UNIVERSITY OF TRENTO

Advanced Programming of Cryptographic Methods

Project Report

SecureSurvey

D'Achille Letizia, Montanari Sara

04/02/2024

Table of Contents

Description

SecureSurvey provides a comprehensive solution for researchers conducting sensitive surveys with binary voting, offering a privacy-preserving environment through homomorphic encryption, zero-knowledge proofs and digital signatures. The system prioritizes the confidentiality and integrity of the voting process while ensuring that only entitled individuals can participate and their choices remain private.

User Roles:

- Voter:
    - Knows his/her username and password to log in.
    - Has the ability to cast a binary vote (Yes/No).
    - Can view the polls and make selections without revealing preferences.
    - Can see only the open surveys to which he/she hasn't already voted.
    - Generates a public/private key pair during registration for digital signature purposes.
- Researcher:
    - Knows his/her username and password to log in.
    - Can create and close polls securely: a pair of keys is generated for each survey.
    - Manages the overall survey process without accessing individual votes: when he/she closes the survey, the results are revealed.

Server Role:

- Stores credentials of every user to manage log in and sign up phases, using hash and salt techniques.
- Manages the creation and closure of polls without handling clear information.
- Receives and tallies encrypted binary votes using homomorphic encryption.
- Verifies votes with signatures and zero knowledge proofs before storing them.
- Does not have knowledge of individual choices but can count votes securely.

Requirements

Functional Requirements

1. **Survey Creation and Management:** Researchers should be able to create new surveys and close them. Additionally, researchers should have the ability to view and manage all active and closed surveys.
2. **Anonymous Binary Voting:** Voters should be able to anonymously cast binary votes (“Yes” or “No”) on the available surveys without revealing their preferences to the system.

3. **User Registration and Authentication:** Voters and researchers should be able to register their accounts. Voters should have the ability to generate a public/private key pair during registration for digital signature purposes. The system should authenticate users during the login process to ensure that only entitled individuals can access their respective functionalities.

4. **Vote Counting:** The system should securely count the votes without decrypting individual choices. It should provide accurate and reliable tallying of votes.

5. **Abstention Option:** Voters should have the option to abstain from voting. The system should handle the "abstained" option to ensure the integrity of the vote count while preserving the anonymity of individual selections.

6. **Survey Participation:** Voters should be able to view the available surveys, make their selections, and submit their votes securely.

7. **Survey Results:** After the closure of a survey, the system should provide researchers with access to the aggregated results while ensuring the anonymity of individual voters. Researchers should be able to analyze the survey results to derive insights.

8. **User-friendly Interface:** The system should feature a user-friendly interface for both voters and researchers, facilitating ease of navigation, clear presentation of survey options, and intuitive interaction with the system functionalities.

Security Requirements

1. **Homomorphic Encryption for Vote Counting**: Utilizes homomorphic encryption to allow the voters to cast a vote and the server to securely count votes without decrypting individual choices. The researcher is then able to retrieve and decrypt the final results.

2. **Digital Signatures for Voter Authentication**: Each voter generates a public/private key pair during registration. The private key is used to sign the vote for digital signature purposes. Ensures that only entitled individuals can cast votes, and the authenticity of the votes can be verified using the respective public key.

3. **Abstention Option for Privacy**: Introduces an "abstained" option to allow users to abstain from voting without revealing their actual choice. The handling of the "abstained" option is designed to uphold vote count integrity while preserving the anonymity of individual selections.

4. **Zero-Knowledge Proofs**: Implement zero-knowledge proofs to enhance privacy. This allows voters to prove they know a piece of information without revealing the information itself. For instance, a voter can prove that he selected an admissible choice between the two possible options (Yes/No).

5. **Hash and Salt**: Protect user passwords from unauthorized access and ensure the confidentiality of sensitive information stored in the system by implementing a robust password hashing mechanism with the use of a unique salt for each user. The salt must

be combined with the user's password before applying the hash function, this ensures that even users with the same password will have different hash values due to their unique salts. Only the hashed password and the corresponding salt should be stored in the system's database.

Technical Details

Architecture[1]

All the project code lies in the folder **code**. This folder contains:

Server side:

- **server**: Folder containing all server-side files. Anything outside of this is user-side.
    - **voters**: folder that contains one file for each voter called ***username.txt***. This file contains the list of “id” of the surveys for which this voter has already voted.
    - **surveys**: folder that contains one file for each survey that has been created, called ***id.txt***. This file contains info about the survey.
      For each “open” survey, it contains: “true” in the first line, the public key of the survey in the second line and all the verified votes in the following lines.
      For each “closed” survey, it contains: “false” in the first line and the encryption of the result in the second line (tot yes, tot no, tot abstain).
    - ***surveys.txt***: this file contains one line for each survey that has been created. The first part of each line contains the username of the researcher that created the survey, the second part contains the content of the survey.
    - ***ServerSurvey.java***: class that extends Survey (see user side below). This class implements a survey through the point of view of the server that has access to more info with respect to the user side. Indeed this class has all the attributes and methods of the Survey class, plus additional features that are hidden from the user side.
    - ***Server.java***: this java class implements and simulates the server with all its methods.
    - ***credentials.txt***: this file contains one line for each user: the first part contains the username, the second part contains the digest of (password+salt), the third part contains the salt and the last one contains the role of this user (either “voter” or “researcher”).

---

[1] **Green**: folders
Bordeaux: java files
Light brown: text files
Black: other files

Application side:

- **`voters`**: folder that contains one file for each voter user called ***`username.txt`***. This file contains the private key of each voter, the one that he/she will use for the signature.
- **`researchers`**: folder that contains one file for each researcher user called ***`username.txt`***. This file contains one line for each survey created by the researcher. Each line is composed of two parts: the first one is the "id" of the survey, the second one is the public key associated with this survey.
- **`lib`**: this folder contains the library used, UniCrypt.
- **`.vscode`**: this folder contains settings to include the library in the Visual Studio Code ambient.
- ***`Application.java`***: file that implements the application with the GUI code for the interface, and other user side methods.
- ***`Survey.java`***: java class for a survey, with attributes and methods related to the user side.
- ***`Vote.java`***: java class that represents a vote, it includes the ciphertext, the signature and the zero knowledge proofs.
- ***`clear_credentials.txt`***: file containing clear-text credentials for logging in (obviously, in the project scenario, this file doesn't exist; credentials should be stored individually by each user).
- ***`Application.jar`***: executable file that runs the project.
- ***`run.bat`***: script file to automate the process of running the Java application.

User Interface Description

The application runs through an interface made of many panels. Here we describe each of them.

- **Login panel**:

  The first panel that opens after the application is executed is the login panel. From this panel, the user can enter credentials to proceed. In case of incorrect credentials, a red error message will appear. In the top left, there is a button to switch to the sign-up panel.

- **Sign up panel**:

  In this panel, it is possible to create a new account by entering a new username and a password. Both have rules for acceptance, regarding the length of the string or the

presence of special characters or uppercase letters (see Guidelines for the use of the software in the final section Instructions for Installation and Execution). Additionally, the password must be re-entered for added security. Finally, the user must select the role: *voter* or *researcher*. In case of errors, red feedback will appear. If everything goes well, the account is created. In the top left, there is a back button to return to the login panel.

In case of successful login, two scenarios are possible:

If the user role is *voter*:

- **Voter main panel**:

  In this panel, the surveys that this voter can vote on (i.e., those that are open and haven't been voted on yet) are visible. For each of them, the title is displayed, and on the right, there is a button to vote. In the top left, there is the logout button to exit and return to the login panel. Above this, the username of the currently logged-in user is indicated.

- **Vote panel**:

  This panel allows the voter to express a preference regarding the selected survey. In the center, there is the title of the survey, while at the bottom, there are three buttons to vote: *yes*, *no*, and *abstain*. By clicking on one of these, the vote will be sent to the server, and the panel will return to the Voter main panel. In the top left, there is a button to return to the Voter main panel without casting the vote.

