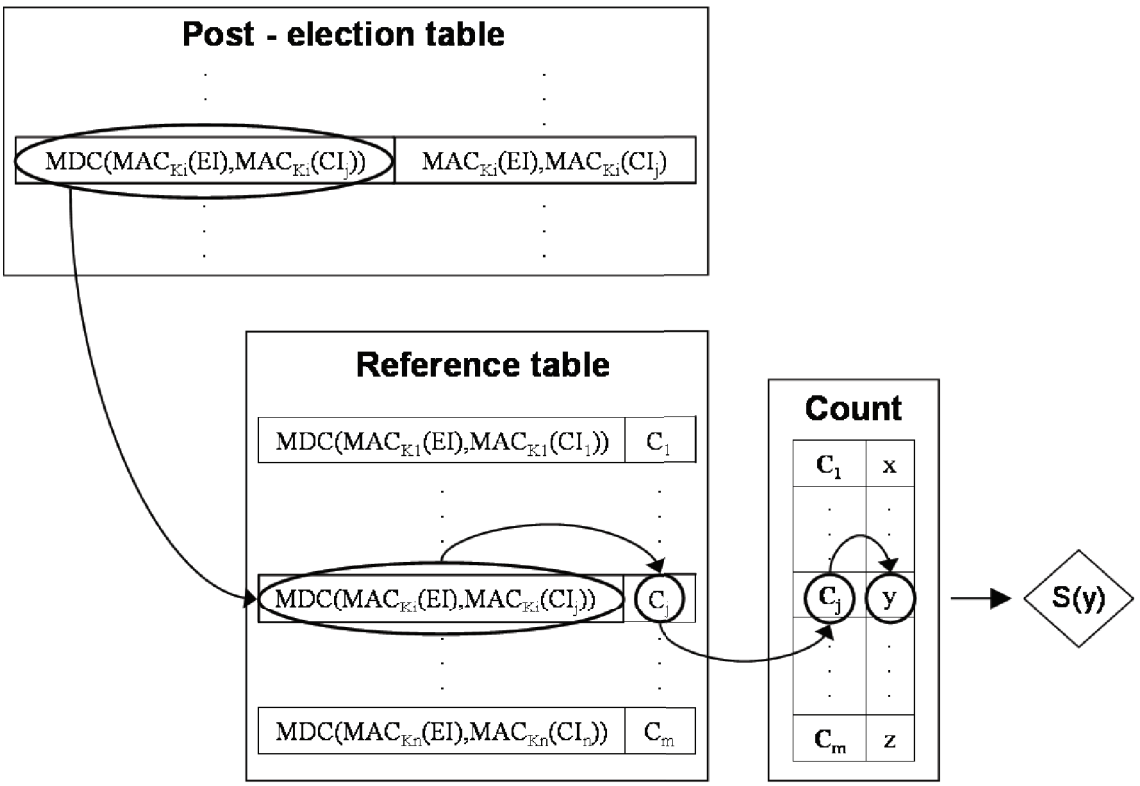

Figure 3: Count procedure

when it concerns Water Board Elections. (*"Rijnland kiest"*, n.d.)

- Independent party check
  An investigation by a independent third party is supported to solve problems and differences.

To be able to verify his personal vote, the voter will need his technical vote. This technical vote is shown on the screen during voting and the voter should save this expression. The voter should enable pop-ups to allow the display of both the technical vote and the acknowledgement. Without these two, the voter is not able to respectively check his individual vote and proof that a particular vote has been received by the server. (*"Kiezen uit..."*, n.d.)

With this technical vote, he can check:

- if the vote has been received by the vote office;
- for which candidate the vote has been counted;
- how the vote has been processed in the outcome.

This check is very important as it could be a great help in detecting fraud. As Piet Maclaine Pont states it (*"Online verkiezingen"*, n.d.):

> 'If just one out of thousand voters performs such a check every big hacker attack will be uncovered. Even if just one voter succeeds in showing that his vote has been tampered with, you can conclude that something fundamental is wrong. As soon as one vote disappears or is incorrectly assigned to a particular candidate, the election should be held again.'

The voter could do a check via the website at www.rijnlandkiest.nl/stemcontrole and www.kiezenuithetbuitenland.nl/stemcontrolesite for respectively the Water Board Election and Second Chamber 2006, but as this check would then be performed by the same party that processed his vote, this would not be reassuring. (Hubbers & Jacobs, 2004) Therefore, another check is made possible: a check by hand.

The first part is easily checked by convincing oneself that the technical vote appears in the received votes table. (Hubbers & Jacobs, 2004) This table lists both the technical vote and the corresponding MDC2 value. (Hubbers, Jacobs & Pieters, 2005) To clarify this action it is captured in figure 4.

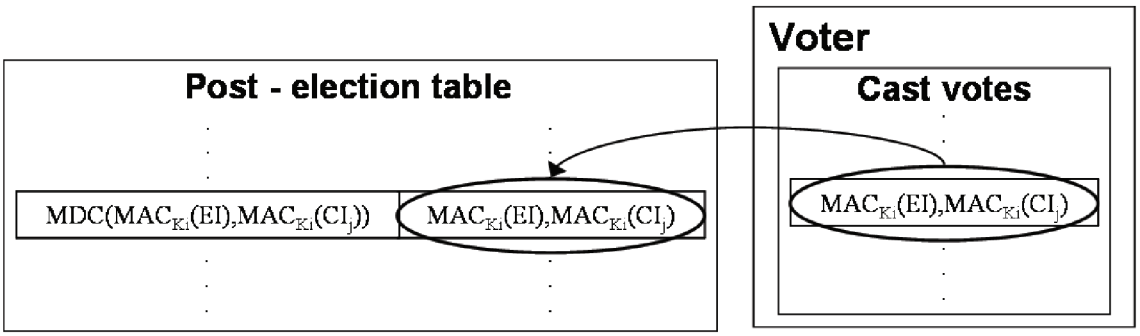

Figure 4: Voter's check on received

The second part is more complicated. The voter will need the MDC2 value of his technical vote as he needs this as an index into the pre-election table to derive the corresponding candidate. The procedure followed here is the same performed by TTPI in computing the count, made visible in figure 3 Recall that the pre-election table contains the MDC2 values of all possible technical votes with a mapping to the candidate. The MDC2 value can be extracted from the post-election table, which contains his technical vote as well as its (supposedly) corresponding MDC2 value. However, this MDC2 value is calculated by TTPI. Therefore, it is to be preferred that the voter calculates this MDC2 value himself given his technical vote, instead of relying on TTPI to present the correct MDC2 values. Such a calculation can easily be performed using some downloadable program or as a service by a third party. (Hubbers, Jacobs & Pieters, 2005)

The third part, determining how the vote has been processed, can be derived from the log entry that is present in case of an invalid vote. It indicates why it was invalid and hence not counted. A voter can then see what happened with a particular vote. (Hubbers, Jacobs & Pieters, 2005)

The second and third part of this check can be performed by anyone, because the hash needed is a key-less hash. Therefore, anyone can compute it, hence anyone can check the result of the elections. Without using any secret key a recount can be performed. All

one needs to do is compute or extract the MDC2 value of all the received technical votes published in the post-election table. These hashes can then be looked up in the pre-election or reference table. In this way one can determine for which candidate each vote should have been counted. Finally, the end result can be compared against the published result. This also means that anyone who has the technical vote at its disposal can determine for which candidate the corresponding voter has voted. (Hubbers, Jacobs & Pieters, 2005)

To check the integrity of the files needed to validate the result these files are published together with their MD5 hashes. These hashes are also published in advertisements to prevent tampering with these hashes and/or files. A voter can check this value by calculating the hash himself and compare the outcome against the hash value that is published. To do so, one can use publicly available MD5-hash software, that can be downloaded from for example www.fastsum.com. (*"Rijnland kiest"*, n.d.)

5 Informal review

To end this informal discussion about RIES a short review is given here.

5.1 Strengths

Strong points offered by the protocol concern aspects like costs, security but also user-friendliness.

This section starts with the measures RIES has taken to make the protocol secure. These features are both procedural and functional, most important of which are separating both knowledge and responsibility of several parties and the protocol's transparency.

Division of power and responsibility The RIES protocol assigned different roles to different parties. In that way not one has both the power and knowledge to affect the election alone. Indirectly, SURFnet is checked by the voter. TTPI has most opportunities and is therefore checked by both SURFnet and the voter.

Irregularities concerning this party are verified with information kept by a notary that is trusted with all keys, ballots, results and other confidential files for years after the election concerned. (Maclaine Pont, 2004)

Transparency

Open source The RIES protocol consists of several software components: a module that performs cryptographic calculations and that generates keys and the reference table, a module that determines the tally result and a module that is used by the voter to cast a vote. The code of the latter is available from the web browser of the voter, but the rest is not open source, although the developers did give an opportunity to examine the code to convince oneself of its correctness. (*"Kiezen uit..."*, n.d.)

Individual and general checks enabled When using voting machines and other alternatives for e-voting (KOA) open source is an important issue. Here this is less relevant, because of the possibility for every voter to check his vote and thus, the result.

Every eligible voter can check how his vote has been processed without having a detailed understanding of the protocol. Everyone interested can recompute the tally result. Both would prove the correctness or incorrectness of the protocol. (Hubbers, n.d.)

Checks are enabled by publication of several files before and after the election. Before the election TTPI publishes the reference table in the form of a .zip file together with its MD5 hash value to guarantee integrity. This hash is also published in advertisements in a newspaper. (*"Rijnland kiest"*, n.d.)

Test stage The ministry of BZK does have access to all documentations and has instructed external experts to perform a research and tests. (*"Kiezen uit..."*, n.d.) These include (Maclaine Pont, 2004):

- TNO, Delft (for an initial feasibility survey)
- A team of Peter Landrocks Cryptomathic, Aarhus Denmark (the cryptographic design)
- TNO Human Factors, Soesterberg (human factor aspects of the voter screens)
- Madison Gurka, Eindhoven (server and network design)
- Bart Jacobs external penetration team of Radboud University Nijmegen (external network and server penetration tests)
- Burger@overheid, ICTU, Den Haag (large-scale end-user validation).

Moreover, the RIES protocol has proven itself by applying it for waterboard elections before utilizing it for the Second Chamber elections.

Risk analysis and counter measures The ministry has made an inventory of the risks and their counter measures. (*"Kiezen uit..."*, n.d.)

Cryptology All sensitive information is protected by encryption. Three hash functions are used: MAC, MDC-2 and MD5. The first is used to encrypt the contents of a vote and to compute the acknowledgement. The second is used to keep the content of votes and public tables secret. The latter is used to guarantee integrity of the public tables.

SSL assures that it is impossible to eavesdrop on packets before they reach the server and therefore to determine the technical votes. (Hubbers, Jacobs & Pieters, 2005)

Acknowledgement The receipt acknowledgement makes it possible for a voter to prove that the vote has been received and the vote has been received in time, while at the same time preventing a proof of vote as the acknowledgement can only be computed with a secret key known to TTPI only. It also helps to ground the claim that fraud is committed by TTPI. When necessary the acknowledgement can be recomputed by an external party. (personal communication, Maclaine Pont, September 2005)

IP information not available to TTPI TTPI's software computes the acknowledgement over the technical vote that has been stripped from IP addresses by SURFnet's software. The acknowledgement is sent back by SURFnet. (personal communication, Maclaine Pont, September 2005)

Authentication without voter identity E-voting systems must authenticate voters, store their identity to avoid voting more than once, and collect their vote. This results in the problem of separating voter identity from vote. RIES has accomplished this while keeping privacy intact.

Supervision A polling station will supervise the course of the election and can decide to cancel the election. Second, the computers processing the votes are protected against unauthorized access by organisational and procedural measures. (*"Kiezen uit..."*, n.d.)

It goes too far to discuss all here. The discussion is continued by listing other features considering strong points.

Very simple and low cost approach of servers Because the computations are done at the client side, a very simple and low cost approach of servers can be taken and allows for the setup of the entire system mainly as a "store-and-forward system". (Maclaine Pont, 2004)

It allows for the acceptance of multiple votes from the same voter in the same election. In the tally part, these entries can be compared and an automated decision can be made about a voters real intended choice without any breach to the required confidentiality of each vote. This creates the possibility for the voter to vote once more by Internet or mail in case he is not sure that the system processed his first attempt correctly, for example in case of a major network disturbance. (Maclaine Pont, 2004)

It allows for the issuing of a replacement election package to a voter, who claims not to have received the original one through the postal mail. To prevent abuse, individual voters and independent parties can validate such changes afterwards. (Maclaine Pont, 2004)

It enables offering multiple election technologies (like postal mail, Internet or GSM) to the same voter, without requiring them to register beforehand. (*"RIES for..."*, 2004)

Low demands on voter All the voter needs is an Internet connection and an Internet browser that supports SSL and JavaScript. (*"RIES - website..."*, n.d.)

Use of Internet browser to load software Most Internet voting systems will depend on election personnel (at poll sites) or voters (at remote locations) to ensure that the hardware and software standards provide the needed levels of security. This task would be

most problematic for remote voters who may possess little or no technical understanding and whose budgets do not allow them to upgrade as necessary. (Mote, 2001) This problem is solved in RIES by using the Internet browser to load software. In this way no additional technical know-how is needed and equality of access is assured at this level.

RIES meets all requirements In general, RIES meets all requirements that should be met by any network oriented election system suitable for formal government elections. (Maclaine Pont, 2004)

5.2 Weaknesses

As every system the RIES protocol has some weaknesses, one more critical than another. Most of them can easily be prevented by changing some aspects of the protocol.

Not all source code is public The processing software on the server is relatively simple and therefore not interesting to publish. However, the counting software is much more complex. (Hubbers & Jacobs, 2004)

Check procedure complicated Verifying a vote is quite complicated as the procedure has been designed to make this possible without disclosing their 'proof of vote'. That could lead to a situation in which voters omit this. Indeed, only 0,5 % did perform a check in the Second Chamber election 2006. (Hubbers, 2007)

Size of check information large The size of the information to be downloaded to enable a more reliable check than the online check is quite large for the average end user. (Hubbers, 2007)

Identity linked to vote The voter's pseudo identity is linked to his vote to enable this check, which does not follow today's procedures.

Relating an ip In theory it is not possible to determine which voter voted for that particular candidate, but if the stripping of network addressing by SURFnet has not taken place correctly, a certain vote can be related to a certain network address (ip). Moreover, SURFnet could could save ip and technical vote together, but this illegal action should be noticed when reviewing the server. Formally, this is not a direct link to voters, but it does give presumptions. (Hubbers, Jacobs & Pieters, 2005)

Key management by voter Another danger concerning the voter has to do with his secret key. His ballot contains the key of that voter. If this key is copied or otherwise available to others, it is possible for someone other than the eligible voter to use this key to cast a vote.

This might result in a valid vote if the eligible voter himself does not cast a vote or has cast the same vote.

The vote cast could also end up to be invalid if the eligible voter has also cast a vote, but for a different candidate. Note that this also invalidates the original vote as a

vote is considered invalid when the same key is used to vote more than once on different candidates. If this would be the interceptor's goal to start with he could suffice with casting two different votes as this would make the original vote invalid no matter what its content.

The interceptor can also prevent that the voter receives his ballot. However, the voter could request a duplicate ballot and votes cast with the original ballot are not taken into account.

If the eligible voter abstained from voting or did not request for a duplicate the interceptor succeeds in his plans, given that the voter does not check public information for receival of votes using his key. It is assumed that such a voter would not be very motivated to do so, in contrast to a voter that did cast a vote.

Thus, it is possible to detect this fraud, but it is necessary in all cases that the voter checks his vote afterwards.

To complicate this kind of fraud one might think of inserting date of birth in the voter key, information which should of course be available to the water management authorities as well.

Postman could violate voter's secrecy The postal channel used to distribute the ballots is not very secure. Note that today credit cards are sent this way as well. He could also save the key to be able to compute the technical votes and can determine the candidate voted for by the voter whose identity is also known to the postman. This violates voter's secrecy.

Technical vote management Everyone having the technical vote of a particular voter at its disposal somehow, can determine for which candidate the corresponding voter has voted. This could be a potential privacy problem. (Hubbers & Jacobs, 2004)

Vote sale/coercion Concerning vote sale and forced votes, the check makes it easier for a voter to prove how he voted but only after the election is closed. The voter is able to prove an actual vote afterwards by making available his unique key. For the corresponding MAC values can be computed using the source code in the JavaScript or some other program downloadable from the Internet and subsequently, the buyer can check if the voter kept his end of the bargain.

Moreover, if he sells his technical vote before the list with received votes has been published, the buyer has a proof that there is indeed voted on a certain candidate. In this particular case, however, the voter can trick the buyer by voting twice on a different candidate making the vote invalid and thus useless to the buyer before selling the technical vote. (Hubbers, Jacobs & Pieters, 2005)

Family voting Another aspect is called family voting. This occurs in all elections not held at poll stations. This means that the most dominant person at home influences the others to vote for a certain candidate. (Hubbers & Jacobs, 2004)

Duplicate ballots prevent automatic checkups Regarding duplicate ballots it is not sufficient to simply compare the MD5 hashes of the two reference tables to conclude

that there has not been committed fraud in the process. A voter can request for another ballot when claiming he did not receive the original one. Subsequently, in the reference table the status of the original ballot should be changed from used to revoked and the status of the duplicate from replacement to used. As such a modification will alter the result of the MD5 hash, fraud can only be detected by manually checking all entries with the original ones. This test could be automated. (Hubbers, Jacobs & Pieters, 2005)

Zips prevent automatic checkups Another reason which hinders such tests is that these MD5 hashes are calculated over the complete .zip file, but zip programs can differ from one another. This makes it possible that two identical files lead to different zipped file results. (Hubbers, Jacobs & Pieters, 2005)

MD5 hash more reliable by SURFnet After the election TTPI starts by computing an MD5 hash over the technical votes received by SURFnet in order to prove that they did not modify this file, but it would have been more trustworthy if SURFnet computed this hash before handing over the votes. Still, the correctness of the file can be manually checked against the file SURFnet makes available for download. (Hubbers, Jacobs & Pieters, 2005)

Hash collision In theory the hash functions mapping candidates and voters to a certain value could create collisions, but this could be detected by TTPI immediately after generation. Furthermore, using a good hash function this possibility is extremely rare. (Hubbers, Jacobs & Pieters, 2005)

JavaScript The script used to vote is downloaded from the Internet and therefore is exposed to certain risks: the JavaScript code can easily be modified in order to send arbitrary data to the server trying to generate a valid vote. However, in order for a vote to be valid a valid encrypted voter identifier as well as a valid candidate identifier is needed and the change that these two are guessed correctly is almost nil. The code could also be manipulated to read the secret key from the voter as he enters it and subsequently to cast a valid vote. This last attack can be detected if the voter checks his vote afterwards. This can be circumvented by using candidate-identities that are different for each voter, so that such a virus does not know which identity to select. (Hubbers, Jacobs & Pieters, 2005)

TTPI essential role The designer Piet Maclaine Pont of Mullpon v.o.f. and the developer Arnout Hannink of Magic Choice B.V. together form TTPI. In particular, this means that they would know the potential weaknesses of the system, including the secret key. (Hubbers, Jacobs & Pieters, 2005)

Moreover, TTPI plays an essential role in the protocol; it has several tasks and together with their knowledge of the protocol this party has a lot of power. It should be wise to have other parties take over some of their responsibilities.

DDOS Distributed Denial of Service attacks are always an issue in Internet based applications. (Hubbers, Jacobs & Pieters, 2005) A DDOS attack : computers that simultaneously burden a server, which therefore cannot handle requests of legitimate users. The requests often come from computers of negligent users, whose computers have been infected by a virus that sends requests to the server. These attacks are based on exhausting rare resources, like network capacity. The overall amount of data becomes more than the network can process resulting in a state in which the targets can not be reached over a network. (*"Intrusion detection"*, n.d.) As the server in the RIES protocol accepts all votes without checking for validity such an attack is realistic.

There is a heavy reliance on the voters. (Hubbers, Jacobs & Pieters, 2005) It is important that the voters check their vote, as no one else can. Fraud could then be undetected.

Here lies the main weakness of the RIES protocol that makes external attacks more of a threat as its detection relies on the voter to do an individual check. The fact that it is not very user friendly does not help. It would be very cumbersome to improve the check procedure as this would diminish security.

Still, even the overly critical group 'Wij vertrouwen stemcomputers niet.nl' (*"Wij vertrouwen..."*, n.d.) admits RIES being fairly secured against external attacks. It however claims that internal fraud can have great consequences, although the chance that it is committed is rather small. Therefore, the scenarios sketched by this group are not very shocking. In practice, quite some preconditions must be satisfied to have any constructive effect. Constructive in the sense that fraud remains undetected. One scenario addresses this last point by sketching a scenario that keeps the voter from detecting fraud. It is easily prevented and it could be that this weakness has already been solved in the protocol.

If an insider replaces the JavaScript by another, it will receive the key from the user that can be used to cast a vote. A check performed by this user would reveal a fraudulous sending. This could in theory be circumvented by returning the technical vote of someone else that voted for the same candidate. This is realized by saving such vote separately. The corresponding technical vote is shown on screen to the voter that has cast a vote for the same candidate. This vote can then be removed and replaced by a fraudulous one. If this voter checks its technical vote it appears in the table with received votes and is assigned to the candidate he cast a vote for and no fraud is detected.

The voter can however calculate it's own MAC values and detect the fraud, although in practice it seems reasonable to believe he would not recompute it. The most reliable way to prevent this is by examining the software before the election starts as the software that only can be accessed by TTPI resides at SURFnet.

This scenario is easily prevented and counter measures might already be used by RIES. This applies for more scenarios mentioned by this group. Moreover, some points stated by this group of opponents seem a little far fetched and others concern acts that are applicable to traditional ways of voting as well. Indeed, I would be very interested in a research about our traditional elections performed by this same group. One claim made by them just seems untrue. The site states that it is not always possible to detect fraud, but it is. Even tricks like the one above can be detected when voters recompute the hashes themselves using publicly available software. The only question here is if the voters are

willing to.

Only individual voter can detect SURFnets removal As SURFnet collects all technical votes, he could compute the MDC2 hash and look up this value in the reference table to be able to conclude for which candidate this vote is meant. Subsequently, SURFnet could drop certain votes. If SURFnet does this before the MD5 hash over the post-election table is calculated, this will be very hard to trace. This kind of fraud can only be detected if the particular voter checks his vote. (Hubbers, Jacobs & Pieters, 2005)

Only individual voter can detect TTPIs illegal key-use It is impossible for a system administrator to add valid votes, because he misses the necessary keys K_i. On the contrary, if TTPI offended the policy by not destroying the keys after generation, they could not only delete votes but they would be capable of adding or even altering (deletion followed by addition) votes too. Such fraud can be detected by voters whose votes have been altered. (Hubbers, Jacobs & Pieters, 2005)

Acknowledgement Thus, it seems like that in theory all fraud can be detected, although in some cases quite some actions have to be performed. Moreover, the acknowledgement needed to proof the fraud seems to suffer from potential weaknesses. Four remarks are discussed next. Mind that the information provided by Mr. Maclaine Pont is rather general that makes it harder to understand the use of the acknowledgements fully.

Need for halves unclear The acknowledgement is divided in two pieces of which one is sent to the voter and one is published by TTPI. In my opinion it would be sufficient to use the MAC value as it is and not to divide it. Presumably, it is needed to provide a proof that the vote has indeed been cast in time. However, it is not clear how this is accomplished by dividing the acknowledgement.

Not possible to prove detection of casting Moreover, it seems that the acknowledgement is designed to prove the receival of a vote, but not the absence of voting. If a personal key has been obtained illegally and used to cast a vote, the eligible voter has no means of proving that he did not send it but someone else did. Although this fraud can be detected by noticing that a vote has been received using this key, no evidence will exist.

No integrity check possible on acknowledgement It is not possible to perform an integrity check on the receipt acknowledgement. Only the independent party having the general secret key at its disposal can. (personal communication, Maclaine Pont, September 2005) However, Salomonson (2004) has some suggestions. A file could be returned that can be verified using an independently distributed tool or .p7c files can be used as acknowledgement, which are verifiable by Windows while distributing the fingerprint on the ballot.

Acknowledgement not unique From a test stage in June 2004 it appears that two different technical votes are associated with the same half of an acknowledgement (personal communication, Hubbers, September 2005).

It is important to note here that the two halves can only serve as a proof if they are indeed unique. This is needed to be able to identify the real vote sent. A third party should be able to recompute the hash and it should correspond exactly to the two halves owned by the voter and TTPI.

Take for example the following presumably technical votes and their voter parts of the acknowledgement in table 6 from a test stage in June, 2004. (personal communication, Hubbers, September 2005) This shows that unique votes are associated with the same half of an acknowledgement.

Technical vote	Receipt acknowledgement
3ae7b369646cd197 49a3d4c15826d0bf	FBAAE73C
3ae7b369646cd197 25267fa64258c8cc	FBAAE73C
830f08d28a5288bc c58212af3e671bfc	7036465B
830f08d28a5288bc 94f280d425c8749e	7036465B
96d2b67f1d314c73 a732dace0da550d3	F15C3345
96d2b67f1d314c73 494c4cd3b7493648	F15C3345

Table 6: Example of technical votes and not-uniquely defined receipt acknowledgements

A minimal chance on collision exists, but the fact that three examples are given here implicates that collision has not caused this. One explanation would be that this half of the acknowledgement is based only on the first part as two votes consisting of the same first part but a different second part result in the same acknowledgement. The second part is assumed to be the candidate part. (personal communication, Hubbers, September 2005) The halves presented here could be linked to some second half but could as well be to another second half.

This would result in a situation in which claims about sending and receiving some vote cannot be refuted or proven using the acknowledgement as it is not unique.

If somehow a vote for candidate A is altered to be counted for candidate B, the voter would have no means to prove that he cast a vote for candidate A.

It would also enable a voter to claim that his vote has been assigned incorrectly although it was not.

However mind that this matter could have been solved in the meanwhile or that crucial information is missing that would lead to other conclusions. In general, to be able to discuss the use of acknowledgements in the RIES protocol meaningfully more detailed information is needed.

From the discussion in this section it should be clear that the integrity of the parties involved is quite important. It seems protection against insider attacks could be improved. As the RIES protocol strength comes from detectability the use of the acknowledgements should be reviewed as well. In the same context a way to motivate the voters to perform the check is encouraged. Maybe a sample taken from the population should be obliged to perform the check, inspired on the procedure followed in the United States where citizens have a duty to serve in a jury.

Of course, if RIES could benefit from improving some points stated above it should. However, attention should not be limited to the aspects mentioned. The whole procedure should be carefully monitored, including organisational aspects. Democracy cannot afford it to be negligent at any subject, but one should keep in mind that 100% secureness can never be guaranteed and it would only be realistic not to demand this. This new way of voting should replace an existing one and it seems fair not to demand a higher level of security than offered by this existing procedure. It should be wise to benefit from advantages offered, even if the security level is not much higher than today's traditional voting.

6 Formal verification

After this detailed understanding of the RIES protocol it is time for a formal verification.

This section explains how model checking is applied to verify the properties stated in section 3. It continues by evaluating the approach of model checking in general. The section ends with discussing which software and formal languages have been used here to enable model checking.

6.1 Approach

The properties used to verify the general concepts verifiability and validity as described in section 3 are verified by automatically traversing the state spaces to show that the properties stated in some logic hold. This is called model checking. Nine scenarios have been designed for this purpose. Eight of them model some form of fraud to show that fraud is detectable (part of the verifiability feature) and that certain other properties still or no longer hold in hostile environments. Section 7 will come back to this. The approach taken to demonstrate that anonymity is guaranteed deviates from this as it is not a property that can be stated in modal logic. Instead, the proof lies in comparing two runs of the protocol.

In RIES the contents of a vote is known to the voter and to the party that is to assign the vote to a certain candidate. But except for this party no one should be able to determine the identity based on observable information. The basic intuition behind anonymity offered here is that actions should be divorced from the agents who perform them, for some set of observers.

In this context it is useful to discuss the concept of unlinkability. Unlinkability of two or more items of interest means that within the system, from the observer's perspective, these items of interest are no more and no less related after his observation than they are related concerning his a-priori knowledge. (Pfitzmann and Hansen, 2006) In practice, this means that the probability of two or more votes being related from the observers perspective stays the same before (a-priori knowledge) and after the observers observation (a-posteriori knowledge).

These concepts are stretched a little to be useful in applying role interchangeability to prove anonymity. Instead of applying the definitions to one system it is applied here to two systems as in two runs of the election.

Then, unlinkability ensures that a user may vote in several elections without others being able to link these votes together. Unlinkability requires that others are unable to determine whether the same voter caused certain specific operations in the system. (*"International Organisation..."*, 1999) Here, operation can be interpreted as a vote, the set of received votes and the resulting count.

Roughly speaking, unlinkability of items means that the ability of the attacker to relate these items does not increase by observing the two runs of the protocol.

This definition is a relative one as opposed to an absolute one. ISO (99) differentiates between absolute unlinkability (no determination of a link between uses) and relative unlinkability (no change of knowledge about a link between uses).

In many cases as in this one unlinkability of two votes depends on whether their content is protected against the observer considered. The message content could leak some information on the sender and thus their linkability. Therefore, the message content is made independent of the voter as a natural person.

Another obvious assumption is that the voters do not emit any information that would be useful to identification, like their personal key.

Thus, the formalization and verification of this property have to be oriented towards demonstrating that no observable difference can be detected between the first and second run of an election.

To achieve verifying anonymity it is useful to express anonymity in terms of unlinkability. When considering the sending and receiving of votes as the items of interest, anonymity may be defined as unlinkability of a vote and a voter. More specifically, anonymity of a voter means it is not linkable to any vote. Note that this also means that anonymity of a vote is also achieved. So we have sender anonymity as the two following properties hold: 1) a particular message is not linkable to any sender and 2) to a particular sender no message is linkable.

Here, it is proven that given two permutations of the voting process no difference can be entailed by some observer. In the first permutation a particular voter A gets assigned a particular key X and votes for a particular candidate V, while another voter B gets assigned a key Y and votes for candidate W. In the second permutation the situation is reversed; voter A is identified by the pair (Y,W) while voter B is identified by the pair (X,V). To an external observer the two situations are exactly the same; no inferences can be made about the origin of the votes.

The idea is that an observer will not notice any difference between the two runs as the output is the same. For all the observer is concerned, voter A could be associated with Key X or Key Y and the same goes for voter B. The observer could assume that voter A used Key X to vote for candidate V or that voter A used Key Y to vote for candidate W. Based on the observables either guess could be true. Hence, unlinkability between the voter and his vote is achieved: anonymity is guaranteed.

6.2 Model checking

To verify certain characteristics of a system in a formal way different techniques can be used, ranging from a manual proof of mathematical arguments to interactive computer-aided theorem proving and finally to automated model checking (Wang, Hidvey, Bailey &

Whinston, 2000). Here, it has been chosen to apply model checking for this purpose.

6.2.1 Model checking defined

Model checking concerns an advanced formal method. It is a powerful analysis method that determines whether a system model satisfies certain requirements under all circumstances. Verifying such requirements means proving or disproving properties with respect to a certain formal specification(*"Formal verification"*, n.d.).

Formal methods are best suited to address functional requirements and quality requirements, mainly correctness and reliability (Kneuper, 1997).

A formal method is a set of tools and notations (with a formal semantics) used to specify unambiguously the requirements of a computer system that supports the proof of properties of that formal specification (Hinchey & Bowen, 1995).

For this an abstract mathematical model of the system is needed. The requirements have to be expressed in some formal language to make reasoning about them possible. These requirements are usually stated in temporal logic formulas. The system or software design of interest is formalized by describing it with process behaviour. Design parts that are not relevant to the specific properties that are to be proven, can be ignored.

The model checker automatically translates these processes and property specifications into finite state machines. A finite state model generates on any given input only a finite number of possible states when executed (Holzmann and Smith, 2000).

The first results in a model usually expressed as a directed graph consisting of nodes (or vertices) and edges (*"Formal verification"*, n.d.), a labeled transition system. The nodes represent states of a program, the edges represent possible executions which alter the state. Atomic propositions are associated with every node that represent the basic properties that hold at a point of execution.

The second concerns property-specifying temporal logic formulas being converted to so-called property automata. Each property automaton specifies the set of feasible execution sequences over the actions (transitions) that correspond to a safety property of interest (Cheung and Kramer, 1999). Temporal logic makes reasoning about propositions qualified in terms of time possible. In a temporal logic, statements can have a truth value which can vary in time. The three basic temporal operators are: Always, sometimes, and never. Using these operators so called safety and liveness properties can be verified. Safety properties are "never" or "always" claims. Liveness properties are eventuality claims. A safety property holds if the integrated process and property automaton never go to a bad state in which the property is not satisfied. A liveness property holds if the automaton always returns to a progress state. (Wang, Hidvey, Bailey & Whinston, 2000)

Together this enables one to prove using a model checker that a particular execution path is or is not possible. Every execution path in the model has to be traversed, until the truth value of the claim can be decided.

