Bachelor Thesis   2013

VoteVerifier

Independent Verifier for UniVote Elections

Division	Computer Science
Students	Giuseppe Scalzi
	Justin Springer
Professors	Dr. Eric Dubuis
	Dr. Rolf Haenni
Expert	Han van der Kleij

Berner Fachhochschule
Haute école spécialisée bernoise
Technik und Informatik
Technique et informatique

Abteilung Informatik
Division informatique

Erklärung der Diplomandinnen und Diplomanden
Déclaration des diplômé-e-s

Selbständige Arbeit / Travail autonome

Ich bestätige mit meiner Unterschrift, dass ich meine Bachelor Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.), die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt.

Par ma signature, je confirme avoir effectué mon mémoire de Bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.), qui m'ont fortement aidées dans mon travail, sont intégralement mentionnées dans l'annexe de mon mémoire.

Name/Nom, Vorname/Prénom   *Springer Justin*

Datum/Date   *10.6.2013*

Unterschrift/Signature   *[signature]*

Dieses Formular ist dem Bachelor Thesis Bericht beizulegen.

Ce formulaire doit être joint au rapport du mémoire de Bachelor.

Berner Fachhochschule
Haute école spécialisée bernoise
Technik und Informatik
Technique et informatique

Abteilung Informatik
Division informatique

Erklärung der Diplomandinnen und Diplomanden
Déclaration des diplômé-e-s

Selbständige Arbeit / Travail autonome

Ich bestätige mit meiner Unterschrift, dass ich meine Bachelor Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.), die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt.

Par ma signature, je confirme avoir effectué mon mémoire de Bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.), qui m'ont fortement aidées dans mon travail, sont intégralement mentionnées dans l'annexe de mon mémoire.

Name/Nom, Vorname/Prénom   *Scalzi Giuseppe*

Datum/Date   *10. 6. 2013*

Unterschrift/Signature   *[signature]*

Dieses Formular ist dem Bachelor Thesis Bericht beizulegen.

Ce formulaire doit être joint au rapport du mémoire de Bachelor.

Contents

List of Illustrations

Contents of the Tables

1   Introduction

The Internet plays an ever-greater role in an even increasing number of aspects of life. Practically all areas off life can be connected in some way to the use of the Internet. From ordering DVDs to watching them online, from researching a topic to planning a vacation, there is no end to the timesaving conveniences that appear on the Internet. It is therefore no small wonder, that the lengthy and arduous process of holding an election be made more convenient by creating a safe and secure service to hold elections electronically.

Perhaps the issue of electronic voting, however, deserves the most emphasis on security than any other branch of life that could be placed on the Internet. Although e-voting presents a great advantage compared to traditional voting systems, it poses an equally great risk that the results could be covertly changed. For all the time-saving that an e-voting system can offers, if the system could be hacked, it would represent a huge temptation for opposing parties to try to easily manipulate the results of an election. Were a party successfully able to compromise the voting system, it would take little effort to swing the results of an election into their own favor and causing profound effects on the future of schools, countries, and the world.

For this reason e-voting systems are implemented with strong security measures, which rely on proven and effective cryptographic functions. The UniVote system, for example, additionally implements a system of checks on every critical step of the election process. Using mathematical proofs the veracity and accuracy of the system can be shown to be trustworthy. But who can check these proofs? Should blind trust be placed on the UniVote team to have honestly and diligently implemented the checks and proofs? Whose word shall we trust to say that the results of an election are valid?

To meet these demanding needs in the e-voting community, the VoteVerifier verification software validates the parameters, proofs, adherence to the system specification, as well as the results of elections held with the UniVote e-voting system. The VoteVerifier software provides participants in the elections of Swiss universities an extra boost of confidence in the integrity of the university student governmental system.

1.1   Purpose

The VoteVerifier project was created to provide an independent verification software to confirm the results from elections held with a specific e-voting software platform called UniVote. The developers of this project are students majoring in IT Security and chose this project to deepen their knowledge of cryptography, IT security, software development and e-voting systems. This project serves as a Bachelor's Thesis for the Bern University of Applied Sciences in Biel and was carried out from February to July 2013 under the supervision of Dr. Dubuis and Dr. Haenni.

1.2   Theme

UniVote is an e-voting system developed by the Bern University of Applied Sciences in Biel. UniVote is the result of many years of research activity at the Bern University of Applied Sciences. VoteVerifier was developed independently from, yet in cooperation with, the UniVote electronic voting system. The voting secret and correctness of the election results are protected cryptographically. By publicizing the cryptographically protected election data, the results of the election can later be scrutinized by independent parties.
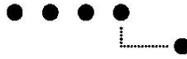
Illustration 1-1 The flow of the information from generation to VoteVerifier results.

The voters cast their votes in
an UniVote election

voter1

The votes are processed
following the specification

When the election period ends,
the result are published on the
public board

voter2

Election

UniVote

Public Board

Everybody can verify the results
in order to trust UniVote

Vote
Verifier

voter3

independant people
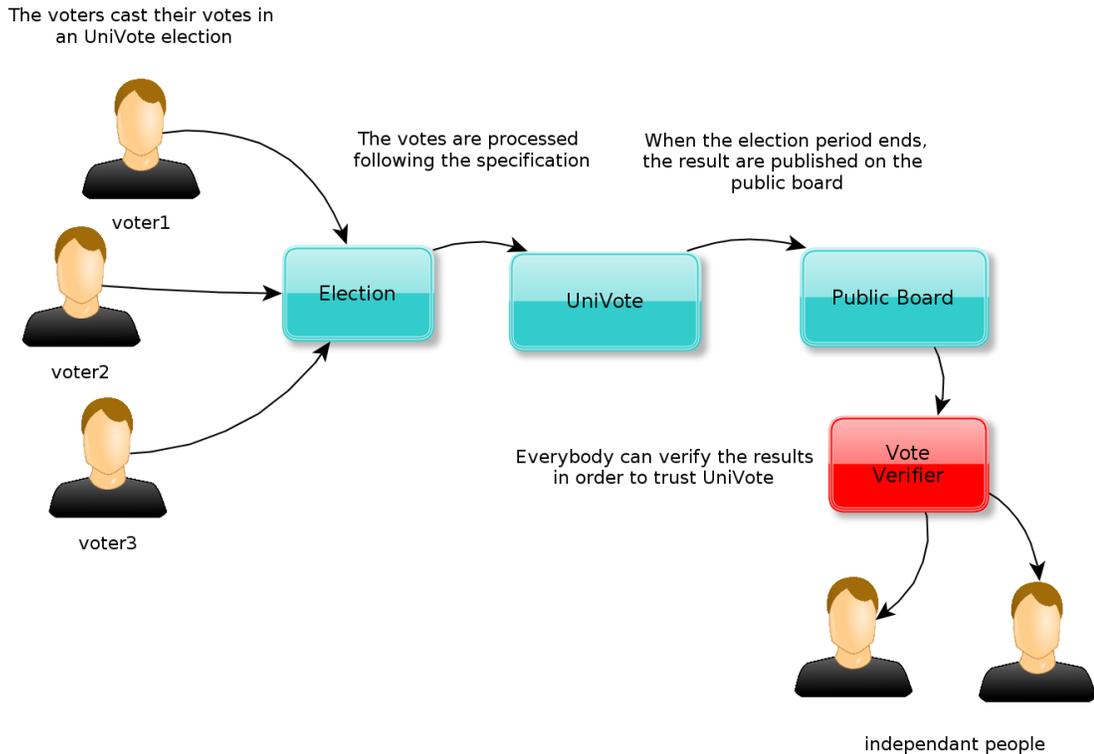
1.3   Goal

The main goal of the software we have to develop is to verify election results from the UniVote e-voting system. To this end, the software should provide a quick and simple tool to accompany and reinforce the validity of the UniVote e-voting system. A graphical user interface (GUI) should provide a user-friendly, attractive appearance for users to input the parameters to start a verification process and the results should be displayed clearly and in an organized fashion.

The manner in which the program evaluates a verification should be created in strict adherence to the specification from the UniVote system. To test the program results from real elections from the student body elections of the University of Bern, the University of Zurich and the Bern University of Applied Sciences should be used.
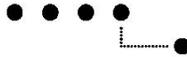
In the development of the program the highest possible level of independence from and impartiality to the UniVote system and team should be maintained. In order to guarantee the impartiality of the verification software the source code from the UniVote system may not be used in development of this verification software. That being said, without cooperation with and instruction from the UniVote team, this project would not have been possible.

1.4 Requirements

This section discusses the requirement that the software is to meet. Detailed explanations of the functionality, architecture and more can be found in the sections system description and technical implementation.

Adherence to the UniVote Specification
The software should be developed in strict adherence to the specification document for the UniVote system. The alternative would be to create the verification software in manner that the UniVote system was actually built. As a consequence deviations from the specification by the UniVote team result in a verification result other than true. It was agreed upon during development, that the deviations were the responsibility of the UniVote team to address, and not that of the development team for VoteVerifier.

Two Verification Types
The program will be able to execute a verification of an entire election, called a universal verification, as well as an individual verification to verify a single election receipt. The two verifications are briefly discussed below.

In a universal verification the system parameters, signatures, certificates, and non-interactive zero knowledge proofs (NIZKP) are verified. To begin a universal verification an election ID must be provided to allow the program to fetch the information to verify. The program displays the output of the verification process in a table.

An individual verification verifies a certificate, signature, and NIZKP, and if the ballot was included in the election results. The input for this verification is a quick response code (QR code), which contains the information of an election receipt. The entire process lasts less than a few seconds, and again the results are displayed in a table.

Information about the cryptography involved in these verifications can be found in the section entitled Cryptography.

Reading QR Code
After voting in an election using the UniVote system, the user is given an electronic receipt, which contains the election ID, encryption of the users vote, signatures, timestamp and other information.

Multiple Views of Results
The program should offer three different views of the results. The views are according to system specification, election entity, and result type. Additionally there must be a section to view the election results, which entails the votes that each political party and candidate received.

Help

The program should offer two methods of receiving help. There should be a user manual available that can be accessed from within the program. Additionally, information should be supplied about why a verification result failed by clicking on or hovering over the verification result.

Installer

The software can be installed using an installer and is available as a digital download and will support the three main computing platforms: Windows, Mac OSX, and Linux. As agreed upon, the software installer will be created and added to the project by Dr. Dubuis.

Internationalization

The program will be available in three languages: English, German, and French.

1.5 Project Planning

For our planning we have chosen to divide the different phases into iterations, so that we can have a better control over the development cycle. We can explain the different tasks with the following table.