If the user role is *researcher*:

- **Researcher main panel**:

  This panel shows the researcher the surveys he/she has created so far. At the top, there is a button to create a new survey, which takes him/her to the New survey panel (see below). Further down, there is the list of surveys of the researcher. For each open survey, the following information is visible (from left to right): the title, the status (open) with the number of votes registered so far, and a button to close the survey. For each closed survey, the following is visible (from left to right): the title, the status (closed), and the results (total yes, total no, total abstain). In the top left, there is a button to log out and return to the login panel.

- **New survey panel**:

  This panel allows the researcher to create a new survey. In the center, there is a space to enter the content of the new survey (max 5 lines), while below there is the button to create and open the survey. Once the button is clicked, we will return to the Researcher main panel. In the top left, there is a button to go back to the previous panel without creating any new survey.

Implementation

Programming language: Java. It allows us to exploit Object-Oriented Programming and several standard libraries.

Library: UniCrypt [3]. This library implements in a secure way all the cryptographic techniques that we need in this project, as we detail below.

Here we describe the implementation of the cryptographic techniques[2] used to achieve each security requirement.
To implement all these techniques we used the library UniCrypt mentioned above.

1. **ElGamal Homomorphic Encryption for Vote Counting**:

   The scheme is built on a SafePrime of 3072 bits.

   - Public parameters:
     Starting from a *safePrime* (type *SafePrime*), we derive a *group* (*GStarModSafePrime*) and a *generator* (*GStarModElement*) of it, together with its *order* (*ZMod*). From the generator we get the scheme *elGamal* (*ElGamalEncryptionScheme*)

   - Key generation:
     From the scheme *elGamal* we get a *keyPair* (type *Pair*). The first component of *keyPair* is the private key, the second component is the public key.

   - Encryption:
     Given a plaintext *mex,* we get *encodedMex* (the corresponding power of the generator). From *encodedMex,* the public key of the survey *publicKey* and a random element *r*, we get the ciphertext *encMex* (through the method *encrypt* of the scheme *elGamal)*. In our project, we need to encrypt a vote composed of three components. For this purpose, we use a different random element in the encryption of each component.

   - Homomorphic addition:
     Given two ciphertexts *ciph1 , ciph2,* we get the ciphertext of the sum between the two plaintexts related to *ciph1* and *ciph2*, by executing: *ciph1.apply(ciph2)*. This method returns an object of the type *Tuple*.

   - Decryption:
     Given a ciphertext and the private key of the survey, we get the power of the generator with the plaintext as exponent through the method *decrypt*() of the scheme *elGamal*. Then we perform brute force on the discrete logarithm to retrieve the plaintext.

2. **Schnorr Digital Signatures Scheme for Voter Authentication**:

   The scheme is built on a (different) SafePrime of 3072 bits.

   - Public parameters:

[2]A more theoretical and detailed explanation of the cryptographic techniques used is provided in the subsection Cryptographic techniques details.

Starting from a *signSafePrime* (type *SafePrime*), we derive a *signGroup* (*GStarModSafePrime*) and a *signGenerator* (*GStarModElement*) of it, together with its sign*Order* (*ZMod*). From the generator we get the scheme *shnorr* (*SchnorrSignatureScheme*).

- Key generation:
  From the scheme *schnorr* we get a *keyPair* (type *Pair*). The first component of *keyPair* is the private key, the second component is the public key.

- Sign:
  Given a ciphertext *ciphertext* and the private key *signPrivateKey* of the user that generates the signature, we create the signature through the method *sign()* of the scheme *schnorr.*

- Verify:
  Given a ciphertext *ciphertext*, a signature *signature* related to it, and the public key *signPublicKey* of the user that generated the signature, we verify the signature through the method *verify()* of the scheme *schnorr.*

3. **Abstention Option for Privacy**:

   The implementation of the Abstention Option for Privacy involves encrypting the vote components as follows:

   - “Yes" vote: Encrypt the tuple $(1, 0, 0)$.
   - "No" vote: Encrypt the tuple $(0, 1, 0)$.
   - "Abstain" vote: Encrypt the tuple $(0, 0, 1)$.

   Encryption is performed component-wise, hence we generate the encryption of 0 (named *encZero*) and the encryption of 1 (*encOne*) for each vote. We also generate a distinct encryption (*encZero2*) for the second 0 in the tuple to ensure that the vote cannot be guessed by examining the three encrypted components, as two of them would always be identical.
   Each encrypted component is a *Pair* of elements of type *GStarModElement*, i.e. two numbers are generated for each component of the vote. As a consequence, overall the encrypted vote consists of six numbers in total.

4. **Zero-Knowledge Proofs**:

   - Scheme generation:
     Starting from the string *username* we get a *proverId*. From the *group* (related to the encryption scheme) and the *proverId* we get a *challengeGenerator* (type *SigmaChallengeGenerator*). Now we create a *subset* (*Subset*) starting from *group* and at least two messages. The zero knowledge attached to a ciphertext proves that the plaintext from which the ciphertext derives belongs to *subset*. From *challengeGenerator, elGamal, publicKey* (the public key of the survey)*, subset* we generate the *proofSystem* (type *ElGamalEncryptionValidityProofSystem*).

   - Proof generation:
     Given a ciphertext *ciphertext* (type *Pair*) and a random element *r* (the same used for the encryption of this ciphertext), we create the zero knowledge proof (type *Triple*) through the *proofSystem*.

   - Proof verification:

Given a ciphertext *ciphertext* and a zero knowledge proof related to it *proof*, we verify it through the method *verify(proof, ciphertexts)* of the object *proofSystem*.

5. **SHA-384 hash and salt**:

   Hashing is performed using an hashing scheme already implemented in UniCrypt. We enforce the use of SHA-384, a strong hash function.
   Moreover, we generate salt values of length 8 bytes.

   - Parameters generation:
     We get a *passwordSpace* (type *StringMonoid*) from which is derived the *scheme* (*PasswordHashingScheme*).
   - Salt generation:
     We generate the *salt* (type *ByteArrayElement*) through the class *ByteArray* of length 64 bits. The *salt* is then converted into a string through the method *Base64.getEncoder().encodeToString()*.
   - Hashing:
     From a string we derive an object of the class *StringElement,* through a method of *passwordSpace*. This object is given as input (together with the salt string) to the method *hash* of *scheme*. This method returns an object of the class *FiniteByteArrayElement*, that is finally converted into a string through the method *Base64.getEncoder().encodeToString()*.

Code Structure

The code is composed of 5 java files. Here we describe each of them, explaining the instructions and how they interact together.

1. Application.java

This java file implements the base structure of the application.
Here we explain the code from top to bottom:

- In the first part we import all we need from the UniCrypt library and other java utilities packages.
- Declaration and initialization of the GUI objects.
- Declaration and initialization of session parameters, server, public parameters for encryption and signature, auxiliary objects.
  In particular, all the public parameters for ElGamal encryption scheme depend on a unique value: *safePrime*.
  Similarly, all the public parameters for Schnorr signature scheme depend on *signSafePrime*.
- ***checkCredentials***[3]: method that checks for the validity of username and password during sign up (username: 5-16 alphanumeric characters; password: at least 8

[3] It appears in Figure 2.

characters, at least one uppercase letter, one lowercase letter, one number, one special character; the two password fields are equal).

INPUT:	*tempusername* (inserted username) *temppassword* (inserted password) *checkpassword* (re-inserted password)
OUTPUT:	*validity* (array of three boolean values: check succeeded if all the three values are true)

- ***createNewUser***[4]: method called when a new account is created.
  If the role is *voter*, a pair of keys for signature is created; moreover the file *username.txt* in the folder *voters* is created and the private key is written on it. The public key is returned.
  If the role is *researcher*, the file *username.txt* in the folder *researchers* is created. An empty string is returned in this case.

INPUT:	*tempusername* (username) *temprole* (role of the user)
OUTPUT:	*publicKey* (public key of the user that will be used to verify the signature) or *empty string*

- ***encryptVote***[5]: method called when a user has voted for a survey, here the vote is encrypted and signed, moreover, two zero knowledge proofs are created: the proof that each element is either 0 or 1, the proof that the sum of the three elements is 1.
  The zero knowledge system is based on the string *username* (username of the voter).
  For the proof that the sum of the three elements is 1, we use an *impossibleElement*: the sum of the three elements (that are either 0 or 1) cannot be 4. In this way we are able to use the same zero knowledge system of the first three proofs (the method *Subset.getInstance* needs at least two elements).
  The message that we sign is the concatenation of the three components (each one made of two values) of the ciphertext, called *fullCiphertext*.