If a specified execution path does not exist while it should, some property cannot be verified and a counterexample execution is given that shows why the formula is not satisfied. The counterexamples have accounted for finding even the most subtle errors in complex transition systems.

If it satisfies minimally one or all execution paths, respectively a "sometimes" or "always" claim is considered proven.

Efficient symbolic algorithms help to reduce the time needed to examine all paths. Extremely large state-spaces can often be traversed in minutes (*"Model checking"*, n.d.). Formal methods analyze the logic of system processes rather than execute those processes (Wang, Hidvey, Bailey & Whinston, 2000).

6.2.2 The value of

Model checking is quite different from testing and simulation, although it is often equationed with these techniques. It offers interesting advantages over them. For one, testing and simulation show only the presence of bugs not their absence (Denning, 1999). However, not any method can guarantee the total absence of errors, but formal methods certainly are one of the most complete and reliable strategies (Wang, Hidvey, Bailey & Whinston, 2000). These are by far the most important reasons to apply such techniques. The latter is achieved by the mathematical logic it is based on so that it can prove certain system properties with mathematical certainty. The former is established by its ability to theoretically account for every possible program execution, something necessary for critical system properties. It attempts to verify every possible input, while a simulation can only explore one selected execution path at a time (Wang, Hidvey, Bailey & Whinston, 2000).

This means that new areas in the state space are automatically covered when the model changes, while in testing new test cases should be designed (Wang, Hidvey, Bailey & Whinston, 2000).

An important difference becomes clear when considering the fact that it has detected hardware and software bugs missed by conventional testing methods. One of the reasons is that when failing to prove certain properties, model checkers provide counterexamples and help identify the input sequence that caused it (Meadows, 2003; Wang, Hidvey, Bailey & Whinston, 2000). Locating a bug is not that simple using other techniques.

In model checking, the user is only required to provide some high level representation of the model and the requirements to be checked. The model checker does the rest. (Clarke, Grumberg & Long, 2002)

It is even possible to mechanically extract verification models from a higher-level programming language when restricting to finite state models (Holzmann and Smith, 2000).

Another advantage of this procedure is that it is quite fast and less error-prone as significant expertise and user intervention is not needed. Also, partial requirement specifications can be checked. (Clarke, Grumberg & Long, 2002)

Proponents of formal methods also claim that it helps describing the true requirements of the user as formal specifications force off unambiguity and properties about these stated requirements can be model checked. Moreover, the well-structured form helps expressing or reasoning about complex issues (Kneuper, 1997).

It will also increase the trustworthiness of the system by proving its correctness. A system for which no evidence on reliability and correctness is available will not be acceptable no matter whether it is indeed correct or not (Kneuper, 1997).

6.2.3 The problem of

However, using formal methods is not the only way to achieve confidence in a system. Test results (Denning, 1999) and experience with similar systems (Kneuper, 1997) have

shown to be as important.

Moreover, formal methods cannot solve all problems and also bring problems of its own.

Model checking in its pure form cannot be applied to every large application, as its state space could exceed the maximal file size it can work with. A related concept is state explosion: the state space grows exponentially as the number of components of the system increases (Clarke, Grumberg & Long, 2002). Tools for model checking operate with methods to deal with this, like partial order reduction, abstraction, on-the-fly verification, state hashing and ordered binary decision diagrams. An important one is abstracting, something inherent at formal methods. However, if abstracting is not possible, a formal specification of such an application could easily be as long and complex as the implementation itself. (Kneuper, 1997)

However, it is often not necessary to specify all of the system formally. (Kneuper, 1997) Moreover, under certain circumstances a finite number of states is sufficient to prove the whole system correct (Meadows, 2003).

If that fails, another solution would be the use of special computer programs with model checkers or theorem proving that systematically control possibilities that could occur within the software (Niemoller, 2004).

However, some statements just cannot be proven as they depend on undecidable concepts. Take for example the Halting problem. The Halting problem is stated by Turing in 1936 (*"Halting problem"*, n.d.): it is undecidable whether a Turing machine terminates or not.

Moreover, every technique is always bound to the expressional power of man. For instance, it is not possible to formally specify what "no defect" means. All one can do is prove that a system does not have any of the defects that one can think of, and has all of the properties that together make it functional and useful. This implicates that there is a risk that some factor is not recognized that undermines the proof (Denning, 1999).

This same human error could be the reason that a correctness proof has not been found yet, although the program is correct. The structure of formal methods itself can also be a reason that not all requirements or aspects of a specification can be specified formally. (Kneuper, 1997) In verifying actual code, formal definitions of certain language constructs and software system components are either not available or too complex to be useful (Kneuper, 1997). In such a situation one needs to reanalyse and then rewrite the program using formal methods as a basis of design, instead of using them to analyze existing programs. This should lead to a correctness proof that serves two goals: the program is correct and we know that the program is correct. (Kneuper, 1997) However, formal methods slow down the design process (Denning, 1999).

Although it is a fundamental problem of all software development methods, one should keep in mind that any proof of correctness is always relative to the formal specification and the required properties (Pitt, 2002). A formal verification considers a formal object, rather than an actual implementation that is affected by its environment (Denning, 1999). The real question here is if the specification really is equivalent with the implementation (Kneuper, 1997). For instance, it could be that the programmer has built in some critical error that obviously would not exist in the model, resulting in incorrectness although the model would have been proven correct.

Using formal methods the absence of problems can only be interpreted as the protocol being error free conceptually, whereas found errors can only imply that the model is incorrect. The latter is not to say that the protocol itself must be faulty too as the implementation could differ from the model for several reasons. The same reasoning goes for proving a model correct. Proving certain conditions for a model might not imply that the real and more complex system satisfies them too.

One reason the model could deviate from the real implementation is irregularities in the translation from informal documentation to a formal specification. A proof might also address other concepts than the concepts one had tried to express in modal logic; again a translation error.

In theory, the tools used to reason with the formal model could contain bugs resulting in erroneous conclusions as well. However, this chance is assumed to be rather small.

The model is also based on assumptions that represent current thinking. A proof thus states that some formal model guarantees some property while discarding or assuming correct other parts. As soon as one of these assumptions does not hold (the environment changed or the designer forgot about a particular aspect), the conclusion (security) is invalid.

A well-known example concerns the Needham-Schroeder protocol for key establishment and the RSA cryptosystem that were believed to be proven correct for years (Denning, 1999). The system was cracked by stepping outside the model, by breaking the rules under which the protocol was proven secure.

6.2.4 A compromise

The theoretical chance that the above stated problems cause a proof to be incorrect will always be present and the one perfect solution to that does not exist. It is only realistic to accept that the smallest theoretical risks can become a real-life practice. Therefore, we should try to keep the risks as low as possible by applying and profiting from all methods available. The addition of the approach of formal methods certainly enlarges the chance of correctness of several types of programs, especially when correctness and reliability is concerned. It complements other techniques like it is itself complemented by others. In principal, correctness of a program also depends on its environment unexamined by formal methods, like the organisational (humans) or technical environment (compiler, hardware, operating system, various peripherals). Being aware of the limitations and possibilities will help in benefiting from formal methods.

6.3 Software and Other Machinery

To apply formal methods in verifying the RIES protocol software and other machinery is needed. In this section this is briefly discussed. The verification consists of several steps. First the protocol is specified, second the attempts to commit fraud are modeled, third the desired properties are stated and fourth and last, the protocol is specified. To this end, the RIES protocol is formalized in muCRL (micro Common Representation Language), a process algebraic language (Groote and Ponse, 1995). muCRL, an extension of ACP (Algebra of Communicating Processes) with abstract data types (Bergstra and Klop, 1984),

is a language for specifying distributed systems and protocols in an algebraic style. A muCRL specification describes a labeled transition system, in which states represent process terms and edges are labeled with actions. Using the CADP (Construction and Analysis of Distributed Processes) tool the requirements of the protocol expressed in the regular alternation-free μ-calculus (Mateescu and Sighireanu, 2003) have been verified.

A large number of distributed systems has already been verified in muCRL. In most cases a model checker or theorem prover have been applied. The CADP toolset is one of them and can be used when the state space is finite. The state space results from the muCRL specification in which the states are process terms and the edges are labeled with actions. The muCRL toolset in combination with the CADP toolset can generate and visualise this state space.

In verification of the stated properties of the RIES protocol several tools offered by the CADP toolbox 2001 "Ottawa" have been applied. It concerns muCRL 2.8.5, Instantiator 3.0, Evaluator 3.0, Aldebaran 6.5, BCG 1.4.

Some properties have been verified for particular scenarios using CADP 2005 at CWI Amsterdam. The newer version offers a more convenient notation of modal logic formulas and should operate faster. However, due to time constraints only a few have been verified this way. All properties and scenarios have been examined at the Vrije Universiteit Amsterdam using the 2001 version.

The toolbox expects a muCRL specification in plain ASCII. My specifications have been accepted by the tool mcrl 2.8.5 which implicates that these specifications are in correct muCRL. A lineariser transforms the specifictions into the Linear Process Operator (LPO) format. All my specifications have been translated to the so-called "tool bus format" (tbf) which is an LPO format using a regular linearisation method. The remaining tools require specifications in tbf format as input. (*"CADP manual..."*, n.d.)

From this .tbf file the tool instantiator 3.0 is used to generate the corresponding state space in .aut format. The -i option is used to accomplish that the internal action τ is represented in a form that is understood by CADP. (*"CADP manual..."*, n.d.)

The model checker EVALUATOR 3.0 from the CADP toolset is used here to verify properties of the protocol. It is used for efficient on-the-fly model checking of regular alternation-free μ-calculus formulas on a given Labeled Transition System (LTS for short). The evaluator tool takes two different inputs: an LTS, in this case expressed as a BCG graph and a temporal logic property, contained in the file prop[.mcl], expressed as a regular alternation-free μ-calculus formula. The result of this verification (TRUE or FALSE) is displayed on the standard output, possibly accompanied by a diagnostic. (*"CADP manual..."*, n.d.)

Options were set to generate a diagnostic in BCG format explaining the truth value of the formula and to divert output to the user's screen. Diagnostics are (usually small) portions of the LTS on which the formula yields the same result as when it is evaluated on the whole LTS. (*"CADP manual..."*, n.d.)

To verify the anonymity property, it is proven that two state spaces are branching bisimilar. This means that every equivalent pair of processes from these state spaces are bisimilar. If two processes are bisimilar, then not only they can execute exactly the same strings of actions, but also they have the same branching structure. (Fokkink, Groote & Reniers, n.d.)

For proving this bisimulation the tool Aldebaran 6.5 was used. The online manual pages of CADP (*"CADP manual..."*, n.d.) state that from October 2005, Aldebaran is a deprecated tool of CADP and is kept only for backward compatibility purpose. The tool bisimulator is recommended instead, but as it was not available to me Aldebaran was used to prove anonymity. BISIMULATOR can be applied directly to descriptions written in high level languages and the diagnostics generated by BISIMULATOR are directed acyclic graphs and are usually much smaller than those generated by other tools (such as ALDEBARAN), which can only generate counterexamples restricted to sets of traces.

Aldebaran allows the minimization or comparison of a LTS with respect to various equivalence and preorder relations. In this case it is applied to completely generated LTSs, which are usually contained in files with the .aut or .bcg suffix.

Options were set to allow comparison of the two LTSs based on the Fernandez & Mounier "on-the-fly" algorithm using branching bisimulation. On-the-fly verification of a property means that the property is verified during state space generation, in contrast to the traditional approach where properties are verified after state space generation. As soon as it is known whether the property holds, the generation of the state space can be stopped. It is important to find errors as soon as possible, because an erroneous system can have a much larger state space than the intended correct system. On the other hand, even in the case that all states become generated, the overhead caused by on-the-fly verification, compared to non-on-the-fly verification, is often negligible. (Varpaaniemi, 1994)

The result can be either TRUE (both LTSs are equivalent) or FALSE; in the latter case, Aldebaran issues a diagnostic consisting either of a set of discriminating sequences displayed on standard output or of an acyclic BCG graph stored in the file aldebaran.bcg (*"CADP manual..."*, n.d.)

Prior to applying Aldebaran the two state spaces concerned have been reduced with BCG_MIN applied to the .aut files using branching equivalence. The reduced state spaces are represented in BCG format. The BCG (Binary Coded Graph) format uses a binary representation of the LTS together with an ad-hoc compression algorithm, leading to very compact LTS files. (*"CADP manual..."*, n.d.)

Finally, the hardware used is specified here. Both the tool mcrl and instantiator are applied using a Linux compute server with the following specification: SUN Fire V440z, 4x AMD Opteron 852, 2594 MHz CPU, 16.0 GB memory. The CADP tool applied in model checking is used on a Solaris compute server with the following specification: SUN Fire V440, 4x Sun UltraSPARC-IIIi, 1062 MHz CPU, 8.0 GB memory.

6.3.1 Formal specification

To formalize the specification of the protocol muCRL is used.

The language microCRL, denoted muCRL, is a process algebraic language that was especially developed to take account of data in the study of communicating processes, something that process algebras tend to lack. It is basically intended to study description and analysis techniques for (large) distributed systems.

Process algebra focuses on the specification and manipulation of process terms as induced by a collection of operator symbols. It includes basic operators for sequential composition (.), non-deterministic choice (+), parallelism ($\|$) and communication ($|$) to express concurrency, deadlock (δ) to represent no behaviour, encapsulation (∂) to enable

remaining all actions of H into δ and enforcing actions into communication, hiding (τ) to enable hiding internal computations by making actions (representing events) invisible and recursive declarations to capture infinite behaviour. It operates on actions that can carry data parameters.

Each operator is given meaning by a characterizing set of equations called axioms, that together form a basis for equational reasoning about processes. The set of axioms state which combinations of function symbols (called data terms) are equal.

System behaviour generally consists of a mix of processes and data. Processes manipulate data types, which are defined by declaring a collection of function symbols. From these function symbols one can build data terms, together with a set of axioms, declaring which data terms are equal.

A conditional expression ($\lhd\rhd$) enables that data elements influence the course of a process and an alternative quantification operator ($\sum$) provides the non-deterministic choice over some sort.

A specification in muCRL serves this goal as it consists of two parts. One part specifies the data types, the second part specifies the processes.

As said, the muCRL toolset can automatically transform a muCRL specification into a linear process operator (LPO). The tool mcrl determines whether a specification is correct muCRL and if possible a so-called lineariser transforms this specification into an LPO. The remaining tools require specifications in LPO format as input. From this file a state space is visualized using instantiator. It generates this LTS if it consists of finitely many states. The LTS can be visualized, analyzed and minimized by the muCRL toolset in combination with CADP toolset. (Fokkink, Groote & Reniers, n.d.)

6.3.2 Formal properties

The properties to be verified are translated to modal logic formulas.

The regular alternation-free μ-calculus is used here to formulate properties of labeled transition systems. It is a fragment of μ-calculus that can be efficiently checked. It is built up from three types of formulas: action formulas, regular formulas and state formulas. For regular formulas the constructors '.', '$\wedge$' and '*' are used for respectively concatenation, choice and transitive-reflexive closure. F and T are used in both action formulas and state formulas, representing *no action* and *any action* in action formulas and *empty set* and *entire state space* in state formulas. The operators $\langle\rangle$ and [ ] are used to denote the existential and universal quantifier.

Finally, μ is the minimal fixed point operator. It is used to determine a set of states that will lead to a state that satisfies a certain property. More formally: Let D be a finite set with a partial ordering $\leq$. Given a mapping $\phi : S \to S$, an element d of D is a *fixpoint* of ϕ if $\phi(d) = d$. It is a *least fixpoint* if $d \leq e$ for all fixpoints e of ϕ. The least fixpoint of ϕ is denoted by $\mu X.\phi$. A set S of states is mapped to those states where ϕ holds, under the assumption that the recursion variable X evaluates to `T` for states in S and to `F` for states outside S. As partial ordering on sets of states set inclusion is taken. This makes the empty set the least element.

μX is computed by first taking as a solution S_0: the empty set of states. Next, $S_{i+1} = FIX(\emptyset, X = S_i)$ for $i \leq 0$ is computed. This results in the set of states that satisfy ϕ under the assumption that S_i is the set of states where the sub-formula X is satisfied.

Since the set of states is finite $S_i = S_{i+1}$ holds for some i. This fixpoint is the actual solution for X in $\mu X.\phi$. (Fokkink, n.d.)

The μ operator is a very powerful one. Deadlock freeness and the liveness property are easily expressed using it (*"CADP manual..."*, n.d.):

To express that a system is deadlock free the following line is used:

$$\mu X(\langle T\rangle T \wedge [T]X)$$

There is an action that leads to TRUE and all actions leads to a state in X.

The liveness property, saying that eventually ϕ happens, is expressed as follows:

$$\mu X(\phi \vee (\langle T\rangle \wedge [T]X)$$

The μ operator can be preceded by a formula in modal logic.

7 Algebraic Specification

Verifying a protocol generally starts with generating its algebraic specification. This involves abstracting the protocol by identifying the key behaviours of the protocol participants and their responses to different situations. For example, part of the key behaviour of a voter as modeled here is checking if his cast vote has been received. If it has, the voter would signal this using an OK action. In the situation that it has not, another response is modeled: a detect action to signal that something is wrong.

The algebraic specification in muCRL that will describe the RIES protocol consists of two parts, as every such specification does. The data part consists of data type declarations and expresses which kinds of data are used and the operations on them. The behaviour part consists of processes and actions describing the protocol's behaviour, including the exchange of data. The latter also includes communication specification. As it does not consider the actual contents transferred by these communications but rather the kinds of data and operations on them, one can abstract away the complex cryptographic primitives, such as encryption and decryption.

I will start with describing the data types used, followed by specifying the process behaviour. The key behaviour is derived from the operation of the RIES protocol as described in section 4. This reflects the normal behaviour of the processes. However, certain hostile environments have been designed as well.

In describing the process behaviour, I will first present the detailed specification of every process when the normal situation is concerned. Then I will proceed with discussing the differences that result when fraudulous attacks are introduced. In most cases this means that a parallel process is added and some sub-process is slightly altered.

Along the way, assumptions made will be clarified. These assumptions exist because the specification provided here is not an exact copy of the RIES protocol. It is designed to enable proving some particular properties. To make this possible some aspects have been added and others have been omitted.

7.1 Data types

An extensive list of data types has been designed to simulate the RIES protocol. Instead of reviewing all of them in detail, they are introduced here. The functions that are associated with each of these sorts will be elaborated on more when discussing the particular process behaviours. However, it goes to far to discuss every function in full detail, especially when the function uses sub-functions. These sub-functions are not named here to keep matters simple. However, the complete formal specification is inserted in Appendix A : The muCRL specification of the RIES protocol.

Naturally, the sort **Bool** of booleans has been specified. Another basic sort is **Nat** for natural number beginning at zero. It serves as a counter to keep score of which candidate has been assigned a vote and it is used to link a natural number to a vote to enable a fixed ordering in saving them. The latter use keeps the state space manageable. In this protocol a vote is made up of a personal **Key**, an election identifier **ElectionId** and a candidate identifier **CandidateId**. The key is used to encrypt both the election and candidate identifier resulting in the so called encrypted voter identity **EncryptedVoterId** and encrypted candidate identity **EncryptedCandidateId**. Together this makes a total of five sorts. The personal key for every voter comes from a collection of keys, called the **keyTable** here. The candidate identifier too originates from a collection, the **candidateTable**. Every election is associated with one election identifier as to assure that the vote containing this identifier concerns this election and not i.e. last year's. This is at least recommended by Salomonson (2004). To enable the casting of an invalid vote, it has been chosen here to initiate every voter process with an incorrect election identifier. It could be seen as associated with former elections and votes consisting of this identifier could represent replay of old messages.

The correct one is received along with the ballot. Both identifiers will then be joined in a table from the sort **elidTable** from which the voter non-deterministically selects one when casting a vote. All cast votes consisting of a pair EncryptedVoterId X EncryptedCandidateId are stored by every individual voter and by SURFnet that receives them. Such a collection is saved in a table of sort **Table**. Here, the $\leq$ ordering is applied while satisfying in succession the relations Key1 $\langle$ Key2, CI1 $\langle$ CI2 and EI1 $\langle$ EI2. Thus, a vote consisting of Key1 will always precede a vote consisting of Key2, irrespective of the remaining contents of the votes. Votes consisting of the same key are ordered by candidate identifier, with CI1 preceding CI2. The ordering for EI is applied when it concerns votes composed of the same Key and CI. Adding a vote to a table that contains an identical vote results in the adding of the vote preceding the existing vote.

After the election is closed SURFnet will transfer this table to TTPI that will use it to perform the count. For this purpose the reference table is needed. It consists of all possible votes for every eligible voter together with a reference to the candidate identifier, with every vote protected by an MDC hash. Here, the hash is modeled as one taken over the whole vote. This results in the so called **referenceTable**. Note that it does not contain votes using an incorrect identifier. Every such vote is captured by the sort **MDCvote**. To enable matching a received vote with a vote in the reference table the MDC value of every received vote is needed. Therefore, the MDC value for every received vote is calculated and published in a table together with the vote in its normal form: a pair encrypted voter identity X encrypted candidate identity. The collection of the pairs

consisting of the MDC value and the vote in its normal form makes up the **MDCTable**. Together, the reference table and this MDC table makes it possible to assign votes to the corresponding candidates. The count is stored as a pair of candidate identifier X natural number. These pairs together form the sort **countsTable**, which is also made public after the election is closed.

7.2 Process behaviour

This section discusses the various scenarios and their specifications.

7.2.1 General scenario

Modeled In the general scenario everybody is honest and no attacks are modeled. The attacker scenarios to be discussed in the next section are based on this general specification. Therefore, the specification given here applies to all scenario's. To start with a general conception of the protocol, the data flow in the RIES protocol is discussed first.

Here two different ways are used to enable data flow. The first is by means of explicit communication between processes. A process X can send a datum d to a process Y only if Y at the same time is in a position to receive it. The synchronization between the send and receive action is realized using the encapsulation functionality. The second way data can be transferred to a process Y is by using parameters. The sending process $X(x_1, \ldots, x_m)$ should be parametrized with particular instances of the data and its body should contain an expression $Y(d_1, \ldots, d_n)$ where Y is a process name and d_i are data terms that contain occurrences of the variables $x_1, \ldots, x_m$. This makes it possible to pass the data or part of it in a possibly altered form to another process. This latter form is also used to provide general knowledge among the participants. This will become evident in this section. The data parameters of a process declaration are initialized using the initial declaration in the **init** section.

The formal specification of the RIES protocol consists of minimally three processes operating in parallel. They make up the initial part of the specification. It concerns the central controller TTPI, voters $V_1, \ldots, V_p$ and the server administrator SURFnet. Every voter is represented by a separate process.

These parties are defined as separate processes to make concurrent actions possible and to keep knowledge separate. The reason not to combine several parties in one process to reduce code and states is that it leads to a situation in which conceptually one party can use another one's knowledge, which is not possible in the system modeled here. Instead of making assumptions about which actions are possible and which are not, which would weaken my conclusions if not supported by formal proofs, this approach is taken.

Another process that is to be seen as a separate entity is the website that publishes the information needed to perform the checks. However, this process is not a parallel one, but is executed as a sequential part of the process term that makes up TTPI as its knowledge is a sub-set of TTPI's.

The data flow between these entities is outlined below. The formal specification implements this flow, which closely follows the original RIES protocol. Some communications, parameters and sub-processes are left out in this section to keep matters simple.

Explicit communication is represented by a two way arrow, whereas transferring data by means of parameterizing is represented by a single arrow.

TTPI starts with sending the ballots to each of the voters.

$$TTPI \leftrightarrow_{(key,ei,ctable)} Voter \tag{1}$$

Then, each voter will either send a vote to the server managed by SURFnet or abstain from casting a vote. In the case of voting the following communication takes place:

$$Voter \leftrightarrow_{(MAC_{K_i}(ei),MAC_{K_i}(ci))} SURFnet \tag{2}$$

The election is closed when the server at SURFnet times out. Casting votes is no longer possible and the collection of received votes is transferred from SURFnet to TTPI using parameters. Together with the reference table *reftable* the tally result can be determined, which is saved in the counts table *cntstable*.

$$SURFnet \rightarrow_{(receivedVotes,reftable,cntstable)} TTPI \tag{3}$$

Subsequently, TTPI determines the tally result and transfers this to the website. The tally result consists of a list with candidates and their corresponding number of votes received and an MDC protected table with all received votes together with a marking. This marking indicates if the vote is valid or invalid. In this specification this is implemented by using a separate table that consists of the received valid votes only. A vote that does not occur in this table is an invalid one.

It is chosen to express this using a table to enable the reuse of the already existing data type and functions of the received votes table.

$$TTPI \rightarrow_{(MDC(receivedVotes),MDC(validVotes),cntstable,reftable,cntstable)} Website \tag{4}$$

Finally, the checks are to be performed by the voter and SURFnet. Both parties will need information that is made public at the website. Therefore, both entities will communicate with the website to obtain the information needed to do checks. The order in which the two parties contact the website is non-deterministic.

$$Website \leftrightarrow_{(MDC(receivedVotes),MDC(validVotes),cntstable,reftable,cntstable)} Voter \tag{5}$$

and

$$Website \leftrightarrow_{(MDC(receivedVotes),MDC(validVotes),cntstable,reftable,cntstable)} SURFnet \tag{6}$$

This data flow is made visible in figure 5.

The explicit communication is specified by the communication of actions in the formal specification:

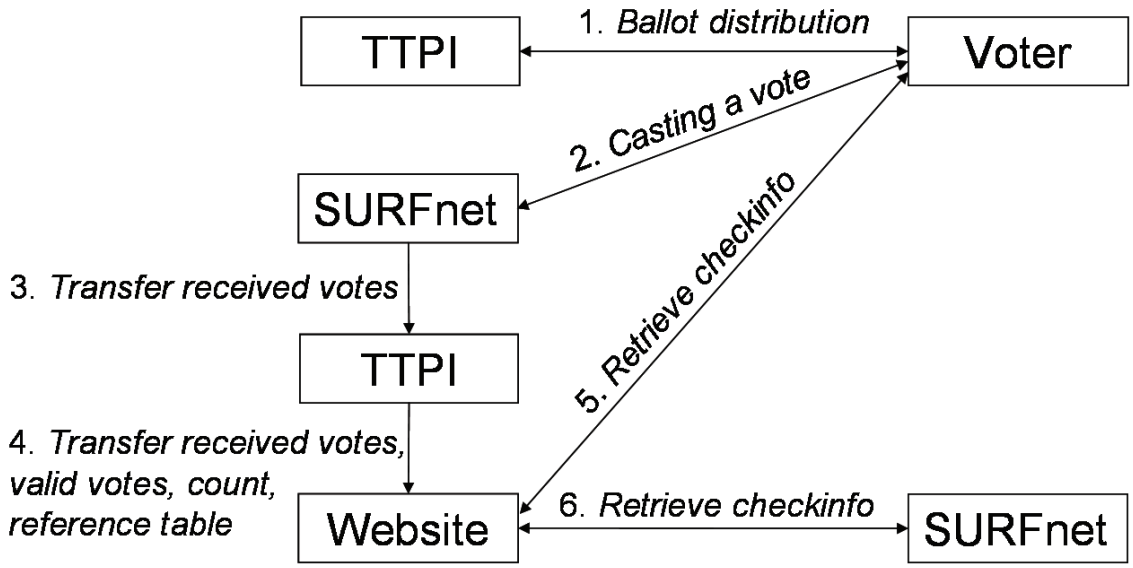

Figure 5: Dataflow

comm

send_ballot	| receive_ballot	= comm_ballot (1)
send_vote	| receive_vote	= comm_vote (2)
send_checkInfo	| receive_checkInfo	= comm_checkInfo (5)
send_checkInfo	| receive_checkInfoSURFnet	=
comm_checkInfoSURFnet (6)		

Another communication is modeled in all but one specification to avoid deadlock. This will become clear later. It concerns:

comm

send_recVotes	| rec_recVotes	= comm_recVotesSURFnet

Additionally, when fraud is committed other communication is needed to enable separate processes as a means to keep fraud information from innocent parties.

comm

sendFraudInfo	| recFraudInfo	= comm_recallFraudInfo
saveFraudInfo	| keepFraudInfo	= comm_storeFraudInfo
saveFraudInfo	| getFraudInfo	= comm_FraudInfo

The communications are presented in the state space by the actions preceded by *comm_*. This form is also used in expressing properties in modal logic.

In this simplistic view of the data flow only three processes exist, plus the website. However, in the formal specification every process, except for the website, is divided into sub-processes. Every name of such a sub-process is always preceded by the name of one of the three core entities.

Specified In all specifications the sum operator is used that offers non-deterministic behaviour, instead of restricting my models by using deterministic functions.

The first concept of the formal specification addressed two voters and at most two duplicate votes per voter. However, the file size of the corresponding state space presumably exceeds the maximum size the CADP tool can operate with. This was the case for an earlier specification that tried to consider duplicate votes. In a later stage the generation of the state space did not even end. A recent modification of the function that adds the votes to a table affected the filesize positively. Instead of first adding a vote and then possibly remove third duplicates the new add function first checks on the occurrence of duplicates before adding one. However, a statespace concerning two voters and two duplicate votes remains to grow out of proportion. Therefore, the properties have been proven correct for specifications built around one voter and allowing two duplicate votes and for specifications built around two voters but allowing only one duplicate vote. The file size is not affected by the existence of different single votes for each candidate and each candidate identifier.

Section 8.2 gives some information about file sizes and verification times for different configurations. The various add functions are included in Appendix A.

The formal specification provided here describes the situation for two voters and two candidates with two duplicates allowed. It consists of four processes operating in parallel. It concerns the central controller TTPI, a voter V_1, a voter V_2 and the server administrator SURFnet. These four process declarations make up the initial state of the specification. In the init section the data parameters of these processes are initiated. The processes have been given more appropriate names that fit this phase, that will alter during the protocol.

In detail, *TTPI_sendBallot* starts with the following arguments: a key table containing all unique keys for the voters, a reference table, a candidate table and a counts table. As two voter processes are modeled, two keys exist. The candidate table is initialized for two candidates. The stated properties are proven for this configuration and it is assumed that the properties will also prove to be true for more voters and candidates.

As in this specification the reference table is given as an argument to the TTPI process that starts the protocol, there is no need to express that the reference table is published. Here, the reference table takes the form of a reference function that can be used by every process. In the RIES protocol a true table is used.

The two voter processes *Voter_receiveBallot* are given as arguments an empty table voterMemory to store every vote the voter casts, a faulty election identifier to simulate invalid votes and an empty election identifier table to store the correct and incorrect election identifier and choose arbitrarily between them. The server administrator *SURFnet* has no arguments.

To begin with the first mentioned, *TTPI_sendBallot* will send a ballot containing a personal key, the election identifier and the table containing all possible candidates to the voter processes until all keys have been distributed. This iteration is guarded with the function *notEmpty* that determines if the table containing all keys is empty. The number of keys should therefore be equal to the total number of voter processes.

Here, the keys, election identifier and candidate identifiers are distributed among the eligible voters unprotected. In the actual system it concerns a postal channel. It has been implemented here though to give the protocol a clear beginning, but in verification

it should not be taken into account. This distribution simulates postal distribution. Of course, postal mail is not 100 percent secure, but the risk is negligible.

The sum operator together with the *element* function guarantees that the keys from the keytable are non-deterministically selected to be sent to some voter process. The key that is sent will be removed from the keytable using the function *remove*. The demand that TTPI destroys all keys after distribution is hereby satisfied. The process then continues with the execution of the sub-process *SURFnet_receive*.

$$
\begin{aligned}
&TTPI\_sendBallot(\quad keytable : keyTable, ei : ElectionId,\\
&\quad reftable : referenceTable, ctable : candidateTable,\\
&\quad cntstable : countsTable) =\\
&\quad (\textstyle\sum_{key:Key} send\_ballot(key, ei, ctable) \quad .\\
&\\
&\quad TTPI\_sendBallot(\quad remove(key, keytable), ei,\\
&\qquad reftable, ctable, cntstable)\\
&\\
&\quad \triangleleft \quad element(key, keytable) \quad \triangleright \quad \delta)\\
&\\
&\triangleleft \quad notEmpty(keytable) \quad \triangleright\\
&\\
&SURFnet\_receive(empty, reftable, cntstable)
\end{aligned}
$$

One voter process *Voter_receiveBallot* at a time is in communication with *TTPI* to receive the ballot. Once received, the voter can either cast a vote or abstain from voting. This is simulated using the sub-process *Voter_choice* that will be executed after receiving the ballot. To enable voting, this sub-process is transferred the needed information: the personal key, the correct and incorrect election identifier in a table and the candidate table. Note that the identifier received with the ballot is always the correct one. The other one comes from initializing the voter process in the init section. Two more arguments are passed to the sub-process, namely two voterMemories. These tables are needed to check if the individual votes have been processed correctly. They will later be filled with the votes this voter has cast and the votes that the voter did not cast. In the original protocol it is also possible for a voter to know this. The first is achieved by presenting the cast vote on screen. The received set complemented with the cast votes will result in the set of uncast but received votes.

$$
\begin{aligned}
&Voter\_receiveBallot(\quad voterMemory : Table, ei2 : ElectionId,\\
&\quad eitable : elidTable) =\\
&\textstyle\sum_{key:Key} \quad \sum_{ei1:ElectionId} \quad \sum_{ctable:candidateTable},\\
&\\
&receive\_ballot(key, ei1, ctable) \quad .\\
&\\
&Voter\_choice(key, ei1, eitable(ei1, eitable(ei2, eitable)),\\
&\qquad\qquad ctable, voterMemory, voterMemory)
\end{aligned}
$$

Next, the sub-process *Voter_choice* is executed. The alternative composition operator enables a choice between casting a vote and abstaining. In the RIES protocol the eligible

voter should first login at some predefined website and enter his password. This is omitted here.