**Illustration 1-2 Planning of the development of the project.**
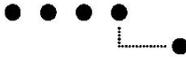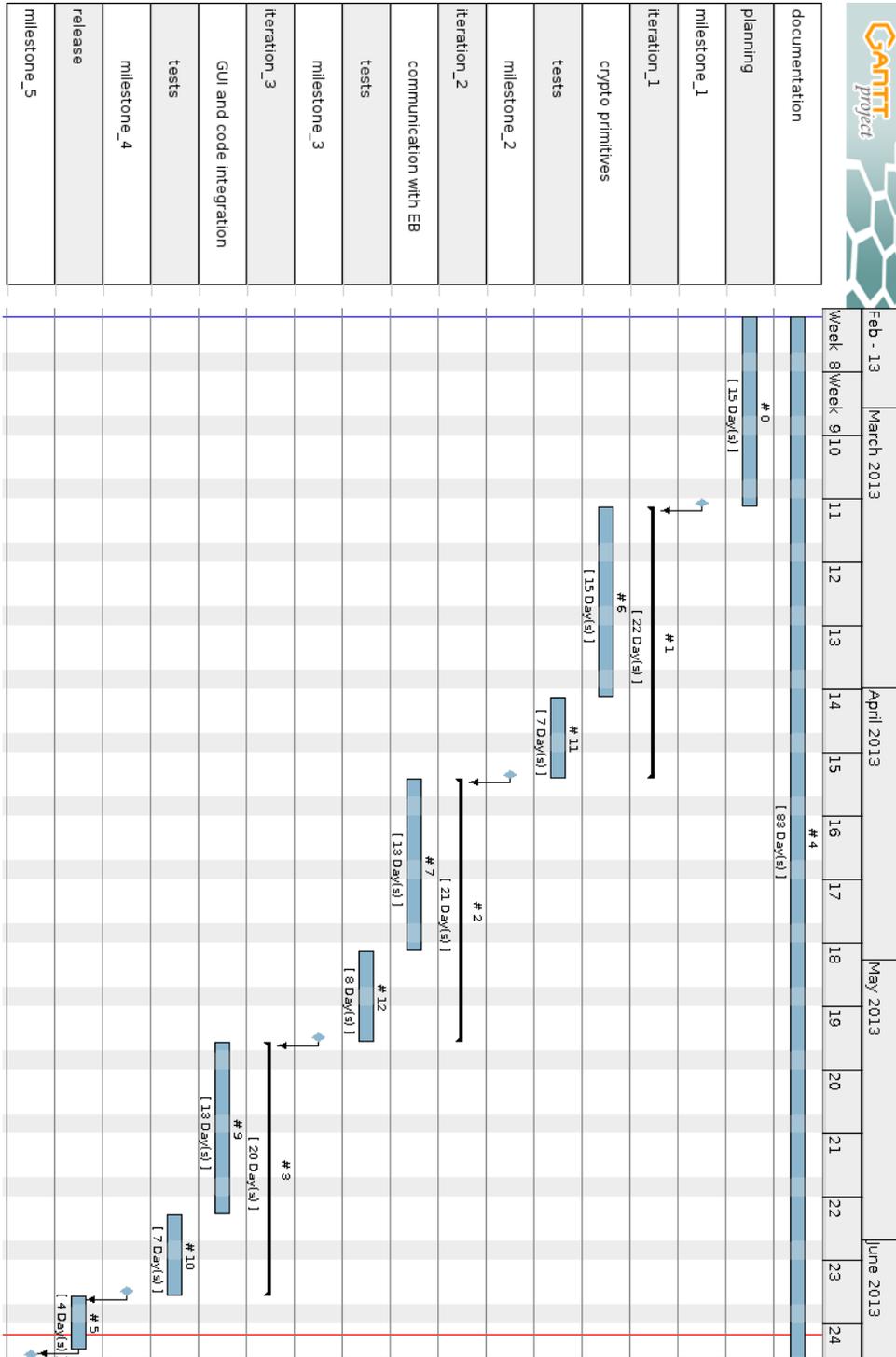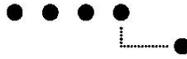
Table 1: Planning of the development

Task	Description
Documentation	The documentation is long as the project duration because we always write some portions of it.
Planning	The planning is the first step of the project and will help us by defining the guide lines for the future tasks.
First Iteration	For the first iteration we have planned the developing of the crypto primitives. These are the basis for the future tasks and they include for example the ability to verify an RSA signature of a non-interactive zero knowledge proof. After each iteration, there will be a phase of test developing in order to provide to the next iteration a code that is fully operational.
Second Iteration	Here we planned to build the infrastructure used to communicate with the election board of UniVote. This means that after this step, we will be able to perform the verification of an election by downloading the necessary data. This step also includes the implementation of the different cryptographic computations.
Third Iteration	During this last iteration, we have to design and develop the graphical user interface. This means that we have to integrate the code of the iteration 2 and at the end we will have the complete software.
Release	This is the final phase where some minor works will be done, such as packing the software into an installer.

We basically followed the plan that was created at the start of the project. The only considerable deviation involves the second iteration. Here we thought that due to the lack of time and the important amount of work for the GUI, to split the work between us. So during the second iteration Mr. Scalzi developed the structure of the software in order to communicate and perform a verification while Mr. Springer designed and implemented the GUI. By doing this we were able to intercept some possible problems in an early stage of development.
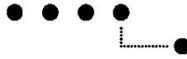
1.6   Results

In the course of development we were able to meet all of the original requirements set at the beginning of the project. We were successfully able to create a simple and user-friendly graphical user interface, which allows the user to create two different verification types to verify the results of election held by UniVote e-voting systems.

Illustration 1-3: The results of the verification process are shown in a table.
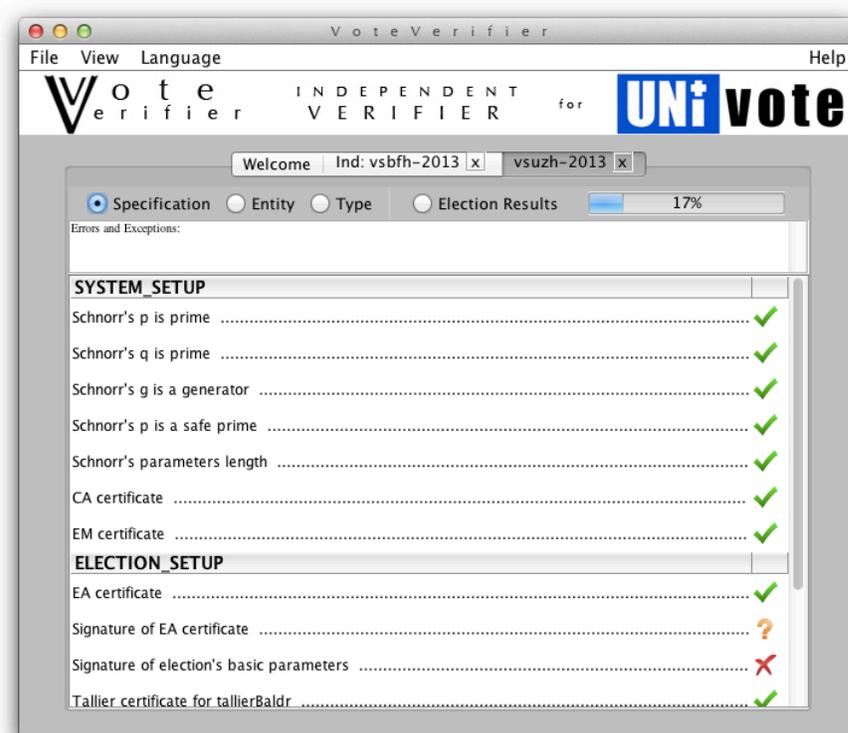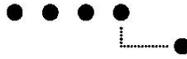
2   System Description

This paragraph explains how the system works in a non-technical way. In particular we will show what cryptographic primitives are involved in our project and we will also see how the voting process of UniVote works.

2.1   Cryptography

Our thesis is strong related to some cryptographic primitives used by UniVote. These permit to the election system to work correctly. These permit to have a secure e-voting platform and more in particular they guarantee the integrity and the secrecy of the voting data as well the anonymity.

We now describe each cryptographic component.

2.1.1   RSA Signatures

RSA is mainly an asymmetric public key algorithm used to encrypt and decrypt the data we want. A party who wants to send a message to one other, must first get the public key of the other so that he can compute the encrypted message. Then the receiver can decrypt the message using the private key.

RSA can also be used as a signature scheme so that it's possible to verify the integrity of data. If a party wants to generate a signature, in order to give the possibility to verify the integrity of some data, it must first hash the data with a given hash function and then compute the signature with its private key. A possible verifier, must first get and hash with the same hash function the data, then with the public key of the signer, he can verify the signature.

RSA signatures are used to guarantee the integrity of the data of UniVote.

2.1.2   ElGamal

ElGamal is another encryption scheme that works with public keys and it's based on the discrete log problem. The particularity of this scheme, is that is randomized so that for each equal input we will obtain every time a different output. As a Schnorr's signature, an ElGamal encryption will result in a tuple with two values. The ElGamal system, in UniVote, is used to encrypt the votes.

2.1.3   Schnorr Signatures

The Schnorr signature scheme is another method to check the integrity of data and the eligibility of a voter to vote. A Schnorr signature is mainly composed by two values and one possible advantage over an RSA signature is that it generates relatively short signatures. In contrast to
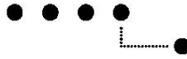
RSA that is based on the difficulty to factorize a number into prime factors, Schnorr signature are based on the discrete log problem.

Schnorr signatures are used to prove the integrity of the encrypted vote of a voter.

2.1.4 Non-Interactive Zero Knowledge Proof

A non-interactive zero knowledge proof is a method used to prove the knowledge of a certain value. The verifier, the one who verifies if the prover knows the value, can through some mathematical operations say if the prover effectively knows a certain value. The particularity of this method, is that it must be impossible for the verifier to gain the access to the value that has to be verified. This because otherwise it will be clearly too trivial to prove the knowledge of the value.

We can distinguish between "Zero Knowledge Proof" and "non-interactive zero knowledge proof", by saying that in the first case, interaction is needed between the verifier and the prover, where in the second not. UniVote uses only the "non-interactive" variant.

NIZKP are mainly used in UniVote to prove the knowledge a particular discrete logarithm.

2.1.5 Certificates

UniVote uses X.509 certificates in order to prove the identity of the different parties involved during the voting process. The parties include, a certificate authority (CA), an election administrator (EA), an election manager (EM), some talliers (T), some mixers (M) and the voters (V). Each of these entities has a certificate and each of them has a public key used for example to generate the various signatures involved.

2.2 UniVote's Voting Process

In this paragraph we want to briefly explain the voting process of UniVote.