INPUT:	*vote* (either “Yes” or “No” or “Abstain”) *publicKey* (public key of the survey) *signPrivateKey* (private key of the voter for signature)
OUTPUT:	*encVote* (object of the class vote containing ciphertext, signature and zero knowledge proofs)

- Methods for generating and refreshing all the interface panel

[4] It appears in Figure 2.
[5] It appears in Figure 5.

- ***main***: standard main method that starts the application with login and signup panels.
- ***actionPerformed***: it manages the action performed by the user. Through a switch-case, each action is followed by instructions:
    - "Login":
      ask the server to check credentials by calling *server.readCredentials* and clear password variable. In case of error, add red feedback messages to the panel; otherwise ask the server for the surveys related to this user by calling *server.serverReadSurveys*. Update the interface according to the role of the user.

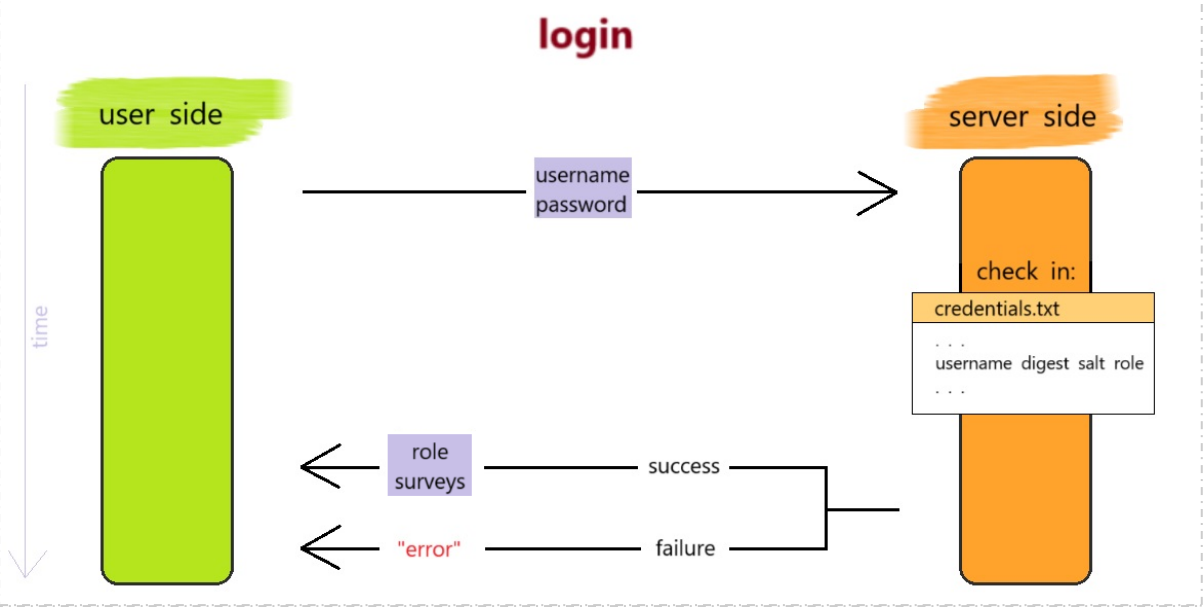

Figure 1: diagram depicting the login process.

    - "Sign up":
      move to the Sign up panel.

    - "Register credentials":
      store credentials and reset text fields. Check for the validity of the credentials by calling *checkCredentials* and clear the variable containing the re-inserted password.
      If the check succeeds, ask the server to register credentials by calling *server.registerCredentials* and clear password variable. In case of no errors, create the new user by calling *createNewUser*. If the user is a voter, store the public key in the server side (for signature verification) by calling *server.registerPublicKey*. Update interface.

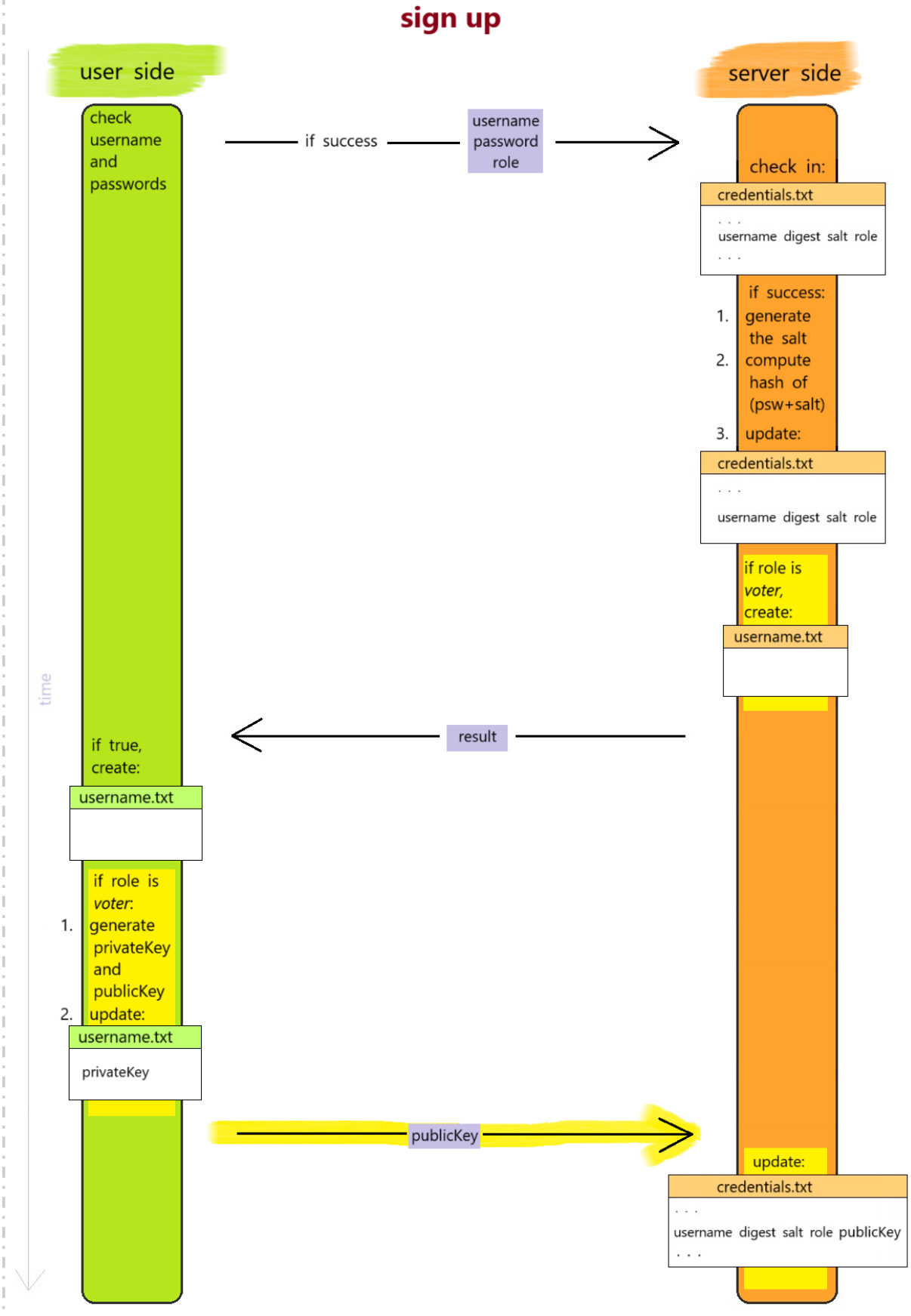

Figure 2: diagram depicting the sign up process.

- "Back to login":
  move to the Login panel.

- "Logout":
  reset variables and panels and move to the Login panel.

- "Create":
  move to the New survey panel.

- "Open":
  create private and public keys for the new survey. Ask the server to add the new survey by calling *server.writeSurvey*, this method returns the updated list of surveys related to the current user. Write the private key of the new survey in the file of the current researcher by calling *survey.writeSurvey.* Update interface.