In the latter case of abstaining, the sub-process *voter_getCheckInfo* is executed after a wait action. This symbolizes an abstaining voter. This voter will not cast a vote, but will directly check if someone has cast a vote using his key. The wait action is introduced here to avoid unguarded recursion.

In the case of voting the sum operator and two *element* functions are used to non-deterministically select both a candidate identifier and election identifier from respectively the candidate table and elidtable. The vote will then be arbitrary. It is sent using the action *send_vote* that communicates with SURFnet's *receive_vote* to *comm_vote* as used in both the state space and the properties in modal logic.

Voters that do not succeed in casting a vote before the election is closed are not modeled here as the encapsulation functionality guarantees communication.

In the RIES protocol the web browser computes the MAC values of the vote sent by the voter using the unique key. This is omitted here. Instead, the approach taken is to let the voter send his technical vote directly. This minor assumption does not affect the properties to be verified. The feature that no information leaves the client's pc unprotected remains untouched.

After casting a vote the voter will remember the vote by adding it to his voter memory. He can then choose to cast a vote again, possibly the same one.

Concerning remembering the cast votes by the individual voter, a special *add* function is used here to ensure that at most two identical votes are stored in memory. This is to avoid infinite enumerations. The same add function is used by the SURFnet process to store at most two identical votes per voter. However note that the communication of identical votes between a voter and SURFnet is possible an infinite number of times, as said earlier.

There are no restrictions placed here, just like in the original protocol. However, the fairness property guarantees that the wait action from the Voter process or the timeout action from the server side is always performed eventually. It means that if an exit from a τ-loop exists, then no infinite execution sequence will remain in this tau-loop forever. Therefore, every voter process will eventually execute the sub-process *voter_getCheckInfo*. This is to say that it is assumed here that every eligible voter will check if his individual votes have been processed correctly, including an abstainer. This will probably not be the case in practice. The assumption that it is, enables proving certain properties true as all traces will be verified.

The sub-process *voter_getCheckInfo* will eventually start sub-processes to perform checks concerning the cast votes and the votes the voter did not cast. The set of cast votes is passed to the sub-process together with an empty set that will later contain the votes that have not been cast. Also, the key and the correct election identifier coming from the ballot are passed. Together they form the EVI, that is part of the voter's knowledge. The voter can use this to quickly decide if votes using his key have been received. This is particularly interesting when checking for reception of non-cast votes, and thus fraud.

$$
\begin{aligned}
&Voter\_choice(\quad key : Key, ei1 : ElectionId,\\
&\quad eitable : elidTable, ctable : candidateTable,\\
&\quad voted : Table, notVoted : Table) =\\
&\\
&(\textstyle\sum_{ci:CandidateId} \quad \sum_{ei:ElectionId}\\
&\quad send\_vote(MAC(key, ei), MAC(key, ci)) \quad .\\
&\\
&\quad Voter\_choice(\quad key, eitable, ctable,\\
&\qquad add(MAC(key, ei), MAC(key, ci), voted), notVoted)\\
&\\
&\quad \triangleleft \quad and(element(ci, ctable), element(ei, eitable)) \quad \triangleright \quad \delta)\\
&\\
&+\\
&\\
&wait \quad . \quad voter\_getCheckInfo(\quad voted, notVoted, key, ei1)
\end{aligned}
$$

Both a casting process and an abstaining process is followed by the execution of the *voter_getCheckInfo* sub-process. This process takes preparations for starting the checking procedure by downloading the needed information from the website. This is simulated by the action *receive_checkInfo* with the *Website* process that communicates to *comm_checkInfo*. Not all information present at the website is needed to perform the checks that are modeled here for the voter. However, all information is communicated here as the same communication type is used to allow the SURFnet process to access some of the information. This is to avoid the need of a separate process to make separate communications with the voter and SURFnet possible.

In detail, the information communicated consists of an MDC protected table containing the set of received votes, an MDC protected table containing the set of valid votes, the table containing the candidates and corresponding count, the reference table and an empty counts table.

As modeled here, the latter is used by SURFnet to save the results of the recount and match it against the count published at the website. Anyone can perform this check, but here it has been chosen to use the SURFnet process for this purpose. A task that can only be fulfilled by SURFnet is matching its set of received votes against the set published by TTPI.

The information used by the voter process includes the personal key and election identifier that form it's EVI and the table with received and valid votes. It is passed to the sub-process *voter_checkVotedRec* together with two tables to contain its cast votes and the votes that were not cast. A copy of the first mentioned is transferred as well to make it possible to have access to the set of cast votes after the original set is emptied by processing every vote. This will become more clear when discussing the last sub-process of the Voter.

$$voter\_getCheckInfo(\quad voted : Table, allVotes : Table, notVoted : Table,$$
$$key : Key, ei : ElectionId) =$$

$$\sum_{receivedVotes:MDCTable} \sum_{validVotes:MDCTable} \sum_{countstable:countsTable}$$
$$\sum_{reftable:referenceTable} \sum(_{nitCnts:countsTable}$$

$$receive\_checkInfo(receivedVotes, validVotes, countstable, reftable,$$
$$initCnts) \quad .$$

$$voter\_checkVotedRec(voted, voted, notVoted, receivedVotes,$$
$$validVotes, key, ei)$$

The sub-process *voter_checkVotedRec* is the first of three sub-processes that simulates the checks performed by an individual voter as modeled here. The other two are named *voter_checkNotVoted* and *voter_checkVotedValid*. Three checks are concerned: first the set of cast votes is checked for receival, second the set of uncast votes is determined and checked for validity and finally the set of cast votes is checked for validity.

Effectively, a blank voter or abstainer performs only the second check as it has not cast a vote. Instead it only checks if his key has been used to cast votes.

In RIES all checks are performed using zipped tables. Here, such compression is omitted.

An essential assumption is made here: every eligible voter will perform all these checks. In practice, it is expected that this will not hold. Here, it is modeled though to enable verifying certain properties. For instance, it is possible to verify if every cast vote is indeed received. It would make no sense to verify that some of the cast votes are received while the receival of others is undetermined. Moreover, only the eligible voter is capable of performing this check.

Another important assumption is that the check if an uncast vote has been used is performed as a separate check, although in practice it is more of a byproduct when concluding that something went wrong. In that sense, abstainers would have even less motivation to perform this check.

Before turning to the details of each sub-process of the voter entity some concepts concerned should be explained first.

In this specification a valid vote is one that appears in the received valid votes set. It will if it satisfies two conditions. One, it must be an element of the reference table. As such it must contain the correct election identifier. Two, other votes with the same key but a different candidate identifier do not exist in the received set.

On the contrary, an invalid vote does not satisfy these two conditions. One, it does not occur in the reference table as it is composed of an incorrect election identifier. Two, the same key is used once or more to cast a vote for another candidate. In the latter case these votes are denoted here by the term false duplicate.

In addition, if two or more votes exist that are an exact copy of one another only one occurrence is seen as valid. The remaining ones are marked invalid. This convention is used both in the protocol as in this specification. The term duplicate is used here. It

concerns two or more votes that are exactly the same, meaning that they contain the same key, election identifier and candidate identifier. A second duplicate denotes the vote that will be marked invalid. Note that it does not matter whether the first or second duplicate is marked invalid, leaving the other valid.

Second duplicates are only introduced here to enable showing that only one occurrence is valid and only one is counted in the tally result. This is part of proving that only one vote is marked valid. Under all circumstances at most one vote using a particular key is marked valid. This includes duplicates, which can be generated by the voter but also by an illegal party. It is not sufficient to only consider one of these two cases. The second duplicates cast by the voter are examined when checking if the cast votes are valid. The duplicates cast by some illegal party should be examined when checking the uncast votes for receival and validity. Therefore, the set of uncast votes should not only be based on the personal sets of all possible and cast votes, but also on the public set of received votes. The received votes could contain duplicates from the voter, which will be examined when checking the cast votes. The remaining duplicates will be originated from the illegal party. As these votes will contain its encrypted voter identity it is easy for the voter to recognize the votes as one of its own. In the original protocol the voter has the knowledge to do so and here it is assumed that he will.

In the RIES protocol valid votes are those left unmarked in the table with received votes. However, in this specification two separate tables are used to specify each set to reuse the sort Table. Another difference lies in marking invalid votes with a log entry. This is omitted here as the properties to be proven do not address this.

As the set of valid received votes should be seen as a sub-set of the received votes fraudulous actions that have an impact on the valid votes will also have an impact on the received votes. Every valid vote is a received vote. The contrary does not hold, as not every received vote is a valid one. Adding or removing an invalid received vote will not affect the set of valid votes. As the sets are defined separately, fraudulous actions affecting both sets are to be performed twice.

The published table containing all received votes that are valid cannot contain duplicates, as only one occurrence of a vote can be valid. Therefore, after it is determined that a vote occurs in the valid set this vote should be removed from the valid set. The second time a duplicate of this vote is checked it will not have a match in the valid set and it is judged invalid. This could be the result of the eligible voter casting duplicates or of another party fraudulously adding a duplicate. In the latter case, it concerns a duplicate not cast by the voter although it did send the same vote. The fraud is detected when concerning the set of uncast votes. When the eligible voter himself sends a second or more duplicate all but one vote are invalid and no fraud is concerned. In both situations it should be the case that when later examining if the second duplicate is a valid one the result is negative. This is achieved in this specification by removing the occurrence from the valid set and then continuing the check procedure.

This set together with the received votes is public information needed for the voter to perform the checks.

Both the presence and absence of a vote in the two sets can either mean that the vote is processed correctly or that fraud is committed concerning this vote.

The presence of a vote in the received set can result from eligible voting or from

fraudulous actions. The same goes for the presence of a vote in the valid set and the absence of a vote in the received set. However, solely the absence of a vote in the valid set does not necessarily implicate that fraud has been committed. It could be that the eligible voter send an invalid vote himself, modeled here by casting false duplicates or by casting a vote using an incorrect election identifier. An originally valid vote could however be made invalid by an attacker by adding false duplicates. This should become apparent when the check confirming that votes from the set of votes not cast do not appear in the table with received votes fails. Therefore, first the checks are made that could detect fraud and then the set of valid votes is checked that is not a sub-set of the set of uncast votes. As all uncast but yet received valid votes are not elements of this set, the presence or absence of a valid vote from this set can only be legal. This closely follows the procedure performed in practice as it seems natural to first confirm that the vote has been received and then to check if and why the vote is marked invalid.

Actions are designed to express the results of the checks. An OK action means that the particular vote is present in some set. A detect action signals that some vote has been added or removed illegally. To make model checking stronger, these actions are parametrized with the vote concerned. For the detect actions this means that they not only signal that some fraud is detected, but also that this specific fraud action is detected. The same goes for the OK actions in that they also specify that it concerns a specific vote.

As the votes are tested against two different sets two different actions are designed to express that the votes are present. The received set is associated with the action *OKr* with the letter r standing for *received*. Using this letter the action distinguishes it from the OKv action that is associated with the valid set where the letter v stands for *valid*. The two actions mark the presence of a particular vote in respectively the set of received votes and the set of received valid votes. This means that an OKr action is also issued when the eligible voter did not cast a vote but a vote has somehow been added. This vote can be valid, but can only be when it is received. Therefore, a valid vote is signalled by both an OKr and OKv action. Similar to the OKr action, an OKv action can be issued although the eligible voter did not cast it.

The vote has thus been added fraudulously, something which should be detected. If the voter does, this is modeled by issuing a *Voter_detectAdd* action. If the fraud concerns the opposite, the removal of a vote, the action concerned is a *Voter_detectRemove* action. The removal of a vote should be detected when confirming that all cast votes have been received. It could concern valid votes, false and true duplicate votes. The adding of a vote should be detected when confirming that the votes not cast have not been received either. Although the adding of a duplicate will not alter the tally result this specification will coin this action as fraud. As they are not saved the adding of third or higher order duplicates will not be noticed.

In practice, an abstainer would have little motivation to perform this check or any other. Voters that did cast a vote will have more reason to trace what has happened to their votes. If they should conclude that a vote that should be valid is marked invalid, this could be the result of fraud. They will now automatically find out if their vote has been invalidated by the adding of a vote they did not cast.

As will be explained later, SURFnet is modeled here to use similar actions to signal detection of fraud.

Now that these implementation decisions are made clear, I will continue with discussing the different sub-processes of the voter entity.

As its name implicates, *voter_checkVotedRec* checks if all votes cast, including second duplicates, have been received. The function *present* that is expressed in a conditional is used to simulate this. It takes the first vote from his set and checks if it is present in the set with received votes. As the vote consists of two parts, both parts are retrieved separately using the *retrieveV* and the *retrieveC* functions to retrieve respectively the *Encrypted Voter Identity* and the *Encrypted Candidate Identity* from the table containing the cast votes.

For every vote not received a detect action is issued. In this case, fraud has been committed and subsequently detected. In the RIES protocol one must prove that he did send a vote by presenting the acknowledgement the server returns upon receiving a vote. The use of acknowledgements is omitted here as the sending of a vote in this specification can only occur when the other party is ready to receive it. All cast votes must therefore have been received and if not, it must be the result of fraud.

If the vote is present, an OKr action signals the presence of the vote in the set of received votes. Both actions are parametrized with the vote concerned using the *retrieveV* and the *retrieveC* function.

As recursion is needed to perform this check on every cast vote, a conditional is needed to stop when done. Every time the sub-process is iterated the vote examined is removed from the set of cast votes with the function *removeFirst* and the remaining set is passed as an argument to the sub-process. If it concerns a vote that occurs in the received set it is also removed from this set using the function *delete*. The *retrieve* are applied to retrieve the concerning vote. The *notEmpty* function determines the outcome of the conditional expression that considers if the set of cast votes is empty. If it is, all cast votes have been examined and the check procedure continues by executing the *voter_checkNotVoted* sub-process. One of its arguments concerns the set of uncast votes by this voter. This could include duplicates introduced by an illegal party.

The existence of illegal duplicates is noticed only when considering the received set. Therefore, the approach to compute the set of uncast votes by taking the complement of the set of cast votes, with all possible votes as the universal set is not taken. Every voter can determine the set of all possible votes by computing the corresponding MAC values using the personal key, the correct election identifier and the candidate table as arguments. However, this computation would not take duplicates into account that are cast by illegal parties. Therefore, it has been chosen to determine the set of uncast votes by considering the received set only. Of course, this table could also contain duplicates generated by the voter himself. Therefore, the voter's duplicates should be removed from the set of received votes. A convenient place to do so is during the procedure when checking the cast votes against the received votes. Every cast vote occurring in the received votes set is removed from this set. Subsequently, the remaining votes consisting of the same encrypted voter identity are added to the set of uncast votes.

The function *retrieveUncast* takes the corresponding *encrypted voter identity* (which can be calculated by anyone, given the key and election identifier) and the received set as arguments. As the cast votes, including duplicate cast votes, have already been removed from the received set at this stage the result would consist of all uncast votes received,

uncast duplicates included.

Technically, the set of uncast votes does not only apply to the set of uncast received votes but also to the set of uncast and unreceived votes. However, this last set is not considered as in practice it would not be either.

In the actual procedure followed one would notice the votes that contain his personal EVI and that have not been cast just by consulting the published table. This table is sorted by EVI.

$$
\begin{aligned}
&voter\_checkVotedRec(\quad voted : Table, copyVoted : Table, notVoted : Table,\\
&\quad receivedVotes : MDCTable, validVotes : MDCTable\\
&\quad key : Key, ei : ElectionId) =\\
&\\
&\quad (\quad (\quad OKr(retrieveV(voted), retrieveC(voted))\quad .\\
&\quad voter\_checkVotedRec(removeFirst(voted), copyVoted, notVoted,\\
&\qquad delete(retrieveV(voted), retrieveC(voted), receivedVotes),\\
&\qquad validVotes)\quad)\\
&\quad \triangleleft\quad present(retrieveV(voted), retrieveC(voted), receivedVotes)\quad \triangleright\\
&\quad (Voter\_detectRemove(retrieveV(voted), retrieveC(voted))\quad)\quad .\\
&\\
&\quad voter\_checkVotedRec(removeFirst(voted), copyVoted, notVoted,\\
&\qquad receivedVotes, validVotes, key, ei)\quad)\quad)\\
&\\
&\triangleleft\quad notEmpty(voted)\quad \triangleright\\
&\\
&voter\_checkNotVoted(copyVoted, retrieveUncast(MAC(key, ei),\\
&\quad receivedVotes, notVoted), receivedVotes, validVotes)
\end{aligned}
$$

The next two sub-processes will perform validity checks. In practice, the voter would directly see that only one vote is valid. To be able to simulate this in muCRL, one check should be made for the cast votes and another for the uncast votes.

As only one vote per voter can be marked valid, only the first validity check can succeed. The next time a check for validity is performed, it should fail. Be it in checking the cast votes or the uncast votes. To enable this, the corresponding vote is removed from the public valid votes table after the first valid check has succeeded.

The sub-process *voter_checkNotVoted* considers all votes that have not been cast but yet received. As such, these votes should not occur in the set of received votes published by TTPI. However, it has just been determined this set is. For all these votes it must be signalled that they are present in the received set and that they have been illegally cast by another party. Furthermore, a check is made to determine if such a vote is valid. This is needed to enable model checking the above mentioned property.

When the set of uncast received votes is not empty this implicates fraud. Apparently, some party other than the eligible voter has added a vote with his key to the set of received votes somehow. As this vote has been received, although not by the eligible voter, an *OKr* action is issued first. It signals the presence of the vote in this set and is followed by a *Voter_detectAdd* action. As the vote concerned can be a valid one, another

conditional expression will subsequently decide if these actions are followed by an *OKv* action or a *continue* action without parameters. The *continue* action is introduced here to avoid unguarded recursion. It is issued when again a *present* function determines that the particular vote is not present in the valid votes. If it is, an *OKv* action indicates this. Each time that an uncast vote is marked valid, this means that the eligible voter did not cast any vote. A vote is valid if it is the only one using a particular personal key. Both the continue action and the OKv action are followed by another execution of the process.

These iterations are needed to consider every vote until the set of uncast votes is empty. In every iteration the vote examined is removed from the set of cast votes with the function *removeFirst* and the remaining set is passed as an argument to the same sub-process. Again, the *notEmpty* function conditions this recursion and will eventually lead to the execution of the sub-process *voter_checkVotedValid*. This sub-process will work with the set of cast votes and a possibly altered set of valid votes.

$$
\begin{aligned}
&voter\_checkNotVoted(\quad voted : Table, notVoted : Table,\\
&\quad receivedVotes : MDCTable, validVotes : MDCTable) =\\
&\\
&(OKr(retrieveV(notVoted), retrieveC(notVoted)) \quad .\\
&Voter\_detectAdd(retrieveV(notVoted), retrieveC(notVoted)) \quad .\\
&\quad (OKv(retrieveV(notVoted), retrieveC(notVoted)) \quad .\\
&\quad voter\_checkNotVoted(voted, removeFirst(notVoted), receivedVotes,\\
&\qquad remove(retrieveV(voted), validVotes))\\
&\quad \triangleleft \quad present(retrieveV(notVoted), retrieveC(notVoted), validVotes) \quad \triangleright\\
&\quad continue \quad . \quad voter\_checkNotVoted(voted, removeFirst(notVoted),\\
&\qquad receivedVotes, validVotes)))\\
&\triangleleft \quad notEmpty(notVoted) \quad \triangleright\\
&voter\_checkVotedValid(voted, validVotes)
\end{aligned}
$$

The last check the Voter process performs is twofold. First the sub-process *voter_checkVotedValid* is designed to check its cast votes for validity. If its votes are not, the voter himself must have voted more than once on different candidates. A vote can be turned invalid or even removed in a fraud action, but that should have become apparent in the former checks. Therefore, this sub-process will not signal any fraud. This process will simply signal that a vote is valid and if it is not it will just continue with examining the next vote. This is repeated until the set of cast votes is empty. As before, the function *notEmpty* guards this recursion and the *present* function will determine if a particular vote is an element of the set of valid votes. A vote is accessed using the *retrieve* functions. If it is not present, a continue action is issued followed by the execution of the sub-process parametrized with the set of cast votes minus the one just examined and the set of valid votes. If it is an element of the valid votes this fact is signalled using the *OKv* action.

Second, if a vote is valid, that vote is checked for correct assignment. The voter will know for which candidate the vote should be counted. Other parties than the voter are not able to perform such a check on the correctness of references as they will miss this knowledge. The individual knowledge of each voter is simulated with the function *recalci* that takes the second part of the vote, the *encrypted candidate identity*, as an argument

using *retrieveC*. The result should be equal to the reference to the corresponding candidate in the published reference table. As the reference table is MDC protected, the MDC value of the vote considered should be looked up in order to determine the candidate. This value must first be calculated by the voter using the function *computemdc* and then matched against the table, simulated by the function *ref*. Again, the vote is accessed with the *retrieve* functions. The value can also be read from the published table containing all received votes as it contains both the MAC and MDC values. However, this value is published by TTPI and thus it is more reliable to recompute it. If the two references are equal this is signalled using an OKci action that is parametrized with the vote. If it is not, fraud with referencing must have been committed and this is signalled with a *Voter_detectci* action that also has the vote checked as an argument.

If a vote is marked invalid the voter is assumed not to be interested in correct assignment as the vote will not be counted.

The process ends with delta, which is needed to terminate the processes.

$$
\begin{aligned}
&voter\_checkVotedValid(\quad voted : Table, validVotes : MDCTable) = \\
&\\
&(\quad OKv(retrieveV(voted), retrieveC(voted)) \quad . \\
&\quad (OKci(retrieveV(voted), retrieveC(voted)) \\
&\quad \lhd \quad eq(ref(computemdc(retrieveV(voted), retrieveC(voted))), \\
&\qquad\quad recalci(retrieveC(voted))) \quad \rhd \\
&\quad Voter\_detectci(retrieveV(voted), retrieveC(voted)) \quad) \quad . \\
&\quad voter\_checkVotedValid(removeFirst(voted), remove(retrieveV(voted), \\
&\qquad\qquad validVotes)) \\
&\quad \lhd \quad present(retrieveV(voted), retrieveC(voted), validVotes) \quad \rhd \\
&\quad continue \quad . \quad voter\_checkVotedValid(removeFirst(voted), validVotes)) \\
&\lhd \quad notEmpty(voted) \quad \rhd \\
&\delta
\end{aligned}
$$

As concurrent processes have to be executed in parallel with the ones discussed, the discussion returns back to the beginning.

Every process that casts one or more votes explicitly communicates with the server managed by SURFnet, simulated by the sub-process
SURFnet_receive. This sub-process is executed after TTPI is done distributing the ballots. The arguments that the sub-process is initiated with consist of an empty received votes table, the reference table and the counts table with each count initialized to zero. Again, only one voter process at a time can communicate with SURFnet to cast a vote. Every vote is then saved by SURFnet. In the RIES protocol the vote must first be stripped to remove NAT info. The SURFnet will save the vote without checking for validity using the *add* function that allows second duplicates. Communication is no longer possible when the election closes, which is simulated by the action *timeout*. *SURFnet_receive* will then continue with executing the sub-process *TTPI_count* after saving the set of received votes using the function *send_recVotes*. This set is communicated to a parallel process SURFnet that is designed separately to make it possible to perform an equality check after the results are published by TTPI. The check will determine if the received set is tampered

with by TTPI. This process must be a parallel one to avoid deadlock, because in this stage the tally result is yet to be counted and then published.

$$
\begin{aligned}
&SURFnet\_receive(\quad receivedVotes : Table, reftable : referenceTable,\\
&\qquad\qquad cntstable : countsTable) =\\
&\\
&\textstyle\sum_{eci:EncryptedCandidateId} \sum_{evi:EncryptedVoterId} \quad receive\_vote(evi, eci) \quad .\\
&SURFnet\_receive(add(evi, eci, receivedVotes), reftable, cntstable)\\
&+ \quad timeout \quad . \quad send\_recVotes(receivedVotes) \quad .\\
&TTPI\_count(receivedVotes, reftable, cntstable)
\end{aligned}
$$

The sub-process *TTPI_count* simply computes the tally result and publishes it by executing the sub-process *Website_sendCheckInfo*. The result of the election is determined by assigning each valid vote to its corresponding candidate. Therefore, the valid votes are first computed using the *clean* function that marks a vote valid if it occurs in the reference table and if no false duplicates have been received. The *count* function looks up the corresponding candidate for every valid vote and saves the count in the counts table. The set of received votes and valid votes are MDC protected, simulated by the *computemdc* function. The counts table and the reference table are also passed to be published. In reality, the reference table is published before the start of the election. But as parts of the other information is published after, it has been chosen to join all in one process.

An empty counts table is communicated as well for use to SURFnet that will perform a count check. This recount will be safed in the empty table, making a comparison with the count of TTPI possible.

$$
\begin{aligned}
&TTPI\_count(\quad receivedVotes : Table, reftable : referenceTable,\\
&\quad cntstable : countsTable) =\\
&\\
&Website\_sendCheckInfo(\\
&\quad computemdc(receivedVotes),\\
&\quad computemdc(clean(receivedVotes, reftable)),\\
&\quad count(computemdc(clean(receivedVotes, reftable)), cntstable),\\
&\quad reftable, cntstable)
\end{aligned}
$$

Publishing the result is simulated by explicitly communicating the information to the Voters and SURFnet using the action *send_checkInfo*. In the state space and modal logic properties the communications are presented by *comm_checkInfo* and *comm_checkInfo-SURFnet*. As before, only one process at a time can be in communication with the website. This sub-process just consists of the recursive execution of this communication. It is ended when all processes have retrieved the check information successfully.

This website is assumed to be innocent, here. Incorrect information is the result of fraud. In practice, a hash value of the information is advertised to be able to check if the information has been altered.

$$
\begin{aligned}
&Website\_sendCheckInfo(\quad receivedVotes : MDCTable,\\
&\qquad validVotes : MDCTable, countstable : countsTable,\\
&\qquad reftable : referenceTable, initCnts : countsTable) =\\
&\\
&send\_checkInfo(receivedVotes, validVotes, countstable, reftable,\\
&\quad initCnts)\\
&.\quad Website\_sendCheckInfo(receivedVotes, validVotes, countstable,\\
&\qquad reftable, initCnts)
\end{aligned}
$$

In this specification it is SURFnet that recomputes the count, but it first matches its own set of received votes against the set made public by *TTPI*. Its own set has been communicated to this parallel process *SURFnet_checkRec* by *SURFnet_receive* after receiving all votes with the action *rec_recVotes* that communicates to *comm_recVotes*. *SURFnet_checkRec* is a parallel process to avoid deadlock and this communication is simply a way to retrieve the knowledge. The remaining needed check information is retrieved from the website using the sum operator and the *receive_checkInfoSURFnet* action as soon as the website sub-process starts executing. The process then proceeds with determining if the two sets match with *presentAll*. If they do, this is signalled with an OK action parametrized with the set of received votes. If the two sets do not match this would mean that the set offered by TTPI either misses a vote or contains too many votes. In the specification the *added* function expresses if a vote has been added or removed which is respectively signalled by the actions *SURFnet_detectAdd* and *SURFnet_detectRemove*. These actions are parametrized with the vote concerned by using the *getAdd* or *getRemove* function together with the *retrieve* functions. The *getRemove* function is based on the fact that TTPI's table is smaller than SURFnet's and that this results in a difference when comparing the tables line by line. The removed vote is equal to the first element from SURFnet's set that differs from TTPI's. The *getAdd* function works in a similar matter: TTPI's table is bigger than SURFnet's and the added vote is equal to the first element from TTPI's set that differs from SURFnet's.

The sub-process continues by executing *SURFnet_checkCnt*.

$$
\begin{aligned}
&SURFnet\_checkRec = \\
&\sum_{SURFnet\_rec:Table} \quad rec\_recVotes(SURFnet\_rec) \quad . \\
&\sum_{receivedVotes:MDCTable} \sum_{validVotes:MDCTable} \sum_{countstable:countsTable} \\
&\quad \sum_{reftable:referenceTable} \sum_{initCnts:countsTable} \\
&receive\_checkInfoSURFnet(receivedVotes, validVotes, countstable, \\
&\qquad\qquad reftable, initCnts) \quad . \\
&\quad (OK(receivedVotes) \\
&\quad \triangleleft \quad presentAll(SURFnet\_rec, receivedVotes) \quad \triangleright \\
&\qquad (SURFnet\_detectAdd(retrieveV(getAdd(SURFnet\_rec, \\
&\qquad\quad receivedVotes)), retrieveC(\\
&\qquad\quad getAdd(SURFnet\_rec, receivedVotes))) \\
&\qquad \triangleleft \quad added(SURFnet\_rec, receivedVotes) \quad \triangleright \\
&\qquad SURFnet\_detectRemove(retrieveV(getRemove(SURFnet\_rec, \\
&\qquad\quad\quad receivedVotes)), retrieveC(getRemove(\\
&\qquad\quad SURFnet\_rec, receivedVotes))) \quad) \\
&\quad) \quad . \\
&SURFnet\_checkCnt(count(mdcclean(receivedVotes, reftable), \\
&\qquad\qquad initCnts), countstable)
\end{aligned}
$$

The recount can be performed by anyone as all information that is needed is published on the website. In my formal specification it is SURFnet that will do the check, following the argument that if it is verified that SURFnet can perform the check everyone can perform the check.

The *SURFnet_checkCnt* sub-process is parametrized with the count as computed by TTPI and with the count as computed by SURFnet. The latter count is based on the set of received votes published by TTPI and not on its own set, as the recount in this specification is meant to reveal fraud in the count procedure, independently of fraud concerning the individual votes. A recount using its own set would differ from TTPI's count if the sets differ and if the procedure differs. At this stage I am only interested in the latter. Again, to perform the *count* function first the valid votes are determined using the *mdcclean* function. The result is saved in *initCnts*.

The sub-process does an equality check on both counts. If they match this is signalled with an *OK* action parametrized with the count and the process terminates with δ. If the counts differ, this should be signalled with an action parametrized with the candidate identifier that causes the mismatch. For this purpose the sub-process *SURFnet_getCorrupt* is executed next.

$$SURFnet\_checkCnt(\quad SURFnet\_countstable : countsTable,$$
$$TTPI\_countstable : countsTable) =$$

$$OK(TTPI\_countstable) \quad . \quad \delta$$
$$\triangleleft \quad eq(SURFnet\_countstable, TTPI\_countstable) \quad \triangleright$$
$$SURFnet\_getCorrupt(SURFnet\_countstable, TTPI\_countstable)$$

In practice, one single glance at the two counts suffices to determine the candidate identifier that causes the mismatch. In muCRL however, it takes a few steps. In every iteration of the sub-process *SURFnet_getCorrupt* the next first elements of both counts is examined. The *notEmpty* function guards this recursion. The process also ends when the remaining counts are equal. In both situations the process is eventually terminated with δ.

If the counts differ, the *added* function determines if a count has been decreased or increased compared to the count of SURFnet. It functions by saving in succession the count from SURFnet and the count from TTPI that differ per candidate identifier using *getDiff*.