First of all we want to explain the different entities involved. Each entity plays a precise role and we want to briefly describe each of them:
- Certificate Authority (CA): this entity is at the top of the hierarchy and is as a trust of anchor for the others.
- Election Manager (EM): as the name says, it manages an election. It verifies the cryptographic statement generated by other parties.
- Election Administrator (EA): the EA defines some parameters such as who are the talliers and who are the mixers or for example the election choices. It usually requests that an election be run.
- Tallier (T): a tallier is responsible for the generation of a part of the encryption key. These parts then form an encryption key that can be used by a voter to encrypt a vote.
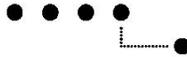
- Mixer (M): a Mixer is used to mix for example the encrypted votes, in order to avoid a possible association between a voter and his vote.
- Voter (V): a voter who takes part to an election.

Now we can list the phases of the voting. Each of them uses the cryptographic primitives we have listed in the previous section and our software follow this phases in order to perform a verification.

In the voting protocol of UniVote there are six phases:
1. Public parameters: where the public parameters and the certificates are generated.
2. Registration phase: during this phase a voter registers itself in the UniVote system.
3. Election setup: where the election-specific parameters are generated.
4. Election preparation: this phase happens shortly before an election starts and here will be for example defined, the voting choices.
5. Election period: during this phase, we are in the middle of an election. Here we can for example find the creation and casting of a vote.
6. Mixing and tallying: during this last phase, the encrypted votes will be mixed and the vote will be decrypted in order to get the final result.

Our software must go through all the phases (except the registration phase), download and verify the data published during each of them. For a detailed explanation of the UniVote protocol, please consult the specification on the project page (http://e-voting.bfh.ch/projects/univote/).

2.3   List of verifications

Here we want to list all the cryptographic checks that we must to perform. Each of the 60 checks belong to a precise voting phase defined in the UniVote specification.

This table contains all of them and in the first column we can find the code that identifies a particular check in our software along with the description in the second column. The "Specification paragraph" column tells us where this check is defined in the specification. The "Entity" column explains which entity, listed in the previous section, is responsible for verifying this check. The last column simply tells in which class this check is located.

Table 2: Cryptographic checks that VoteVerifier performs.

Code	Description	Specification paragraph	Entity	Class
100	SETUP_SCHNORR_P	1.3.1	Parameters	ParametersImplementer
110	SETUP_SCHNORR_Q	1.3.1	Parameters	ParametersImplementer
120	SETUP_SCHNORR_P_SAFE_PRIME	1.3.1	Parameters	ParametersImplementer
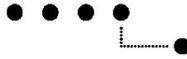
130	SETUP_SCHNORR_G	1.3.1	Parameters	ParametersImplementer
140	SETUP_SCHNORR_PARAM_LEN	1.3.1	Parameters	ParametersImplementer
150	SETUP_EM_CERT	1.3.2	CA	CertificatesImplementer
160	SETUP_CA_CERT	1.3.2	CA	CertificatesImplementer
200	EL_SETUP_EA_CERT	1.3.4 a	CA	CertificatesImplementer
205	EL_SETUP_EA_CERT_ID_SIGN	1.3.4 a	EM	RSAImplementer
210	EL_SETUP_BASICS_PARAMS_SIGN	1.3.4 b	EA	RSAImplementer
215	EL_SETUP_TALLIERS_CERT	1.3.4 b	CA	CertificatesImplementer
220	EL_SETUP_MIXERS_CERT	1.3.4 b	CA	CertificatesImplementer
225	EL_SETUP_T_CERT_M_CERT_ID_SIGN	1.3.4 b	EM	RSAImplementer
230	EL_SETUP_ELGAMAL_P	1.3.4 c	Parameters	ParametersImplementer
231	EL_SETUP_ELGAMAL_Q	1.3.4 c	Parameters	ParametersImplementer
232	EL_SETUP_ELGAMAL_G	1.3.4 c	Parameters	ParametersImplementer
233	EL_SETUP_ELGAMAL_SAFE_PRIME	1.3.4 c	Parameters	ParametersImplementer
234	EL_SETUP_ELGAMAL_PARAM_LEN	1.3.4 c	Parameters	ParametersImplementer
235	EL_SETUP_ELGAMAL_PARAMS_SIGN	1.3.4 c	EM	RSAImplementer
240	EL_SETUP_T_NIZKP_OF_X	1.3.4 d	Tallier	ProofImplementer
245	EL_SETUP_T_NIZKP_OF_X_SIGN	1.3.4 d	Tallier	RSAImplementer
250	EL_SETUP_T_PUBLIC_KEY	1.3.4 d	Parameters	ParametersImplementer
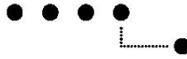
255	EL_SETUP_T_PUBLIC_KEY_SIGN	1.3.4 d	EM	RSAImplementer
260	EL_SETUP_M_NIZKP_OF_ALPHA	1.3.4 e	Mixer	ProofImplementer
265	EL_SETUP_M_NIZKP_OF_ALPHA_SIGN	1.3.4 e	Mixer	RSAImplementer
270	EL_SETUP_ANON_GEN	1.3.4 e	Parameters	ParametersImplementer
275	EL_SETUP_ANON_GEN_SIGN	1.3.4 e	EM	RSAImplementer
300	EL_PREP_C_AND_R_SIGN	1.3.5 a	EA	RSAImplementer
310	EL_PREP_EDATA_SIGN	1.3.5 b	EM	RSAImplementer
320	EL_PREP_ELECTORAL_ROLL_SIGN	1.3.5 c	EA	RSAImplementer
330	EL_PREP_VOTERS_CERT	1.3.5 c	CA	CertificatesImplementer
335	EL_PREP_VOTERS_CERT_SIGN	1.3.5 c	EM	RSAImplementer
350	EL_PREP_M_PUB_VER_KEYS	1.3.5 d	Parameters	ProofImplementer
360	EL_PREP_M_PUB_VER_KEYS_SIGN	01.03.05	Mixer	RSAImplementer
370	EL_PREP_PUB_VER_KEYS	1.3.5 d	Parameters	ParametersImplementer
380	EL_PREP_PUB_VER_KEYS_SIGN	1.3.5 d	EM	RSAImplementer
400	EL_PERIOD_LATE_NEW_VOTER_CERT	1.3.6 a	CA	CertificatesImplementer
405	EL_PERIOD_LATE_NEW_VOTER_CERT_SIGN	1.3.6 a	EM	RSAImplementer
410	EL_PERIOD_M_NIZKP_EQUALITY_NEW_VRF	1.3.6 a	Mixer	ProofImplementer
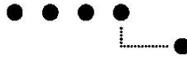
415	EL_PERIOD_M_NIZKP_EQUALITY_NEW_V RF_SIGN	1.3.6 a	Mixer	RSAImplementer
420	EL_PERIOD_NEW_VER_KEY	1.3.6 a	Parameters	ParametersImplementer
425	EL_PERIOD_NEW_VER_KEY_SIGN	1.3.6 a	EM	RSAImplementer
430	EL_PERIOD_M_VER_KEY_NIZKP_OF_ALP HA	1.3.6 b	Mixer	ProofImplementer
435	EL_PERIOD_M_VER_KEY_NIZKP_OF_ALP HA_SIGN	1.3.6 b	Mixer	RSAImplementer
440	EL_PERIOD_LAST_M_VER_KEY	1.3.6 b	Parameter	ParametersImplementer
445	EL_PERIOD_LAST_M_VER_KEY_SIGN	1.3.6 b	EM	RSAImplementer
450	EL_PERIOD_BALLOT	1.3.6 d	Paramerers	ElectionPeriodRunner
455	EL_PERIOD_BALLOT_SIGN	1.3.6 d	EM	RSAImplementer
500	MT_M_ENC_VOTES_SET	1.3.7 a	Mixer	ProofImplementer
510	MT_M_ENC_VOTES_SET_SIGN	1.3.7 a	Mixer	RSAImplementer
515	MT_ENC_VOTES_SET	1.3.7 a	Parameter	ParametersImplementer
520	MT_ENC_VOTES_ID_SIGN	1.3.7 a	EA	RSAImplementer
530	MT_T_NIZKP_OF_X	1.3.7 b	Tallier	ProofImplementer
540	MT_T_NIZKP_OF_X_SIGN	1.3.7 b	Tallier	RSAImplementer
550	MT_VALID_PLAINTEXT_VOTES	1.3.7 b	Parameter	ParametersImplementer
560	MT_VALID_PLAINTEXT_VOTES_SIGN	1.3.7 b	EM	RSAImplementer
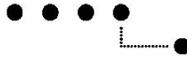
600	SINGLE_BALLOT_RSA_SIGN	QR-Code	EM	RSAImplementer
610	SINGLE_BALLOT_IN_BALLOTS	1.3.6 c	Parameter	ParametersImplementer
620	SINGLE_BALLOT_VERIFICATION_KEY	1.3.6 c	Parameter	ParametersImplementer
630	SINGLE_BALLOT_PROOF	1.3.6 c	Voter	ProofImplementer
640	SINGLE_BALLOT_SCHNORR_SIGN	1.3.6 c	Voter	SchnorrImplementer

2.4 Graphical User Interface

The GUI is organized into three sections, the title panel at the top and the main content panel in the middle, both of which are visible by default upon starting up the program, and the console panel at the bottom, which contains text relevant to the verification process being run.

The first section is the title panel, which contains the logo and description of the program. Below the title panel is the main content panel, or middle panel, which constitutes the bulk of the graphical user interface, and contains the interface required to begin verifying an election, check verification results, and view the votes the various parties and candidates received.

The third panel is hidden by default, but can be viewed by toggling the visibility in the "view" menu. The panel contains a console in which the results of the verifications are output in as text. When multiple verifications have been initiated, then there is a console window that corresponds to each verification process that has been created. Switching between these verifications switches automatically the text of the console.
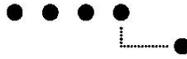
**Illustration 2-1: Sections of the graphical user interface.**
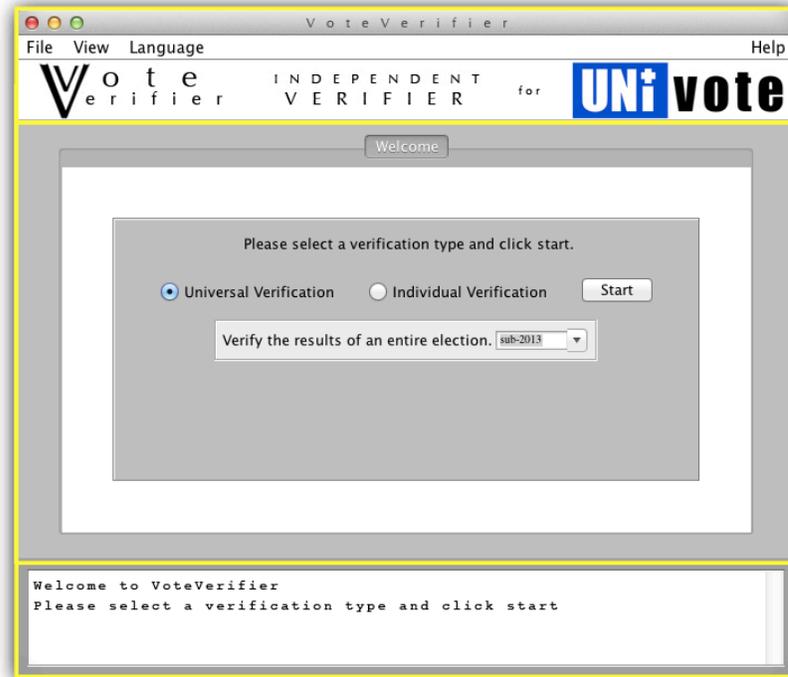
The class organization of the components of the user interface reflects the visual layout. The main window is constructed in the class MainGUI, which contains, in turn in a vertical layout, the classes TitlePanel, MiddlePanel, and ConsolePanel. The MiddlePanel contains then the tabbed panel with the welcome tab and creates any necessary new tabs for verification results. Below is the simplified class diagram of the GUI components.