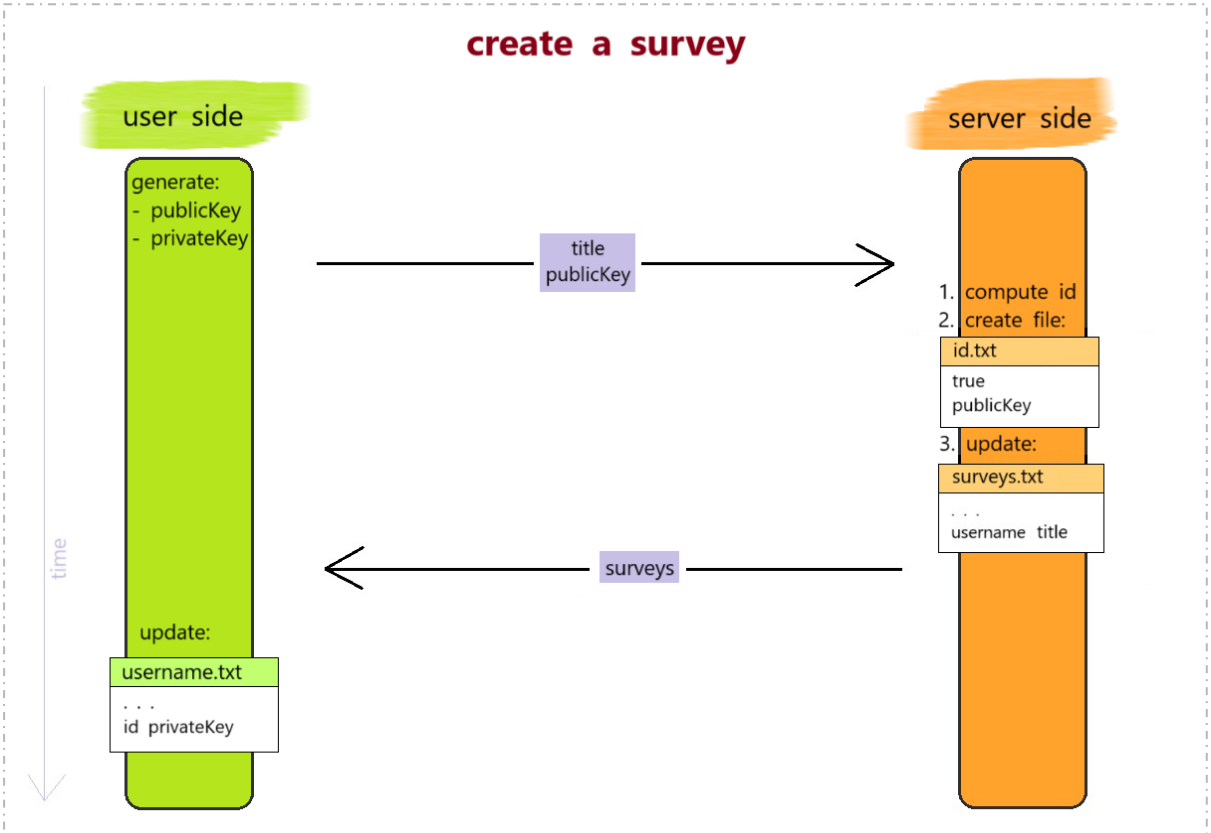

Figure 3: diagram depicting the process of creating a new survey.

- "Close":
  ask the server to close this survey by calling *server.serverCloseSurvey* and refresh panels.

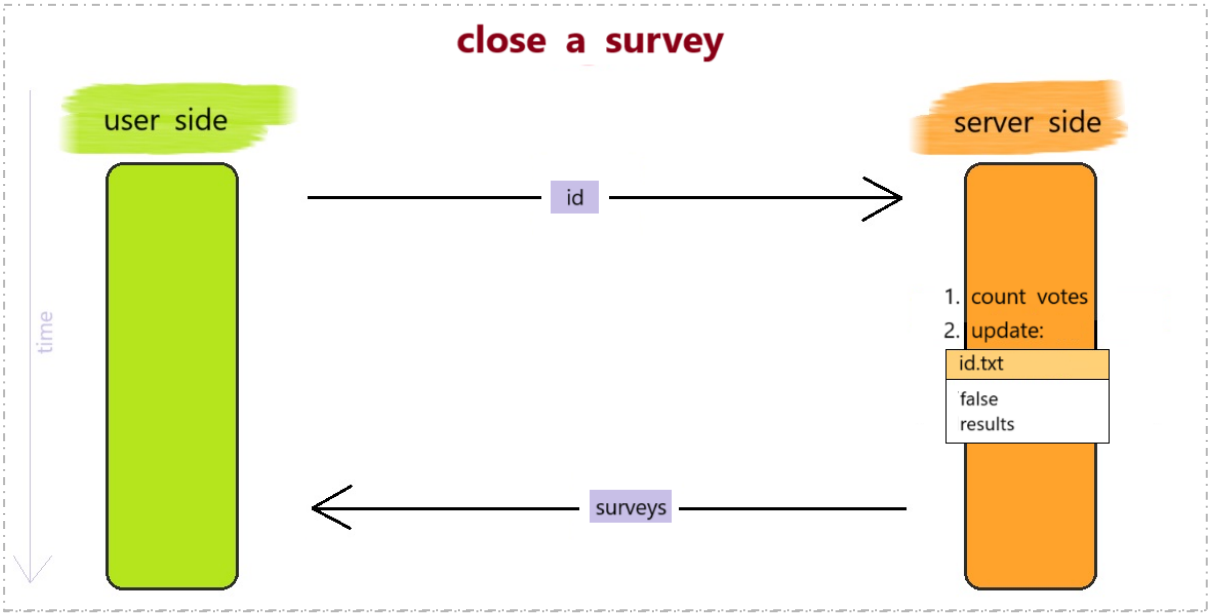

Figure 4: diagram depicting the process of closing a survey.

- "Vote":
  individuate the survey and move to the Vote panel.

- "Yes", "No", "Abstain":
  individuate the survey and recover the public key by calling *survey.voteSurvey*. Encrypt the vote and generate signature and zero knowledge proofs by calling *encryptVote*. Send the vote to the server by calling *server.serverVoteSurvey*. Update interface.

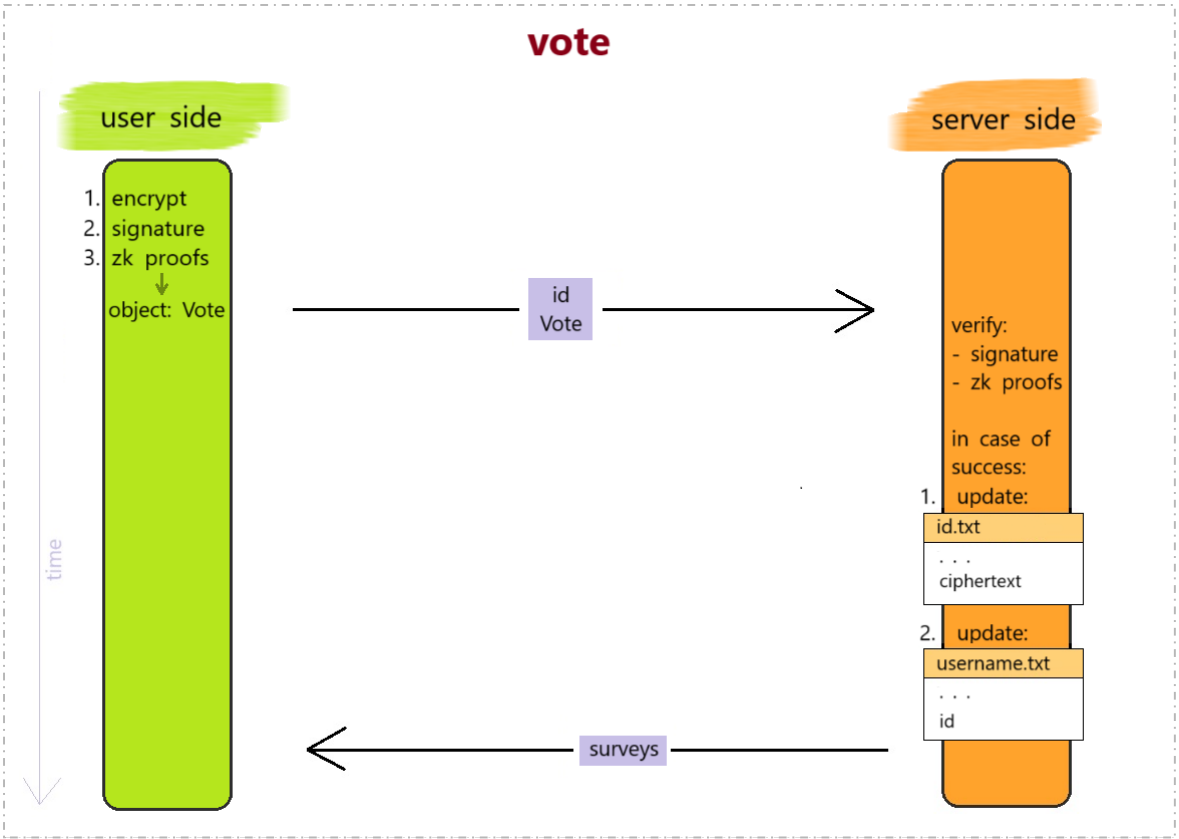

Figure 5: diagram depicting the voting process.