If this table consisting of the differing counts contains a count that has been increased, this is signalled with the action *SURFnet_detectAdd*
parametrized with the candidate identifier whose count has been decreased using the *getAdd* function. The process is then iterated with this count removed from both tables and checked again for adding. This means that every fraud concerning the count is signalled, although in the current specification only one count will be altered. The procedure followed here results in first signalling all increased counts and then all decreased counts. The latter are processed and signalled in a similar way.

$$
\begin{aligned}
&SURFnet\_getCorrupt(\quad SURFnet\_countstable : countsTable,\\
&\quad TTPI\_countstable : countsTable) =\\
&\\
&\quad (\delta\\
&\quad \lhd \quad eq(SURFnet\_countstable, TTPI\_countstable) \quad \rhd\\
&\qquad (\\
&\qquad (SURFnet\_detectAdd(getAdd(getDiff(SURFnet\_countstable,\\
&\qquad\qquad TTPI\_countstable))) \quad .\\
&\qquad SURFnet\_getCorrupt(remove(getAdd(getDiff(\\
&\qquad\qquad SURFnet\_countstable, TTPI\_countstable)),\\
&\qquad\qquad SURFnet\_countstable),\\
&\qquad\qquad\quad remove(getAdd(getDiff(\\
&\qquad\qquad SURFnet\_countstable, TTPI\_countstable)),\\
&\qquad\qquad TTPI\_countstable)))\\
&\qquad \lhd \quad added(getDiff(SURFnet\_countstable, TTPI\_countstable)) \quad \rhd\\
&\qquad (SURFnet\_detectRemove(getRemove(getDiff(\\
&\qquad\qquad SURFnet\_countstable, TTPI\_countstable))) \quad .\\
&\qquad SURFnet\_getCorrupt(remove(getRemove(getDiff(\\
&\qquad\qquad SURFnet\_countstable, TTPI\_countstable)),\\
&\qquad\qquad SURFnet\_countstable),\\
&\qquad\qquad\quad remove(getRemove(getDiff(\\
&\qquad\qquad SURFnet\_countstable, TTPI\_countstable)),\\
&\qquad\qquad TTPI\_countstable)))\\
&\qquad)\\
&\quad)\\
&\lhd \quad notEmpty(SURFnet\_countstable) \quad \rhd\\
&\delta
\end{aligned}
$$

The specifications used to enable fraud differ on some points from the processes as discussed. The next section will look at this in more detail.

7.2.2 Attacker Scenarios

Fraud can be committed by different participants, namely TTPI, SURFnet, the voter and an external party. It is assumed that only one type of fraud is committed at once. As such, a scenario only models one type.

Normally, a security protocol is verified in a hostile environment by adding an intruder into the protocol. The Dolev-Yao model (Dolev & Yao, 1983) is the most widely used one and it specifies an intruder having the following capabilities (adjusted a little to fit the current protocol):

1. The other voters may try to set up a session with it;
2. It can overhear any message exchanged between the server and the voters;
3. It decrypts messages that are encrypted with its own public key and store parts of a message in its knowledge database;

4. It may replay an old message it has seen before, even if it cannot decrypt the encrypted part;

5. It can generate messages using any combination of its knowledge database and send them.

The first attack is not considered a big risk according to a report of TNO as they claim that server authentication prevents other websites from pretending to be the election site. To anticipate misuse in this way the voters are explained how to validate the certificate. (Esch-Bussemakers, Geers, Maclaine Pont & Vink, 2002).

SSL assures that it is impossible to eavesdrop on packets before they reach the server and therefore to determine the technical votes. (Hubbers & Jacobs, 2004) Moreover, even if these votes could be intercepted this would only yield information that is already publicly available as soon as the election is closed. The votes are published on the webpage as they are received: in MAC format.

The RIES protocol makes use of a one way Message Authentication Code to hash the votes sent over Internet using private keys (*"Kiezen uit..."*, n.d.) and cannot be decrypted as such. Therefore, the third attack is not applicable here. It is assumed that the voter handles his private key in a proper way and that the distribution of these keys by the mailman is safe, making it impossible for anyone else to know this key.

For the above two discussions it is not likely that an intruder succeeds in adding usable information to its knowledge base to generate messages that will match a vote in the reference table.

Replaying an old message (being the whole vote in this case) will not affect the security of the protocol as it is allowed to cast duplicate votes. This is something that will be verified here. If the old message has originated from a prior election it is assumed that the format deviates from the one used in the current election as the MAC function encrypts votes based on a unique election identifier associated with one particular election. It should be best if the candidate identifiers also differ from the ones used before. This should yield differing encrypted votes and if not, this should be uncovered and solved when composing the reference table.

An intruder could however be an observer that tries to break voter secrecy. To demonstrate that the RIES protocol offers voter secrecy two scenarios have been designed, which will be explained shortly.

In my formal specification the keys, election identifier and candidate identifiers are distributed among the eligible voters unprotected. This would mean that an Observer capable of eavesdropping has every information needed to construct a vote. He would only have to calculate the MAC values. With this information he could send a vote or could find out what the contents are of other observed votes by matching them against his calculated values or by decrypting them.

However, this practice is not considered as in the actual system it concerns a postal channel. It has been implemented here though to give the protocol a clear beginning and ending, but in verification it should not be taken into account. This distribution simulates postal distribution. Of course, postal mail is not 100 percent secure, but the change is negligible.

I will focus on internal fraud as the most important factor in determining the vulnerability of a system is the people involved (Fisher, 2003). However, the chance that internal parties commit fraud is assumed to be rather small.

Among these entities TTPI has the most and most powerful opportunities. Several forms can be distinguished, the most important ones of which are to be described in this section.

One way a voter could consciously or unconsciously commit fraud is to cast a vote that does not appear in the reference table. This vote will be marked invalid by the protocol. Such a vote will be discarded and other votes using the same key are treated independently. Thus, it is not treated as an illegal, fraudulous action. It is modeled in all scenarios.

An abstainer could also falsely state that his cast vote has been removed. He would have to validate this statement by presenting the acknowledgement he received from the application TTPI wrote at the server administered by SURFnet. This act is not modeled.

Finally, SURFnet may well have the best chance to commit fraud that remains undetected.

I will start describing the scenarios in which the fraudulous actions occur during the election, followed by the scenarios in which the actions take place after the closing of the election. A chronological ordering is preserved as the construction of the protocol is followed by determining this ordering. Therefore, the sending of a vote is treated first, followed by removal of a received vote. The protocol will then continue to determine the valid votes and the count. Hence, the scenarios in which a vote is added, removed or made invalid are described next. I will end with the scenarios that will alter the count directly by changing the count for a certain candidate and indirectly, by using a altered reference table. As explained, attacks resulting from weaknesses of the cryptographic functions used in the protocol are not considered. These include the use of SSL, MAC and MDC. An intruder attack is also omitted from this analysis.

Modeled In all scenarios all but one participant is honest. Only the entity committing fraud is stated in the brief description of the scenarios. The remaining participants are assumed honest.

But before turning to the discussion of the attacker models the two scenarios needed to demonstrate anonymity are discussed first.

Anonymity scenarios In the first scenario voter process 1 is assigned Key 1 and votes for candidate 1, whereas voter process 2 is assigned Key 2 and casts a vote for candidate 2. The reversed situation is realized in the second scenario in which voter process 1 is assigned Key 2 and votes in favour of candidate 2, whereas voter process 2 is assigned Key 1 which is used to cast a vote for candidate 1. To show that in the presence of an observer anonymity is preserved the two state spaces are proven to be equal. The underlying thought here is that an Observer will not note a difference between the two. Note that in these two models the distribution of keys and the choice for a particular candidate are both deterministic. This is needed to verify the anonymity property as both the key and the candidate determine the look of the traffic and of the contents of the table containing the received votes. It has been shown that the two state spaces that are

based on these two scenarios are branching bisimilar equivalent. The assumption made here is that the true identity of an eligible voter is not defined by his choice for a particular candidate or by a key. Refraining here from deterministically distributing the keys leads to a discrepancy between the two state spaces that follow from these two scenarios. Take for example the situation in which voter process 1 decides to cast a valid vote for his favourite candidate, being candidate 1, whereas voter process 2 prefers candidate 2. If voter process 1 is assigned key 1, leaving key 2 for voter process 2 this would lead to two valid votes. In encrypted form the set would consist of the votes displayed in column one of the following table. If Key 2 is assigned to voter process 1 and Key 2 to voter process 2, the results are those in the next column. The results have been ordered by Key. The difference is visible immediately and will be revealed when observing the traffic (the sending of these votes) and the table containing the received votes.

Scenario 1	Scenario 2
$(MAC_{(Key1,EI1)}, MAC_{(Key1,CI1)})$	$(MAC_{(Key1,EI1)}, MAC_{(Key1,CI2)})$
$(MAC_{(Key2,EI1)}, MAC_{(Key2,CI2)})$	$(MAC_{(Key2,EI1)}, MAC_{(Key2,CI1)})$

Table 7: Visible difference when role interchangeability is based on key only

As my specification also models the distribution of the keys to the voter, the order in which the keys are assigned should not differ in both models. Moreover, this order should not be fixed either. Otherwise, a difference is created. Therefore, a choice is modeled in both between first assigning a key to process 1 followed by assigning one to process 2 and first assigning a key to process 2 followed by the assignment to process 1.

The Observer can be an insider; a voter. In my formal specification modeling two voters and two candidates that means that the Observer knows for which candidate the other voter voted in the situation that at most two votes have been cast. This information can be derived from the count. The count can be zero, one or two for a certain candidate. If it is one and the insider did not choose this candidate, he will know that the other voter cast a vote for this candidate. If a count is two, the insider can conclude that both he and the other voter voted for this candidate. Therefore, the situation is considered in which both voters are genuine and the Observer is an external party.

The eight attacker scenarios concern actively committed fraud. First, TTPI can pretend being a voter by taking a key from the keytable and using it for sending an arbitrary vote real time. This is modeled using the sum operator to achieve that the key is selected in a non-deterministic way. In this case TTPI apparently did not destroy the keytable, thereby breaking the protocol. At this moment TTPI cannot know which voters abstain from voting, that would leave their keys unused and attractive to be misused in such a way. Therefore, this fraud can result in a valid vote if that key is not used by the voter, an invalid vote if the key is used by the voter for another valid vote or an invalid vote in the case that votes using that key were already invalid. It could also concern duplicates.

Only in the first case this fraud will prove effective, but only if the voter refrains from checking its vote afterwards.

This form of fraud can be detected if every voter checks if his unused votes have been received. This also goes for a blank voter. If from this check it followed that they did fraud is signalled. This is expressed by stating that the sending of a vote by TTPI is

always followed by a detection of that vote.

The scenarios have been partly designed to show that fraud can be detected. This fraud can also be detected, although it seems unlikely that an abstainer would take the trouble to check if his key has been used. If he did, proving it seems not to be possible. Both reasons will contribute to the chance being small that such fraud is revealed and proven.

Second, SURFnet can remove a vote that has been received before handing over the file containing all received votes to TTPI. SURFnet does not have the key table and candidate table to its disposable and therefore cannot cast or alter a vote. Furthermore, this party cannot obtain the key by deciphering the incoming votes either as a MAC is a one-way hash.

However, as SURFnet manages the servers that receive incoming votes, this party is capable of removing one of them. The algorithm MDC-2 is widely available and can thus be computed and used together with the reference table to determine for whom a certain vote is intended. In this way, SURFnet can remove particular votes in a directed way: valid votes for a certain candidate or invalid votes making one valid. SURFnet can also decide to remove votes in an arbitrary way. This routine is modeled here to show that every removal during the election will be detected by the voter, not only the valid ones. In removing a random vote this could be a valid or invalid vote. This can have several consequences. In the case of a single valid or invalid vote this will result in zero votes for that particular voter. However, if that voter has cast a vote more than once maximal one vote can be valid. Remember that all duplicates but one are invalid and that all votes are invalid if the same key has been used for different candidates. In the latter situation removal of one such vote will lead to one or more remaining votes for that voter. If only one remains that vote has become valid. If more than one are left these votes will remain invalid. It could also be that a duplicate vote is removed. In that case the workload of TTPI is slightly released, but the net result (the count) will remain the same. Still, this fraud is detected by the voter whose vote has been removed. Note that this form of fraud cannot be detected by TTPI, because TTPI receives the votes from SURFnet.

If an invalid or duplicate vote has been removed it could be that the voter will not notice it as he would have no motivation to check. Note however that detection is possible in theory.

Third, TTPI has the ability to remove a valid vote from the table with received votes after the election is closed. As TTPI needs to know which votes are valid, this form of fraud can only occur when the election is closed. The valid votes have to be computed using the same procedure as when the valid votes are computed normally. In this model such a vote is selected non-deterministically. This form of fraud can be detected by both the voter and SURFnet. When performing the check afterwards, the voter will notice that the vote does not occur in the table with received votes. SURFnet will uncover this type of fraud as well when equating the table TTPI will publish after this action with the table containing the votes that were received by SURFnet. As such, this fraud will not be very successful.

Fourth, this same party can add valid votes after the election is closed by simply adding one to the table containing the received votes. This would mean that TTPI did not destroy the key table. As the election is closed, TTPI is now in the position to determine which

keys have not been used. TTPI identifies the unused keys by combining every key with the election identifier and then computing the MAC hash of this combination resulting in the encrypted voter identity.

This voter identity will be present in the table with received votes that TTPI has received from SURFnet if it concerns a voter that has cast a vote. TTPI could try several combinations until a combination does not occur in the table with received votes. The key used to generate this combination would then be unused. Adding a vote using such a key guarantees that this is a valid vote. This fraud is detected by the particular voter, because the table with received votes now contains a vote that he did not cast. SURFnet will find that the two tables differ.

Fifth, a valid vote can be made invalid by TTPI by adding a vote with the same election identifier and same key, but different candidate identifier. Again, TTPI needs to determine the set of valid votes applying the same procedure used normally to do so. In this model one such vote is selected in an non-deterministic way using the sum operator. Subsequently, the key is determined using a similar strategy in finding an unused key. TTPI has to find out which key is used in the vote selected. Therefore, TTPI combines every key with the election identifier and computes its hash until the combination matches the part of the vote that represents the voter identity. That key will then be used to cast a vote. TTPI could now add two votes using this key for two different candidates. This will make all votes using that same key invalid, including the previous valid vote, cast by the eligible voter. Another strategy for TTPI is to find out what candidate the voter voted for. In that case only one vote is needed. As TTPI is capable of determining this in the same way the key is revealed, I have chosen to model this. This time TTPI will use the key discovered and the candidate table, that is part of TTPI's knowledge as this party has used it before when computing the ballot collection for every voter before the election. TTPI tries every candidate identifier from the candidate table it distributes using the key discovered until it matches the second part of the technical vote: the encrypted candidate identity. For this fraud also goes that it is detectable by the particular voter and SURFnet. The voter will find that his vote has been received but has been marked invalid as his key has been used to cast another vote for another candidate. SURFnet will note a difference between the table with received votes this entity collected and the table which TTPI has published.

Sending a vote using an arbitrary key during the election with the chance that the owner will also cast a vote seems to have more potential for TTPI than adding votes after the election is closed. Moreover, the sending of an arbitrary vote can likewise result in the adding of a valid vote and making a valid vote invalid. However, the adding of votes afterwards are modeled here as it lies in TTPI's ability to do so and to prove explicitly that this strategy does not work either.

Here, the two are modeled as fraud afterwards as TTPI seems to have no access to the received votes during the election, but maybe someone can manage to commit this fraud near the end of the election. It still holds that the voter is able to detect it.

Sixth and seventh, TTPI can change the count for a certain candidate. It can either increase or decrease the count associated with a particular candidate by changing the number of votes presented in the count table online. It is assumed that the two types mostly occur at the same time to favour a particular candidate to the expense of another.

This can only be carried through if the count can be decreased, that is, is not zero. To allow verification for the situation in which the count is equal to zero the scenario in which the count is only increased is used next to the scenario that models both increasing and decreasing. In both scenarios the fraud is only detected if someone performs the count procedure and compares its result with the count published by TTPI. In my model SURFnet will recompute the count and will obviously be the one who will detect this fraud.

Assuming that this recount is always performed this fraud will not have much chance to succeed either.

Eighth, TTPI can assign certain votes to his favourite candidate during the count. In the model this is implemented by using a different reference table. It is assumed that it is published. Obviously, an attacker is in favor for some candidate and would like to convert votes for another candidate to his favourite. All votes meant for candidate A will then be assigned to candidate B, for example. The voter will notice this when checking if his vote has been given to the candidate of his choice. The main goal here is to show that every voter is able to verify his choice and to detect fraud with this.

A recount by SURFnet will result in an equal output and hence, fraud is not noticed by this party.

Note that the reference table used by TTPI and SURFnet is the same in this scenario. The MD5 hash will not have changed and cannot signal that is has been corrupted from the beginning. To detect this one should note that an unusual number of votes are linked to the same candidate. If one does not the voter should notice the incorrect referencing, given that the voter will check his vote. As the change that none of them will is very small this fraud seems of low risk.

Note that if an altered method of assigning is applied without publishing these different assignments, a recount by SURFnet will reveal the fraud. A check of his vote by a voter will not.

This fraudulous action seems to be a popular one as it is mentioned in almost every article on this subject. However, it is detectable in both the situation that the altered references are and are not made public.

Combining removing and adding a vote results in altering a vote. No separate scenario is used to verify that altering a vote will be detected as it consists of detecting removal and adding of a vote. Altering a vote is assumed to be detected as the separate actions have been verified to be detected as well.

Specified In this paragraph the algebraic specifications of the processes that differ from those presented in the general scenario are discussed.

Anonymity The two scenarios used for proving the anonymity property only differ from the general scenario in the undeterministic assignment and choice of respectively the keys and the candidates.

To enable proving anonymity it is necessary to deterministically provide two Voters with a different pair of Key X Candidate Identifier in two runs the sum operator should be omitted in distributing. Therefore, the process *TTPI_sendBallot* now only consists of the deterministic distribution of ballots followed by the execution of *SURFnet_receive*

from where the specification continues as normal. One process is sent a ballot using the communication *send_ballot1* and the other using *send_ballot2*. For one specification this results in :

$$
\begin{aligned}
&TTPI\_sendBallot(\quad keytable : keyTable, ei : ElectionId,\\
&\qquad\qquad\qquad\quad reftable : referenceTable, ctable : candidateTable,\\
&\qquad\qquad\qquad\quad cntstable : countsTable) =\\
&(send\_ballot1(Key1, ei, ctable(CI1, ect)) \quad .\\
&send\_ballot2(Key2, ei, ctable(CI2, ect))\\
&+\\
&send\_ballot2(Key2, ei, ctable(CI2, ect)) \quad .\\
&send\_ballot1(Key1, ei, ctable(CI1, ect)))\\
&. \quad SURFnet\_receive(empty, reftable, cntstable)
\end{aligned}
$$

Before discussing the processes being altered when fraud is committed one more assumption should be clarified.

If fraud cannot be committed due to the arbitrariness of the course of the specification it is run as normal to ensure that properties will always hold and to simulate the behaviour of attackers in practice instead of ending it. For example, if not one valid vote has been received an attacker cannot remove a valid vote. The election will then take place without this fraud and i.e. the property that ensures that every voter is able to check its vote will hold. The latter is true for every specification but can only be tested for its truthfulness if the specification is run totally as the check can be performed only after execution of the *TTPI_count* process. The same goes for other properties and other fraudulous actions.

For the same reason checks are performed even after a fraudulous action has already been detected. As only one fraudulous action is active in the specifications no other actions will be found. However, this is an assumption which should be proven. Instead, checks continue after detecting. This also enables a specification that does models two or more fraudulous actions.

Next, the detailed specifications of the attackers scenarios are presented. The order differs somewhat from the one introducing the models. Here, the models are ordered by the adjustments needed to implement them. This is to avoid repetition as some concepts are reused.

changeRef In one scenario TTPI commits fraud by using an altered reference table that is made public to be used by everyone. The corresponding specification differs from the normal specification at only one point: the *ref* function that simulates use of the reference table by TTPI, SURFnet and the voter. Normally it is defined as follows:

$$ref(MDC(MAC(key1, ei), MAC(key1, ci1))) = ci1$$

But in the changeRef specification one candidate identifier is falsely linked to another. The remaining references stay unchanged.

In this case CI1 is linked to CI2, whereas CI2 stays linked to CI2, as usual:

$$
\begin{aligned}
ref(MDC(MAC(key1, ei), MAC(key1, CI1))) &= CI2\\
ref(MDC(MAC(key1, ei), MAC(key1, CI2))) &= CI2
\end{aligned}
$$

After determining the tally result using this reference, the *TTPI_count* process will execute the website process as normal, but only after an action changeRef is issued. This action is later used in modal logic to enable proving that this fraud can be detected.

removeRec As SURFnet manages the server that receives all votes this party can remove votes before passing the received set over to TTPI. Using the publicly available reference table SURFnet can determine for which candidate the vote is meant by computing its MDC value. In this specification, this is omitted. Here, SURFnet will simply remove a vote from the received set arbitrarily if the table is not empty. The *SURFnet_receive* sub-process that simulates this fraud is almost equal to the normal sub-process. It differs in that an action *removeRec* is issued for modal checking reasons and that the *TTPI_count* process is parametrized with the received set minus the vote removed.

The vote is non-deterministically selected by first selecting an EVI from the received votes and then selecting an ECI from the set of votes consisting of this EVI, using *elementV*, *elementC* and *evichoice*. This work-around is not needed in practice, but it is in this specification.

$$
\begin{aligned}
&SURFnet\_receive(\quad receivedVotes : Table, reftable : referenceTable,\\
&\quad cntstable : countsTable) =\\
&\\
&(\textstyle\sum_{eci:EncryptedCandidateId} \sum_{evi:EncryptedVoterId}\\
&\\
&receive\_vote(evi, eci) \quad .\\
&\\
&SURFnet\_receive(add(evi, eci, receivedVotes), reftable, cntstable)\\
&\\
&+\\
&\\
&timeout \quad) \quad .\\
&\\
&((\quad \textstyle\sum_{evi:EncryptedVoterId} \sum_{eci:EncryptedCandidateId}\\
&\\
&\quad removeRec(evi, eci) \quad .\\
&\\
&\quad TTPI\_count(delete(evi, eci, receivedVotes), reftable, cntstable)\\
&\\
&\quad \triangleleft \quad and(elementV(evi, receivedVotes), elementC(eci, evichoice(evi,\\
&\quad receivedVotes, empty))) \quad \triangleright\\
&\\
&\quad \delta)\\
&\\
&\triangleleft \quad notEmpty(receivedVotes) \quad \triangleright\\
&\\
&TTPI\_count(receivedVotes, reftable, cntstable) \quad)
\end{aligned}
$$

The corrupt party will not perform a check to match its own received set with TTPI's

as it would reveal its own fraudulous actions. SURFnet can however do a recount to discover possible incorrectnesses in the count procedure as performed by TTPI. This enables verifying properties stated about the count.

As this check usually follows the check on the received set this process is slightly altered. However, it does remain a parallel process to avoid deadlock and is denoted with *SURFnet_checkCnt* here.

First, the checkinfo is communicated and then the check is performed. If a difference is found *SURFnet_getCorrupt* is executed, like before.

$$
\begin{aligned}
&SURFnet\_checkCnt = \\
&\sum_{receivedVotes:MDCTable} \sum_{validVotes:MDCTable} \sum_{countstable:countsTable} \\
&\quad \sum_{reftable:referenceTable} \sum_{initCnts:countsTable} \\
&receive\_checkInfoSURFnet(receivedVotes, validVotes, countstable, \\
&\quad reftable, initCnts) \quad . \\
&(OK(TTPI\_countstable) \quad . \quad \delta \\
&\lhd \quad eq(count(mdcclean(receivedVotes, reftable), initCnts), countstable) \quad \rhd \\
&SURFnet\_getCorrupt(count(mdcclean(receivedVotes, reftable), initCnts), \\
&\quad countstable) \quad) \quad)))))
\end{aligned}
$$

removeValid The entity TTPI also has the opportunity to remove a valid vote from the received set before sending the tally results to the website. The vote is selected non-deterministically by using the sum operator to first select an *encrypted voter identity* with the function *elementV* from the valid set. As the second part of the vote, the *encrypted candidate identity*, should contain the same key as the first part, it is selected from those valid votes that consist of the EVI just selected and saved in the empty table using the *evichoice* and then *elementC* function. The resulting vote is subsequently *delete*d from the received and valid set, both arguments of the *Website_sendCheckInfo* sub-process. For modal logic reasons the action *removeValid* is issued first.

This fraud can only be committed when valid votes exist. This is determined with the *notEmpty* function. If valid votes exist, the fraud is committed as stated above. If no valid votes exist, the *Website_sendCheckInfo* is executed as normal to allow the protocol to finish and to enable model checking particular properties.

$$
\begin{aligned}
&TTPI\_count(\quad receivedVotes : Table, reftable : referenceTable,\\
&\quad cntstable : countsTable) =\\
&\\
&\quad (\textstyle\sum_{evi:EncryptedVoterId} \sum_{eci:EncryptedCandidateId}\\
&\\
&\quad removeValid(evi, eci) \quad .\\
&\\
&\quad Website_s\, endCheckInfo(\\
&\qquad computemdc(delete(evi, eci, receivedVotes)),\\
&\qquad computemdc(clean(delete(evi, eci, receivedVotes), reftable)),\\
&\qquad count(computemdc(clean(delete(evi, eci, receivedVotes),\\
&\qquad\quad reftable)), cntstable),\\
&\qquad reftable, cntstable)\\
&\\
&\quad \triangleleft \quad and(elementV(evi, clean(receivedVotes, reftable)),\\
&\qquad elementC(eci, evichoice(evi, clean(receivedVotes, reftable),\\
&\qquad\quad empty))) \quad \triangleright\, \delta)\\
&\\
&\triangleleft \quad notEmpty(clean(receivedVotes, reftable)) \quad \triangleright\\
&\\
&Website\_sendCheckInfo(\quad computemdc(receivedVotes),\\
&\qquad computemdc(clean(receivedVotes, reftable)),\\
&\qquad count(computemdc(clean(receivedVotes,\\
&\qquad\quad reftable)), cntstable),\\
&\qquad reftable, cntstable)
\end{aligned}
$$

makeInvalid In this scenario TTPI commits fraud by making a valid vote invalid. This is achieved by adding a vote by using the same key but another candidate identifier as the original one. Obviously, this fraud can only occur when valid votes are present. The function *notEmpty* guards this precondition. If no valid votes exist, the *Website_sendCheckInfo* is executed as normal.

To commit this fraud a keytable, a candidate identifier and the election identifier is needed. The keytable should be destroyed after distribution, but TTPI could have saved this knowledge illegally.

Normally, the candidate table, keytable and election identifier will no longer be part of the knowledge at this stage. Therefore, in this fraud specification this knowledge is saved by TTPI before distributing the ballot using the action *saveFraudInfo*.

$$
\begin{aligned}
&TTPI\_sendBallotCORRUPT(\quad keytable : keyTable, ei : ElectionId,\\
&\quad reftable : referenceTable, ctable : candidateTable,\\
&\quad cntstable : countsTable) =\\
&\\
&saveFraudInfo(ctable, keytable, ei) \quad .\\
&TTPI\_sendBallot(vtable, keytable, ei, reftable, ctable, cntstable)
\end{aligned}
$$

This process replaces *TTPI_sendBallot* in the init section and executes *TTPI_sendBallot* after saving the information.

A parallel process *FraudInfoMem* functions as the memory of TTPI that recalls this information when *TTPI_count* issues a communication with its memory process. Here, *saveFraudInfo* communicates with *keepFraudInfo*. In the state space this communication is denoted by *comm_storeFraudInfo*.

It is designed to be a parallel process to divide knowledge which keeps it from other innocent consecutive processes.

$$
\begin{aligned}
&FraudInfoMem = \\
&\textstyle\sum_{ctable:candidateTable} \sum_{keytable:keyTable} \sum_{ei:ElectionId} \\
&keepFraudInfo(ctable, keytable, ei) \quad . \\
&sendFraudInfo(ctable, keytable, ei) \quad . \quad \delta
\end{aligned}
$$

The information is recalled when the action *sendFraudInfo* communicates with *recFraudInfo* in the *TTPI_count* sub-process to *comm_recallFraudInfo*.

These processes and actions are also used in this exact form when modeling the adding of a valid vote and increasing or changing the count.

The vote that will be made invalid here is selected non-deterministically by using the sum operator to first select an *encrypted voter identity* with the function *elementV* from the valid set. As the second part of the vote, the *encrypted candidate identity*, should contain the same key as the first part, it is selected from those valid votes that consist of the EVI just selected and saved in the empty table using the *evichoice* and then *elementC* function.

To be able to cast a vote that differs from this one the key and the candidate identifier should be determined somehow. TTPI can compute all possible votes using the keys, the candidate identifiers and the election identifier. The function *getKey* will simulate this by comparing all combinations to the EVI and ECI part of the original vote. When they correspond the combination must have been build with the same key as the original vote. The function *getci* determines the used candidate identifier in the same way, which is removed from the set of candidate identifiers using the *del* function. Subsequently, this key is used to add a vote for an arbitrary candidate identifier selected from this set. The MAC value of the vote is added to the received set with the function *makeInvalid* that uses the *add* function. This will make the original and this new vote both invalid. As the valid set and the count are both based on the received votes these are altered too before passing these arguments to the website.

To make model checking easier and more powerful the action *makeInvalid* is issued first, which is parametrized with the vote being added and not the vote being made invalid. This is because in this specification the vote added to invalidate this vote is detected. In modal logic the advantage is to address the same vote in both actions.

$$
\begin{aligned}
&TTPI\_count(\quad receivedVotes : Table, reftable : referenceTable,\\
&\quad cntstable : countsTable) =\\
&\\
&\textstyle\sum_{ctable:candidateTable} \sum_{keytable:keyTable} \sum_{ei:ElectionId}\\
&\\
&recFraudInfo(ctable, keytable, ei) \quad .\\
&(\\
&\quad (\textstyle\sum_{evi:EncryptedVoterId} \sum_{eci:EncryptedCandidateId} \sum_{ci:CandidateId}\\
&\\
&\quad makeInvalid(MAC(getkey(evi, eci, keytable, ei), ei),\\
&\qquad MAC(getkey(evi, eci, keytable, ei), ci)) \quad .\\
&\\
&\quad Website\_sendCheckInfo(\\
&\qquad computemdc(makeInvalid(getkey(evi, eci, keytable, ei), ei, ci,\\
&\qquad\quad receivedVotes)),\\
&\qquad computemdc(clean(makeInvalid(getkey(evi, eci, keytable, ei), ei, ci,\\
&\qquad\quad receivedVotes), reftable)),\\
&\qquad count(computemdc(clean(makeInvalid(getkey(evi, eci, keytable, ei),\\
&\qquad\quad ei, ci, receivedVotes), reftable)), cntstable),\\
&\qquad reftable, cntstable)\\
&\\
&\quad \triangleleft \quad and(elementV(evi, clean(receivedVotes, reftable)), and(elementC(\\
&\qquad eci, evichoice(evi, clean(receivedVotes, reftable), empty)), element(\\
&\qquad ci, del(getci(evi, eci), ctable)))) \quad \triangleright\\
&\\
&\quad \delta)\\
&\\
&\triangleleft \quad notEmpty(clean(receivedVotes, reftable)) \quad \triangleright\\
&\\
&Website\_sendCheckInfo(\\
&\qquad computemdc(receivedVotes),\\
&\qquad computemdc(clean(receivedVotes, reftable)),\\
&\qquad count(computemdc(clean(receivedVotes,\\
&\qquad\quad reftable)), cntstable),\\
&\qquad reftable, cntstable) \quad)
\end{aligned}
$$

addValid A valid vote can be added to the received set by TTPI using an unused key. Again, the keytable, candidate identifiers and the election identifier is illegally recalled from memory. The need for an unused key implicates that this form of fraud cannot be committed if every key has already been used. The *notEmpty* function determines if the set of unused keys formed with the function *getUnusedKeys* is empty or not. If it is empty the website sub-process is executed as normal. If an unused key exists it is selected from this set non-deterministically together with a candidate identifier. Together they are used to form a vote that is added with the *add* function to the received and valid set after the

addValid action is issued that is reused in modal logic.

$$
\begin{aligned}
&TTPI\_count(\quad receivedVotes : Table, reftable : referenceTable,\\
&\quad cntstable : countsTable) =\\
&\textstyle\sum_{ctable:candidateTable} \sum_{keytable:keyTable} \sum_{ei:ElectionId}\\
&\\
&recFraudInfo(ctable, keytable, ei) \quad .\\
&\\
&(\textstyle\sum_{key:Key} \sum_{ci:CandidateId}\\
&\\
&\quad addValid(MAC(key, ei), MAC(key, ci)) \quad .\\
&\\
&\quad Website\_sendCheckInfo(\\
&\qquad computemdc(add(MAC(key, ei),\\
&\qquad\quad MAC(key, ci), receivedVotes)),\\
&\qquad computemdc(clean(add(MAC(key, ei), MAC(key, ci),\\
&\qquad\quad receivedVotes), reftable)),\\
&\qquad count(computemdc(clean(add(MAC(key, ei), MAC(key, ci),\\
&\qquad\quad receivedVotes), reftable)), cntstable),\\
&\qquad reftable, cntstable)\\
&\\
&\quad \triangleleft \quad and(element(key, getUnusedkeys(keytable, receivedVotes, ei)),\\
&\qquad element(ci, ctable)) \quad \triangleright\\
&\\
&\quad \delta)\\
&\triangleleft \quad notEmpty(getUnusedkeys(keytable, receivedVotes, ei)) \quad \triangleright\\
&\\
&Website\_sendCheckInfo(\\
&\qquad computemdc(receivedVotes),\\
&\qquad computemdc(clean(receivedVotes, reftable)),\\
&\qquad count(computemdc(clean(receivedVotes,\\
&\qquad\quad reftable)), cntstable),\\
&\qquad reftable, cntstable)
\end{aligned}
$$

sendVote TTPI can also commit fraud by pretending to be a voter using a key from the keytable. The vote to be sent is select non-deterministically by using the sum operator and *element* function to determine the candidate identifier and the key from respectively the candidate table and the keytable.