The GUI was designed to be as simple and user friendly as possible. All attempts were made at keeping all functionality within one main GUI Frame and to minimize the occurrences of annoying pop-up windows. In the first version of the program, it was imagined that the welcome window, in which verification process could be setup, would be a different window than where verification process results would be shown. The design of this concept can be seen in the image below.

It was then decided that additional windows were not desirable, as the program could be simplified by organizing multiple windows into a tab system within the main window of the user interface. In the second concept, the setup controls were designed to be placed above the results windows. This simplified the organization of the windows, but created a bit of a cramped feel, as space was very limited at the top of the window, and it was difficult to place the necessary information to guide the user through the setup process.

In the final version of the program, the setup controls are placed into a welcome tab, which is visible when the program stats up, and to which the user can switch back after a verification process has begun.
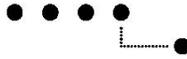
2.4.1 Interpreting Results

There are 4 possible outcomes for the verification results. A result may pass, fail, not be implemented, or have caused an error. In the case that the result passed the verification a green checkmark will appear at the end of the line where the verification is described. If the verification was unsuccessful, then in place of a green checkmark, a red "x" will appear.

The two remaining results are special cases, in which either the UniVote team has not yet implemented the content for the verification, or an error occurred during the processing of the verification. If a verification has yet to be implemented, an orange question mark will be shown. If an error was produced then a yellow warn symbol is shown. In any case except a successful result, additional information about why the test failed and what this means can be obtained through the Helper Text. More information about this feature is available in the section Helper Text.

After a verification has been completed, a Verification Event object is sent to the GUI over the Messenger class. The Verification Event is a helper class, which contains various results from the verification. An algorithm evaluates the Verification Event to decide which image should be shown and which information should be added into a pop-up, helper text. Below is a table, which illustrates the criteria, upon which the algorithm acts to choose appropriate results and additional information to show.

Table 3: How the verification result and additional information are interpreted.

	Implemented	Not implemented
True	✔ No additional text	❓ Failure code text
False	✘ Failure code text	✘ Failure code text
Exception	⚠ Exception	⚠ Exception

2.4.2 Helper Text

If a verification was not successful additional information about what went wrong may be obtained by clicking on or placing the mouse over the verification will cause a small blue helper text to appear providing a brief explanation of why the verification was not a success.

In the table above, for example, the results of a successful verification are shown with a green check. If the verification has not been implemented then the icon will be an orange question mark and the failure code text will be shown in the pop-up, helper text in the table containing the verification results. If the result is false, the icon is always a red "x" and the failure code will
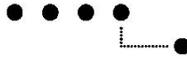
again be shown in the helper text. Lastly, if an exception occurred then this exception will appear in the helper text.

With this mechanism, the user can get more information about what the results mean and how serious of a situation it is if a certain verification has not succeeded. The descriptions in the helper text, however, are not incredibly detailed and require a certain level of technical knowledge. In upcoming versions of this software, it would be recommended to introduce descriptions that are also more useful for the layperson.

2.4.3   View Modifications

The view of the program will change based on the system that the program is running on. The program will identify the operating system upon which it is running and select an appropriate look and feel for the graphical components such as buttons, menu bars, and scroll bars.

Additionally, the user may choose to show or hide a console-like text area in which messages will be displayed. This text area is by default hidden, but will appear at the bottom of the screen upon request. To show the text area the user will click on the 'view' menu and then check or uncheck the 'show console' option.

2.4.4   Passing Messages to the GUI

This software respects the convention of the separation of the programming of the GUI from the implementation of rest of the code. This architecture is achieved through the use of the listener pattern in which the GUI registers a listener object, implemented as an inner class, with a subject object, also an inner class, contained within a class of the main code.

In this program the subject inner class has been implemented in a class called Messenger, which is solely dedicated to sending messages to the GUI or to the console, in the case of running the program from the terminal. Initially it was desirable that all messages be sent over one point of entry to the GUI, and so there was only one instance of the Messenger class, which was passed upon creation to all objects, which send messages to the GUI.

With this organization, the classes in the main code are prevented from directly calling methods within classes that compose the GUI. This is advantageous because changes in the GUI do not necessarily require changes in the main code. The GUI simply is alerted that an event has occurred and is given information, but is otherwise free to act to this information according to its own programming.

An example of this pattern involves the verification process during which the GUI must be informed of the results from a certain verification. Upon completion of a verification, the appropriate runner (see verification process) can call the method sendVrfResult in its messenger object. A helper class called Verification Event is sent to the GUI, which reacts appropriately to the Verification Event based on its contents. The Verification Event contains various instance variables such as Strings for messages or even another helper class called Verification Result, which contains the results of a verification.
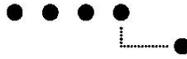
With the extension of the functionality of the software to encompass execution of multiple verification processes at a time, the single Messenger instance was no longer sufficient. To identify incoming messages by their process ID, there were two options. Either the verification Runner classes would need to know for which process they were working for, or the Messenger over which the Runner communicated with the GUI would know to which process ID they belonged.

The Messenger Manager allows multiple instances of the Messenger class to be organized and distributed to the appropriate verification threads. Before a thread is created, the Start Action contacts the Messenger Manager and asks for an instance of a Messenger for a given Process ID. The Messenger Manager maintains a single reference to the Listener subclass in the Main GUI and registers this reference with each new Messenger class that it creates. This Messenger instance is then passed on to the thread for the verification process. In this way, there is only one point of entry for messages to arrive in the GUI, but the messages a differentiated by process ID without bogging down the Runner classes with this information.
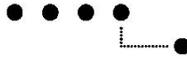
3 Technical Implementation

This section will explain in detail how the verification software works. Certain aspects, such as the organization behind a verification or the technology used, will be analyzed and described.

3.1 Technology

This paragraph explains the technology we have used, from the coding language to the various tools and libraries that are present in our software.

3.1.1 Programming Language

As a programming language we have used Java. This was beneficial first of all because it is a language that we had previously learned, and secondly it offers flexibility in terms of software development. In addition a Java program can be used on every platform by installing a Java Virtual Machine, so our software is not bound to a particular architecture or operating system.

3.1.2 Maven

Maven as the project's website says, is a "is a software project management and comprehension tool". We have used by suggestion of Dr. Dubuis. Thanks to Maven we are able to generate project builds in a faster and cleaner way. In addition it manages all the importation and downloading of additional libraries. Thanks to the cobertura plugin, we can also generate test reports and other types of report.

3.1.3 Web Services

In order to get the data from the public board, we have used the web service provided by UniVote. Thanks to this technology we can easily communicate with another system without having to worry about managing the communication. For this purpose, we have created in our project a "Web Service Client" by giving the address of the board (http://univote.ch:8080/ElectionBoardService/ElectionBoardServiceImpl?wsdl). After the creation the artifacts from UniVote will be created and made available in our project. Then we can send SOAP requests and use the data directly as Java objects, so we don't have to parse any kind of file.

3.1.4 Git

As version control system, we have chosen Git, so that we are able to manage the changes of the code between the project members. Git permits the consultation of the source code online and our project is hosted at https://github.com/EVGStudents/VoteVerifier.git.
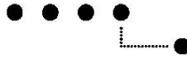
3.1.5 XStream

This is a library we used to store the data from an election to XML files so we can use them to perform JUnit testing without having the need to use the network.

3.1.6 ZXing

ZXing, pronounced Zebra Crossing, is an open source library implemented in Java and developed by Google. The library provides handling of 1D and 2D barcodes. In the VoteVerifier software it was necessary to decode the information contained in a special kinds of 2D barcode called a Quick Response Code, or QR-Code. A QR-Code is square shaped, grid-like image in which certain cells of the grid a black. An example of QR Code, which represents the data contained in an election receipt, can be seen in the figure below.

Illustration 3-1: Example of a quick response code.

3.2 Class Diagram

We can now explain the structure of our program. This section explains the main organization of the classes in the program. First we have a look at the class diagram.
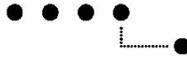
Illustration 3-2: VoteVerifier class diagram.
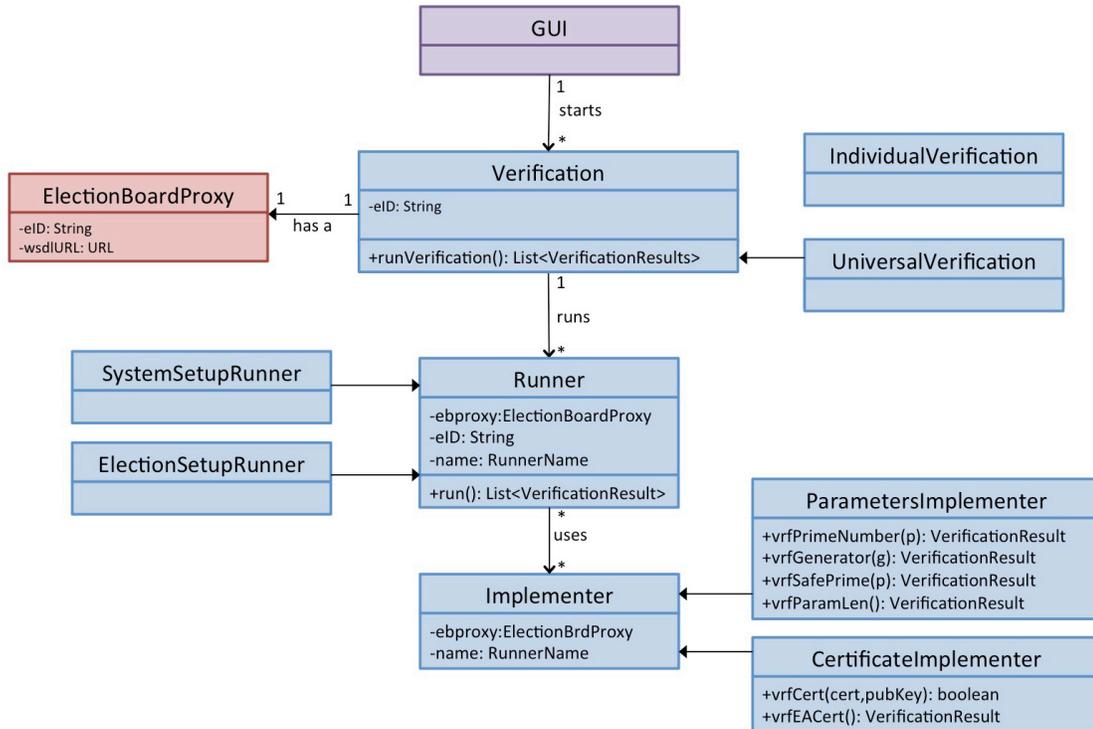
We mainly have three super classes: a Verification, a Runner and an Implementer.
We can create the specialized version of each class in order to fit our needs. For example, we can see from the diagram that we have two kinds of verifications: individual and universal.