- "Back":
  move to the Researcher main panel or Voter main panel, according to the role of the current user.

2. Survey.java

This java file implements the class *Survey*: methods and attributes of a survey related to the application side (user side).

Here we explain the code from top to bottom:

- In the first part we import all we need from the UniCrypt library and other java utilities packages.
- Private attributes:
  - *id:* id of the survey
  - *researcher:* username of the researcher that has created this survey.
  - *title:* content of the survey.
  - *status:* open (true) / close (false).
  - *numvotes:* number of votes stored for this survey at the moment.
  - *result:* before the closure it is empty, after the closure it contains the encryption of the results for this survey.
  - *publicKey:* public key of this survey, it is used to encrypt votes for this survey.

- Declaration and initialization of public parameters for encryption and signature.
  In particular, all the public parameters for ElGamal encryption scheme depend on a unique value: *safePrime*.
  Similarly, all the public parameters for Schnorr signature scheme depend on *signSafePrime*.

- Public constructor *Survey* that sets values of *id, researcher* and *title*.

- Public methods *getId, getResearcher, getTitle, getStatus, getNumvotes, getPublicKey* to return attributes.

- ***getResult***: this method decrypts the encrypted results of this closed survey, using the private key of this survey that is known only by the researcher (application side) in the file *username.txt*.
  The last phase of the decryption is a brute force on the discrete logarithm.

INPUT:	—
OUTPUT:	*decodedResult* (array of three int: tot yes, tot no, tot abstain)

- ***writeSurvey***[6]: method called when a researcher has opened a new survey. It writes the private key of the new survey in the researcher file *username.txt* (application side).

INPUT:	*userpath* (path of the file *username.txt* of the researcher), *privateKey* (private key of this survey)
OUTPUT:	—

- ***voteSurvey***: method called when a voter has voted to this survey. It recovers the private key of the voter from the file *username.txt* of the current voter (application side). This key will be used to create the signature.

INPUT:	*userpath* (path of the file *username.txt* of the voter)
OUTPUT:	*signPrivateKey* (private key of the voter)

[6] It appears in Figure 3.

3. Vote.java

This java file implements the class *Vote*: methods and attributes related to a single encrypted and signed vote with its zero knowledge proofs.

Here we explain the code from top to bottom:

- In the first part we import java utilities packages.
- Private attributes:
  - *ciphertexts:* encryption of the vote, array of three elements of type *Pair*.
  - *proofs:* array of three zero knowledge proofs related to the three elements of *ciphertexts*. One proof proves that the plaintext from which a component of *ciphertexts* derives is either 0 or 1.
  - *proofSum*: zero knowledge proof, it proves that the sum of the three plaintexts is 1.
  - *signature*: signature of the vote using the private key of the voter.
- Public constructor *Vote* that sets values of *ciphertexts, proofs, proofSum, signature*.
- Public methods *getCiphertexts, getProofs, getProof, getSignature* to return attributes.

4. Server.java

This java file implements the server.

Here we explain the code from top to bottom:

- In the first part we import all we need from the UniCrypt library and other java utilities packages.
- Declaration and initialization of session parameters, public parameters for signature and parameters for password hashing.
  In particular, all the public parameters for Schnorr signature scheme depend on *signSafePrime*.
  Session parameters:
  - *numsurveys*: total number of existing surveys.
  - *username*: username of the user currently logged in.
  - *serverSurveys*: arraylist of objects of the class *ServerSurvey* representing the surveys related to the user currently logged in.
  - *role*: role of the user currently logged in.
  - *signPublicKey*: public key of the user currently logged in, to verify his/her signature.
- Public methods *setNumsurveys, setUsername, setSurveys, setRole* to set the value of the related attributes.

- Public methods *getNumsurveys, getUsername, getRole* to get the value of the related attributes.

- ***registerCredentials***[7]: method called when a user is trying to sign up, it checks if the inserted credentials are already present in the file *credentials.txt*. In case of valid credentials, it generates a salt and the digest of (password + salt) and it stores them in the file *credentials.txt*. Moreover, if the user is a voter, it creates the new file *username.txt* (server side).

INPUT:	*tempusername* (inserted username) *temppassword* (inserted password) *temprole* (inserted role)
OUTPUT:	*result* (boolean value: true if credentials are accepted)

- ***registerPublicKey***[8]: method called when a voter has signed up, it stores the public key of the voter in the file *credentials.txt*.

INPUT:	*tempSignPublicKey* (public key of the voter that has signed up)
OUTPUT:	—

- ***readCredentials***[9]: method called when a user is trying to log in, it checks his/her credentials comparing the digest of (password + salt) with the related stored value in *credentials.txt*. If the check succeeds, it sets the values of the session parameters *username*, *role* and *signPublicKey* (the last one only if the user is a voter).

INPUT:	*username* (inserted username) *password* (inserted password)
OUTPUT:	*role* (if the check succeeds) or *"error"*

- ***serverReadSurveys***: this method returns an arraylist of the surveys (objects of the class *Survey*) related to the user currently logged in.
  If the user is a researcher, *surveys* contains all the surveys created by the researcher. This is done by reading the file *surveys.txt* (server side) and comparing the username of the user with the username of the researcher that created each survey.

[7] It appears in Figure 2.
[8] It appears in Figure 2.
[9] It appears in Figure 1.

If the user is a voter, *surveys* contains all the open surveys for which the voter has not already voted. This is done by reading the file *username.txt* (server side) and storing the id of the surveys (for which the user has already voted) in the sorted array *alreadyVoted* (using insertion sort). Then we read the file *surveys.txt* (server side) and we select only the open surveys whose id does not appear in *alreadyVoted* (using linear search on a sorted array).
While this process is executed, this method also creates an arraylist of surveys from the class *ServerSurvey,* called *serverSurveys* (server side)*.* This list corresponds to the session parameter *serverSurveys* described above. It contains the surveys related to the user currently logged in. Note that the two lists *serverSurveys* and *surveys* contain the same list of surveys with the important difference that in the first one is made of surveys from the class *ServerSurvey* (server side), while the second one is made of objects of the class *Survey* (application side). This difference is relevant in terms of security: the two classes have distinct access to elements as security parameters, keys and others, according to their role in our scenario.

INPUT:	—
OUTPUT:	*surveys* (arraylist of surveys of the class *Survey*, related to the user)

- ***writeSurvey***[10]: method called when a researcher has opened a new survey. It writes the new survey in the file *surveys.txt* and it creates a new file *id.txt* for the new survey writing on in "true" and the public key of the survey. Moreover, it updates the two arraylists *surveys* and *serverSurveys.* The list *surveys* is finally returned.
  Obviously, it updates also the arraylist *serverSurveys*

INPUT:	*title* (content of the new survey) *publicKey* (public key of the new survey)
OUTPUT:	*surveys* (updated list of surveys related to the user)

- ***serverCloseSurvey***[11]: method called when a researcher closes a survey. It closes the selected survey through the method *closeSurvey()* of the class *ServerSurvey*. Moreover, it updates the two arraylists *surveys* and *serverSurveys.* The list *surveys* is finally returned.