Again, these tables and the election identifier are not part of its knowledge at this stage and should be obtained from its memory. As the process to cast a vote is already a parallel one this information comes directly from *TTPI_sendBallotCORRUPT*. There is no need for a parallel memory process as used before. Thus, the communicating actions here to retrieve it are *saveFraudInfo* and *getFraudInfoTTPI*, that are together represented by *comm_FraudInfo*. Subsequently, the process waits until the SURFnet process is ready to communicate its vote and terminates with δ.

As modeled here, this party will not check what has happened to his vote like the voter will, although in practice it is exposed to fraudulous actions of other parties.

Only the naming of the send action used by TTPI differs from the voter's: *send_voteCORRUPT*, which communicates with *receive_vote* at the server's side to *comm_voteCORRUPT*. In practice, however, no difference can be observed. Here, it is necessary to distinguish between fraudulous and eligible voting as TTPI will not check his vote and therefore, i.e. the property that states that every vote is followed by a check will fail. Furthermore, the action is in property statements about this fraud.

$$
\begin{aligned}
&TTPI\_castCORRUPT = \\
&\textstyle\sum_{ctable:candidateTable} \sum_{keytable:keyTable} \sum_{ei:ElectionId} \\
&getFraudInfoTTPI(ctable, keytable, ei) \quad . \\
&\textstyle\sum_{ci:CandidateId} \sum_{key:Key} \\
&send\_voteCORRUPT(MAC(key, ei), MAC(key, ci)) \quad . \quad \delta \\
&\lhd \quad and(element(ci, ctable), element(key, keytable)) \quad \rhd \\
&\delta
\end{aligned}
$$

IncreaseCount Fraudulously modifying the counts table only affects the sub-process *TTPI_count*. It will differ in its arguments that are passed to the website. The count will be altered for some candidate identifier, that is non-deterministically selected using the sum operator applied to elements of the counts table.

Subsequently, the candidate identifier is non-deterministically selected and its count is increased in the counts table using the function *increase*. An *addcount* action is applied here for use in the modal logic properties to prove that fraud is detected. Note that a count can always be increased.

$$
\begin{aligned}
&TTPI\_count(\quad receivedVotes : Table, reftable : referenceTable, \\
&\quad cntstable : countsTable) = \\
&\textstyle\sum_{ci:CandidateId} \quad addcount(ci) \quad . \\
&Website\_sendCheckInfo(\\
&\quad computemdc(receivedVotes), \\
&\quad computemdc(clean(receivedVotes, reftable)), \\
&\quad increase(ci, count(computemdc(clean(receivedVotes, reftable)), \\
&\qquad cntstable)), reftable, cntstable) \\
&\lhd \quad element(ci, cntstable) \quad \rhd \quad \delta
\end{aligned}
$$

changeCount To change the count by increasing one count and decreasing another a slightly different *TTPI_count* process is designed. As decreasing a count can only occur when some count is non-zero a conditional expressed with the *notEmpty* function is used to decide if the published table contains any valid votes. If it does not, all counts would be zero and none of them can be decreased. This process would then continue executing the website sub-process as normal. If the table containing all valid votes is not empty, at least one count should be non-zero. In that case, both a count that can be decreased and another count is non-deterministically selected using the sum operator to increase. The first is selected using the *voted* function that determines if a valid vote exists for some candidate identifier and the second should be an *element* of the counts table minus the candidate identifier decreased. Before executing the website process, parametrized with a counts table that has been altered by the functions *increase* and *decrease*, both a *subcount* and an *addcount* action is issued for use in modal logic.

$$
\begin{aligned}
&TTPI\_count(\quad receivedVotes : Table, reftable : referenceTable,\\
&\quad cntstable : countsTable) =\\
&\\
&\quad (\textstyle\sum_{ci:CandidateId} \sum_{ci2:CandidateId}\\
&\\
&\quad subcount(ci) \quad . \quad addcount(ci2) \quad .\\
&\\
&\quad Website\_sendCheckInfo(\\
&\qquad computemdc(receivedVotes),\\
&\qquad computemdc(clean(receivedVotes, reftable)),\\
&\qquad increase(ci2, decrease(ci, count(computemdc(clean(receivedVotes,\\
&\qquad\quad reftable)), cntstable))), reftable, cntstable)\\
&\quad \triangleleft \quad and(voted(ci, count(computemdc(clean(receivedVotes, reftable)),\\
&\qquad\qquad cntstable)), element(ci2, remove(ci, cntstable))) \quad \triangleright\\
&\quad \delta)\\
&\\
&\triangleleft \quad notEmpty(clean(receivedVotes, reftable)) \quad \triangleright\\
&\\
&Website\_sendCheckInfo(\\
&\quad computemdc(receivedVotes),\\
&\quad computemdc(clean(receivedVotes, reftable)),\\
&\quad count(computemdc(clean(receivedVotes,\\
&\qquad reftable)), cntstable), reftable, cntstable)
\end{aligned}
$$

With this specification the discussion of process behaviour and some high-level functions ends. Please remember that functions presented in this section might use subfunctions. The complete muCRL specification can be examined in Appendix A.

8 Properties and Verification results

In this section the results obtained from the formal analysis of the protocol concerning the stated properties are described. The formalization of the stated properties and the conclu-

sion of verification are presented. The properties are expressed in regular alternation-free μ-calculus and are briefly explained.

A thorough analysis will be complicated due to the state explosion problem. Besides the software limitation, time constraints also form a barrier. Therefore, the analysis performed here considers a small number of concurrent sessions of the protocol and not all exceptions and attacks are modeled. Although this cannot constitute a theoretical proof of correctness and security for a protocol, correctness of the protocol for small configurations does provide confidence that the protocol is free of flaws.

The analysis presented here is fully automatic and the verification algorithms do not need human intervention.

The formal analysis considers nine scenarios designed to show that some properties are preserved and some fail under different circumstances. Most properties have been proven correct for more than one model as they should; which will be explained later. Due to fraud obviously not all properties hold in some of the scenarios modeling some type of fraud. The models distinguish themselves in the type of fraud that is committed. Fraud is introduced here, because the protocol implicates that the voter can always verify what has happened to his and other votes. This would mean that the voter can find out if his vote has been received, if it has been marked valid, if it has been assigned to the intended candidate and if the count has been performed correctly. Respectively, the voter could detect removal of his vote, adding of a vote using his key, incorrect referencing and incorrect counting. The latter can be detected by performing a recount by anyone. This task is assigned to SURFnet in this formal specification. As it is able to, this party is also assumed to check if the table containing the received votes published by TTPI contains precisely those votes that have been received by SURFnet. These verifiability issues are inspected for situations that will put this to the test. The simplest model does not allow fraud. The other scenarios all model one type of fraud that is only committed once as discussed.

I would like to point out in general that the models are model checked concerning these properties for two situations. One concerns two voters and two candidates with no duplicate votes allowed. The other models one voter and two candidates with a maximum of second duplicate votes allowed. In the latter situation the removal and adding of second duplicates are detected like normal votes. Allowing duplicates means that a maximum of two identical votes are saved by both the voter as SURFnet. When duplicates are not allowed, this is to say that the voter and SURFnet only store differing votes and if duplicates are concerned only one 'duplicate' is saved.

In both situations it is allowed to cast an arbitrary number of identical votes. However, if only one is stored reasoning with duplicates will not be necessary although they have been cast. If two duplicates are stored, one of them is judged valid and the other is invalid. Which one will be judged invalid is arbitrary. Note that both sides should follow the same saving rules here to enable a correct check. For example, if the voter saves two duplicates and checks if they both have been received while the server only stores one the check will fail.

A valid vote in this sense is one that appears in the reference table, is not accompanied by votes for different candidates using the same key and if it concerns a duplicate only one is judged valid. Note that it does not matter which duplicate is judged valid. A valid

vote is signalled by an OKv action in my formal specification. Remember that in case of duplicates one is judged valid.

It should be noted that the OKv actions corresponding to duplicates are identical as the action is parametrized with the same vote. This also goes for the OKr actions that may occur several times, in contrast to identical OKv actions. Despite the fact that the OKr actions for duplicates cannot be distinguished the voter process can still determine if a second duplicate vote has been added or removed. Every time his cast vote has a match in the table with received votes, this vote is removed from both sets. The next time the same vote is checked it will only have a match if a duplicate has been received. If this succeeds, both identical votes have indeed been received.

In the property statements it is not possible to apply the same trick and therefore, the OKr actions of the first and second duplicate cannot be kept apart. To illustrate this, take a situation in which some voter casts two duplicate votes and checks if they both have been received after the election is closed. This means that both votes should eventually be followed by a corresponding OKr action. A general statement that every vote is followed by a corresponding OKr action does not do the job here, as the corresponding OKr action is the same for duplicates. In that case the specification would only verify that *a* corresponding OKr action follows and not a particular one. It leaves the option open that one of the votes does not have a corresponding vote in the table with received votes. This could be solved by adding an additional identifier in the case of duplicates but this complicates things and such practice would deviate from the original protocol. Therefore, it is checked if the casting of duplicates is followed by two identical OKr actions.

Moreover, correct handling of duplicates is also implicated by verifying that removal and adding of such duplicates is detected.

Some properties have only been verified for the situation that does not take second duplicates into account to simplify the matter. It regards the properties that state something about the set of received votes. The formulas are quite lengthy when formulated for two voters and two candidates. They will grow even more when issuing duplicate votes. Therefore, it is assumed that if the property holds for this case, it also holds for the situation where double votes are taken into consideration. To be specific, this concerns the properties D_3 : *receivedSet*, A_1 : *receivedSet_CR* and A_9 : *receivedSet_HIT*.

It is my believe that this little adjustment will not affect the value of verification of this protocol in general.

8.1 Macros

Voting protocols are designed to meet particular properties. In order to analyse such protocols formally, it is necessary to provide a formal definition of the property that they are intended to provide.

For the sake of readability macros have been used to state the properties here.

Note that the following macros are not necessarily an exact match of those used in the CADP toolset. They have been categorized.

Casting a vote In this text the macro $vote_{ij}$ is used to mean the casting of a vote by a voter using key i for candidate j and $i, j \in \{1, 2\}$. Other letters are used as indices as well,

in particular the set $\{i, k, p, r\}$ is used to indicate keys and the set $\{j, l, q, s\}$ is reserved for indexing candidates. If not stated otherwise the vote is encrypted using the correct election identifier. The use of indices to represent a particular vote is not meant to imply that the contents of the vote (the key and the candidate) are known. It is used to a point out a certain encrypted vote that distinguishes itself from the encrypted forms of other votes.

Furthermore, the convention is followed for the $\leq$ ordering. Thus, $ij \leq kl$ means that $i < k$ or that $i = k$ and $j < l$.

The macro $validvote_{ij}$ encompasses several actions as a vote to be a valid one should not be preceded or followed by votes using the same key and another candidate. In case of casting duplicates one vote remains valid. Therefore, a valid vote is one that has the following structure:

$(not(vote_{il}))^*.(vote_{ij})^+.(not(vote_{il}))^*$ *with* $i, j, l \in \{1, 2\}$ *and* $j \neq l$.

A variation on this macro is a macro like $validVote_{i1}$, meaning that the candidate is the constant 1 and as before $i \in \{1, 2\}$.

Positive verification The macro OKr_{ij} is used to represent the action as used by a voter process to indicate that a vote encrypted with key i and candidate j has been received.

With the macro OKv_{ij} the action is meant that is issued by a voter process confirming that his vote using key i for candidate j and $i, j \in \{1, 2\}$ has been marked valid.

A slightly different one is OKv_{ijc}. Here, the same action is meant but now the confirmed vote contains an invalid election identifier.

$OKci_j$ represent the action that signals that a particular vote has correctly been assigned to candidate j. In muCRL this action is parametrized with this vote, but here this is omitted.

The macro $OKrec_{empty}$ is used to mean that an empty received table published by TTPI has positively been matched by SURFnet, signalled by the action OKrec with the votes as arguments.

Of course, other macros are needed to indicate that one or more particular votes have been received and that this set has been verified:

$OKrec_{(ij)}$ says something about a table containing a vote using key i for candidate j and $i, j \in \{1, 2\}$.

$OKrec_{(ij,kl)}$ makes the OKrec statement about a table containing a vote using key i for candidate j and a vote using key k for candidate l and $i, j, k, l \in \{1, 2\}$ and ij $\neq$ kl.

$OKrec_{(ij,kl,pq)}$ represents an OKrec action for a table containing a vote using key i for candidate j, a vote using key k for candidate l and a vote using key p for candidate q with $i, j, k, l, p, q \in \{1, 2\}$ and $ij < kl < pq$.

In the same way $OKrec_{(ij,kl,pq,rs)}$ is used to mean the OKrec action for a table containing a vote using key i for candidate j, a vote using key k for candidate l, a vote using key p for candidate q and a vote using key r for candidate s with $i, j, k, l, p, q, r, s \in \{1, 2\}$ and $ij < kl < pq < rs$.

The next macro to discuss is the OKcnt action that is issued by SURFnet to indicate that a particular count published by TTPI is correct. The following structure is used with

n,m representing the number of votes counted for respectively candidate 1 and candidate 2.

OKcnt1n2m with $n, m \in \{z, s, ss\}$ *and* $z = zero, s = 1, ss = 2$.

Negative verification or detection of fraud In this section the macros are briefly explained that represent signalling some kind of fraud concerning a particular candidate identifier (CI) or vote by SURFnet and the voter.

First to mention are *SURFnet_detectAdd(*CI_n*)* and *SURFnet_detectRemove(*CI_n*)* with $n \in \{1, 2\}$. The first signals the increasing of the count and the latter the decreasing of the count for candidate n.

Both SURFnet and the voter can detect the adding of a vote using key i intended for candidate j to the set of received votes:

SURFnet_detectAdd$_{ij}$ and *Voter_detectAdd*$_{ij}$ with $i, j \in \{1, 2\}$.

The same goes for detecting the removal of such a vote:

SURFnet_detectRemove$_{ij}$ and *Voter_detectRemove*$_{ij}$ with $i, j \in \{1, 2\}$.

The last macro in this category is the detection of not assigning a particular vote to candidate j: *Voter_detectci*$_j$ with $j \in \{1, 2\}$. Again, in muCRL this action is parametrized with the vote.

Communication of information needed to perform a check The voter and SURFnet both have access to the same information published that is needed to check the status of a particular vote, the table with received votes as a whole and the count. All information is communicated between the website and the voter and the website and SURFnet, although not all information is needed for every check sort. The contents of that information will not matter and therefore wild cards are used in the CADP toolset to allow these parts to be arbitrary. To verify certain properties it is needed to explicitly specify specific parts of that information in modal logic, dependent on the part the property is interested in. The macro name outlines these parts.

To outline the communication of the check information to SURFnet including a particular count for the two candidates the following structure is used. The indices n, m represent the number of votes counted for respectively candidate 1 and candidate 2.

checkInfoCnt1n2m with $n, m \in \{z, s, ss\}$ *and* $z = zero, s = 1, ss = 2$.

In the case of communication with a voter the action and macro is preceded by "Voter": i.e. *voter_checkInfoCnt1n2m*.

For the properties that say something about the set of received votes that is published by TTPI the following macro is used for SURFnet:

checkInfoRec$_{(empty)}$ is used to indicate that an empty table is published.

checkInfoRec$_{(ij)}$ includes a table containing a vote using key i for candidate j and $i, j \in \{1, 2\}$.

checkInfoRec$_{(ij,kl)}$ includes a table containing a vote using key i for candidate j and a vote using key k for candidate l and $i, j, k, l \in \{1, 2\}$ and $ij < kl$.

checkInfoRec$_{(ij,kl,pq)}$ includes a table containing a vote using key i for candidate j, a vote using key k for candidate l and a vote using key p for candidate q with $i, j, k, l, p, q \in \{1, 2\}$ and $ij < kl < pq$.

$checkInfoRec_{(ij,kl,pq,rs)}$ includes a table containing a vote using key i for candidate j, a vote using key k for candidate l and a vote using key p for candidate q with $i, j, k, l, p, q, r, s \in \{1, 2\}$ and $ij < kl < pq < rs$.

Fraud actions In committing and detecting fraud particular votes and candidates are concerned. To be able to say something about these particular entities in certain properties they are outlined in the actions that signal that fraud is being committed as well.

The macro $voteCORRUPT_{ij}$ represents the casting of a vote for candidate j by TTPI using key i with $i, j \in \{1, 2\}$.

The action $removeRec_{ij}$ is used to indicate that SURFnet removes a vote for candidate j that is encrypted with key i and $i, j \in \{1, 2\}$.

The removal of a valid vote for candidate j using key i by TTPI is described by the following macro: $removeValid_{ij}$ with $i, j \in \{1, 2\}$.

The same goes for adding a valid vote by TTPI: $addValid_{ij}$ with $i, j \in \{1, 2\}$.

and making a vote invalid by adding a vote for candidate j using key i represented by the macro $makeInvalid_{ij}$ with $i, j \in \{1, 2\}$.

The action of decreasing and increasing the count for candidate n by TTPI is indicated by respectively the macros $subcount(CI_n)$ and $addcount(CI_n)$ with $n \in \{1, 2\}$.

The last action to discuss in this section is *changeRef* to denote the action of using an altered reference table to achieve assigning votes to some favourite candidate by TTPI. No macro is needed here.

Other The action *timeout* is used to simulate that the election is closed. Communication with SURFnet is no longer possible. Voter processes can check their vote as soon as the election is closed. Therefore, the timeout action is needed in some modal logic formulas to indicate that all voters have left their casting loop.

As usual the action $true^*$ represents zero or more random actions.

8.2 Results

This section presents the formalization of the stated properties in μ-calculus and their truth value. A truth value equal to TRUE means that the property holds for that particular scenario, whereas FALSE indicates that the property does not hold for that scenario.

The evaluator tool has been used to determine the truth value, but has been applied only after I used CADP's standard simulator to step through the specifications to ensure their correct operation. This has helped by identifying a particular instruction for use that I have experienced as a minus of the software: white space used in a composite action statement in a formula will always cause the formula to be evaluated to TRUE, as this white space is interpreted as part of the syntax of the action name.

Simulation also helped to assure that an implication does not result in a positive truth value only because the antecedent is always false.

For certain properties it will be shown that they do not hold as they should not. The requirement that some property is satisfied under particular circumstances implicates that this property is not always satisfied under other circumstances. In fact, for several properties stated here they are expected to fail for particular (fraud) scenarios. Let it

suffice to say that all properties should evaluate to TRUE, except for certain properties when a fraud scenario is concerned. The results of the verification of the properties will show to correspond to the expected truth values. This contributes to the conclusion that the protocol is correctly formalized and that the protocol is correct concerning these properties. For example, the sending of a vote should always be followed by the receival of that vote indicated by an OKr action. However, if that vote is fraudulously removed it should be the case that the sending of that vote is not followed by an OKr action. If it does, this could mean an incorrectness in the protocol or the formalization of it.

The properties formulated here are used to prove something for the situation in which at most one fraudulous act is performed. However, the formulas are also suitable to prove these properties when more than one fraud is allowed.

All properties have been categorized into the concepts validity (V), determinability (D), accurateness (A) and detectability (DC). The latter three are grouped under Verifiability. The anonymity property is treated separately; no properties in modal logic have been defined.

I will start with describing those properties that have been proven to be satisfied in all scenarios. Subsequently, the properties that hold in the nofraud scenario are discussed. Together these two sections cover all properties but the detectability properties. To prevent repetition the consecutive sections discuss only those properties that have been proven to be false. The only exception are the detectability properties. Every scenario is associated with at most two detectability properties that have been proven TRUE for that particular scenario only. This mode of operation leads to a number of properties that is quite smaller than when discussing all properties for all scenarios. In discussing these properties the ordering used in discussing the attacker models is maintained. It ends with the verification result of the anonymity property that deviates from the others as it concerns state space comparison.

8.2.1 All scenarios

Before turning to the discussion of the different scenarios and the truth values of the properties concerned, I will first list the properties that have been found to be true in every model.

Validity

Result V_0 : *invalidEI*
Votes that do not occur in the reference table are not marked valid. In the formal specification such a vote uses an invalid election identifier. To verify that such a vote is not taken into account it must be shown that it cannot be followed by an OKv action.

$$[true^*.OKv_{ijc}]false$$

Result V_1 : *doubleOnce*
Solely valid votes are counted. A vote is valid when it appears in the reference table and when votes using the same key for another candidate are not present. The latter is regarded here. Every such vote is counted only once, which will become apparent when

computing the final count. The table with received votes will contain also the duplicate votes. Only one of these is marked valid. In my formal specification this is simulated by constructing a new table containing all valid votes. This table does not contain duplicates. This means that such a mark will thus only be given to one vote belonging to a particular key, which is signalled by an OKv action, and that two or more OKv actions for this vote cannot exist.

$$[true^*.(OKv_{ij})^+.true^*.OKv_{ij}]false$$

Determinability

Result D_0 : *sendVote*
The RIES protocol makes it possible for a voter to find out what has happened with his vote. For just any vote a voter can conclude that it has indeed been received or has been removed, be it a valid, invalid, first or second duplicate one. Model checking this property should lead to the conclusion that a particular vote is in some situations followed by an OKr action and in others by a detectRemove. In case of duplicate votes no distinction in first, second or more order vote can be made.

$$[true^*.vote_{ij}."timeout"]mu\ X.(\langle true\rangle\ true\ and\ [not(OKr_{ij}\ or\ Voter\_detectRemove_{ij})]X)$$

Result D_1 : *validVote*
The voter should also be able to check what has happened with his valid vote. A valid vote is the sending of a vote, possibly more than once as in that case one of them will be judged valid. The sending of differing votes using the same key will make the vote invalid. A valid vote can remain valid, which is the case when no fraud takes place. It could also have been removed or it could have been made invalid due to the adding of another vote using the same key. Therefore, a valid vote is sometimes followed by a confirmation that it is valid, sometimes followed by detection of its removal and sometimes followed by the detection of the adding of a vote that makes the original one invalid.

$$[validvote_{ij}."timeout"]$$
$$mu\ X.(\langle true\rangle true\ and\ [not(OKv_{ij}\ or\ Voter\_detectRemove_{ij}\ or\ Voter\_detectAdd_{ik})]X)$$

Result D_2 : *choice*
The voter should also be able to verify that his vote has been assigned to the candidate of his choice. To be assigned the vote should be a valid one. It is assumed that the voter will not check the assignment of a vote that is not valid. The verification will lead to the conclusion that the vote has been assigned correctly, but in other situations to the detection of fraudulous assigning.

$$[validvote_{ij}."timeout"]mu\ X.(\langle true\rangle true\ and\ [not(OKci_j\ or\ Voter\_detectci_j)]X)$$

Both actions are parametrized with the vote being checked to differentiate between a check from another voter that has cast a vote for the same candidate.

Result D_3 : receivedSet
In my formal specification SURFnet will verify if the published table containing the received votes is equal to the table composed of the votes received by SURFnet. Such a verification can result in a match (represented by an OKrec action for that particular table) or in a mismatch. The mismatch can be caused by the removal or adding of a vote. The former is signalled by a *SURFnet_detectRemove* action for that vote and the latter by a *SURFnet_detectAdd* action. This property is verified for the model that assumes that no double votes exist to simplify the matter. It is assumed that if the property holds for this case, it also holds for the situation where double votes are taken into consideration. With two voters, two candidates and no double votes already sixteen possibilities exist. A distinction between tables with zero, one, two, three or four votes is to be made. This property is stated using a hybrid notation. In the case where only one situation exists, all instances are presented literally. In all others, indices $\{i, j, k, l\}$ with no relation between them assumed and $\{x, y, v, w\}$ are used to indicate an arbitrary instance, again with no relation between them assumed.

The first part states that if the published table contains zero votes, it could be that this is true: SURFnet also has received zero votes. It could also be that some vote has been removed. In the case of two votes and two candidates, one of four eligible votes could have been deleted.

The same reasoning goes for the situation in which the published table does contain one or more votes. Not only could a vote have been removed, but now one vote could have been added as well.

The more votes that can be cast, the more possibilities exist. After the information needed to perform the check is consulted in some situations SURFnet will conclude that the published votes are correct, in some situations that a particular vote has been removed and in others that a particular vote has been added.

$$
\begin{aligned}
&[true^*.checkInfoRec_{empty}]mu\ X.(\langle true\rangle true\ and\ [not(OKrec_{empty}\\
&or\ SURFnet\_detectRemove11\ or\ SURFnet\_detectRemove12\ or\\
&SURFnet\_detectRemove21\ or\ SURFnet\_detectRemove22)]X)\ and\\
&[true^*.checkInfoRecxx]mu\ X.(\langle true\rangle true\ and\ [not(OKrecxx\\
&or\ SURFnet\_detectAddxx\ or\ SURFnet\_detectRemoveik\ or\\
&SURFnet\_detectRemoveil\ or\ SURFnet\_detectRemovejk)]X)\ and\\
&[true^*.checkInfoRecxxyy]mu\ X.(\langle true\rangle true\ and\ [not(OKrecxxyy\\
&or\ SURFnet\_detectAddxx\ or\ SURFnet\_detectAddyy\ or\\
&SURFnet\_detectRemoveik\ or\ SURFnet\_detectRemovejl)]X)\ and\\
&[true^*.checkInfoRecxxyyvv]mu\ X.(\langle true\rangle true\ and\ [not(OKrecxxyyvv\\
&or\ SURFnet\_detectAddxx\ or\ SURFnet\_detectAddyy\ or\\
&SURFnet\_detectAddvv\ or\ SURFnet\_detectRemovejk]X)\ and\\
&[true^*.checkInfoRec11122122]mu\ X.(\langle true\rangle true\ and\ [not(OKrec11122122\\
&or\ SURFnet\_detectAdd11\ or\ SURFnet\_detectAdd12\ or\\
&SURFnet\_detectAdd21\ or\ SURFnet\_detectAdd22)]X)
\end{aligned}
$$

Result D_4 : *count*
SURFnet is also responsible for recomputing the count in my formal specification. This computation must have the same result as the count published by TTPI. The count could reveal that some count has been decreased or increased for some candidate. This is signalled by an *SURFnet_detectAdd* or *SURFnet_detectRemove* action for that particular candidate.

With two voters, two candidates and no duplicate votes already nine possibilities exist. In the simplest situation TTPI claims both candidates should be allocated zero votes. If this is not true, one of the two counts have been decreased. If TTPI has assigned one or more votes to one or more candidates this adds the possibility of the increase of a count for one of the two candidates. Therefore, after consulting the published count information SURFnet concludes in some situations that the counts match, in some situations that the count for a particular candidate has been decreased and in others that the count has been increased.

As it does not pay to use indices to state this property all instances are presented here.

$[true^*.checkInfoCnt1z2z]mu\ X.(\langle true\rangle true\ and\ [not(OKcnt1z2z\ or$
$"SURFnet\_detectRemove(CI1)"\ or\ "SURFnet\_detectRemove(CI2)")]X)\ and$
$[true^*.checkInfoCnt1z2s]mu\ X.(\langle true\rangle true\ and\ [not(OKcnt1z2s\ or$
$"SURFnet\_detectRemove(CI1)"\ or\ "SURFnet\_detectRemove(CI2)"\ or$
$"SURFnet\_detectAdd(CI2)")]X)\ and\ [true^*.checkInfoCnt1z2ss]mu$
$X.(\langle true\rangle true\ and\ [not(OKcnt1z2ss\ or\ "SURFnet\_detectRemove(CI1)"\ or$
$"SURFnet\_detectAdd(CI2)")]X)\ and\ [true^*.checkInfoCnt1s2z]mu$
$X.(\langle true\rangle true\ and\ [not(OKcnt1s2z\ or\ "SURFnet\_detectRemove(CI1)"\ or$
$"SURFnet\_detectRemove(CI2)"\ or\ "SURFnet\_detectAdd(CI1)")]X)\ and$
$[true^*.checkInfoCnt1s2s]mu\ X.(\langle true\rangle true\ and\ [not(OKcnt1s2s\ or$
$"SURFnet\_detectRemove(CI1)"\ or\ "SURFnet\_detectRemove(CI2)"\ or$
$"SURFnet\_detectAdd(CI1)"\ or\ "SURFnet\_detectAdd(CI2)")]X)\ and$
$[true^*.checkInfoCnt1s2ss]mu\ X.(\langle true\rangle true\ and\ [not(OKcnt1s2ss\ or$
$"SURFnet\_detectRemove(CI1)"\ or\ "SURFnet\_detectAdd(CI1)"\ or$
$"SURFnet\_detectAdd(CI2)")]X)\ and\ [true^*.checkInfoCnt1ss2z]mu$
$X.(\langle true\rangle true\ and\ [not(OKcnt1ss2z\ or\ "SURFnet\_detectRemove(CI2)"\ or$
$"SURFnet\_detectAdd(CI1)")]X)\ and\ [true^*.checkInfoCnt1ss2s]mu$
$X.(\langle true\rangle true\ and\ [not(OKcnt1ss2s\ or\ "SURFnet\_detectRemove(CI2)"\ or$
$"SURFnet\_detectAdd(CI1)"\ or\ "SURFnet\_detectAdd(CI2)")]X)\ and\ [true^*$
$.checkInfoCnt1ss2ss]mu\ X.(\langle true\rangle true\ and\ [not(OKcnt1ss2ss\ or$
$"SURFnet\_detectAdd(CI1)"\ or\ "SURFnet\_detectAdd(CI2)")]X)$

Accurateness

Result A_0 : *count_CR*
This property is needed to assure that an OK action for a particular counts table is only given for that counts table as the *count_HIT* (A_9) formula does not ensure that another OK action also follows the communication of that counts table. It states that if some particular count is not published the conclusion that the recount matches it, could not

follow: a correct rejection. Note its twofold purpose as it also enforces the count to be published.

$$[(not(checkInfoCnt1n2m))^*.OKcnt1n2m]false$$

Result A_1 : *receivedSet_CR*
Again, a HIT property is backed up by a Correct Rejection property. It states that if some particular table with received votes is not published the conclusion that SURFnet own table matches it, could not follow. Again, the property stated below does not only enforces a match but also the publication itself. A distinction in five cases is made.

$$[(not(checkInfoRec_{empty}))^*.OKrec_{empty}]false \; and$$
$$[(not(checkInfoRecxx))^*.OKrecxx]false \; and$$
$$[(not(checkInfoRecxxyy))^*.OKrecxxyy]false \; and$$
$$[(not(checkInfoRecxxyyvv))^*.OKrecxxyyvv]false \; and$$
$$[(not(checkInfoRecxxyyvvww))^*.OKrecxxyyvvww]false$$

result A_2 : *choice_CR*
Of course, this hit statement is accompanied by a CR statement. It should not be the case that a valid vote for candidate j is not cast while an OKci action for candidate j is issued. Note that the assignment is only checked if the voter has cast a valid vote himself.

$$[(not(validvote_{ij}))^*.(OKci_j)]false$$

If no fraud is committed, it should not be detected either. This is verified using the following formulas. They are complemented by DC formulas.

Result A_3 : *noRemove_CR*
It should be the case that if a particular vote has not been removed then the removal of this vote is not signalled by a *voter_detectRemove* or *SURFnet_detectRemove* of this vote. As the removal of a random vote and a valid vote are regarded as separate fraudulous action it is treated as distinct cases here too. Thus, if only actions occur that are not equal to fraudulous removing a (valid) vote the actions of detecting the removal of votes cannot take place.

$$[(not(removeRec_{ij} \; or \; removeValid_{ij}))^*.(voter\_detectRemove_{ij} \; or$$
$$SURFnet\_detectRemove_{ij})]false$$

Result A_4 : *noAdd_CR*
The same goes for the adding of a vote. Here, a distinction is made between adding a valid vote, adding a invalid vote to a valid one and randomly adding a vote by sending one real time. The formula below states that if a particular vote has not been added in some way, it is also not detected by the voter and SURFnet.

$$[(not(voteCORRUPT_{ij} \; or \; addValid_{ij} \; or \; makeInvalid_{ij}))^*.$$
$$(voter\_detectAdd_{ij} \; or \; SURFnet\_detectAdd_{ij})]false$$

Result A_5 : *noSubcount_CR*
If the count for a particular candidate has not been decreased, then SURFnet will not detect a remove for this candidate either.

$$[(not(\text{"}subcount(CIn)\text{"}))^*.\text{"}SURFnet\_detectRemove(CIn)\text{"}]false$$

Result A_6 : *noAddcount_CR*
Similarly, if the count for a particular candidate has not been increased, then SURFnet will not detect an add for this candidate either.

$$[(not(\text{"}addcount(CIn)\text{"}))^*.\text{"}SURFnet\_detectAdd(CIn)\text{"}]false$$

Result A_7 : *noChangeRef_CR*
All voters that perform a check on their vote will notice it if a different reference table has been used. In that case their vote is assigned to another candidate than they had intented. As it will concern all candidates instead of a specific detect action for a particular vote and candidate wild cards are used. Of course, if a different reference table is not used, this cannot be detected by a voter either.

$$[(not(\text{"}changeRef\text{"}))^*.\text{"}Voter\_detectci_j\text{"}]false$$

8.2.2 noFraud scenario S_0

In the simplest model all participants are honest; no fraud is committed.