The idea is to divide the whole verification process into small pieces called "Runners", which ask an Implementer to compute some cryptographic checks.
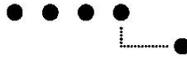
3.3 System Sequence Diagram

Before introducing the classes we can illustrate the sequence diagram of a verification.

Illustration 3-3: VoteVerifier system sequence diagram.
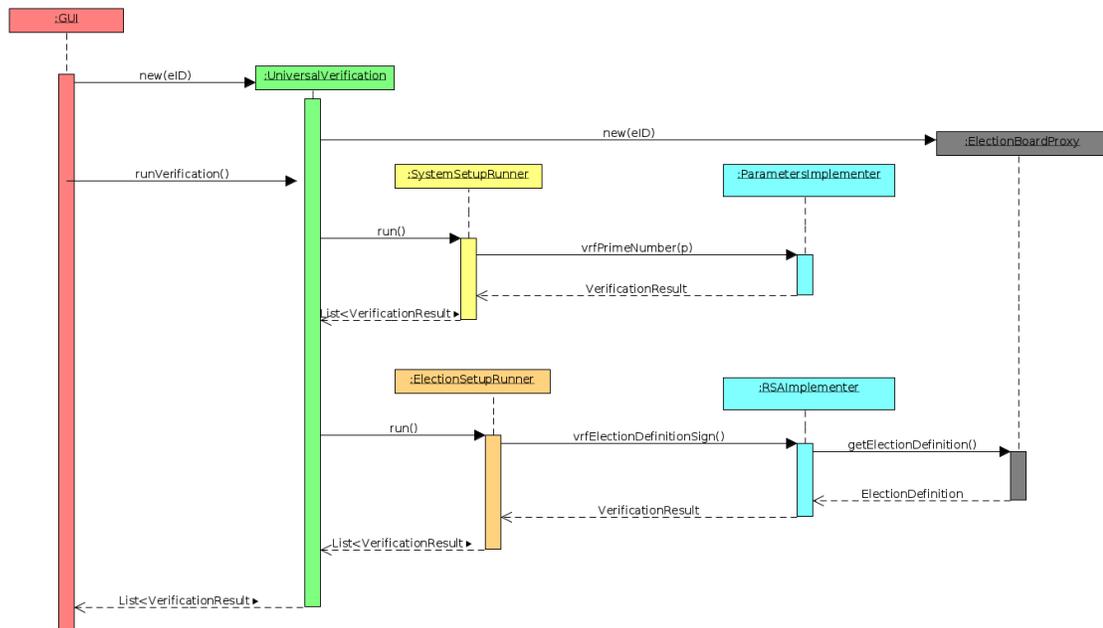
We can see how the verification will be performed. In this case a class UniversalVerification has two runners, SystemSetupRunner and ElectionSetupRunner. These runners then have some implementers and we can see that for the ParametersImplementer a method will be called to verify prime numbers. The second runner uses the RSAImplementer to verify a signature.
The implementers can communicate with the board thanks to the ElectionBoardProxy object created by the verification class.

Basically, we must first create a specialized Verification class and add some Runners. For each Runner we choose the necessary Implementer used to perform the cryptographic. We can see a certain level of flexibility because we can create the structure we want. For example a UniversalVerficiation will use only certain Runners, whereas, an IndividualVerification will use a different set of runners.

The next chapters, gives more detail about verifications, runners and implementers.
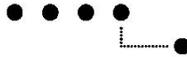
3.4   Implementer

An implementer consists of a class specialized in the implementation of the various cryptographic checks. We have 5 types of implementers at the bottom of our class infrastructure.

- CertificatesImplementer: used to check certificates.
- RSAImplementer: used to verfiy RSA signatures.
- SchnorrImplementer: for Schnorr signatures.
- ParametersImplementer: for other checks like prime numbers or parameters length.
- ProofImplementer: for the verification of NIZKP.

Each method in an Implementer also produces a VerificationResult, which is an object that contains all the necessary information about the outcome of a particular check.

We want to briefly give an example of each Implementer in order to give a better idea of what they do.

3.4.1   Certificates Implementer

The Implementer Class, called CertificatesImplementer, is dedicated to the verification of X509 certificates, which can belong to the different entities involved in UniVote.

We can see the main method used to verify a certificate:

```
public boolean vrfCert(List<X509Certificate> certList) {
    CertificateFactory cf =
                CertificateFactory.getInstance("X.509");
    CertPath cp = cf.generateCertPath(certList);
    TrustAnchor anchor = new
                TrustAnchor(certList.get(certList.size() - 1), null);
    PKIXParameters params = new
                PKIXParameters(Collections.singleton(anchor));
    params.setRevocationEnabled(false);
    CertPathValidator cpv =
                CertPathValidator.getInstance("PKIX");
    cpv.validate(cp, params);
    return true;
}
```

This method validates a certificate chain composed by one ore more certificates by using the certificate path validator algorithm provided by Java. The methods in this class can wrap vrfCert() in order to verify a certificate and this is the basic idea that we tried to follow in the other Implementers. The idea is to have a general method that can be wrapped by other method that can pass what they want as arguments.
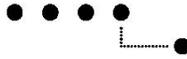
3.4.2 RSA Implementer

This class, called RSAImplementer, does example as the name implies, and that is to verify RSA signatures.

The method that provides the verification of a signature as follows:

```
public boolean vrfRSASign(RSAPublicKey pubKey, String clearText,
            BigInteger signature) {
    BigInteger hash = CryptoFunc.sha256(clearText);
    BigInteger decSign =
                signature.modPow(pubKey.getPublicExponent(),
                pubKey.getModulus());
    boolean result = decSign.equals(hash);
    return result;
}
```

As input it takes the public key to verify the signature, the message to be verified and the signature itself. The public key of RSA is composed of two values, the exponent (e) and the modulus (n) so to verify a signature it computes $signature^e$ mod n. Then if this value corresponds to the sha-256 hash of the message the signature is correctly verified.

All the other methods of this implementer wrap `vrfRSASign()` and pass the relative values.

3.4.3 Schnorr Implementer

The implementer class called SchnorrImplementer, verifies a Schnorr signature. The method is:

```
public boolean vrfSchnorrSign(BigInteger verificationKey, String
        message, BigInteger a, BigInteger b, BigInteger gen) {
    BigInteger r = gen.modPow(b,
                p).multiply(verificationKey.modPow(a, p)).mod(p);
    sc.pushLeftDelim();
    sc.pushObjectDelimiter(message,
                StringConcatenator.INNER_DELIMITER);
    sc.pushObject(r);
    sc.pushRightDelim();
    String concat = sc.pullAll();
    BigInteger hashResult = CryptoFunc.sha256(concat).mod(q);
    return hashResult.equals(a);
}
```

The input for the method is the verification key used to verify the signature, the message, the two values of the Schnorr signature and the generator for which we want to compute the "r" values. First it computes $r = g^b \cdot vk^a$ mod p, which is the value that is concatenated to the message and the sha-256 mod q hash will be computed. If this value corresponds to the "a" value, then the signature is verified. GCHECK
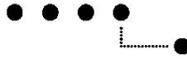
3.4.4 Parameters Implementer

This implementer, called ParametersImplementer, is used to compute different values and it is not specific to a particular domain, as is for example the RSAImplementer for RSA signatures.

Here we can find different methods, and we can briefly list the purpose of them:
1. Verification of prime numbers and safe primes
2. Correctness of the length of the encryption parameters (p,q and g for Schnorr and ElGamal)
3. Validity of the ballots
4. Verification of the last set of data for the last mixer

Here we have an example where we check if a ballot belongs to the set of all ballots:

```
public VerificationResult vrfBallotInSet(BigInteger verificationKey) {
            Exception exc = null;
            boolean r = false;
            Report rep;

            Ballot qrCodeBallot = ebp.getBallot(verificationKey);

                //check if the ballot belongs to the set of all ballots
                for (Ballot b : ebp.getBallots().getBallot()) {
                        if (qrCodeBallot.equals(b)) {
                                r = true;
                                break;
                        }
                }

        VerificationResult v = new
            VerificationResult(VerificationType.SINGLE_BALLOT_IN_BALLOTS, r,
            ebp.getElectionID(), rn, it, EntityType.PARAMETER);

        return v;
    }
```

3.4.5 Proof Implementer

This last implementer, which is called ProofImplementer, is used to perform the verification of the NIZKP. There are mainly two types of proofs to verify, the proof of knowledge of discrete log and the equality of discrete log.

We want to give as example the proof of knowledge of discrete log, because they are used most frequently:
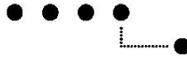
```
public boolean knowledgeOfDiscreteLog(BigInteger t, BigInteger s, BigInteger c,
            BigInteger paramV, BigInteger paramW, BigInteger prime, boolean
            vExponentSign) {
    if (vExponentSign) {
                s = s.negate();
    }

    BigInteger v = paramV.modPow(s, prime);
    BigInteger w = t.multiply(paramW.modPow(c, prime)).mod(prime);
    return v.equals(w);
}
```

This method, which verifies the knowledge of discrete log, takes the following information as parameters (see the specification of UniVote for additional information about NIZKP, paragraph 1.1.4 and 1.4.1 for a practical example):

1. t: the commit value of a proof.
2. s: the response value of a proof.
3. c: the hash of the some precomuted values .
4. paramV: the parameter used in the computation of the v value.
5. paramW: the parameter used for w.
6. prime: the prime number used for modulus operations.
7. vExponentSign: the sing of the s parameters, sometimes we need a negative s.

After the various computations, the knowledge of discrete log is verified only if the v is equal to w.

3.5   Runner

We decided to create these "Runners" in order to give to the program logic more organization. A Runner is a component used to check only a portion of the verification process, so we have created a Runner for each phase of the voting protocol.

For a UniversalVerificaiton we have created the following Runners:
• SystemSetupRunner
• ElectionSetupRunner
• ElectionPreparationRunner
• ElectionPeriodRunner
• MixingTallingRunner
• ResultRunner

On the other hand, an IndividualVerification has only two runners:
• IndividualRunner
• ResultRunner
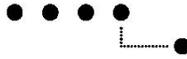
Note that one can create a Runner for other purposes too. For example we have the ResultRunner, which is responsible for computing the results of an election.