INPUT:	*id* (id of the survey that needs to be closed)

[10] It appears in Figure 3.
[11] It appears in Figure 4.

OUTPUT:	*surveys* (updated list of surveys related to the user)

- ***serverVoteSurvey***[12]: method called when a voter has voted in a survey. It verifies the vote through the method *verifyVote()* of the class *ServerSurvey*. If the check succeeds, it writes the id of the survey in the voter file *username.txt* (this voter can not vote for this survey anymore), through the method *voteSurvey* of the class *ServerSurvey*. Moreover, it updates the two arraylists *surveys* and *serverSurveys*. The list *surveys* is finally returned.

INPUT:	*id* (id of the survey related to the vote) *encVote* (object of the class *Vote*, it is the vote that needs to be verified and eventually stored)
OUTPUT:	*surveys* (updated list of surveys related to the user)

5. ServerSurvey.java

This java file implements the class *ServerSurvey*: methods and attributes of a survey related to the server side. This class extends the class *Survey*.

Here we explain the code from top to bottom:

- In the first part we import all we need from the UniCrypt library and other java utilities packages.
- Private attributes (in addition to those of the class *Survey*):
  - *path*: path of the survey file *id.txt* (server side).
- Public constructor *ServerSurvey* that sets values of *id, researcher* and *title* (through the constructor of the superclass); sets the value of *path* and calls the method *readSurvey(path)*.
- ***readSurvey***: method called by the constructor of this class, it reads the survey's file *id.txt* to set the values of *status, publicKey, numvotes* and *result* (attributes of the superclass *Survey*).

INPUT:	—
OUTPUT:	—

[12] It appears in Figure 5.

- ***verifyVote***[13]: method called by the method *serverVoteSurvey* of the class *Server*. It verifies the vote given as input.
  The proof systems are reconstructed from the string *username*.
  The message *fullCiphertext* is reconstructed from the ciphertext (it is the message on which the signature is created).
  The and (*&&*) concatenation of the five checks is returned.

INPUT:	*username* (username of the voter, used to verify the zero knowledge) *signPublicKey* (public key of the voter, used to verify the signature) *encVote* (object of the class *Vote*, it is the vote that needs to be verified)
OUTPUT:	boolean value: *true* if the check succeeds, *false* otherwise

- ***voteSurvey***[14]: method called by the method *serverVoteSurvey* of the class *Server* in case of successful verification of a vote, it appends the ciphertext to the survey file *id.txt* and writes in the voter file *username.txt* (server side) the id of this survey.

INPUT:	*encVote* (object of the class *Vote*, it is the already verified vote that needs to be stored) *userpath* (path of the voter file *username.txt* (server side))
OUTPUT:	—

- ***closeSurvey***[15]: method called by the method *serverCloseSurvey* of the class *Server*. It reads the votes from the survey file *id.txt*, counts the votes and writes the encrypted results on the survey file *id.txt*.
  The attribute *status* becomes *false*, the attribute *result* is set to the outcome of the tallying.

INPUT:	—
OUTPUT:	—

[13] It appears in Figure 5.
[14] It appears in Figure 5.
[15] It appears in Figure 4.

Security Considerations

Security achievements

- **Confidentiality (ElGamal Homomorphic Encryption):** The system ensures the confidentiality of voting choices by utilizing ElGamal homomorphic encryption. This encryption scheme allows the server to securely count votes without decrypting individual choices, thus preserving the privacy of participants.
- **Verifiability by the Server (Zero-Knowledge Proofs for ElGamal Homomorphic Encryption):** Verifiability of vote counts by the server is achieved through the implementation of zero-knowledge proofs for ElGamal homomorphic encryption. These proofs enable the server to verify the correctness of the encrypted vote without revealing individual voting choices.
- **Eligibility (Schnorr Digital Signature):** The system ensures the eligibility of voters through digital signatures generated by voters using public/private key pairs. Although the Digital Signature Algorithm (DSA) was not implemented, the chosen parameters for Schnorr digital signature ensure strong cryptographic security.
- **Authenticity (Password Hashing):** User authentication is enforced through password security measures. Passwords are hashed using the SHA-384 algorithm.
- **Mitigation to Coercion (Abstention Option):** The system offers an "abstained" option to mitigate coercion, allowing users to abstain from voting without revealing their actual choice. This feature enhances the privacy of the voting process by preserving the anonymity of individual selections.

Cryptographic techniques details:

- **Homomorphic Encryption (ElGamal):** This technique allows the server to perform sums among encrypted votes, yielding results that remain meaningful when decrypted, without exposing sensitive information during computation.
  The scheme is based on the ElGamal encryption scheme [2].
  A vote is encoded as

  $$v = (v_1, v_2, v_3) \in \{0,1\}^3$$

  where $(1,0,0) \equiv Yes$, $(0,1,0) \equiv No$, $(0,0,1) \equiv Abstain$.
  Given

  g, g': generators of a cyclic group of prime order p
  x: private key in $\mathbb{Z}_p$
  $h = g^x$: public key
  r: random in $\mathbb{Z}_p$
  m: message (plaintext)

  Then

$$E(m) = (g^r, g'^m \cdot h^r)$$

$$D(c_1, c_2) = \texttt{brute-force}\,(c_2 \cdot (c_1^x)^{-1}) = \texttt{brute-force}\,(g'^m) = m$$

In this formulation, ElGamal is additively homomorphic, i.e.

$$E(m_1) \cdot E(m_2) = E(m_1 + m_2)$$

Indeed

$$E(m_1) \cdot E(m_2) = (g^{r_1}, g'^{m_1} \cdot h^{r_1}) \cdot (g^{r_2}, g'^{m_2} \cdot h^{r_2}) =$$
$$= (g^{r_1+r_2}, g'^{m_1+m_2} \cdot h^{r_1+r_2}) = E(m_1 \cdot m_2)$$

Let $v^{(i)}$ denote the vote of a voter *i*.
Computing

$$E(v^{(i)}) = (E(v_1^{(i)}), E(v_2^{(i)}), E(v_3^{(i)}))$$

for each voter, the server can perform the sums

$$(E(\sum v_1^{(i)}), E(\sum v_2^{(i)}), E(\sum v_3^{(i)})) = (\sum E(v_1^{(i)}), \sum E(v_2^{(i)}), \sum E(v_3^{(i)}))$$

and the results can be retrieved by the researcher decrypting each component.
The system employs a safe prime of length 3072 bits, that ensures a security strength of 128 bits in accordance with NIST recommendations for discrete logarithm-based schemes [1].

- **Zero-Knowledge Proof (for ElGamal Homomorphic Encryption):** Zero-knowledge proofs are integrated into the SecureSurvey system to enhance voter privacy by allowing voters to prove that each component of the ciphertext is the encryption of either 0 or 1, and that the sum of the three components is equal to 1, hence proving that the vote is correct.
  The zero-knowledge proof used in the system allows to prove that:

  Given a ciphertext y, $\exists$ a plaintext m and a random element r, such that

  $y = E(m)$ (using as randomness r) and $m \in M$,

  where M is the set of permitted plaintexts.
  Hence, given a ciphertext and a zero-knowledge related to it, it is possible to prove that the ciphertext corresponds to the encryption of a permitted plaintext.
  Using $M = \{0, 1\}$ for the single components, and $M = \{0, Imp\}$ for the sum (where Imp represents an impossible value) we achieve the desired requirements.
  The zero-knowledge proofs are generated while the vote is encrypted, and their verification is enforced as soon as the vote reaches the server, ensuring the integrity of the voting process.

- **Digital Signature (Schnorr):** The scheme is obtained by applying the Fiat-Shamir transformation to Schnorr's identification protocol [7].
  The digital signature is generated as soon as the vote is encrypted and its verification is enforced when the vote reaches the server, preventing unauthorized tampering and ensuring that the vote was casted by an eligible user.

The system employs a safe prime of length 3072 bits, that ensures a security strength of 128 bits in accordance with NIST recommendations for discrete logarithm-based schemes [1].