This scenario was model-checked using the CADP toolset, confirming that all properties stated below have been proven correct.

Validity

Result V_2 : *diffOKv*
This property is twofold. It states that the sending of a vote for a particular candidate cannot be followed by an action that validates a vote for another candidate by the same voter. This excludes the situation in which a voter can cast a valid vote for both candidate 1 and candidate 2 (represented by an OKv for that vote). It also covers the statement that a vote for one candidate cannot be assigned to another, something that I will return to later. This is expressed as follows:

$$[true^*.vote_{ij}.true^*.OKv_{ik}]false$$

Result V_3 : *doubleValid*
The casting of a valid vote will always be followed by an OKv action for that vote to indicate that the vote has been marked valid. Note that this statement also includes the so-called duplicates one of them is judged valid and thus followed by an OKv action.

$$[(validvote_{ik})^+.\text{"}timeout\text{"}]mu\ X.(\langle true\rangle true\ and\ [not(OKv_{ik})]X$$

Result V_4 : *notVotedReceived*
A vote that has not been cast by the eligible voter cannot be received. Obviously, this also goes for an abstaining voter. This is represented by the absence of an OKr action.

$$[(not(vote_{ij}))^*.OKr_{ij}]false$$

Result V_5 : *doubleOnceCount_HIT*
The correct assigning of a vote to a candidate is also visible in the counts table that is published after the election is closed. This property shows that if a valid vote is cast once or more (duplicates) for a particular candidate the count for this candidate is precisely one. This only holds if other voters have not cast a valid vote for that same candidate. As in my formal specification two candidates are used this formula is illustrated for both cases.

$$[(not(validVote_{j1}))^*.(validVote_{i1})^+.(not(validVote_{j1}))^*."timeout"]$$
$$mu\ X.(\langle true\rangle true\ and\ [not(voter\_checkInfoCnt1s2z)]X)\ and$$
$$[(not(validVote_{j2}))^*.(validVote_{i2})^+.(not(validVote_{j2}))^*."timeout"]$$
$$mu\ X.(\langle true\rangle true\ and\ [not(voter\_checkInfoCnt1z2s)]X)$$

Result V_6 : *doubleOnceCount_CR*
This property is added to ensure that the counts table containing one vote for a particular candidate and zero for the other is only published when only one (possibly duplicate) valid vote for that candidate is cast.

The former formula does not enforce that another publication could also follow the casting of such a vote. It states that if a valid vote for a certain candidate is not cast a counts table containing a count for that candidate cannot have been published. Again, it is illustrated for two candidates.

$$[(not(validVote_{j1}\ or\ validVote_{i1}))^*."timeout".true^*.$$
$$voter\_checkInfoCnt1s2z]false\ and\ [(not(validVote_{j2}\ or\ validVote_{i2}))^*.$$
$$"timeout".true^*.\ voter\_checkInfoCnt1z2s]false$$

Result V_7 : *notReceivedValid*
Obviously, a vote that does not occur in the received table (indicated by the absence of an OKr action) cannot be marked valid (indicated by an OKv action) either.

$$[(not(OKr_{ij}))^*.OKv_{ij}]false$$

Result V_8 : *notVotedValid*
The protocol also ensures that if a vote has not been cast, it cannot be marked valid. Hence, in this case an OKv action for that vote does not occur when the voter does his check.

$$[(not(vote_{ij}))^*.OKv_{ij}]false$$

Accurateness

Result A_8 : *count_HIT*
In the ideal situation it should be that the two counts match using the published information needed to do a recount. This means that after SURFnet recomputes the count and inspects the published counts table the conclusion that the two match should eventually follow, a HIT. This is signalled by an OKcnt action for that table. This is easily explained using indices m and n that represent natural numbers, starting at zero.

$$[true^*.checkInfoCnt1n2m]\ mu\ X.(\langle true\rangle true\ and\ [not(OKcnt1n2m)]X)$$

Result A_9 : *receivedSet_HIT*
After SURFnet has consulted the published table with received votes eventually the conclusion should follow that its own table confirms the correctness of the published one. Again, it is easiest to distinguish the receival of zero, one, two, three and four votes in the situation with two voters, two candidates and no double votes. Every possibility is expressed using the mu operator to express that the consultation of the check information is always followed by the confirmation that the table consisting of the received votes is correct.

$$\begin{aligned}&[true^*.checkInfoRec_{empty}]\ mu\ X.(\langle true\rangle true\ and\\ &[not(OKrec_{empty})]X)\ and\ [true^*.checkInfoRecxx]\\ &mu\ X.(\langle true\rangle true\ and\ [not(OKrecxx)]X)\ and\\ &[true^*.checkInfoRecxxyy]\ mu\ X.(\langle true\rangle true\ and\\ &[not(OKrecxxyy)]X)\ and\ [true^*.checkInfoRecxxyyvv]\\ &mu\ X.(\langle true\rangle true\ and\ [not(OKrecxxyyvv)]X)\ and\\ &[true^*.checkInfoRecxxyyvvww]\ mu\ X.(\langle true\rangle true\\ &and\ [not(OKrecxxyyvvww)]X)\end{aligned}$$

Result A_{10} : *receivedVote_HIT*
After the election is closed all voters can find out what happened to their vote. For every cast vote it should be the case that it is eventually received, signalled by an OKr action for that particular vote. This also goes for an invalid vote. When duplicate votes are concerned at most two OKr actions exist for that vote. This simplification is built into the formal specification to keep the state space manageable. All voter processes can cast votes non-stop, while only recording the casting of the fresh and second-duplicate votes. Third and more duplicate votes are not recorded, but yet cast.

In the generic formulation below second-duplicate votes are not taken into account as a separate formula would be needed to cover them.

$$[vote_{ij}.\text{"}timeout\text{"}]\ mu\ X.(\langle true\rangle true\ and\ [not(OKr_{ij})]X)$$

Result A_{11} : *receivedDuplicateVote_HIT*
For a vote that has been cast twice or more an OKr action for precisely two of them should eventually follow. This condition is not met in the former formula as it only considers one OKr action, which should be in the case of only one vote. The following states that the casting of minimally two duplicates is followed by at least two OKr actions.

$$[true^*.vote_{ij}.(vote_{ij})^+.timeout]\ [true^*.OKr_{ij}.true^*.OKr_{ij}]true$$

It is not possible to formulate this stronger using a mu operator like:
$[true^*.vote_{ij}.true^*.(vote_{ij})^+.true^*."timeout"]mu\ X.(\langle true\rangle true$

$and\ [not(OKr_{ij}.true^*.OKr_{ij})]X)$

It does not accept more than one action.

Result A_{12} : *receivedVote_CR*
This property states that an OKr action for a particular vote should always be preceded by a communication of that vote.

$[(not(vote_{ij}))^*.OKr_{ij}]false$

Result A_{13} : *validVote_HIT*
All voters have the opportunity to check if their valid vote has indeed been marked valid. This is signalled by an OKv action for that vote. In this context, a valid vote is one that is cast one or more times. Remember that one duplicate is judged valid. The key used for this vote has not been used to cast one or more other votes.

$[validVote_{ij}."timeout"]mu\ X.(\langle true\rangle true\ and\ [not(OKv_{ij})]X)$

Result A_{14} : *validVote_CR*
To complement the latter HIT formula this Correct Rejection property ensures that if another or no casting of a particular vote takes place the conclusion that this vote is valid cannot follow. Separate formulas to distinguish between a valid vote and a random vote are not needed here, as the former is a special case of the latter.

$[(not(vote_{ij}))^*.OKv_{ij}]false$

result A_{15} : *choice_HIT*
A valid vote should be assigned to the correct candidate. Thus, a valid vote for candidate j should eventually be followed by an assignment to candidate j, signalled by an OKci action for j when the voter checks this.

$[(validvote_{ij})^+."timeout"]mu\ X.(\langle true\rangle true\ and\ [not(OKci_j)]X)$

8.2.3 sendVote scenario S_1

TTPI corrupts by using a key to send a vote in real time. The voter will detect this fraud when trying to confirm that his uncast votes do not occur in the table with received votes. In this scenario one does.

In my formal specification it is not noted if it concerns a third-duplicate vote as only second duplicate votes are allowed. If TTPI adds a second duplicate, it is detected. As such, this fraud is only uncovered if the voter has not already cast the same vote; be it a first duplicate in the situation not allowing duplicates and a second duplicate when duplicate are allowed.

This scenario was model-checked using the CADP toolset for all properties concerning validity and verifiability, as described in discussing the no fraud scenario. I have to remark here that the properties *Result* D_3 *: receivedSet*, *Result* A_9 : *receivedSet_HIT* and *Result* A_1 : *receivedSet_CR* have been split up in several formulas regarding part of the original formula to make model checking for these properties possible. When model checking the original formula for this state space evaluator gives a table overflow error in the resolution of the formula.

Most of those properties have been proven valid for this scenario as well. Because they have already been explained I will not elaborate on them further. Instead, I will illustrate why some properties have been shown to be invalid in this scenario, a result that should be expected.

Validity

Result V_3 : *doubleValid*
The casting of a valid vote will always be followed by an OKv action for that vote to indicate that the vote has been marked valid. This property does not hold in the model in which TTPI can send an arbitrary vote. Such a vote could be using the same key as the previous valid one, thereby making the previously valid vote invalid. Therefore, this property is not satisfied in this scenario.

Result V_4 : *notVotedReceived*
A vote that has not been cast by the eligible voter cannot be received. This property does not hold in the current model either as TTPI can cast this particular vote. If the voter later inspects the table with received votes this vote will be present. This is signalled by an OKr action for this vote that this voter did not cast.

Result V_8 : *notVotedValid*
This property states that if a vote has not been cast, it cannot be marked valid. However, TTPI could send this vote resulting in an OKv action from the voter that will check his vote collection. He will find that a vote using his personal key has been marked valid, although he did not cast it. Therefore, this property does not hold in this scenario.

Accurateness

Result A_{13} : *validVote_HIT*
Every valid vote should eventually be marked valid. As before, TTPI could send another vote using the same key, thereby making the vote invalid. In that case this property is not satisfied.

Result A_{14} : *validVote_CR*
The RIES protocol should also ensure that if another or no casting of a particular vote takes place the conclusion that this vote is valid cannot follow. Again, this conclusion will nonetheless follow if TTPI corrupts by casting this vote.

Detectability

Result DC_0 : *sendVote_HIT*
This property states that if TTPI commits fraud by casting a vote it will eventually be detected by the voter concerned, given that the the voter has not already cast a second duplicate. The adding of second duplicates can be detected. SURFnet is not able to detect this fraud, because TTPI pretends to be a voter. This remains unnoticed by SURFnet as TTPI has the unique keys to its use that makes him indistinguishable from other voters. In model-checking this property has been verified to be true, in contrast to the above properties.

$[(not(vote_{ij}))^*.voteCORRUPT_{ij}.(not(vote_{ij}))^*."timeout"]$
$mu\ X.(\langle true\rangle true\ and\ [not(Voter\_detectAdd_{ij})]X)\ and$
$[voteCORRUPT_{ij}.vote_{ij}."timeout"]$
$mu\ X.(\langle true\rangle true\ and\ [not(Voter\_detectAdd_{ij})]X)\ and$
$[vote_{ij}.voteCORRUPT_{ij}."timeout"]$
$mu\ X.(\langle true\rangle true\ and\ [not(Voter\_detectAdd_{ij})]X)$

This property has not been model checked in the no fraud model as it concerns fraud. It would automatically return TRUE as the fraud action specified in the formula does not occur in this state space. For the same reason in this model only this property is regarded and not the other formulas expressing other forms of fraud.

8.2.4 removeReceived scenario S_2

In this model all participants are honest, except for SURFnet. SURFnet will randomly remove a vote from the set of received votes in real time. Only the voter can detect this fraud when checking what has happened to his cast vote. In this case it will not appear in the table with received votes as it should. Note that this type of fraud implies that minimally one vote has been cast.

In this model most of the properties in the categories validity and verifiability as discussed in the no fraud scenario hold. Exceptions will be discussed below. These results have been verified as such using the CADP toolset. The toolset has not been applied to a few properties that assume SURFnet to be honest. Obviously, those properties that state that SURFnet will detect removal of a vote are not applicable here as SURFnet itself is responsible for this fraud. It is assumed that in this case SURFnet will omit checking the table with its received votes against the published table for this will only uncover its own fraudulous actions. Therefore, the following properties have not been model-checked as it would make no sense in doing.

Determinability

Result D_3 : *receivedSet*
This property states that SURFnet will eventually either detect removal of some vote or find that the table composed by SURFnet matches the published one by TTPI.

Accurateness

Result A_1 : *receivedSet_CR*
If some particular table with received votes is not published the conclusion that SURFnet's own table matches it, could not follow.

Result A_9 : *receivedSet_HIT*
Eventually the conclusion should follow that SURFnet's table containing received votes confirms the correctness of the published one.

Next, the exceptions that in model checking result in falsum will be discussed briefly.

Validity

Result V_2 : *diffOKv*
Votes for two or more different candidates using the same key are not valid, which is represented by the absence of an OKvalid action (OKv). In the model described, it could be that one of two invalid votes is removed, leaving one invalid vote that now will be regarded as valid. Therefore, this property will no longer hold. The other interpretation of this property remains unaffected as it states that a vote that has not been cast cannot be marked valid. The removal of a vote will not change this.

Result V_3 : *doubleValid*
The casting of a valid vote will normally be followed by an OKv action for that vote to indicate that the vote has been marked valid. Of course, if this vote is removed by SURFnet it cannot be marked valid anymore.

Accurateness

Result A_{10} : *receivedVote_HIT*
For every cast vote it should be the case that it is eventually received. This property does not longer hold as SURFnet could remove a vote.

Result A_{11} : *receivedDuplicateVote_HIT*
For a vote that has been cast twice or more an OKr action for precisely two of them should eventually follow. This is not true when SURFnet can remove a vote. Then only one vote is confirmed to have been received.

Result A_{13} : *validVote_HIT*
Every valid vote should eventually be marked valid. As SURFnet could have removed this vote the property is not satisfied in this model.

Detectability

The last property to discuss in this scenario regards detecting fraud. It has not been model checked in the before mentioned models as the fraud action specified in the formula does not occur in those state spaces. It would automatically return TRUE. For the same reason in this model only the following property is model checked and not the other formulas expressing other forms of fraud.

Result DC_1 : *removeReceived_HIT*
As the removal is arbitrary it could concern a valid vote, an invalid vote and a second-duplicate vote. In all cases the voter will eventually notice this fraud.

$$[true^*.removeRec_{ij}]mu\ X.(\langle true\rangle true\ and\ [not(voter\_detectRemove_{ij})]X)$$

8.2.5 removeValid scenario S_3

In this scenario it is TTPI that commits fraud. A valid vote is removed by TTPI after the election. Both SURFnet and the voter will note this fraud. SURFnet will notice a discrepancy between its own received set and TTPI's. The voter whose valid vote has been removed will conclude that this vote does not occur in the table with received votes. Obviously, this fraud can only take place if minimal one valid vote has been cast.

Again, this model satisfies most properties discussed in the no fraud scenario. This has been verified using the CADP toolset. The properties that have been shown to be false by this same toolset concern statements about the marking valid and receival of valid votes.

Validity

Result V_3 : *doubleValid*
The first one to be discussed regards the marking of a valid vote valid. A cast of a valid vote will normally be followed by an OKv action for that vote. This is not the case when TTPI removes this valid vote.

Accurateness

Result A_9 : *receivedSet_HIT*
The table containing the received votes by SURFnet should eventually be signalled to be equal to TTPI's table. Obviously, this is not the case when TTPI removes a vote from TTPI's table.

Result A_{10} : *receivedVote_HIT*
In general, it should be the case that every vote is eventually received, including valid votes. This property does not longer hold as TTPI could remove it.

Result A_{11} : *receivedDuplicateVote_HIT*
This property states that a vote that has been cast twice or more should be eventually followed by two corresponding OKr actions. This is not satisfied when TTPI remove one, resulting in only one confirmation.

Result A_{13} : *validVote_HIT*
Eventually, every valid vote should be marked valid. This property does not longer hold when TTPI removes it.

Detectability For this scenario also goes that all properties regarding the detection of

fraud will return true, because the fraud action does not occur in the state space. Again, there is one exception and that concerns the fraud action that does occur in this scenario.

Result DC_2 : *removeValid_HIT*
If TTPI removes a valid vote, it is eventually detected by the voter that has cast the vote and by SURFnet that will find a mismatch between the two tables containing the received votes. The fraud action is therefore always followed by the detection by the voter and always followed by the detection by SURFnet.

$$([true^*.removeValid_{ij}]mu\ X.(\langle true\rangle true\ and\ [not(Voter\_detectRemove_{ij})]X)\ and\ [true^*.removeValid_{ij}]\ mu\ X.(\langle true\rangle true\ and\ [not(SURFnet\_detectRemove_{ij})]X))$$

8.2.6 addValid scenario S_4

In this scenario TTPI adds a valid vote after the election. Again, both SURFnet and the voter detect this action. SURFnet will conclude that TTPI's table will contain one more vote than its own received set and the voter will notice that one of his uncast votes occurs in TTPI's table while it should not.

All properties discussed in the no fraud scenario concerning validity and verifiability have been model checked using the CADP toolset. Except for the following they all resulted in TRUE.

Validity

Result V_4 : *notVotedReceived*
A vote that has not been cast by the eligible voter cannot be received. As TTPI adds the valid vote to the table with received votes this property does not hold in the current model.

Result V_8 : *notVotedVali*
A vote that has not been cast, cannot be marked valid normally. Of course, a valid vote added by TTPI will be marked valid although it has not been cast by the eligible voter.

Accurateness

Result A_9 : *receivedSet_HIT*
If TTPI adds a valid vote to the published table, it will contain an extra vote in comparison to the set of received votes by SURFnet. Normally, the tables match.

Result A_{12} : *receivedVote_CR*
Likewise, this adding results in the voter signalling that the vote is present in the received table although the voter did not cast it. Normally, this would only be the case when the voter has send the vote.

Result A_{14} : *validVote_CR*
This property ensures that if another or no casting of a particular vote takes place the

conclusion that this vote is valid cannot follow. Again, this is not satisfied in this model as TTPI adds a valid vote.

Detectability

As before, only one fraud detectability property is model checked as just one type of fraud is modeled in this scenario. The next property is satisfied.

Result DC_3 : *addValid_HIT*

Eventually, the fraudulous adding of a valid vote to the table containing the received votes will be detected by both the voter and SURFnet. This is expressed as follows:

$$([true^*.addValid_{ij}]mu\ X.(\langle true\rangle true\ and\ [not(Voter\_detectAdd_{ij})]X)\ and\ [true^*.addValid_{ij}]\ muX.(\langle true\rangle true\ and\ [not(SURFnet\_detectAdd_{ij})]X))$$

8.2.7 makeInvalid scenario S_5

TTPI makes a valid vote invalid by adding a vote for a different candidate after the election. This is detected by SURFnet and the voter for the same reasons that uncover the previous mentioned actions: SURFnet will find a mismatch between TTPI's and its own received votes and the voter fails to confirm that his uncast vote is not present in the table containing the received votes. Note that this fraudulous action can only occur when at least one valid vote is present.

Validity

Result V_3 : *doubleValid*

The casting of a (duplicate) valid vote will always be followed by an OKv action for that vote to indicate that the vote has been marked valid. Note that a second duplicate is invalid. If TTPI makes the valid vote invalid by adding a vote, this does not longer hold.

Result V_4 : *notVotedReceived*

This property states that a vote that has not been cast by the eligible voter cannot be received. This is not satisfied here, as TTPI can add a vote to the table with received votes.

Accurateness

Result A_9 : *receivedSet_HIT*

In a normal situation, SURFnet and TTPI agree about the votes received. This does not hold if TTPI adds a vote that has not been received by SURFnet.

Result A_{12} : *receivedVote_CR*

This adding also results in the voter signalling that the vote is present in the received table although the voter did not cast it. This should only be the case when the voter has sent the vote.

Result A_{13} : *validVote_HIT*
Every valid vote should be followed by a positive checkup by its voter as being indeed marked valid. As TTPI makes it invalid by adding a vote using the same key this is no longer true.

Detectability

The following formula is only model checked in this particular model as the fraudulous action concerned only occurs in this state space.

Result DC_4 : *makeInvalid_HIT*
This property states that the making of a valid vote invalid by adding a vote with the same key is eventually noticed by SURFnet and the voter concerned.

$([true^*.makeInvalid_{ij}]mu\ X.(\langle true\rangle true\ and$
$[not(voter\_detectAdd_{ij})]X)\ and\ [true^*.makeInvalid_{ij}]$
$mu\ X.(\langle true\rangle true\ and\ [not(SURFnet\_detectAdd_{ij})]X))$

8.2.8 changeCount *scenario*S_6

The count is changed by TTPI after the election is closed by publishing an increased count for a particular candidate as well as a decreased count for another. As SURFnet recomputes the count, this party will notice it. The recount will fail at two places: for the candidate whose count has been increased and for the candidate whose count has been decreased.

As this form of fraud only affects the count in the counts table directly according to this formal specification, all properties that have been satisfied in the no fraud model have also found to be true here. There is only one exception, being the following.

Accurateness

Result A_8 : *count_HIT*
If no fraud is committed, recomputing the count will show that the published count is correct. If the count has been altered the original count by TTPI and the recount by SURFnet will no longer match.

Detectability

In contradiction to all former described models the discussion of this scenario ends with two properties about detecting fraud. The following two statements have been proven to be correct.

Result DC_5 : *subCount_HIT*
When the count for a particular candidate has been decreased by directly altering the count in the counts table to be published, this is eventually always noted by SURFnet.

$$[true^*.subcount(CI_n)]mu\ X.(\langle true\rangle true \\ and\ [not(SURFnet\_detectRemove(CI_n))]X)$$

Result DC_6 : *addCount_HIT*
Similarly, when the count for a particular candidate has been increased, it is eventually always detected by SURFnet.

$$[true^*.addcount(CI_n)]mu\ X.(\langle true\rangle true\ and\ [not(SURFnet\_detectAdd(CI_n))]X)$$

8.2.9 addCount scenario S_7

To complement the former scenario in which the fraud can only take place when a candidate is assigned one or more votes to allow decreasing a count, this scenario is used. Here, TTPI can directly increase the count of a particular candidate, even if it is zero. Again, this is detected by SURFnet as the recount fails for the candidate whose count is increased.

In this scenario the same properties are satisfied as in the former scenario in which a particular count is increased and another decreased. Furthermore, the property that has been proven to be false using the CADP tool is identical as well.

Accurateness

Result A_8 : *count_HIT*
If no fraud is committed, recomputing the count will show that the published count is correct. If the count has been altered the original count by TTPI and the recount by SURFnet will no longer match.

Detectability

The fraudulous action in this scenario consists of increasing the count. The model should guarantee that this is always detected. Again, other properties that are designed to ensure that a particular form of fraud is detected are not model checked as they will return TRUE because of the absence of such fraud.

Result DC_6 : *addCount_HIT*
The protocol ensures that the act of increasing a count is eventually detected. This property has shown to be satisfied in this model.

8.2.10 changeRef scenario S_8

In this last fraud scenario TTPI performs the count using an altered reference table, resulting in the assignment of a vote to a different candidate as it was meant for. In particular, all votes for candidate 1 are assigned to candidate 2.

This particular fraud is discovered by the voter only. The voter would fail to verify that his vote has been assigned to the correct candidate. SURFnet's recount will however find no mismatch as it uses the incorrect reference table as well.

This form of fraud only affects the count, leaving the properties stating something about votes during the election untouched. This means that only the features regarding the assignment of the candidates in some way are not satisfied here.

Validity

Result V_5 : *doubleOnceCount_HIT*
If a valid vote is cast once or more (duplicates) for a particular candidate the count for this candidate is precisely one. Because this vote is assigned to a different candidate, this will not hold.

Result V_6 : *doubleOnceCount_CR*
For the same reason it is no longer true that if a valid vote for a certain candidate is not cast a counts table containing a count for that candidate cannot have been published.

Accurateness

Result A_{15} : *choice_HIT*
In checking if his vote has been assigned to the correct candidate in this scenario, it does not longer hold for all votes that a valid vote for candidate j is eventually followed by an assignment to candidate j, signalled by an OKci action for j and that vote.

Detectability

As discussed in all former scenarios there is one particular property that should be satisfied in this specific scenario: the detection of the form of fraud committed here.

Result DC_7 : *changeRef_HIT*
After the election, the voter inspects the reference table to see if the correct candidate has been assigned to his valid vote. This means that after a valid vote has been cast and the reference have been changed, this fraudulous action should eventually be detected, signalled by a detectci action for the intended vote for candidate 1.

Note that voters casting a vote for candidate 2 will not detect this fraud.

$$[(validvote_{i1})^{+}."timeout".true^{*}."refchange"]mu\ X.(\langle true\rangle true \\ and\ not(Voter\_detectci_1)X)$$

8.2.11 Anonymity scenarios

The following two scenarios are used to verify the anonymity property.

anonymity111222 S_9 In this scenario voter process 1 uses Key 1 to vote for candidate 1, whereas voter process 2 uses Key 2 to vote for candidate 2.

anonymity122211 S_{10} This scenario is the reverse situation in that voter process 1 uses Key 2 to vote for candidate 2 and voter process 2 uses Key 1 to vote for candidate 1.

Both scenarios have been translated to a formal specification which has resulted in two corresponding state spaces. The states spaces have been reduced using branching bisimulation equivalence and applying the tool Aldebaran it has been verified "on the fly" that the two state spaces are branching bisimilar. This means that an observer would not note a difference between the two situations, thereby guaranteeing the anonymity of the voter.

To end this section an overview of the verification results obtained by state space traversal is given. Figure 6 lists all scenarios and the truth value of each property verified for that scenario. A result is coded with 1 to indicate that the property has been proven TRUE and with 0 to indicate that the property has been proven FALSE. A horizontal bar (-) is used to code that model checking has not been performed in that particular case. Note that these results apply to the two configurations discussed: one voter, two candidates and duplicate votes, and two voters, two candidates and single votes. However, remember that three properties have not been verified for the first configuration. It concerns D_3 : *receivedSet*, A_1 : *receivedSet_CR* and A_9 : *receivedSet_HIT*.

The results of the verification of the properties all correspond to the expected truth values. This contributes to the conclusion that the protocol is correctly formalized and that the protocol is correct concerning these properties.

To give an impression of the size of the states spaces and the verification times some statistics for two arbitrary scenarios and 4 properties are presented in table 8. The scenarios considered are the noFraud and sendVote scenario. In the table every first row of numbers is associated with the noFraud scenario, whereas the second row is associated with the sendVote scenario.

The properties concern the fairly easily formalized property V_0 : invalidEI and three liveness properties; D_3 : *receivedSet*, A_9 : *count_HIT* and A_{10} : *receivedSet_HIT*. For both scenarios statistics on the state space and verification times are determined concerning three different configurations. The first one, single1, considers one voter and no duplicate votes. This configuration has not been used in verifying the protocol. It is only used here to indicate the impact of the number of voters and duplicate votes on the state space and verification times. The second configuration, dup1, also considers just one voter, but allows second duplicate votes. Finally, the third one considers two voters and only single votes, no duplicates. Therefore, it is named single2. The latter configuration results in the largest state spaces and verification times. This also goes for the other attacker scenarios. The statistics show that the CADP toolset gives a table overflow when trying to verify the properties D_3 : *receivedSet* and A_{10} : *receivedSet_HIT*, whereas the other configurations and the noFraud scenario show no problem. In verifying RIES this has been solved by dividing these properties in several parts that can be handled by every scenario and configuration as used here.

The table also includes a configuration named dup2 that concerns two voters and duplicate votes. It is not used for verification, but presented here for statistical reasons. Concerning the noFraud scenario an attempt has been made to determine its size and generation time. After two days and eight hours the generation has been manually interrupted. At that time the state space had reached a size of 2,043,328,153 bytes, which

equals 1,9 GB. Most likely this state space exceeds the size manageable by the CADP toolset. Verification times of these four properties have not been determined for this configuration.

The verification times for the other configurations are. These are given in seconds to make a clear comparison with smaller verifications possible. The smallest takes 2 seconds and the largest 3769 seconds which equals 1 hour and almost 3 minutes. Together with the immense size of a configuration of two voters and duplicates this demonstrates that the tool might not be the most suitable to perform such model checking.

	single1	dup1	single2	dup2
State space				
Generation time (seconds)	4	5	31	201,600
	4	6	132	
Size (bytes)	143,004	1,358,185	61,540,495	2,043,328,153
	431,251	4,013,246	258,585,952	
States explored	545	4,033	119,109	
	1,570	11,414	488,406	
Transitions generated	937	6,976	302,333	
	2,798	20,387	1,264,554	
Levels	21	33	36	
	23	35	38	
Verification time (seconds)				
V_0 : invalidEI	2	5	11	
	3	9	78	
D_3 : receivedSet	3	6	221	
	3	3	table overflow	
A_9 : count_HIT	6	24	749	
	10	69	3,769	
A_{10} : receivedSet_HIT	2	6	222	
	5	5	table overflow	

Table 8: Statistics on generation and verification of two scenarios and four properties

	noFraud	sendVote	removeReceived	removeValid	addValid	makeInvalid	changeCount	addCount	changeRef
V_0 : invalidEl	1	1	1	1	1	1	1	1	1
V_1 : doubleOnce	1	1	1	1	1	1	1	1	1
V_2 : diffOKv	1	1	0	1	1	1	1	1	1
V_3 : doubleValid	1	0	0	0	1	0	1	1	1
V_4 : notVotedReceived	1	0	1	1	0	0	1	1	1
V_5 : doubleOnceCount_HIT	1	1	1	1	1	1	1	1	0
V_6 : doubleOnceCount_CR	1	1	1	1	1	1	1	1	0
V_7 : notReceivedValid	1	1	1	1	1	1	1	1	1
V_8 : notVotedValid	1	0	1	1	0	1	1	1	1
D_0 : sendVote	1	1	1	1	1	1	1	1	1
D_1 : validVote	1	1	1	1	1	1	1	1	1
D_2 : choice	1	1	1	1	1	1	1	1	1
D_3 : receivedSet	1	1	-	1	1	1	1	1	1
D_4 : count	1	1	1	1	1	1	1	1	1
DC_0 : sendVote_HIT	-	1	-	-	-	-	-	-	-
DC_1 : removeReceived_HIT	-	-	1	-	-	-	-	-	-
DC_2 : removeValid_HIT	-	-	-	1		-	-	-	-
DC_3 : addValid_HIT	-	-	-	-	1	-	-	-	-
DC_4 : makeInvalid_HIT	-	-	-	-	-	1	-	-	-
DC_5 : subCount_HIT	-	-	-	-	-	-	1	-	-
DC_6 : addCount_HIT	-	-	-	-	-	-	1	1	-
DC_7 : changeRef_HIT	-	-	-	-	-	-	-	-	1
A_0 : count_CR	1	1	1	1	1	1	1	1	1
A_1 : receivedSet_CR	1	1	-	1	1	1	1	1	1
A_2 : choice_CR	1	1	1	1	1	1	1	1	1
A_3 : noRemove_CR	1	1	1	1	1	1	1	1	1
A_4 : noAdd_CR	1	1	1	1	1	1	1	1	1
A_5 : noSubcount_CR	1	1	1	1	1	1	1	1	1
A_6 : noAddcount_CR	1	1	1	1	1	1	1	1	1
A_7 : noChangeRef_CR	1	1	1	1	1	1	1	1	1
A_8 : count_HIT	1	1	1	1	1	1	0	0	1
A_9 : receivedSet_HIT	1	1	-	0	0	0	1	1	1
A_{10} : receivedVote_HIT	1	1	0	0	1	1	1	1	1
A_{11} : receivedDuplicateVote_HIT	1	1	0	0	1	1	1	1	1
A_{12} : receivedVote_CR	1	1	1	1	0	0	1	1	1
A_{13} : validVote_HIT	1	0	0	0	1	0	1	1	1
A_{14} : validVote_CR	1	0	1	1	0	1	1	1	1
A_{15} : choice_HIT	1	1	1	1	1	1	1	1	0

Figure 6: Overview of verification results

9 Conclusion

The main purpose of this paper has been to describe the RIES protocol in full detail, offer a formal specification and finally verify several properties concerning anonymity, verifiability and some validity features using the CADP tool. Although several researches exist a formal analysis has not been performed before.