We can make a small example of what the SystemSetupRunner effectively does:

```
@Override
  public List<VerificationResult> run()  {
          //is Schnorr p prime
          VerificationResult v1 = paramImpl.vrfPrime(Config.p,
              VerificationType.SETUP_SCHNORR_P);
          msgr.sendVrfMsg(v1);
          partialResults.add(v1);
          Thread.sleep(SLEEP_TIME);

          //is Schnorr q prime
          VerificationResult v2 = paramImpl.vrfPrime(Config.q,
              VerificationType.SETUP_SCHNORR_Q);
          msgr.sendVrfMsg(v2);
          partialResults.add(v2);
          Thread.sleep(SLEEP_TIME);

          ....

          //verifiy EM certificate
          VerificationResult v7 = certImpl.vrfEMCertificate();
          msgr.sendVrfMsg(v7);
          partialResults.add(v7);

          return Collections.unmodifiableList(partialResults);
    }
```

When the `run()` method of this Runner is called, we see that some checks of prime numbers will be performed as well as a check of a certificate. We also see that it sends the result to the GUI through the messenger (`msgr` variable).

The list of results that this method returns is mainly used for testing purposes.

Each of these runners corresponds to a phase of an election according to the UniVote specification.

3.6   Verification

This class represents a Verification and it is the key point of the infrastructure. From here the whole verification process starts for both the universal and individual verification.
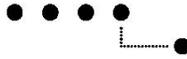
This super class is responsible for creating the instance of the ElectionBoardProxy so that each verification has its own data. Each verification can be identified by an election ID.

A Verification is also responsible for the creation of the necessary runners. A specialized class of this kind must override the `createRunners()` method in order to delegate a certain number of "Runners".

The `runVerification()` method is the most important, because if a verification is created this method must be called to start the whole process. It iterates over the list of runners, for which it executes `run()` each time. The list of all the result returned is primarily used for JUnit testing.

This is an extract of `runVerification()`:

```
    ….
        createRunners();
        ….
        //run the runners  and get the results
        for (Runner r : runners) {
            //the result runner doesn't support run() but runResult().
              if (r instanceof ResultsRunner) {
                     ResultsRunner rr = (ResultsRunner) r;
                     rr.runResults();
              } else {
                     List<VerificationResult> l = r.run()
                     …..
              }
        }
    ...
    return Collections.unmodifiableList(res);
}
```

3.7   Thread Manager

It is necessary to start the verification processes in their own thread due to the intensive calculations that they carry out. If this were not the case, the demands placed on the CPU would block Event Dispatcher Thread of the graphical user interface, and actions performed by the user would hang. When a thread is created by the StartAction, it is registered with the ThreadManager, which exists as a single instance for the entire program.

The task of the ThreadManager is to reply if a given thread exists, to interrupt a given thread, or to interrupt all threads that are running. These features come into play in two cases. If a verification result window is closed while the verification process is still running then it is necessary to signal to the verification process that all further activity should cease. In this case a call is made to the ThreadManager of `killThread(processID)` with the given process ID, which uniquely identifies the running verification process. If the thread is active, it will be
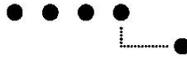
interrupted and removed from the list of threads that the Thread Manager maintains. If the thread was no longer running, then no action is taken.

3.8   Verification Result

A VerificationResult is created by a method of an Implementer. These objects are mainly used to store information about the state of the cryptographic check.

We can illustrate an example of a VerificationResult created by the `vrfElGamalParamSing()` of the RSAImplementer:

```
new VerificationResult(VerificationType.EL_SETUP_ELGAMAL_PARAMS_SIGN, result,
ebp.getElectionID(), RunnerName.ELECTION_SETUP, ImplementerType.RSA, EntityType.EM);
```

The parameters are explained here:

- `VerificationType.EL_SETUP_ELGAMAL_PARAMS_SIGN`: the type of cryptographic check.
- `result`: usually a boolean value indicating the success or failure of this check.
- `Ebp.getElectionID()`: the ID of the election for this check.
- `RunnerName.ELECTION_SETUP`: the name of the runner, which has requested the check. This field is used to sort the results by Runner.
- `ImplementerType.RSA`: the type of implementer that has performed the check. This field is used to sort the results by Implementer.
- `EntityType.EM`: the UniVote entity that has verified the data of this check. This field is used to sort the result by entity.

In addition a VerificationResult can contain a report, which is an object that contains additional information about a VerificationResult. The following cases are possible:

- The result in the VerificationResult is false.
- An exception has occurred during computation.
- The cryptographic check is not implemented.
- We want to add some additional information about a verification.
- We can use a report to print additional information about a certain check, for example we use it to explain which plausibility tests are performed in some NIZKP.

3.9   ElectionBoardProxy

Each verification process queries this class for election data. The concept behind an ElectionBoardProxy is that each verification will have only one object of this type so that we can cache the downloaded results for the started verification process. This will speed up the verification procedure.

We can see how we to construct an ElectionBoardProxy:

```
    public ElectionBoardProxy(String eID) {
```
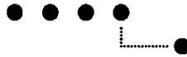
```
        this.eID = eID;

    ...

            this.wsdlURL = new URL(Config.wsdlLocation);

    ...

        getElectionBoard();
    }
```

First of all we have to provide an election ID. This is because the public board of UniVote needs it in order to send the relative data.

The communication between the public board and our software is done through web services.

From the constructor above, we can see the `getElectionBoard()` method by which we initialize the public board:

```
    private void getElectionBoard() {
        ElectionBoardService ebs = new ElectionBoardService(wsdlURL);
        eb = ebs.getElectionBoardPort();
    }
```

The ElectionBoardService is an artifact generated during the creation of the web service client and give us back an object of the type ElectionBoardPort, which is really used to communicate with the other party.

We now explain the simple cache mechanism introduced at the beginning. The concept is simple. We want to return an object that has already been created so that we avoid the creation of a new one and, as a result, do not have to download all the data once again .
We can explain the situation with a practical example:

```
    public Ballots getBallots() throws ElectionBoardServiceFault {
        if (ballots == null) {
            ballots = eb.getBallots(eID);
        }

        return ballots;
    }
```

This method is used to get the ballots of an election. We can see that thanks to the "if" statement we can control the access to the object "ballots". At the first call the object will be "null" and we will use the previously created ElectionBoardPort object to get the ballots from the network. When someone calls `getBallots()` one more time, we can deliver the previously created "ballots" object. This mechanism speeds up the verification process because we need the same data in different points of the software.
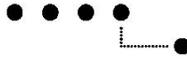
3.10 Customization of Messages

In our software it is possible to modify the output messages. We have three properties files:

- `messages.properties`: prints the title of a cryptographic check
    - i.e. 160 = CA certificate
- `failurecode.properties`: contains failure messages
    - i.e. if an RSA signature is invalid

The format of messages.properties and addinfo.properties is the same. There is a key, which is a number and the value, which is the text. The key is the ID of a cryptographic check. These IDs can be found in the enumeration VerificationType located in the *common* package.

The format of failurecode.properties is the same except that the key does not correspond to a certain cryptographic check. This is because the same failure can occur for different cases.

3.11 JUnit Tests

Basically we tried to follow a logic in our testing and we first test each cryptographic check that we can find in the Implementer classes, and then we test if each Runner performs the various call to the Implementer in the right order. Finally we test a Verification by checking, for example, the size of the result list and the results. Other tests check QR Code processing, string concatenation and hashing functions.

We have chosen not to establish a network connection during testing, and so data from the public board has been stored locally for running the tests. We have used the class LocalBoardProxyDownloader to download the data.

We test our methods with data that is assumed to be valid, but due to time constraints we have not tested them using data that we know to be false. For example, we could use an invalid QR-Code with fake data. This can be done in the future, in order to test that our computation really works.

3.11.1 Test Failures

Although we strictly followed the specification of UniVote, there are still some problems. The possible nature of these problems can be identified in the actual implementation of UniVote or in a bug in our code. For the first reason, we can find some discrepancies between the implementation and the specification that can cause an error during the verification process of some cryptographic data. We cannot, however, exclude the possibility of a bug in our code and for this reason there will be the need to modify the actual code in order to fix problems or to adapt what we have written to a possible new version of UniVote.

The tests have been executed with the data of "risis-2013-1" and of "vsbfh-2013". The tests involving a QR-Code involve only the election ID "vsbfh-2013" since we have a QR-code from only that election.

We can now group here the tests that are not successful and give a possible explanation:
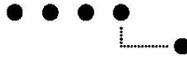
Table 4: Explanation of unsuccessful tests.

| Test	Comments	Possible cause
ParamImplTest.testMixedExncryptedVotes	This test fails because the set of encrypted vote is not equal to the set of encrypted vote of the last mixer.	Check if `getEncryptedVotes()` and `getEncryptedVotesMixedBy()` are correct.
ParamImplTest.testVotes	Here we have a problem with the computation of a value (m'). We have seen that our computations give back a m' too big as the expected one.	Bug in our code or problem with the order of the set given by `getPartiallyDecryptedVotes()`.
RSAImplTest.testSingleBallot	Here we don't know how concatenate the RSA signature contained in a QR-Code	Concatenation of the string used to compute the signature.
RSAImplTest.testBasicParam	Failure in the verification of the signature.	Same as above.
RSAImplTest.testLatelyRegisteredVotersKeysBySign	Same as above	Same as above.
RSAImplTest.testMixedVerificationKeysBySign	Same as above	Same as above.
RSAImplTest.testShuffledEncVotesSign	Same as above.	Same as above.
RSAImplTest.testElectionData	Same as above.	Same as above.
RSAImplTest.testElectoralRoll	Same as above.	Same as above.
RSAImplTest.testElectionOptions	Same as above.	Same as above.
RSAImplTest.testMixedEncVotesBySign	Same as above.	Same as above.
RSAImplTest.testPlaintextVotesSign	Same as above.	Same as above.