- **Hash Function (SHA-384):** The scheme is fully described in [6].
  The hash of the password (together with the salt) is computed as soon as the password reaches the server, and compared to the stored digest in order to prevent unauthorized access.
  The use of SHA-384 is motivated by the fact that it is the only algorithm implemented by the library which mitigates the risk of length extension attacks, since it utilizes a truncation of the Merkle–Damgård construction.
  The algorithm ensures a security strength of 192 bits in accordance with NIST recommendations for discrete logarithm-based schemes [1].
  The system employs a unique salt of 64 bits (8 bytes), exceeding the recommended 32-bit salt size [5], to reduce the risk of rainbow table attacks.

Security implementation details:

- **Change of randomness:** Using a different random value in the ElGamal encryption of each value in the vote, ensures that the three ciphertexts composing one vote are all distinct (even if two plaintext entries are always equal to 0). Not enforcing this policy would have led to a triple of ciphertexts composed of a couple of identical entries and a third distinct one in some kind of order, hence making it clear which vote was cast by simply observing the position of the distinct entry.
- **Strong password:** The GUI enforces strong password policies, requiring passwords to be at least 8 characters long and to contain at least one lowercase letter, one uppercase letter, one number, and one special character, aligning with standard recommendations for robust password security.
- **Password management:** User passwords are transmitted to the server in clear text, assuming a secure communication channel, and are then hashed upon receipt. Hence this does not compromise the overall security of the system. The server stores only the hashed passwords and their corresponding salts.
- **No Access to Voted Surveys:** Voters do not have direct access to the file of voted surveys, preventing the possibility of multiple voting if only one registration per user is ensured. This restriction enhances the integrity of the voting process by preventing the modification of the file by the user.
- **Client-side Encryption of Votes:** Votes are encrypted locally within the voter's client application before being transmitted to the server. The server does not have access to the private keys required for decrypting the votes, ensuring the confidentiality of the voting data.
- **Client-side Decryption of Votes:** Results of tallying are decrypted locally within the researcher's client application after being transmitted from the server. Again, the server

does not have access to the private keys required for decrypting the result, ensuring the confidentiality of the resulting data.

- **Key Storage:** Private keys for both the ElGamal encryption scheme and the Schnorr signature scheme are stored securely on the local machine, while public keys are stored on the server, acting solely as a key distribution center. Researchers possess the private keys for decrypting their respective surveys, and voters possess the private key for signing the votes, ensuring controlled access to cryptographic keys.
- **Limited Access to Encrypted Votes:** Only the server has access to the pool of encrypted votes, ensuring that neither users nor researchers can access partial encrypted results. This prevents researchers from computing partial encrypted results and maintains the confidentiality of voting data.
- **Storage of signatures and proofs:** The digital signatures and zero-knowledge proofs are not stored with the votes. The researchers cannot receive partial encrypted votes, hence they are not interested in verifying them one by one. The researchers would only be interested in other types of verifiability, that are not currently implemented in the application. Hence only the server has interest in utilizing these pieces of information, and does not store them.
- **Hardcoded public parameters:** Public parameters used in cryptographic operations are hardcoded. This ensures consistency and stability in cryptographic operations without compromising security.
- **Memory management:** The system actively invokes the garbage collector after setting the variables of private keys to null, ensuring that no references remain to the objects storing sensitive cryptographic information that can then be removed from the memory. Additionally, char arrays storing passwords are systematically cleared one position at a time as soon as the clear password is no longer required, further enhancing security by minimizing the exposure of sensitive data.

Threats and vulnerabilities:[16]

- **Brute Force Attacks on Passwords:** In the event of the credentials file `credentials.txt` being compromised, containing salts in clear text, the system is susceptible to brute force attacks on user passwords. Additionally, the absence of adaptive hashing mechanisms, due to alignment with the library used, may further expose passwords to brute force attacks.
- **Vulnerability of Signatures:** The library `unicrypt` only implements RSA and Schnorr digital signatures. The absence of ECDSA/EDDSA implementation leaves the system vulnerable to potential attacks targeting finite fields-based signature verification

[16] The identified security challenges and proposed solutions are comprehensively addressed in the subsection *Cryptographic and Security Limitations*, providing users with a clear understanding of the system's security posture and any potential areas for improvement or further development.

processes, such as index calculus attacks. This vulnerability compromises the authenticity of digital signatures.

- **Hashing implementation:** The library `unicrypt` only implements SHA-2 digital signatures. Hence the system does not employ SHA-3, the latest standard cryptographic hash function, that uses the sponge construction instead of the Merkle–Damgård one. This choice may reduce the confidence in the hashing procedure.

- **Quantum Threat:** The system does not exploit quantum-resistant algorithms, hence it is possibly susceptible to quantum attacks. Threats arise from algorithms like Shor's algorithm, which can compromise the security of cryptographic schemes relying on discrete logarithms.

- **Exposure of Secret Key Files:** Private keys are stored in clear text on local clients. If these files containing private keys are compromised, it could lead to the decryption of individual votes, posing a risk to the confidentiality of voting data.

- **Disclosure of Researcher's Surveys:** Surveys generated by each researcher are stored in clear, potentially revealing the nature of their research projects. This could pose confidentiality risks and may compromise the integrity of the research process.

- **Potential for Multiple Registrations:** The system allows for multiple registrations by the same voter, enabling individuals to exert disproportionate influence on survey outcomes through repeated voting.

- **Analysis of Voter Preferences:** Since the server has knowledge of which individuals participated in each survey, there exists the potential to analyze individuals' preferences irrespective of their actual votes. Additionally, managing this information securely during transit poses challenges.
  An alternative solution could involve storing the voted survey files locally, but this introduces the issue of preventing voter manipulation of files to avoid double voting. Moreover, this would only solve the concerns about the transit, since the server must always know who the survey comes from in order to verify the signature.

- **Server Integrity Concerns:** The server may delete votes, generate votes, or produce outcomes different from those indicated by the votes themselves by leveraging its access to public keys. This compromises the integrity of the server's operations, raising questions about the reliability of the voting process.

- **Confidentiality Failures for Skewed Results:** If the result of a survey is heavily skewed, such as a concentrated response of "Yes", and there is a leakage of the information about the surveys voted by each user, it becomes possible to deduce the votes of individual participants based on the overall outcome. This failure of confidentiality undermines the anonymity and privacy of participants' voting choices.

- **Coercion Vulnerability:** While the implementation of the "abstention option" ensures that individuals cannot be coerced into expressing a preference, the system lacks

protection against coercion aimed at forcing abstention. It's essential to note that this vulnerability may not be considered critical in the context of the specific use-case scenario for this system.

- **Storage of Files in Clear Text:** All files, including sensitive cryptographic materials and survey data, are stored in clear text, posing risks of unauthorized access and potential data breaches, even assuming that the server can be trusted.
- **Availability and Integrity Concerns:** The system does not consider availability, posing risks of inaccessible or deleted files without backup. Additionally, integrity of the files is not ensured, leaving files vulnerable to tampering and potential data corruption by an external player.
- **Side-channel attacks:** While `unicrypt` may provide features to mitigate side-channel attacks, the system could still be vulnerable to such attacks, potentially compromising the confidentiality of sensitive data.
- **Suboptimal Garbage Collection Management**: Although an attempt has been made to guarantee the immediate removal of private cryptographic keys from memory by actively invoking the garbage collector, the system's implementation may not guarantee the immediate removal of private cryptographic keys from memory once they are no longer required. This inadequacy presents a security risk as private keys may persist in memory, potentially susceptible to unauthorized access. Moreover, other sensitive information, such as private votes and information about surveys voted by the user, which were not adequately considered, may remain in memory, posing additional security risks if not properly managed.

Final considerations:

- **Uniformity within the library:** The decision to source all cryptographic functions from a single library was made to maintain consistency and readability within the system. However, this approach entails reliance on the correctness of the imported library, adherence to potentially outdated standards, and the risk of inadequate maintenance and updates. Despite these concerns, opting for a single library fosters internal uniformity, coherence, and facilitates potential code auditing efforts. By adhering strictly to the library's recommended practices [4], unnecessary casting between different libraries is avoided, mitigating the introduction of potential vulnerabilities. This centralized approach streamlines development and reduces the complexity associated with integrating multiple libraries, enhancing the overall security posture of the system.
- **Cryptographic Primitives Compliance:** The SecureSurvey system assumes that the cryptographic primitives provided by the external library comply with the recommendations set forth by the National Institute of Standards and Technology (NIST) and other relevant standards organizations. However, it's important to note that the implementation details and security assurances of these primitives have not been independently verified or audited.