The RIES protocol has been formalized using μCRL, a process algebraic language that allows data to affect the course of the processes. Some assumptions have been made to simplify the specification.

A total of nine scenario's have been designed to show that certain verifiability and validity features hold under differing circumstances. As a special interest lies in internal fraud, these specifications each model a particular instance of fraud. It concerns the removal of a vote by both SURFnet and TTPI, the adding of a vote by TTPI in several ways, and the practicing of various incorrect count procedures by TTPI. As it should, not all properties are true in every specification. For example, a vote is not received if it is fraudulously removed. Keeping this in mind, the following can be stated.

RIES is praised for its verifiability feature: every voter is capable of finding out what happened with his vote and everyone can perform a recount. It has been shown that this feature is also satisfied in different fraudulous environments. In other words, (internal) fraud can be detected.

The protocol should also ensure proper authentication, receival, validity judgement and count of votes. These features are classed under the term validity here and have been shown to hold in all nine models where they should.

In all voting protocols the greatest concern is privacy. Here, it is verified that the RIES protocol guarantees anonymity by applying the concepts unlinkability and role interchangeability. The observables give no indication of a link between the vote and the voter's identity. This is shown by designing two specifications in which two voters change roles and subsequently demonstrating their branching bisimulation equivalence. The anonymity property is tested using formal specifications in which no fraud is committed.

Thus, all stated properties have been proven to be satisfied when they should with respect to the specifications provided here. This supports the conclusion that the RIES protocol is well-designed concerning the properties verified.

However, the process of formalizing contributed in identifying two particular weaknesses. One is that although the RIES protocol is theoretically safe, in practice it is not. The main reason is that RIES relies on the voter to perform checks, one of the essential assumptions made here in proving its correctness. This especially goes for the scenarios in which votes are removed at the server and keys are misused by a third party to cast a vote during the election. In these cases only the voter compromised can detect this fraud, but as it is very reasonable to assume that not all voters are motivated to do so this fraud can succeed without being detected. This is even worse in the scenario that votes are sent during the election as it could now concern voters that abstain from voting. Such irregularities should be solved by considering the acknowledgements, but an abstainer would have no means to prove that he did not send it although it is detected.

The experiment during the Second Chambre Election 2006 shows the magnitude of this problem as only 0,5 % of the Internet voters performed the check. This could undermine

the reliability of the protocol.

This risk is assumed to be quite low in practice as only one voter is needed to prove fraud and the chance that several unused keys are selected to vote is not very big either.

Therefore, it seems that this protocol can be effectively employed. Especially when one takes a pragmatic view in the realizability and need of the desired features of such a system. Concerning the latter it would be fair to demand this new way of voting to be as safe as the procedure it replaces. Traditional voting also suffers from certain weaknesses, but that seems to be ignored in reviewing Internet voting. However, as possibilities exist to improve RIES like the ones mentioned in this thesis, one should correct RIES on these points before introducing it nationally.

If the government should decide to benefit from the advantages it offers over traditional voting, it would be wise to apply and profit from all methods available that are designed to ensure correctness and reliability. The approach of formal methods could prove to be a valuable instrument in this. Especially when future methods mature and improve its applicability. The specification in this thesis results in a state space that is ineffectively large when only considering two voters, two candidates and second duplicates. The maximum size that the toolset can operate with is most likely exceeded by this configuration. This may be improved in the future.

In all cases it should however be complemented by other techniques that may also consider the organisational aspects of the procedure. Especially when it concerns such a fundamental concept as democracy.

References

[1] Bergstra, J. and Klop, J. (1984) Process algebra for synchronous communication. *Information and Control, 60(3)*:109-137.

[2] Cheung, S.C., Kramer, J. (1999) *ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 8, Issue 1. 1999*

[3] Clarke, E., Grumberg, O. and Long, D. (2002) *Verification tools for finite-state concurrent systems.*

[4] Denning, D. E. (1999) *The limits of Formal Security Models.*

[5] Dolev, D. and Yao, A. (1983) On the security of public key protocols. *IEEE Transactions on Information Theory.*

[6] Esch-Bussemakers, van M.P., Geers, J.M.E., Maclaine Pont, P.G., Vink, H.J. (2002) *Beveiligings- en gebruikersaspecten van elektronisch stemmen voor het Hoogheemraadschap van Rijnland.* TNO.

[7] Fischer, E. A. (2003) *Election Reform and Electronic Voting Systems (DREs): Analysis of Security Issues.*

[8] Fokkink, W., Groote, J.F. and Reniers, M. No date. *Modelling Distributed Systems.* Retrieved from http://www.cs.vu.nl/∼tcs/pv/mcrl-book.pdf.

[9] Groote, J.F. and Ponse, A. (1995) The syntax and semantics of muCRL. In *Algebra of Communication Processes '94, Workshops in Computing Series*, pages 26-62. Springer-Verlag.

[10] Hinchey, M.G., Bowen, J.P. (1995) *Applications of Formal Methods.* Prentice Hall.

[11] Holzmann, G.J., Smith, M.H. (2000) Automating software feature verification. *Bell Labs Technical Journal, Vol. 5, No. 2*, pp. 72-87, April-June 2000.

[12] Hubbers, E. (2007) Hoe veilig is stemmen via internet? In *I/O InformaticaOnderzoek. Magazine van het Informaticaonderzoek Platform Nederland (IPN), 4:1*, pp. 8, January 2007.

[13] Hubbers, E. *"Kiezen uit het buitenland: een onafhankelijke bijdrage door de Radboud Universiteit Nijmegen"* No date. Retrieved from https://www.sos.cs.ru.nl/ research /sosries.

[14] Hubbers, E. and Jacobs, B. (2004) Stemmen via internet geen probleem. *Automatisering Gids*, 42:15, 2004

[15] Hubbers, E., Jacobs, B. and Pieters, W. (2005) RIES: Internet voting in action. In: R. Bilof (ed.). *Proceedings of the 29th Annual International Computer Software and Applications Conference, COMPSAC'05*, IEEE Computer Society, 2005, pp. 417-424.

[16] Kneuper, R. (1997) *Limits of formal methods.*

[17] Maclaine Pont, P.G. (2004) *RIES facts and features sheet, version 1.0.* Retrieved September 2005 from www.surfnet.nl/bijeenkomsten/ries/RIES_Word1.doc.

[18] Maclaine Pont, P.G. (2004) *Simpel, goedkoop, doelmatig, pragmatisch: RIES!* Utrecht.

[19] Mateescu, R. and Sighireanu, M. (2003) Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In *Science of Computer Programming, 46(3)*: 255-281.

[20] Meadows, C. (2003) *Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends*

[21] Mote, C.D. (2001) *Report of the National Workshop on Internet Voting: Issues and Research Agenda.*

[22] Niemoller, K. (2004) *Experience with voting machines in the Netherlands and Germany.* The policy institute, Trinity College Dublin.

[23] Pfitzmann, A. and Hansen, M. (2006) *Anonymity, Unlinkability, Unobservability, Pseudonymity, and Identity Management A Consolidated Proposal for Terminology.* Version v0.28. Retrieved from http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.28.pdf.

[24] Pitt, M. (2002) *Modeling and verification of security protocols. Part 1: Basics of cryptography and introduction to security protocols.* Dresden University of Technology.

[25] Salomonson, G. (2004) *RIES and better?* Retrieved from http://www.surfnet.nl/bijeenkomsten/ries/salomonson.ppt.

[26] Turing, A.M. (1936) On computable numbers, with an application to the Entscheidungs problem. *Proc. London Mathematical Soc., Ser. 2-42, pp. 230-265 (see p. 247), 1936.*

[27] Varpaaniemi, K. (1994) On-the-fly verification with PROD. In *Jrg Desel, Andreas Oberweis, and Wolfgang Reisig, editors, Algorithmen und Werkzeuge fr Petrinetze: Workshop der GI-Fachgruppe 0.0.1, "Petrinetze und verwandte Systemmodelle", Berlin, 10.11. Oktober 1994*, pages 8083. Forschungsberichte, Bericht 309, Institut fr Angewandte Informatik und Formale Beschreibungsverfahren, Universitt Karlsruhe (TH), Germany, 1994.

[28] Wang, W., Hidvegi, Z., Bailey, A.D., Whinston, A.B. (2000) E-Process Design and Assurance Using Model Checking. In *Computer, October 2000.*

Items retrieved from the World Wide Web:

[29] *BBC NEWS - Europe - Italian right 'tried to rig poll'.* November, 2006. Consulted at http://news.bbc.co.uk/2/hi/europe/6180412.stm.

[30] *CADP On-Line Manual Pages.* No date. Consulted at http://www.inrialpes.fr/vasy/cadp/man.

[31] *Documenten - Verkiezingen 2003* No date. Consulted December 2006 at http://www.nrc.nl.

[32] *Eerste internetstemmen Tweede Kamerverkiezingen.* (2006) Retrieved from http://www.nu.nl/news/890089/2000/Eerste_internetstemmen_Tweede_Kamerverkiezingen.html.

[33] *Electronic voting in the Netherlands.* (2006) Consulted December 2006 at http://www.sos.cs.ru.nl/research/society/voting/main.html.

[34] *English - wijvertrouwenstemcomputersniet.* (2006) Consulted October 2006 from http://www.wijvertrouwenstemcomputersniet.nl/English.

[35] *First e-ballot via SMS text message proves a success in Switzerland.* (2005) Retrieved from http://www.heise.de/english/newsticker/news/65619.

[36] *Functionaliteiten RIES 2004* (2004) Retrieved from http://www.ttpi.nl/data/docs/TTPI Functionaliteiten RIES 20041116v24.pdf.

[37] *Herverkiezing Rijnland 2005. Protocol: Stemming en stemopneming.* March 2005. Retrieved from http://www.rijnlandkiest.nl.

[38] *Het Waterschapshuis - Internet Verkiezingen.* (2006) Consulted December 2006 at http://www.hetwaterschapshuis.nl.

[39] *International Organization for Standardization.* (1999) ISO99 ISO IS 15408. Originated from http://www.commoncriteria.org/.

[40] *Internetverkiezing in juni kwetsbaar.* (2004) Consulted at http://www. netkwesties .nl/editie92/artikel1.php.

[41] *Intrusion detection.* No date. Retrieved from http://winsnort.com.

[42] *Kiezen uit het buitenland.* No date. Consulted December 2006 at http://www.kiezenuithetbuitenland.nl.

[43] *Message Authentication Code.* No date. Retrieved from http://en.wikipedia.org/wiki/Message_authentication_code.

[44] *Ministerie van Binnenlandse Zaken en Koninklijkheidsrelaties* No date. Consulted December 2006 at http://www.minbzk.nl.

[45] *Online verkiezingen.* No date. Consulted December 2006 at http://www.kennislink.nl.

[46] *Open Standaarden en Open Source Software voor de overheid.* No date. Consulted at http://www.OSOSS.nl.

[47] *Parlement en Politiek - Stemmen in het buitenland.* No date. Consulted December 2006 at http://www.parlement.com.

[48] *Prime factors* No date. Retrieved October 2005 from http://www.primefactors.com

[49] *RIES KOA TALLY RESULTAAT.* No date. Retrieved from http://www.kiezenuithetbuitenland.nl/upload/uitslag.txt.

[50] *RIES - Rijnland Internet Election System.* May 2005. Retrieved from http://www.terena.nl/tech/task-forces/tf-csirt/meeting15/ries-meijer.pdf.

[51] *RIES (Rijnland Internet Election System) for 1.7 million voters.* (2004) Retrieved from http://www.surfnet.nl/bijeenkomsten/ries/RIES_gen_proj_descr_English.doc

[52] *Rijnland kiest.* No date. Consulted October 2005 at http://www.rijnlandkiest.nl.

[53] *Rijnland - website - Fully transparent election system.* No date. Consulted October 2005 at http://www.rijnland.net/externe_toegang/ries_rijnland/ries_fully

[54] *SSL.* No date. Retrieved from http://en.wikipedia.org/wiki/SSL.

[55] *Uitslag van de Tweede Kamerverkiezing van 22 november 2006.* November 2006. Consulted December 2006 at http://www.kiesraad.nl.

[56] *Verified Voting Foundation : Index.* No date. Consulted February 12th, 2007 at http://verifiedvotingfoundation.org.

[57] *Verkiezingsuitslag van de Tweede Kamerverkiezingen.* November 2006. Consulted December 2006 at http://statline.cbs.nl.

[58] *VVD pleit voor stemmen via internet bij komende Tweede-Kamerverkiezingen.* (2005) Retrieved from http://www.vvd.nl/index.aspx?Contentid=2321&Chapterid=1147&Filterid=974.

[59] *What is SSL?* No date. Retrieved from http://www.webopedia.com/TERM/S/SSL.html.

[60] *Wij vertrouwen stemcomputers niet.* No date. Consulted at http://www.wijvertrouwenstemcomputersniet.nl.

A μCRL specification of the RIES protocol

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Datatypes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sort Bool
func T,F :-> Bool
map and, or : Bool # Bool -> Bool
  not : Bool -> Bool
  eq : Bool # Bool -> Bool
  if : Bool # Bool # Bool -> Bool
var b,b' : Bool
rew and(b,T) = b
    and(b,F) = F
    or(b,T) = T
    or(b,F) = b
    not(T) = F
    not(F) = T
    eq(T,T) = T
    eq(F,F) = T
    eq(T,F) = F
    eq(F,T) = F
    if(T,b,b') = b
    if(F,b,b') = b'

sort Nat
func zero : -> Nat
   S : Nat -> Nat
map eq,less,leq : Nat # Nat -> Bool
var nat1,nat2 : Nat
rew eq(zero,zero)=T
    eq(zero,S(nat1))=F
    eq(S(nat1),zero)=F
    eq(S(nat1),S(nat2))=eq(nat1,nat2)
    less(zero,zero)=F
    less(S(nat1),zero)=F
    less(zero,S(nat1))=T
    less(S(nat1),S(nat2))=less(nat1,nat2)
    leq(zero,zero)=T
    leq(S(nat1),zero)=F
    leq(zero,S(nat1))=T
    leq(S(nat1),S(nat2))=leq(nat1,nat2)
```

```
sort ElectionId
func EI1,EI2 :-> ElectionId
map eq: ElectionId # ElectionId -> Bool
rew eq(EI1,EI1) = T
    eq(EI2,EI2) = T
    eq(EI1,EI2) = F
    eq(EI2,EI1) = F

sort Key
func Key1,Key2 :-> Key
map eq: Key # Key -> Bool
rew eq(Key1,Key1)= T
    eq(Key2,Key2)= T
    eq(Key1,Key2) = F
    eq(Key2,Key1) = F

sort keyTable
func ktable : Key # keyTable -> keyTable
   kempty :-> keyTable
map eq : keyTable # keyTable -> Bool
  remove : Key # keyTable -> keyTable
  notEmpty : keyTable -> Bool
  element : Key # keyTable -> Bool
    if: Bool #  keyTable # keyTable -> keyTable
var key,key1,key2 : Key
  kt1,kt2 : keyTable
rew eq(ktable(key1,kt1), ktable(key2,kt2)) =
      and(eq(key1,key2),eq(kt1,kt2))
  eq(kempty, ktable(key1,kt1)) = F
  eq(ktable(key1,kt1),kempty) = F
  eq(kempty,kempty) = T
remove(key1,ktable(key2,kt1)) =
      if(eq(key1,key2),kt1,ktable(key2,remove(key1,kt1)))
  notEmpty(kempty) = F
  notEmpty(ktable(key1,kt1)) = T
  element(key,kempty) = F
    element(key,ktable(key1,kt1)) = if(eq(key,key1),T,element(key,kt1))
  if(T,kt1,kt2) = kt1
  if(F,kt1,kt2) = kt2

sort EncryptedVoterId
func MAC : Key # ElectionId -> EncryptedVoterId
map eq : EncryptedVoterId # EncryptedVoterId -> Bool
    illegal : EncryptedVoterId -> Bool
    value : EncryptedVoterId -> Nat
```

```
var key1,key2 : Key
    ei1,ei2 : ElectionId
rew eq(MAC(key1,ei1),MAC(key2,ei2)) = and(eq(key1,key2),eq(ei1,ei2))
    value(MAC(Key1,EI1)) = zero
    value(MAC(Key2,EI1)) = S(zero)
    value(MAC(Key1,EI2)) = S(S(zero))
    value(MAC(Key2,EI2)) = S(S(S(zero)))
    illegal(MAC(key1,EI1)) = F
    illegal(MAC(key1,EI2)) = T

sort CandidateId
func CI1,CI2 :-> CandidateId
map eq : CandidateId # CandidateId -> Bool
rew eq(CI1,CI1) = T
    eq(CI2,CI2) = T
    eq(CI1,CI2) = F
    eq(CI2,CI1) = F

sort VoterId
func VI1,VI2 :-> VoterId
map eq : VoterId # VoterId -> Bool
rew eq(VI1,VI1) = T
    eq(VI2,VI2) = T
    eq(VI1,VI2) = F
    eq(VI2,VI1) = F

sort EncryptedCandidateId
func MAC : Key # CandidateId -> EncryptedCandidateId
map eq: EncryptedCandidateId # EncryptedCandidateId -> Bool
    value : EncryptedCandidateId -> Nat
    recalci : EncryptedCandidateId -> CandidateId
var key1,key2,key : Key
    ci1,ci2,ci : CandidateId
rew eq(MAC(key1,ci1),MAC(key2,ci2)) = and(eq(key1,key2),eq(ci1,ci2))
    value(MAC(key,CI1)) = zero
    value(MAC(key,CI2)) = S(zero)
    recalci(MAC(key,ci)) = ci

sort candidateTable
func ctable : CandidateId # candidateTable -> candidateTable
     ect :-> candidateTable
map eq : candidateTable # candidateTable -> Bool
    element : CandidateId # candidateTable -> Bool
    del,remove : CandidateId # candidateTable -> candidateTable
    notEmpty : candidateTable -> Bool
```

```
    if: Bool #  candidateTable # candidateTable -> candidateTable
var ci1,ci2 : CandidateId
    ct1, ct2 : candidateTable
rew eq(ctable(ci1,ct1),ctable(ci2,ct2)) =
      if(eq(ci1,ci2),eq(ct1,ct2), F)
    eq(ect,ect) = T
    eq(ect,ctable(ci1,ct1)) = F
    eq(ctable(ci1,ct1),ect) = F
    element(ci1,ctable(ci2,ct1)) = if(eq(ci1,ci2),T,element(ci1,ct1))
    element(ci1,ect) = F
    del(ci1,ctable(ci2,ct1)) =
      if(eq(ci1,ci2),
        ct1,
      ctable(ci2,del(ci1,ct1)))
    del(ci1,ect) = ect
    remove(ci1,ctable(ci2,ct2)) =
      if(eq(ci1,ci2),ct2,ctable(ci2,remove(ci1,ct2)) )
    remove(ci1,ect) = ect
    notEmpty(ctable(ci1,ct1)) = T
    notEmpty(ect) = F
    if(T,ct1,ct2) = ct1
  if(F,ct1,ct2) = ct2

sort countsTable
func cntstable : CandidateId # Nat # countsTable -> countsTable
     cempty :-> countsTable
map eq : countsTable # countsTable -> Bool
    if : Bool # countsTable # countsTable -> countsTable
    ifci : Bool # CandidateId # CandidateId -> CandidateId
    increase,decrease,remove : CandidateId # countsTable -> countsTable
    voted,element : CandidateId # countsTable -> Bool
    getAdd, getRemove : countsTable -> CandidateId
    getDiff : countsTable # countsTable -> countsTable
    added,notEmpty : countsTable -> Bool
var evi1 : EncryptedVoterId
    eci1 : EncryptedCandidateId
    ci1,ci2 : CandidateId
    nat1,nat2 : Nat
    ct1,ct2 : countsTable
rew eq(cntstable(ci1,nat1,ct1),cntstable(ci2,nat2,ct2)) =
      if(and(eq(ci1,ci2),eq(nat1,nat2)),eq(ct1,ct2),F)
    eq(cempty,cempty) = T
    eq(cempty,cntstable(ci1,nat1,ct1)) = F
    eq(cntstable(ci1,nat1,ct1),cempty) = F
    if(T,ct1,ct2) = ct1
```

```
    if(F,ct1,ct2) = ct2
    ifci(T,ci1,ci2) = ci1
    ifci(F,ci1,ci2) = ci2
    increase(ci2,cntstable(ci1,nat1,ct1)) =
      if(eq(ci2,ci1),cntstable(ci1,S(nat1),ct1),
      cntstable(ci1,nat1,increase(ci2,ct1)))
    increase(ci2,cempty)=cempty
    decrease(ci2,cntstable(ci1,S(nat1),ct1)) =
      if(eq(ci2,ci1), cntstable(ci1,nat1,ct1),
      cntstable(ci1, S(nat1), decrease(ci2,ct1)) )
    decrease(ci2,cntstable(ci1,zero,ct1)) =
      cntstable(ci1,zero,decrease(ci2,ct1))
    remove(ci1,cntstable(ci2,nat2,ct2)) =
      if(eq(ci1,ci2),ct2,cntstable(ci2,nat2,remove(ci1,ct2)))
    remove(ci1,cempty) = cempty
    voted(ci2,cntstable(ci1,nat1,ct1)) =
      if( eq(ci2,ci1),if(eq(nat1,zero),F,T),voted(ci2,ct1))
    voted(ci1,cempty) = F
    element(ci1,cntstable(ci2,nat2,ct2)) =
      if(eq(ci1,ci2),T,element(ci1,ct2))
    element(ci1,cempty) = F
    getAdd(cntstable(ci1,nat1,cntstable(ci2,nat2,ct2))) =
      ifci(less(nat1,nat2),ci1,getAdd(ct2))
    getRemove(cntstable(ci1,nat1,cntstable(ci2,nat2,ct2))) =
      ifci(less(nat2,nat1),ci1,getRemove(ct2))
    getRemove(cempty) = CI1
    getDiff(cntstable(ci1,nat1,ct1),cntstable(ci2,nat2,ct2)) =
      if(eq(nat1,nat2),getDiff(ct1,ct2),
      cntstable(ci1,nat1,cntstable(ci2,nat2,getDiff(ct1,ct2))))
    getDiff(cempty,cempty) = cempty
    added(cntstable(ci1,nat1,cntstable(ci2,nat2,ct2))) =
      if(less(nat1,nat2), T, added(ct2))
    added(cempty) = F
    notEmpty(cntstable(ci1,nat1,ct1)) = T
    notEmpty(cempty) = F

sort MDCvote
func MDC : EncryptedVoterId # EncryptedCandidateId -> MDCvote
map eq : MDCvote # MDCvote -> Bool
    computemdc : EncryptedVoterId # EncryptedCandidateId -> MDCvote
var evi1, evi2 : EncryptedVoterId
    eci1, eci2 : EncryptedCandidateId
rew eq(MDC(evi1,eci1), MDC(evi2,eci2)) = and(eq(evi1,evi2),eq(eci1,eci2))
    computemdc(evi1,eci1) = MDC(evi1,eci1)
```

```
sort referenceTable
func rtable : MDCvote # referenceTable -> referenceTable
     rempty :-> referenceTable
map  eq : referenceTable # referenceTable -> Bool
     ref : MDCvote -> CandidateId
var  rt1,rt2 : referenceTable
     evi1,evi2 : EncryptedVoterId
     eci1,eci2 : EncryptedCandidateId
     key1,key2 : Key
     ci1 : CandidateId
     ei : ElectionId
rew  eq( rtable(MDC(evi1,eci1), rt1), rtable(MDC(evi2,eci2),rt2)) =
      and( eq(evi1,evi2), and( eq(eci1,eci2),eq(rt1,rt2)))
     eq(rempty,rempty) = T
     eq(rempty, rtable(MDC(evi1,eci1),rt1)) = F
     eq(rtable(MDC(evi1,eci1),rt1), rempty) = F
%%%%%%%%%%%%%%%%%%%%%%%%%---------ref---------%%%%%%%%%%%%%%%%%%%%%%%%%
% the following definition is used in all scenario's but changeRef
     ref(MDC(MAC(key1,ei),MAC(key1,ci1))) = ci1
% the following definition is used only in changeRef
     ref(MDC(MAC(key1,ei),MAC(key1,CI1))) = CI2
     ref(MDC(MAC(key1,ei),MAC(key1,CI2))) = CI2
% the following definition cannot occur but is needed for completeness
% a vote composed of two different keys cannot exist
     ref(MDC(MAC(key1,ei),MAC(key2,ci1))) = ci1
%%%%%%%%%%%%%%%%%%%%%%%%%---------------------%%%%%%%%%%%%%%%%%%%%%%%%%

sort MDCTable
func mdctable : MDCvote # EncryptedVoterId # EncryptedCandidateId
              # MDCTable -> MDCTable
     mdcempty : -> MDCTable
map eq : MDCTable # MDCTable -> Bool
    if : Bool # MDCTable # MDCTable -> MDCTable
    if : Bool # EncryptedVoterId # EncryptedVoterId  -> EncryptedVoterId
    present,invalid : EncryptedVoterId # EncryptedCandidateId # MDCTable
                    -> Bool
    presentAll : Table # MDCTable -> Bool
    getAdd,getRemove : Table # MDCTable -> Table
    added : Table # MDCTable -> Bool
    mdcclean,correct : MDCTable # referenceTable -> MDCTable
    getValid: MDCTable -> MDCTable
    more : EncryptedVoterId # MDCTable -> Bool
    remove:EncryptedVoterId # MDCTable -> MDCTable
    delete : EncryptedVoterId # EncryptedCandidateId # MDCTable
           -> MDCTable
```

```
    computemdc : Table -> MDCTable
var mdc1,mdc2 : MDCTable
    t,v : Table
    rt1: referenceTable
    mdcvote1,mdcvote2, mdcvote3 : MDCvote
    evi1, evi2, evi3 : EncryptedVoterId
    eci1, eci2, eci3 : EncryptedCandidateId
rew eq(mdctable(mdcvote1,evi1,eci1,mdc1),
      mdctable(mdcvote2,evi2,eci2,mdc2)) =
        and(eq(evi1,evi2),and(eq(eci1,eci2),eq(mdc1,mdc2)))
    eq(mdcempty,mdcempty) = T
    eq(mdcempty, mdctable(mdcvote1,evi1,eci1,mdc1)) = F
    eq(mdctable(mdcvote1,evi1,eci1,mdc1), mdcempty) = F
    if(T,mdc1,mdc2) = mdc1
    if(F,mdc1,mdc2) = mdc2
    if(T,evi1,evi2) = evi1
    if(F,evi1,evi2) = evi2
    if(T,t,v) = t
    if(F,t,v) = v
    present(evi1,eci1,mdctable(mdcvote1,evi2,eci2,mdc1)) =
      if(and(eq(evi1,evi2),eq(eci1,eci2)),T,present(evi1,eci1,mdc1))
    present(evi1,eci1,mdcempty) = F
    presentAll(table(evi1,eci1,t),mdctable(mdcvote1,evi2,eci2,mdc1)) =
      if(and(eq(evi1,evi2),eq(eci1,eci2)),presentAll(t,mdc1),F)
    presentAll(empty,mdcempty) = T
    presentAll(empty,mdctable(mdcvote1,evi1,eci1,mdc1)) = F
    presentAll(table(evi1,eci1,t),mdcempty) = F
    getAdd(table(evi1,eci1,t),mdctable(mdcvote2,evi2,eci2,mdc1)) =
    if(and(eq(evi1,evi2),eq(eci1,eci2)),
      getAdd(t,mdc1),
    table(evi2,eci2,empty))
    getAdd(empty,mdctable(mdcvote2,evi2,eci2,mdc1)) =
      table(evi2,eci2,empty)
    getRemove(table(evi1,eci1,t),mdctable(mdcvote2,evi2,eci2,mdc1)) =
    if(and(eq(evi1,evi2),eq(eci1,eci2)),
      getRemove(t,mdc1),
    table(evi1,eci1,empty))
    getRemove(table(evi1,eci1,t),mdcempty) = table(evi1,eci1,t)
    added(table(evi1,eci1,t),mdcempty) = F
    added(empty,mdctable(mdcvote2,evi2,eci2,mdc1)) = T
    added(table(evi1,eci1,t),mdctable(mdcvote2,evi2,eci2,mdc1)) =
      added(t,mdc1)
    mdcclean(mdcempty, rt1) = mdcempty
    mdcclean(mdc1,rt1) = getValid(correct(mdc1,rt1))
    correct( mdctable(mdcvote1,evi1,eci1,mdc1), rt1 ) =
```

```
      if(elementRef(mdcvote1, rt1),
        mdctable(mdcvote1,evi1,eci1,correct(mdc1,rt1)),
      correct(mdc1,rt1))
    correct(mdcempty,rt1) = mdcempty
    getValid(mdctable(mdcvote1,evi1,eci1,mdc1)) =
      if(more(evi1,mdc1),
        if(invalid(evi1,eci1,mdc1),
          getValid(remove(evi1,mdctable(mdcvote1,evi1,eci1,mdc1))),
        mdctable(mdcvote1,evi1,eci1,getValid(remove(evi1,mdc1)))),
      mdctable(mdcvote1,evi1,eci1,getValid(mdc1))  )
    getValid(mdcempty) = mdcempty
    invalid(evi1,eci1,mdctable(mdcvote2,evi2,eci2,mdc1)) =
    if(eq(evi1,evi2),
      if(eq(eci1,eci2),
        invalid(evi1,eci1,mdc1),
      T),
    invalid(evi1,eci1,mdc1))
    invalid(evi1,eci1,mdcempty) = F
    more(evi1,mdctable(mdcvote2,evi2,eci2,mdc1)) =
      if(eq(evi1,evi2),T,more(evi1,mdc1))
    more(evi1,mdcempty) = F
    remove(evi1,mdctable(mdcvote2,evi2,eci2,mdc1)) =
      if(eq(evi1,evi2),
        remove(evi1,mdc1),
      mdctable(mdcvote2,evi2,eci2,remove(evi1,mdc1)))
    remove(evi1, mdcempty) = mdcempty
    delete(evi1,eci1,mdctable(mdcvote2,evi2,eci2,mdc1)) =
   if(eq(evi1,evi2),
     if(eq(eci1,eci2),
      mdc1,
     mdctable(mdcvote2,evi2,eci2,delete(evi1,eci1,mdc1))),
   mdctable(mdcvote2,evi2,eci2,delete(evi1,eci1,mdc1)))
    computemdc(empty) = mdcempty
    computemdc(table(evi1,eci1,t)) =
      mdctable(MDC(evi1,eci1),evi1,eci1,computemdc(t))

sort Table
func table : EncryptedVoterId # EncryptedCandidateId # Table -> Table
     empty : -> Table
map eq : Table # Table -> Bool
    ifct : Bool # countsTable # countsTable -> countsTable
    ifk : Bool # Key # Key -> Key
    if : Bool # Table # Table -> Table
    add,addDup, remove,find,addsorted,delete,removeValid :
      EncryptedVoterId # EncryptedCandidateId # Table -> Table
```

```
    makeInvalid : Key # ElectionId # candidateTable # Table
                -> Table
    retrieveUncast : EncryptedVoterId # MDCTable # Table -> Table
    count : MDCTable # countsTable -> countsTable
    up : CandidateId # countsTable -> countsTable
    clean,correct : Table # referenceTable -> Table
    elementRef : MDCvote # referenceTable -> Bool
    getValid,undouble,removeFirst : Table -> Table
    more,elementV,used : EncryptedVoterId # Table -> Bool
    remove : EncryptedVoterId # Table -> Table
    more,elementC : EncryptedCandidateId # Table -> Bool
    remove : EncryptedCandidateId # Table -> Table
    retrieveV : Table -> EncryptedVoterId
    retrieveC : Table -> EncryptedCandidateId
    element,invalid : EncryptedVoterId # EncryptedCandidateId  # Table
                    -> Bool
    evichoice : EncryptedVoterId # Table # Table -> Table
    getci : EncryptedVoterId # EncryptedCandidateId  -> CandidateId
    getkey : EncryptedVoterId # EncryptedCandidateId # keyTable
           # ElectionId -> Key
    getUnusedkeys : keyTable # Table # ElectionId -> keyTable
    computeNotVoted: Table # Table -> Table
    notEmpty : Table -> Bool
var evi1,evi2 : EncryptedVoterId
    eci1,eci2 : EncryptedCandidateId
    t,v : Table
    ct1,ct2 : countsTable
    key1,key2 : Key
    mdcvote1,mdcvote2 : MDCvote
    mdc1 : MDCTable
    ci1,ci2 : CandidateId
    cit1 : candidateTable
    nat1 : Nat
    rt1 : referenceTable
    ei : ElectionId
    kt1,kt2 : keyTable

rew eq(table(evi1,eci1,t),table(evi2,eci2,v)) =
     and(eq(evi1,evi2),and(eq(eci1,eci2),eq(t,v)))
    eq(empty,empty) = T
    eq(empty, table(evi1,eci1,t)) = F
    eq(table(evi1,eci1,t), empty) = F
    ifct(T,ct1,ct2) = ct1
    ifct(F,ct1,ct2) = ct2
    ifk(T,key1,key2) = key1
```