RSAImplTest.testLatelyRegisteredVotersKeysSign	We did not find the signature.	-
RSAImplTest.testLatelyRegisteredVotersCertSign	Same as above.	-
RSAImplTest.testLateRenewalOfRegSignBy	Data for this proof were not available.	-
RSAImplTest.testEACertId	We did not find the signature.	-
RSAImplTest.testTMCert	Same as above.	-
RSAImplTest.testVoterCertID	We don't know how to concatenate the certificate.	-
CertImplTest.testLatelyVotersCerts:	This check fails because of this: "basic constraints check failed: this is not a CA certificate". This is caused by a missing extension in the CA certificate.	Add to the extension field of the CA certificate the relative value.
CertImplTest.testVotersCert	Same as above.	Same as above.
ProofImplTest.testLateRenewalOfRegistrationProofBy	Data for this proof were not available.	-
ProofImplTest.testBallotProofFromBallot	The proof fails since v != w.	The concatenation can be a possible problem.
ProofImplTest.testBallotProofFromQRcode	The proof fails since v != w.	The concatenation can be a possible problem.
SchnorrImplTest.testSignatureVerificationFromBallot	We were not able to verify a Schnorr signature even if the computations seem to be correct.	The concatenation of the value for the hash inside the `vrfSchnorrSign()` can be the problem.
SchnorrImplTest.testSignatureVerificationFromQRCode	Same as above.	Same as above.
ElectionPreparationRunnerTest.testResultList:	These test fails because not all methods return true as verification result.	It will work when there will be no problems anymore.
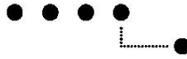
ElectionPeriodRunnerTest.testResultList	Same as above.	Same as above.
IndividualRunnerTest.testResultList:	Same as above.	Same as above.
ElectionSetupRunnerTest.testResultList	Same as above.	Same as above.
MixerTallierRunnerTest.testResultList	Same as above.	Same as above.

Please note that this list can change with the election ID. In fact the data for the "risis-2013-1" election are the most recent, so they have been computed after some bug fixing. Those bugs were still present during the "vsbfh-2013" and "sub-2013" elections.
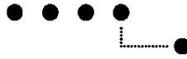
4 Conclusion

The goal of this project was to provide a simple utility that independently evaluates the trustworthiness of the UniVote e-voting system. This software achieves its goal of simplicity and ease of usability, as well as the independent nature. That it provides a boost of confidence in the veracity and accuracy of the UniVote e-voting system is, however, debatable. Unfortunately quite of few of the verifications fail, and without an in-depth, expert knowledge of the system and its specifications, the results of this verification software could be misleading.

4.1 Remaining Tasks to Complete

In large part all of the tasks of this project were completed to a satisfactory level. There are, of course, some minor improvements that could be made, which would render the software even more effective, user-friendly and efficient.

- The internationalization should be extended to include all verification results and additional information.

- The additional information of a tooltip should be placed in a separate properties files because, as it stands today, these are hard coded inside Implementer classes.

4.2 Acquired Knowledge

In this project we deepened our knowledge of software development including the planning and organization of a large-scale, Bachelor project. We learned first hand how creating a clear and specific plan at the beginning of the project can help guide the development of the software, and also how it can keep the progress on track. It is sometimes easy to get lost in small details of a project, but while working under pressure, we were able to keep and eye on the plan, and make sure the most central features of the project received the most attention.

Our knowledge of cryptography, especially signatures, encryption, and zero-knowledge proofs has also increased. Above all it was interesting to see how these different aspects of cryptography and security interact to create a secure system. Up until now most of our knowledge had regarded individual cases of each technology, but after this project, it is much more clear how different security features compliment each other and can be used together.

This was our first experience with the maven build manager, and through its use we have come to appreciate what a great service it provides. From being able to manage the project details, to quickly adding external add-ons, such as ZXing, or Web Services, Maven really makes development easier.

Finally it was a great project to be able to deepen out knowledge of GUI development. This was the first GUI that we designed, and it is clear that some problems cannot be explained in advanced when it comes to GUI development. Rather they must be experienced before they can be rightly appreciated, understood and avoided in the future. Now that the GUI is complete, it would be nice to be able to go back to square-one and rebuild it in a cleaner and more organized fashion.
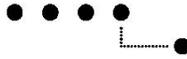
Bibliography

[1] R. Haenni,"UniVote System Specification," 2013. [Online]. Available : http://e-voting.bfh.ch/app/download/5874743461/specification.pdf?t=1365082914

[2] G. Scalzi, J. Springer, S. Bärtsch, "Bases de mathématique, cryptographie asymétrique et preuves à divulgation nulle de connaissance," Semester Project, Project 2, 2013.
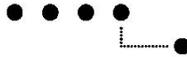
A. Use Cases

List of the Actors

The verification in VoteVerifier involves only one actor, who initiates a desired verification type. After the initial setup phase all functionality is automated and involves only the fetching of data from the election board. For this reason there is only one actor who fills the primary role in the use cases.

Actors	Type	Description
User	Primary	The user who starts a desired verification process

List of Use Cases

Menu Bar Functionality
1. Exit the program.
2. Console View Selection
3. Language Choice
4. View the About screen.

Setup
5. Select Universal Verification
6. Select Individual Verification
7. Select an Election to Verify
8. Select a File
9. Select Start

After Verification Started
10. Return to Welcome Screen
11. Close a Tab
12. Sort Election Verification Results
13. View Candidate Election Results
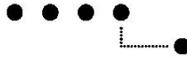
Use Cases in Fully Dressed Format

Menu Bar Functionality

Use Case 1

Use Case UC1 : User exits the program.
Primary Actor : Voter
Success Guarantee : The program will close and shutdown.

Main Success Scenario :

1. The user clicks on "File" option in the menu bar.
2. The user scrolls down to the "Exit" option and clicks on it.
3. The main program window will disappear and .

Use Case 2

Use Case UC2 : Toggling visibility of console.
Primary Actor : Voter
Success Guarantee : The console will be shown or hidden at the bottom of the program window.

Main Success Scenario :

1. The user clicks on "View" option in the menu bar.
2. The user scrolls down to the "Show Console" option and clicks on it.
3. The console window will be shown if it was hidden and hidden if it was already shown.

Use Case 3

Use Case UC3 : Changing the language.
Primary Actor : Voter
Success Guarantee : The main window will be rebuilt and set to the initial state with the newly chosen language.

Main Success Scenario :

1. The user clicks on "Language" option in the menu bar.
2. The user scrolls down to the desired language option and clicks on it.
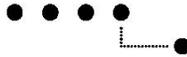
3. The console window is rebuilt with the text in the language selected.

Use Case 4

Use Case UC4 : View the about screen.
Primary Actor : Voter
Success Guarantee : The information about the project will be shown in a pop-up window.

Main Success Scenario :

1. The user clicks on "Help" option in the menu bar.
2. The user scrolls down to the "About" option and clicks on it.
3. A pop-up window appears with the information about the software.

Setup

Use Case 5

Use Case UC5 : Select Universal Verification.
Primary Actor : Voter
Success Guarantee : The choices for beginning an universal verification for a given election ID are shown.

Main Success Scenario :

1. The user starts the program or clicks on the "Welcome" tab.
2. The user clicks on the radio button "Universal Verfication".
3. Directions for the following step and a comboBox with possible election IDs are shown in the welcome tab.


Use Case 6

Use Case UC6 : Select Individual Verification.
Primary Actor : Voter
Success Guarantee : The choices for beginning an individual verification for a given election receipt are shown.

Main Success Scenario :
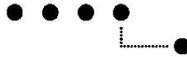
1. The user starts the program or clicks on the "Welcome" tab.

2. The user clicks on the radio button "Individual Verfication".
3. Directions for the following step and a button to allow file selection are shown in the welcome tab.

Use Case 7

Use Case UC7 : Select an Election to Verify.
Primary Actor : Voter
Success Guarantee : The selected election ID appears in the combo box and will be used in the universal verification.

Main Success Scenario :

1. The user starts the program or clicks on the "Welcome" tab.
2. The user clicks on the combo box containing possible election IDs, which can be used in a universal verification.
3. The selected election ID is showing in the combo box and will be used when the user clicks start.

Use Case 8

Use Case UC8 : Select a File to Verify .
Primary Actor : Voter
Success Guarantee : The path for the selected file is saved in a variable and shown to the user.

Main Success Scenario :

1. The user starts the program or clicks on the "Welcome" tab.
2. The user clicks on the radio button individual verification.
3. The user clicks on the "select" button.
4. A pop-up file chooser window appears.
5. The user selects a file and clicks "ok".
6. The path to the selected file is stored in a variable and the file name is shown to the user.

Use Case 9

Use Case UC9 : Select Start.
Primary Actor : Voter
Success Guarantee : The verification is carried out.

Main Success Scenario :

1. The user starts the program or clicks on the "Welcome" tab.
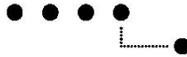
2. The user clicks on the "start" button.
3. A new tab appears with the name of the election ID for which a verification is being performed.
4. The tab becomes the active, visible tab.
5. The user sees the verification results appear in the default organization, which is according to specification.

After Verification has been Started

Use Case 10

Use Case UC10 : Return to Welcome Screen.
Primary Actor : Voter
Success Guarantee : After starting a verification the use clicks on the "Welcome" tab and is presented with the setup options to begin a new verification process.

Main Success Scenario :

1. The user clicks on the "Welcome" tab.
2. The "Welcome" tab is set as the active, visible tab.
3. The choices to start a new verification are visible.

Use case 11

Use Case UC11 : Close a Tab.
Primary Actor : Voter
Success Guarantee : The tab in which the user has clicked the "x" closes and the next tab to the left become the active, visible tab.

Main Success Scenario :

1. The user clicks on the "x" in the tab section of the tabbed window.
2. The currently active, visible tab disappears
3. The next tab to the left becomes the active, visible tab.
4. The thread that was running the verification for the closed tab is interrupted.

Use case 12

Use Case UC12 : Sort Election Verification Results.
Primary Actor : Voter

Success Guarantee : The user clicks on a sort option for the verificaiton results and sees the corresponding organization in the results window.
Preconditions: The user has started a verification which may be running or terminated.

Main Success Scenario :

1. The user clicks on one of the three organization options above the area where verification results are being displayed.
2. The verification results are reorganized according to the selected organisation.

Use case 13

Use Case UC13 : View Candidate Election Results.
Primary Actor : Voter
Success Guarantee : The user sees the results for the election in the results window..
Preconditions: The user has started a verification which may be running or terminated.

Main Success Scenario :

1. The user clicks on the button "Election Results".
2. The results for the votes for the candidates are displayed.
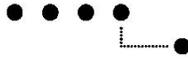
B. User Manual

The user manual is available to the user through the help menu from within the program. It describes how to proceed to start both a universal and individual verification process, as well as how to understand the verification results, and get additional help. The complete user manual is displayed on the following pages.

Vote Verifier

User Manual

Contents

1   Introduction

Welcome to VoteVerifier, the independent verifier for elections held with the UniVote electronic voting system. This system was designed to boost confidence in the UniVote system by verifying the election results of a given election or election receipt. You should already have general knowledge of the UniVote system, but you can also find more information at www.UniVote.ch.