```
    ifk(F,key1,key2) = key2
    if(T,t,v) = t
    if(F,t,v) = v
%%%%%%%%%%%%%%%%%%%%%%%%%%---------add---------%%%%%%%%%%%%%%%%%%%%%%%%%
% The following definition is used when duplicates are not allowed.
    add(evi1,eci1,empty) = table(evi1,eci1,empty)
    add(evi1,eci1,table(evi2,eci2,t)) =
      if(less(value(evi1),value(evi2)),
        table(evi1,eci1,table(evi2,eci2,t)),
      if(eq(value(evi1),value(evi2)),
        if(less(value(eci1),value(eci2)),
          table(evi1,eci1,table(evi2,eci2,t)),
        if(eq(value(eci1),value(eci2)),
          table(evi1,eci1,t),
        table(evi2,eci2,add(evi1,eci1,t)))),
      table(evi2,eci2,add(evi1,eci1,t))))

% The following definition is used when duplicates are allowed. It is
% based on two uses of the general add function. If the adding of a 2nd
% duplicate is concerned, it is only added if the remaining table does not
% contain another duplicate.
    add(evi1,eci1,empty) = table(evi1,eci1,empty)
    add(evi1,eci1,table(evi2,eci2,t)) =
      if(less(value(evi1),value(evi2)),
        table(evi1,eci1,table(evi2,eci2,t)),
      if(eq(value(evi1),value(evi2)),
        if(less(value(eci1),value(eci2)),
          table(evi1,eci1,table(evi2,eci2,t)),
        if(eq(value(eci1),value(eci2)),
          table(evi2,eci2,addDup(evi1,eci1,t)),
        table(evi2,eci2,add(evi1,eci1,t)))),
      table(evi2,eci2,add(evi1,eci1,t))))

    addDup(evi1,eci1,empty)=table(evi1,eci1,empty)
    addDup(evi1,eci1,table(evi2,eci2,t))=
      if(less(value(evi1),value(evi2)),
       table(evi1,eci1,table(evi2,eci2,t)),
      if(eq(value(evi1),value(evi2)),
        if(less(value(eci1),value(eci2)),
          table(evi1,eci1,table(evi2,eci2,t)),
        if(eq(value(eci1),value(eci2)),
          table(evi2,eci2,t),
        table(evi2,eci2,add(evi1,eci1,t)))),
      table(evi2,eci2,add(evi1,eci1,t))))
```

```
% The following definition to allow duplicates creates an inefficiently
% large statespace. It first adds a vote and then checks to see if the
% table already contained second duplicates. If it does, one duplicate is
% removed.
    add(evi1,eci1,t) = addsorted(evi1,eci1,find(evi1,eci1,t))
    addsorted(evi1,eci1,empty) = table(evi1,eci1,empty)
    addsorted(evi1,eci1,table(evi2,eci2,t)) =
      if(less(value(evi1),value(evi2)),
        table(evi1,eci1,table(evi2,eci2,t)),
      if(eq(value(evi1),value(evi2)),
        if(leq(value(eci1),value(eci2)),
          table(evi1,eci1,table(evi2,eci2,t)),
        table(evi2,eci2,addsorted(evi1,eci1,t))),
      table(evi2,eci2,addsorted(evi1,eci1,t))))
    find(evi1,eci1,table(evi2,eci2,t)) =
      table(evi2,eci2,
      if(and(eq(evi1,evi2),eq(eci1,eci2)),
        remove(evi1,eci1,t),
      find(evi1,eci1,t)))
    find(evi1,eci1,empty) = empty
    remove(evi1,eci1,table(evi2,eci2,t)) =
    if(and(eq(evi1,evi2),eq(eci1,eci2)),
      t,
    table(evi1,eci1,remove(evi1,eci1,t)))
    remove(evi1,eci1,empty) = empty
%%%%%%%%%%%%%%%%%%%%%%%%%%----------------------%%%%%%%%%%%%%%%%%%%%%%%%%%

    removeValid(evi1,eci1,table(evi2,eci2,t)) =
    if(eq(evi1,evi2),
      if(eq(eci1,eci2),
          removeValid(evi1,eci1,t),
      table(evi2,eci2,removeValid(evi1,eci1,t)) ),
    table(evi2,eci2,removeValid(evi1,eci1,t)) )
    removeValid(evi1,eci1,empty) = empty
    makeInvalid(key1,ei,ci1,t)= add(MAC(key1,ei),MAC(key1,ci1),t)
    retrieveUncast(evi1,mdcempty,t) = t
    retrieveUncast(evi1,mdctable(mdcvote2,evi2,eci2,mdc1),t) =
      if(eq(evi1,evi2),
        retrieveUncast(evi1,mdc1,add(evi2,eci2,t)),
      retrieveUncast(evi1,mdc1,t))
    count(mdcempty,ct1) = ct1
    count( mdctable(mdcvote1,evi1,eci1,mdc1),cntstable(ci1,nat1,ct1)) =
      count(mdc1,up(ref(mdcvote1),cntstable(ci1,nat1,ct1)))
    up(ci2,cntstable(ci1,nat1,ct1)) =
      if(eq(ci2,ci1),
```

```
    cntstable(ci1,S(nat1),ct1),
  cntstable(ci1,nat1,up(ci2, ct1)))
clean(t,rt1)=getValid(correct(t,rt1))
correct( table(evi1,eci1,t), rt1 ) =
  if(elementRef(computemdc(evi1,eci1), rt1),
    table(evi1,eci1,correct(t,rt1)),
  correct(t,rt1))
correct(empty,rt1) = empty
elementRef(mdcvote2,rtable(mdcvote1,rt1)) =
  if(eq(mdcvote2,mdcvote1),T,elementRef(mdcvote2,rt1))
elementRef(mdcvote2,rempty)=F
getValid(table(evi1,eci1,t)) =
  if(more(evi1,t),
    if(invalid(evi1,eci1,t),
      getValid(remove(evi1,table(evi1,eci1,t))),
    table(evi1,eci1,getValid(remove(evi1,t)))),
  table(evi1,eci1,getValid(t)) )
getValid(empty) = empty
more(evi1,table(evi2,eci2,t)) =
  if(eq(evi1,evi2),
    T,
  more(evi1,t))
more(evi1,empty) = F
more(eci1,table(evi2,eci2,t)) =
  if(eq(eci1,eci2),
    T,
  more(eci1,t))
more(eci1,empty) = F
invalid(evi1,eci1,table(evi2,eci2,t)) =
  if(eq(evi1,evi2),
    if(eq(eci1,eci2),
      invalid(evi1,eci1,t),
    T),
  invalid(evi1,eci1,t))
invalid(evi1,eci1,empty) = F
remove(evi1,table(evi2,eci2,t)) =
  if(eq(evi1,evi2),
    remove(evi1,t),
  table(evi2,eci2,remove(evi1,t)))
remove(evi1,empty) = empty
undouble(table(evi1,eci1,t)) =
  if(more(eci1,t),
    table(evi1,eci1,undouble(remove(eci1,t))),
  table(evi1,eci1,undouble(t)))
undouble(empty) = empty
```

```
    remove(eci1,table(evi2,eci2,t)) =
      if(eq(eci1,eci2),
        remove(eci1,t),
      table(evi2,eci2,remove(eci1,t)))
    remove(eci1,empty) = empty
    removeFirst(table(evi1,eci1,t)) = t
    used(evi2,table(evi1,eci1,t)) =
      if(eq(evi2,evi1),T,used(evi2,t))
    used(evi2,empty) = F
    element(evi1,eci1,empty) = F
    element(evi1,eci1,table(evi2,eci2,t)) =
    if(eq(evi1,evi2),
      if(eq(eci1,eci2),
      T,
      element(evi1,eci1,t)),
    element(evi1,eci1,t))
    elementV(evi1,empty) = F
    elementV(evi1,table(evi2,eci2,t))  =
    if(eq(evi1,evi2),
      T,
    elementV(evi1,t))
    elementC(eci1,empty) = F
    elementC(eci1,table(evi2,eci2,t))  =
   if(eq(eci1,eci2),
     T,
   elementC(eci1,t))
    retrieveV(table(evi1,eci1,t)) = evi1
    retrieveC(table(evi1,eci1,t)) = eci1
% the following two definitions cannot occur but are needed for
% completeness
    retrieveV(empty) = MAC(Key1,EI1)
    retrieveC(empty) = MAC(Key1,CI1)
    evichoice(evi1,table(evi2,eci2,t),v) =
    if(eq(evi1,evi2),
      evichoice(evi1,t,add(evi1,eci2,v)),
    evichoice(evi1,t,v))
    evichoice(evi1,empty,v) = v
    delete(evi1,eci1,table(evi2,eci2,t)) =
    if(eq(evi1,evi2),
      if(eq(eci1,eci2),
      t,
      table(evi2,eci2,delete(evi1,eci1,t))),
    table(evi2,eci2,delete(evi1,eci1,t)))
    delete(evi1,eci1,empty)=empty
    getci(evi1,eci1) = ref(MDC(evi1,eci1))
```

```
    getkey(evi1,eci1,ktable(key1,kt1),ei) =
    ifk(eq(evi1,MAC(key1,ei)),
    key1,
    getkey(evi1,eci1,kt1,ei))
% the following definition cannot occur but is needed for completeness
    getkey(MAC(key1,ei),eci1,kempty,ei) =  key1
    getUnusedkeys(ktable(key1,kt1), t, ei) =
    if(used(MAC(key1,ei),t),
      getUnusedkeys(kt1,t,ei),
      ktable(key1,getUnusedkeys(kt1,t,ei)))
    getUnusedkeys(kempty,t,ei) = kempty
    computeNotVoted(table(evi1,eci1,t),table(evi2,eci2,v)) =
      if(and(eq(evi1,evi2),eq(eci1,eci2)),
        computeNotVoted(t,v),
      if(illegal(evi2),
        table(evi1,eci1,t),
      table(evi1,eci1,computeNotVoted(t,table(evi2,eci2,v)))))
    computeNotVoted(t,empty)=t
    computeNotVoted(empty,v)=empty
    notEmpty(empty) = F
    notEmpty(table(evi1,eci1,t)) = T

sort elidTable
func eitable : ElectionId # elidTable -> elidTable
     eiempty :-> elidTable
map eq : elidTable # elidTable -> Bool
    element : ElectionId # elidTable -> Bool
var ei1,ei2 : ElectionId
    eit1,eit2 : elidTable
rew eq(eitable(ei1,eit1), eitable(ei2,eit2)) =
    and(eq(ei1,ei2), eq(eit1,eit2))
    eq(eitable(ei1,eit1), eiempty) = F
    eq(eiempty,eitable(ei2,eit2)) = F
    element(ei1,eitable(ei2,eit1)) = if(eq(ei1,ei2),T,element(ei1,eit1))
    element(ei1,eiempty) = F

sort voterTable
func vtable : VoterId # voterTable -> voterTable
     vempty :-> voterTable
map eq : voterTable # voterTable -> Bool
    remove : VoterId # voterTable -> voterTable
    notEmpty : voterTable -> Bool
    element : VoterId # voterTable -> Bool
    if : Bool # voterTable # voterTable -> voterTable
var vi,vi1,vi2 : VoterId
```

```
    vt1,vt2 : voterTable
rew eq(vtable(vi1,vt1), vtable(vi2,vt2)) = and(eq(vi1,vi2),eq(vt1,vt2))
    eq(vempty, vtable(vi1,vt1)) = F
    eq(vtable(vi1,vt1), vempty) = F
    eq(vempty,vempty) = T
    remove(vi1,vtable(vi2,vt1)) =
      if(eq(vi1,vi2),
        vt1,
      vtable(vi2,remove(vi1,vt1)))
    notEmpty(vempty) = F
    notEmpty(vtable(vi1,vt1)) = T
    element(vi,vempty) = F
    element(vi,vtable(vi1,vt1)) = if(eq(vi,vi1),T,element(vi,vt1))
    if(T,vt1,vt2) = vt1
    if(F,vt1,vt2) = vt2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Actions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act

% General
send_ballot, receive_ballot, comm_ballot
    : Key # ElectionId # candidateTable

send_vote, receive_vote, comm_vote
    : EncryptedVoterId # EncryptedCandidateId

send_checkInfo, receive_checkInfoSURFnet, receive_checkInfo,
  comm_checkInfoSURFnet, comm_checkInfo
    : MDCTable # MDCTable # countsTable # referenceTable # countsTable

send_recVotes, rec_recVotes, comm_recVotesSURFnet : Table

continue
timeout
wait

% Verifiability actions: OK and detect
OKv, OKr, OKci,Voter_detectci, SURFnet_detectAdd, SURFnet_detectRemove,
  Voter_detectAdd, Voter_detectRemove
     : EncryptedVoterId # EncryptedCandidateId

SURFnet_detectAdd,SURFnet_detectRemove : CandidateId
```

```
OK : countsTable
OK : MDCTable

% Actions to signal fraud.
% Every action appears at most once in the scenario's.
send_voteCORRUPT, removeRec, makeInvalid, addValid, removeValid
  : EncryptedVoterId # EncryptedCandidateId

changeRef

addCount,subCount : candidateId

% Actions to communicate fraud information
sendFraudInfo, recFraudInfo, comm_recallFraudInfo, saveFraudInfo,
  keepFraudInfo, comm_storeFraudInfo, saveFraudInfo, getFraudInfo,
    comm_FraudInfo : candidateTable # keyTable # ElectionId

% Communication
comm  send_ballot | receive_ballot = comm_ballot
      send_vote | receive_vote  = comm_vote
      send_checkInfo | receive_checkInfo = comm_checkInfo
      send_checkInfo | receive_checkInfoSURFnet = comm_checkInfoSURFnet
% the following communications only occur in fraud scenario's
      send_recVotes | rec_recVotes = comm_recVotesSURFnet
      sendFraudInfo | recFraudInfo = comm_recallFraudInfo
      saveFraudInfo | keepFraudInfo = comm_storeFraudInfo
      saveFraudInfo | getFraudInfo = comm_FraudInfo

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Processes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc

%%%%%%%%%%%%%%%%%%%%%%---------TTPI_sendBallot---------%%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the following scenario's:
% makeInvalid, addValid, sendVote

TTPI_sendBallotCORRUPT(keytable:keyTable, ei:ElectionId,
  reftable:referenceTable, ctable:candidateTable, cntstable:countsTable) =

saveFraudInfo(ctable,keytable,ei) .
TTPI_sendBallot(keytable,ei,reftable,ctable,cntstable)

% The following definition is used in all other scenario's
TTPI_sendBallot(keytable:keyTable, ei:ElectionId,
```

```
  reftable:referenceTable, ctable:candidateTable,
    cntstable:countsTable) =

  (sum(key:Key,
    send_ballot(key,ei,ctable) .
    TTPI_sendBallot(remove(key,keytable), ei, reftable, ctable, cntstable)
    <| element(key,keytable) |> delta  )

<| notEmpty(keytable) |>

  SURFnet_receive(empty,reftable,cntstable)
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%----------FraudInfoMem-----------%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the following scenario's
% makeInvalid, addValid

FraudInfoMem =

sum(ctable:candidateTable, sum(keytable:keyTable, sum(ei:ElectionId,

keepFraudInfo(ctable,keytable,ei) . sendFraudInfo(ctable,keytable,ei)
  . delta
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%--------TTPI_castCORRUPT---------%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the sendVote scenario
TTPI_castCORRUPT =

sum(ctable:candidateTable, sum(keytable:keyTable, sum(ei:ElectionId,

getFraudInfoTTPI(ctable,keytable,ei) .

  (sum(ci:CandidateId, sum(key:Key,

  send_voteCORRUPT(MAC(key,ei),MAC(key,ci)) . delta

  <| and(element(ci,ctable),element(key,keytable)) |>

  delta )) ) )))
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%---------SURFnet_receive---------%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the removeReceived scenario
SURFnet_receive(receivedVotes:Table, reftable:referenceTable,
```

```
  cntstable:countsTable) =

(sum(eci:EncryptedCandidateId, sum(evi:EncryptedVoterId,
  receive_vote(evi,eci) .
  SURFnet_receive(add(evi,eci,receivedVotes),reftable,cntstable) ))
  + timeout) .

  (
  (sum(evi:EncryptedVoterId, sum(eci:EncryptedCandidateId,
      removeRec(evi,eci) .
      TTPI_count(delete(evi,eci,receivedVotes),reftable,cntstable)

      <| and(elementV(evi,receivedVotes),
         elementC(eci,evichoice(evi,receivedVotes,empty))) |>
      delta  )) )

  <| notEmpty(receivedVotes) |>

  TTPI_count(receivedVotes,reftable,cntstable)
  )

% The following definition is used in all other scenario's
SURFnet_receive(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

(sum(eci:EncryptedCandidateId, sum(evi:EncryptedVoterId,
  receive_vote(evi,eci) .
  SURFnet_receive(add(evi,eci,receivedVotes),reftable,cntstable) ))
  + timeout) .

(send_recVotes(receivedVotes) .
  TTPI_count(receivedVotes,reftable,cntstable))
%%%%%%%%%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%----------SURFnet_check----------%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the removeReceived scenario
SURFnet_checkCnt =

sum(receivedVotes:MDCTable, sum(validVotes:MDCTable,
  sum(countstable:countsTable, sum(reftable:referenceTable,
    sum(initCnts:countsTable,

receive_checkInfoSURFnet(receivedVotes,validVotes,countstable,reftable,
  initCnts) .
```

```
  (OK(countstable) . delta

  <| eq(count(mdcclean(receivedVotes,reftable),initCnts),countstable) |>

  SURFnet_getCorrupt(count(mdcclean(receivedVotes,reftable),initCnts),
  countstable) )

% The following definition is used in all other scenario's
SURFnet_checkRec =

sum(SURFnet_rec:Table, rec_recVotes(SURFnet_rec) .

sum(receivedVotes:MDCTable, sum(validVotes:MDCTable,
  sum(countstable:countsTable, sum(reftable:referenceTable,
    sum(initCnts:countsTable,

receive_checkInfoSURFnet(receivedVotes,validVotes,countstable,reftable,
  initCnts)
 .

(OK(receivedVotes)

<| presentAll(SURFnet_rec,receivedVotes) |>

  (SURFnet_detectAdd(retrieveV(getAdd(SURFnet_rec, receivedVotes)),
    retrieveC(getAdd(SURFnet_rec, receivedVotes)))

<| added(SURFnet_rec,receivedVotes) |>

SURFnet_detectRemove(retrieveV(getRemove(SURFnet_rec, receivedVotes)),
    retrieveC(getRemove(SURFnet_rec, receivedVotes))) ) ) .

SURFnet_checkCnt(count(mdcclean(receivedVotes,reftable),initCnts),
  countstable) ))))))

%%%%%
SURFnet_checkCnt(SURFnet_countstable:countsTable,
  TTPI_countstable:countsTable) =

OK(TTPI_countstable) . delta

<| eq(SURFnet_countstable, TTPI_countstable) |>

SURFnet_getCorrupt(SURFnet_countstable,TTPI_countstable)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%
SURFnet_getCorrupt(SURFnet_countstable:countsTable,
  TTPI_countstable:countsTable) =

  (delta

  <| eq(SURFnet_countstable, TTPI_countstable) |>

    ((SURFnet_detectAdd(getAdd(getDiff(SURFnet_countstable,
        TTPI_countstable))) .

    SURFnet_getCorrupt(
      remove(getAdd(getDiff(SURFnet_countstable,TTPI_countstable)),
        SURFnet_countstable),
      remove(getAdd(getDiff(SURFnet_countstable,TTPI_countstable)),
        TTPI_countstable)) )

    <| added(getDiff(SURFnet_countstable, TTPI_countstable)) |>

    (SURFnet_detectRemove(getRemove(
      getDiff(SURFnet_countstable,TTPI_countstable))) .

    SURFnet_getCorrupt(
      remove(getRemove(getDiff(SURFnet_countstable,TTPI_countstable)),
        SURFnet_countstable),
      remove(getRemove(getDiff(SURFnet_countstable,TTPI_countstable)),
        TTPI_countstable)) )
    )
  )

<| notEmpty(SURFnet_countstable) |>

delta

%%%%%%%%%%%%%%%%%%%%%%----------TTPI_count----------%%%%%%%%%%%%%%%%%%%%
TTPI_count(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

Website_sendCheckInfo(
  computemdc(receivedVotes),computemdc(clean(receivedVotes,reftable)),
count(computemdc(clean(receivedVotes, reftable)),cntstable),
  reftable,cntstable)

% The following definition is used only in the removeValid scenario
```

```
TTPI_count(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

  (sum(evi:EncryptedVoterId, sum(eci:EncryptedCandidateId,

  removeValid(evi,eci)  .

  Website_sendCheckInfo(
    computemdc(delete(evi,eci,receivedVotes)),
    computemdc(clean(delete(evi,eci,receivedVotes),reftable)),
   count(computemdc(clean(delete(evi,eci,receivedVotes),reftable)),
      cntstable), reftable,cntstable)

  <| and(elementV(evi,clean(receivedVotes,reftable)),
     elementC(eci,evichoice(evi,clean(receivedVotes,reftable),empty))) |>

  delta )

<| notEmpty(clean(receivedVotes,reftable)) |>

Website_sendCheckInfo(
  computemdc(receivedVotes),computemdc(clean(receivedVotes,reftable)),
count(computemdc(clean(receivedVotes, reftable)),cntstable),
  reftable,cntstable)

% The following definition is used only in the makeInvalid scenario
TTPI_count(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

sum(ctable:candidateTable, sum(keytable:keyTable, sum(ei:ElectionId,

recFraudInfo(ctable,keytable,ei)  .

(
    (sum(evi:EncryptedVoterId, sum(eci:EncryptedCandidateId,
      sum(ci:CandidateId,

    makeInvalid(MAC(getkey(evi,eci,keytable,ei),ei),
      MAC(getkey(evi,eci,keytable,ei),ci))  .

    Website_sendCheckInfo(
      computemdc(makeInvalid(getkey(evi,eci,keytable,ei),ei,ci,
      receivedVotes)),computemdc(clean(makeInvalid(getkey(evi,eci,
        keytable,ei),ei,ci,receivedVotes), reftable)),
      count(computemdc(clean(makeInvalid(getkey(evi,eci,keytable,ei),ei,
```

```
        ci,receivedVotes),reftable)),cntstable),reftable,cntstable)

    <| and(elementV(evi,clean(receivedVotes,reftable)),
        and(elementC(eci,evichoice(evi,clean(receivedVotes,reftable),
          empty)),element(ci,del(getci(evi,eci),ctable)) )) |>

    delta ) )))

  <| notEmpty(clean(receivedVotes,reftable)) |>

  Website_sendCheckInfo(
    computemdc(receivedVotes),computemdc(clean(receivedVotes,reftable)),
    count(computemdc(clean(receivedVotes, reftable)),cntstable),
    reftable,cntstable)

)
)))

% The following definition is used only in the addValid scenario
TTPI_count(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

sum(ctable:candidateTable, sum(keytable:keyTable, sum(ei:ElectionId,

recFraudInfo(ctable,keytable,ei)  .

(
  (sum(key:Key, sum(ci:CandidateId, addValid(MAC(key,ei),MAC(key,ci)) .

  Website\_sendCheckInfo(
    computemdc(add(MAC(key,ei),MAC(key,ci),receivedVotes)),
    computemdc(clean(add(MAC(key,ei),MAC(key,ci),receivedVotes),
      reftable)),
    count(computemdc(clean(add(MAC(key,ei),MAC(key,ci),receivedVotes),
      reftable)),cntstable),reftable,cntstable)

  <| and(element(key,getUnusedkeys(keytable,receivedVotes,ei)),
      element(ci,ctable)) |>

  delta)

<| notEmpty(getUnusedkeys(keytable,receivedVotes,ei)) |>

Website_sendCheckInfo(
  computemdc(receivedVotes),computemdc(clean(receivedVotes,reftable)),
```

```
  count(computemdc(clean(receivedVotes, reftable)),cntstable),
  reftable,cntstable)
)

% The following definition is used only in the increaseCount scenario
TTPI_count(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

sum(ci:CandidateId, addcount(ci) .

Website_sendCheckInfo(
  computemdc(receivedVotes),computemdc(clean(receivedVotes,reftable)),
  increase(ci,count(computemdc(clean(receivedVotes,reftable)),
    cntstable)),reftable,cntstable)

<| element(ci,cntstable) |> delta

% The following definition is used only in the changeCount scenario
TTPI_count(receivedVotes:Table, reftable:referenceTable,
  cntstable:countsTable) =

  (sum(ci:CandidateId, sum(ci2:CandidateId, subcount(ci) . addcount(ci2) .

  Website_sendCheckInfo(
    computemdc(receivedVotes),
    computemdc(clean(receivedVotes,reftable)),
    increase(ci2,decrease(ci,count(computemdc(clean(receivedVotes,
      reftable)),cntstable))),reftable,cntstable)

  <| and(voted(ci,count(computemdc(clean(receivedVotes,reftable)),
     cntstable)),element(ci2,remove(ci,cntstable)))  |>

  delta )

<| notEmpty(clean(receivedVotes,reftable)) |>

Website_sendCheckInfo(
  computemdc(receivedVotes),computemdc(clean(receivedVotes,reftable)),
  count(computemdc(clean(receivedVotes, reftable)),cntstable),
  reftable,cntstable)
%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%
Website_sendCheckInfo(receivedVotes:MDCTable, validVotes:MDCTable,
  countstable:countsTable, reftable:referenceTable,
```

```
    initCnts:countsTable) =

send_checkInfo(receivedVotes,validVotes,countstable,reftable,initCnts) .
Website_sendCheckInfo(receivedVotes,validVotes,countstable,reftable,
  initCnts)

%%%%%
Voter_receiveBallot(voterMemory:Table, ei2:ElectionId,
  eitable:elidTable) =

sum(key:Key, sum(ei1:ElectionId, sum(ctable:candidateTable,

receive_ballot(key,ei1,ctable) .
Voter_choice(key,ei1,eitable(ei1,eitable(ei2,eitable)),ctable,
  voterMemory,voterMemory) )))

%%%%%
Voter_choice(key:Key, ei1:ElectionId, eitable:elidTable,
  ctable:candidateTable, voted:Table, notVoted:Table) =

  (sum(ci:CandidateId, sum(ei:ElectionId,

  send_vote(MAC(key,ei),MAC(key,ci)) .
  Voter_choice(key,ei1,eitable,ctable,add(MAC(key,ei),MAC(key,ci),voted),
    notVoted)

  <| and(element(ci,ctable),element(ei,eitable)) |> delta )) )
+
  wait . voter_getCheckInfo(voted,notVoted,key,ei1)

%%%%%
voter_getCheckInfo(voted:Table, notVoted:Table, key:Key, ei:ElectionId) =

sum(receivedVotes:MDCTable, sum(validVotes:MDCTable,
  sum(countstable:countsTable, sum(reftable:referenceTable,
    sum(initCnts:countsTable,

receive_checkInfo(receivedVotes,validVotes,countstable,reftable,initCnts).
voter_checkVotedRec(voted,voted,notVoted,receivedVotes,validVotes,key,ei)
)))))

%%%%%
voter_checkVotedRec(voted:Table, copyVoted:Table, notVoted:Table,
  receivedVotes:MDCTable, validVotes:MDCTable, key:Key, ei:ElectionId) =
```

```
(
  ( OKr(retrieveV(voted),retrieveC(voted)) .
  voter_checkVotedRec(removeFirst(voted),copyVoted,notVoted,
    delete(retrieveV(voted),retrieveC(voted),receivedVotes),validVotes,
      key,ei) )

  <| present(retrieveV(voted), retrieveC(voted), receivedVotes) |>

  ( Voter_detectRemove(retrieveV(voted),retrieveC(voted)) .
  voter_checkVotedRec(removeFirst(voted),copyVoted,notVoted,receivedVotes,
    validVotes,key,ei) )
)

<| notEmpty(voted) |>

voter_checkNotVoted(copyVoted,retrieveUncast(MAC(key,ei),
  receivedVotes,notVoted),receivedVotes,validVotes)

%%%%%
voter_checkNotVoted(voted:Table, notVoted:Table, receivedVotes:MDCTable,
  validVotes:MDCTable) =

(
OKr(retrieveV(notVoted),retrieveC(notVoted)) .
Voter_detectAdd(retrieveV(notVoted),retrieveC(notVoted)) .
  (
  OKv(retrieveV(notVoted),retrieveC(notVoted)) .
  voter_checkNotVoted(voted,removeFirst(notVoted),receivedVotes,
    remove(retrieveV(voted),validVotes))

  <| present(retrieveV(notVoted),retrieveC(notVoted),validVotes) |>

   continue . voter_checkNotVoted(voted,removeFirst(notVoted),
                receivedVotes,validVotes)
  )
)

<| notEmpty(notVoted) |>

voter_checkVotedValid(voted,validVotes)

%%%%%
voter_checkVotedValid(voted:Table, validVotes:MDCTable) =

  (
```

```
  OKv(retrieveV(voted), retrieveC(voted)) .

    (
    OKci(recalci(retrieveC(voted)))

    <|eq(ref(computemdc(retrieveV(voted),retrieveC(voted))),
    recalci(retrieveC(voted)))|>

    Voter_detectci(recalci(retrieveC(voted)))
    ) .
    voter_checkVotedValid(removeFirst(voted),
      remove(retrieveV(voted),validVotes))

  <| present(retrieveV(voted), retrieveC(voted), validVotes) |>

  continue . voter_checkVotedValid(removeFirst(voted),validVotes)
  )

<| notEmpty(voted) |>

delta

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Initialization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init

encap({send_ballot,receive_ballot,send_vote,receive_vote,send_checkInfo,
  receive_checkInfo,receive_checkInfoSURFnet,send_recVotes,rec_recVotes,
    % Actions that communicate to fraudulous actions:
    send_voteCORRUPT,sendFraudInfo,recFraudInfo,saveFraudInfo,
      keepFraudInfo,getFraudInfo},

%%%%%%%%%%%%%%%%%%%%---------TTPI_sendBallot---------%%%%%%%%%%%%%%%%%%%%
% The following initialization is used only in the following scenario's:
% makeInvalid, addValid, sendVote

TTPI_sendBallotCORRUPT(
ktable(Key1,ktable(Key2,kempty)),
EI1,
rtable( MDC( MAC( Key1,EI1), MAC(Key1, CI1)),
rtable( MDC( MAC( Key1,EI1), MAC(Key1, CI2)),
rtable(MDC( MAC( Key2,EI1), MAC(Key2, CI1)),
rtable(MDC( MAC( Key2,EI1), MAC(Key2, CI2)),
rempty)))) ,
```

```
ctable(CI1,ctable(CI2,ect)),
cntstable(CI1,zero, cntstable(CI2,zero,cempty)))

% The following initialization is used in all other scenario's
TTPI_sendBallot(
ktable(Key1,ktable(Key2,kempty)),
EI1,
rtable( MDC( MAC( Key1,EI1), MAC(Key1, CI1)),
rtable( MDC( MAC( Key1,EI1), MAC(Key1, CI2)),
rtable(MDC( MAC( Key2,EI1), MAC(Key2, CI1)),
rtable(MDC( MAC( Key2,EI1), MAC(Key2, CI2)),
rempty)))) ,
ctable(CI1,ctable(CI2,ect)),
cntstable(CI1,zero, cntstable(CI2,zero,cempty)))
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%----------FraudInfoMem-----------%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the following scenario's
% makeInvalid, addValid
|| FraudInfoMem
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%---------TTPI_sendBallot---------%%%%%%%%%%%%%%%%%%%%
% The following initialization is used only in the removeReceived scenario
|| SURFnet_checkcnt

% The following initialization is used in all other scenario's
|| SURFnet_check
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%--------TTPI_castCORRUPT---------%%%%%%%%%%%%%%%%%%%%
% The following definition is used only in the sendVote scenario
|| TTPI_castCORRUPT
%%%%%%%%%%%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%%%%%%

|| Voter_receiveBallot(empty, EI2, eiempty)
|| Voter_receiveBallot(empty, EI2, eiempty)

)
```