2   Overview

This manual will guide you through the basic features and operations of the VoteVerifier program. Upon loading you will see the main welcome window and menu bar at the top. You will be guided through use of the program beginning with periphery operations, followed by initiation of a verification, and ending with viewing and interpreting the results that are displayed by the program.
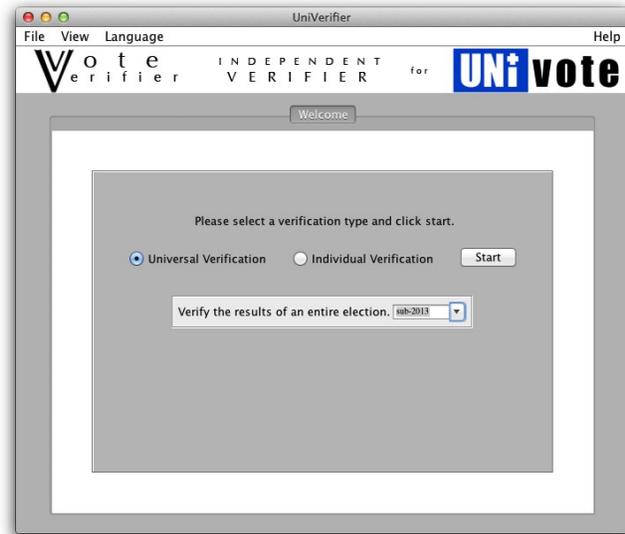

Illustration 1: The program is started up and the welcome panel is shown.

3   Menu and Options

The menu bar provides periphery functionality, which include exiting the program, displaying the text-only output of a verification, changing the language, and viewing information about the development of the program, as well as this manual.

Illustration 2: The menu bar provides periphery functionality.

3.1 File Menu

The *File* menu contains the option to exit and close the program.

3.2 View Menu

The *View* menu contains the option to show or hide the text console. To toggle the visibility, simply click on the check box next to the option *show console*. The console will show up at the bottom of the screen and although the menu has disappeared, the checkbox will now contain a check. To hide the console again return the to the *View* menu, and click once more upon the *show console*.

Illustration 3: A handy text output console can be show and hidden. When the program begins it is hidden.

3.3 Language Menu

VoteVerifier supports use in English, German, and French. The language can be changed through the *Language* menu, in which the options for the available languages appear. Upon clicking on one of the options, the program's visual content will be rebuilt with the text in the newly selected language. It is important not to change the language while a verification is in progress, or the progress will be lost and the verification will have to be started again.

3.4 Help Menu

To view information about the development of this software or to link quickly to this user manual options have been made available in the Help menu. In this menu you will first see the About menu, which will display a pop-up window with information about the development of this program.

To have quick access to this manual, the second option of the *Help* menu comes in quite handy. Click on *Manual*, and the document will automatically be loaded for viewing in your computer's preferred applications for PDF documents.

4 Beginning a Verification

There are two possible verification types that can be carried out. The differences between the two are primarily thoroughness. The Universal Verification option will evaluate every applicable parameter of the election system of a given election ID, whereas the individual verification verifies comparably much less.

4.1 Universal Verification

The length of a typical verification process using the Universal Verification is approximately 2 to 10 minutes depending on the processing power of the computer. In this verification all certificates, signatures, mathematical proofs, are verified and displayed for the user.

When the program begins you are all set to begin the universal verification, which is already selected by default. If an active internet connection was detected and a connection was successfully established with the election board, a list of possible election IDs will be displayed in the text box in the middle of the screen. This list constitutes the only possible election IDs that may be used to start a verification. If, however, no connection was possible, you are still able to manually enter an election ID and try to click start. If a connection was possible, the appropriate verification process will begin upon clicking start.

To choose one of the possible election IDs provided click on the downward-facing arrow in the text box in the middle of the screen. A list of the possible election IDs will pop-up and you will be able to select a new election ID from this list. After clicking on one of the possibilities, your selection will now be displayed in the text box, and you are now ready to begin the verification process by clicking on the blue *Start* button.
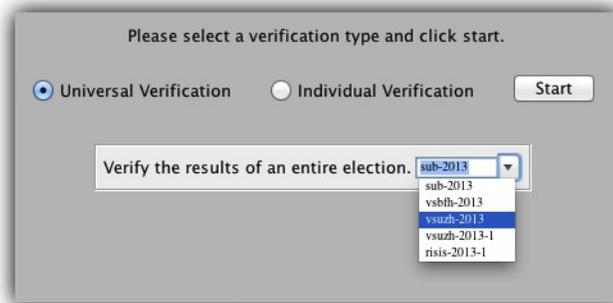
Illustration 4: The election IDs can be selected out of a provided list or inputed manually.

Clicking start causes a new tab to appear and its content will be immediately displayed. The verification results begin to appear and the status of the verification process is displayed. More about this verification tab can be found in the section entitled **Viewing Results**.

4.2   Individual Verification

The individual verification process is comparably much faster than universal verification. In this verification the certificates, signatures, mathematical proof, and if the ballot was included in the election results. The entire process lasts less than a few seconds.

To begin an individual verification change the selected verification type to individual verification by clicking on the button. The setup choices under your selection will change to display instructions for selecting a


Illustration 5: Execute an individual verification by selection an file which contains an election receipt in the form of a QR code.

file which contains an election receipt in the form of a Quick Response Code (QR Code). To choose the election receipt you wish to verify click on the *Select* button and a file chooser window will be displayed. Navigate to the file which contains the election receipt and click *Select*. The program shows you the name of the file your have selected. If you are content with your choice, you may continue by clicking on *Start*.

The contents of the file you provided will be checked for validity. If the file contains a valid QR Code a new tab will appear containing the verification results. More about this verification tab can be found in the section entitled **Viewing Results**. If the file provided did not contain a QR Code, you will be notified of this, and instructed to choose a different file.

4.3   Additional Verification Processes

It is possible to begin multiple verification processes that run parallel to each other. To begin another verification process while another is running, return at any time to the *Welcome* tab by clicking on it. Here you will be able to follow the steps described in the sections *Universal Verification* and *Individual Verification* to begin another verification.

5   Viewing Results

5.1   Tabs

The verification results are displayed in a table in separate tabs for each election verification process. The text at the top of each tab identifies the process to the user. A tab with results for a universal election displays the election ID of the results it contains. If the results are for an individual election the tab shows the prefix *Ind:* followed by the election ID. Clicking on a tab will show the results for that election verification process. Clicking on the Welcome tab will allow you to begin a new election verification process.
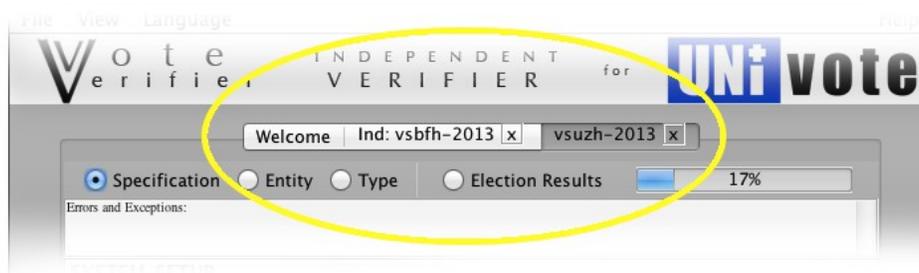

Illustration 6: The different verification processes are organized into tabs with the corresponding name of the election ID.

Each tab is organized into three parts. At the top there is a view panel which shows choices to view the results organized in a different way or to view the results of the election, which shows how many votes the parties and candidates received. After the button panel comes a text area in which *Errors and*

Exceptions will be displayed. And finally there is the main results panel in which the verification results are displayed. Each component of the tab is discussed in detail in the rest of this section.

5.2   Organizing Verification Results

When the verification tab appears that default organization for the results is set to specification. This means that the results are ordered according to how they are listed in the system specification of the UniVote electronic voting system.

System Specification Organization Sections
System Setup
Election Setup
Election Preparation
Election Period
Mixing and Tallying

By clicking on the other buttons next to *Specification* it is possible to change the organization. Clicking on Entity will order the verification results according to the entity in the UniVote system responsible for generating the content that was verified.


Illustration 7: Control how the verification results are organized with three different views.

Entity Organization Sections
Parameters
CA
EM
EA
Tallier
Mixer
Voters

Finally, it is possible to view the results sorted according to the type of verification that was carried out.

For example if a certificate was verified, than the results of this verification will appear with all the other results for certificate verifications.

> **Type Organization Sections**
>> Parameters
>> Certificate
>> RSA
>> NIZKP : proofs

5.3   Interpreting results

There are 4 possible outcomes for the verification results. A result may pass, fail, not be implemented, or have caused an error. In the case that the result passed the verification a green checkmark will appear at the end of the line where the verification is described. If the verification was unsuccessful, then in place of a green checkmark, a red "x" will appear.

Illustration 8: Icons representing the four possible verification results.

The two remaining results of a verification are special cases in which either the content for the verification has not yet been implemented by the UniVote team, or an error occurred during the processing of the verification. If a verification has yet to be implemented an orange question mark will be shown. If an error was produced then a yellow warn symbol is shown. In any case except a successful result, additional information about why the test failed and what this means can be obtained through the *Helper Text*. More information about this feature is available in the section *Helper Text*.
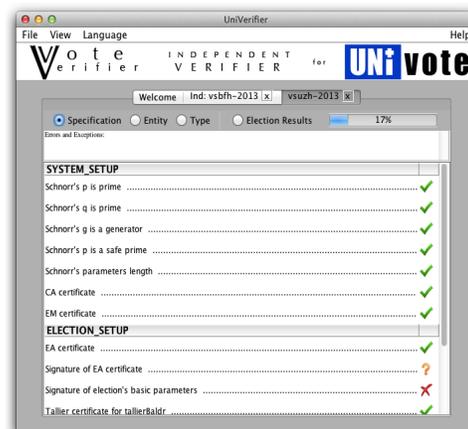

Illustration 9: The verification output is displayed in a corresponding tab.

5.4   Helper Text

If a verification was not successful additional information about what went wrong may be obtained by clicking on or placing the mouse over the verification for which you would like to retrieve more information. A small blue helper text will appear and provide a brief explanation of why the verification was not a success.

5.5   Candidate Results

This view shows the results of the elections. It provides the information of how many votes were received by which candidates and parties. The choice *Election Results*, which is located after the choices for organizing the verification results, will take you to this view. The results are also organized in table form according to political party. In this section on one view is possible.

The table header shows the name of the party and how many votes it received. The content of each table lists the candidate for that party, as well as how many votes each candidate received.

Illustration 10: Election Results displays the voting results of the elections.

5.6   Errors and Exceptions

If there were any serious problems during the verification process such as a system crash, or broken contact with the election board, you will be notified of them in the *Errors and Exceptions* text area at the top of the results panel. In the odd case that there were so many errors that the screen became filled, this text area will contain all this information and allow you to scroll through the information.

6   Thanks

Thank you for using the VoteVerifier independent verification software for UniVote electronic elections. We hope that you feel more assured in the discretion, veracity, and exactitude of the UniVote system, and that you will continue to use this software in the future. For questions and comments, please visit the UniVote website at www.UniVote.ch and contact one of the members of the UniVote team.