Known Limitations

Protocol Implementation Limitations

- **No Key Distribution:** The system does not consider key distribution mechanisms, which can pose challenges in securely distributing cryptographic keys among users.
- **No Security in Transit:** The system lacks Transport Layer Security (TLS) implementation for secure communication between the server and client applications. Instead, a simulated setup is used with separate Java files.
- **No Key Management:** The system lacks proper key management mechanisms, with regard to storage and rotation of encryption keys.

Cryptographic and Security Limitations

- **No Protection of Secret Key Files:** The private keys are stored in clear. One potential solution could involve deriving keys from user passwords for encrypting key files.
- **No Availability/Integrity:** Implementing measures such as regular backups to address availability concerns and ensuring integrity through mechanisms such as hashing, access controls and audit trails can enhance the overall reliability and trustworthiness of the system.
- **Suboptimal Password Security:** Users must trust the server with their passwords, assuming that it will securely handle and protect this sensitive information.
- **No Password Expiration:** Implementing a mechanism that exhorts every user to change their password every few months would reduce the risk of a password breach by an attacker.
- **Limited Verifiability:**[17] While users must trust the server to maintain the integrity of the voting system, mechanisms should be in place to enable users to verify that the votes were correctly recorded and counted. This can be achieved by making previously voted surveys visible to users and providing mechanisms such as zero-knowledge proofs for verifying the accuracy of vote tallies.
- **File Disclosure:** Users must trust the server not to disclose or use the files of voted surveys, so no one utilizes the information on which surveys each person voted for statistical purposes, preserving the anonymity and confidentiality of users.
- **Limited Coercion Resistance:**[18] Techniques such as linkable group signatures can enhance coercion resistance by concealing individual voter identities while allowing for eligibility checking. This would also solve the concerns about skewed results.

[17] While the verifiability of votes is crucial for researchers to ensure the integrity of survey results, it may be of less significance for individual voters.
[18] Less relevant in this context since surveys do not involve sensitive or political decisions.

- **Limited Garbage Collection Management:** The effectiveness of invoking the garbage collector to remove sensitive information from memory requires further scrutiny and validation. To improve this aspect of memory management, one could implement the project using a different programming language such as, for example, C, which provides a high degree of control over memory, allowing direct manipulation of pointers and memory addresses.
- **No Protection Against Multiple Registrations:** The system does not prevent multiple registrations by the same user. One possible solution could involve replacing registration with email verification (OTP) and tying user accounts to their identities.
- **Separation between Name of Researcher and Surveys:** The system lacks a dissociation between the name of the researcher and the surveys created, potentially leaking the research being conducted. Implementing measures to anonymize the creator of surveys could enhance the overall privacy of the system.
- **No Resistance Against Side Channel Attacks:** While the cryptographic library used in the system may offer mechanisms to resist side-channel attacks, they should be explicitly verified or implemented.
- **Possible Update to Recent Standards:** The system may require updates to align with recent cryptographic standards and recommendations. The system currently utilizes the SHA-384 hashing algorithm instead of SHA-3 as advised by the NIST. Additionally, the system employs finite fields discrete logarithms instead of elliptic curve discrete logarithms, which avoid specific types of attacks such as index calculus method while requiring shorter keys. The implementation of ECC-ElGamal homomorphic scheme and ECDSA are strongly recommended.
- **No Quantum Resistance:** Migration to quantum-resistant algorithms is suggested in order to provide resistance against attacks from quantum computers.
- **No Adaptive Hashing:** Adaptive hashing algorithms, like `bcrypt`, incorporate multiple iterations of hashing to increase the computational cost of password hashing, thereby may significantly enhance password security by mitigating brute-force and dictionary attacks on hashed passwords.

System Functionality Limitations

- **No Multiple/Non-binary Votes:** The system currently does not support non-binary surveys or the selection of more than one option, thereby restricting the types of surveys that can be conducted.
- **Limited Platform Compatibility:** The system is only compatible with specific operating systems, browsers, or devices, limiting its accessibility and usability.
- **Limited Error Handling:** The system may lack comprehensive error handling mechanisms, potentially leading to user confusion or system instability in the event of errors or exceptions.

- **Absence of Auditing:** The system lacks auditing capabilities, which are essential for tracking and monitoring user activities, system events, and data access for security and compliance purposes.

- **Computational Overhead:** The system incurs computational overhead due to cryptographic operations, potentially impacting performance, especially in resource-constrained environments. It is imperative to optimize cryptographic algorithms and utilize efficient implementations to reduce computational costs

Instructions for Installation and Execution

System requirements and prerequisites:

- Operating System: Windows 11 (64-bit)
- Java Development Kit (JDK): Version 17 or higher

Installation Steps:

- Extract the SecureSurvey system files from the provided source file `SecureSurvey.zip` to the desired location on the local machine.

Execution Steps:

- Navigate to the directory where you extracted the SecureSurvey system files.
- Locate the `Application.jar` file and double click on it to execute the SecureSurvey system.[19]
- The SecureSurvey system should now start running.

Guidelines for the use of the software

Username: Must be 5 to 16 alphanumeric characters.
Password: Must be at least 8 characters long and contain at least one lowercase letter [a-z], one uppercase letter [A-Z], one number [0-9], and one special character [%=_?!@#$&*-]. It should be changed every six months.
Survey Length: Surveys must not exceed 5 rows in length due to GUI limitations.

[19] If the `Application.jar` file fails to execute or gives an error, follow the steps below to use the `run.bat` file as a fallback option.

1. Navigate to the same directory where the `Application.jar` file is located.
2. Right-click on the `run.bat` file and select "Edit" to open it with a text editor.
3. Replace "C:\Program Files (x86)\Java\jdk-17\" with the actual path to the downloaded JDK version on your system. For example, if the JDK is installed in C:\Java\jdk-21\, the line should be modified to "C:\Java\jdk-21\bin\javaw.exe".
4. Save the changes to the `run.bat` file and close the text editor.
5. Right-click on the `run.bat` file and select "Run as administrator" to ensure proper execution permissions.

Running the `run.bat` file explicitly sets the Java runtime environment to the specified JDK version, which may help resolve errors encountered when executing the `Application.jar` file directly.

Adhering to these guidelines ensures the security of your account and the smooth functionality of the survey creation process. Each user is responsible for safeguarding their credentials and should be vigilant against potential attacks.

Bibliography

[1] Barker, Elaine B. “Recommendation for Key Management: Part 1 - General.” *Special Publication (NIST SP) 800-57 Part 1 Revision 5*, 2020. *National Institute of Standards and Technology*, https://doi.org/10.6028/NIST.SP.800-57pt1r5. Accessed 3 February 2024.

[2] ElGamal, Taher. “A public key cryptosystem and a signature scheme based on discrete logarithms.” IEEE Transactions on Information Theory, vol. 31, no. 4, 1985, pp. 469-472, doi: 10.1109/TIT.1985.1057074. Accessed 3 February 2024.

[3] E-Voting Group at the Bern University of Applied Sciences. “UniCrypt.” *GitHub*, https://github.com/bfh-evg/unicrypt. Accessed 3 February 2024.

[4] E-Voting Group at the Bern University of Applied Sciences. “UniCrypt - Samples.” *GitHub*, https://github.com/bfh-evg/unicrypt-samples. Accessed 3 February 2024.

[5] Grassi, Paul A., et al. “Digital Identity Guidelines.” *NIST Special Publication 800-63B*, 2023. *National Institute of Standards and Technology*, https://pages.nist.gov/800-63-3/sp800-63b.html. Accessed 3 February 2024.

[6] National Institute of Standards and Technology. “Secure Hash Standard (SHS).” FIPS PUB 180-4, 2015, https://doi.org/10.6028/NIST.FIPS.180-4. Accessed 3 February 2024.

[7] Schnorr, Claus P. “Efficient Identification and Signatures for Smart Cards.” Advances in Cryptology — CRYPTO’ 89 Proceedings, 1990, https://doi.org/10.1007/0-387-34805-0_22. Accessed 3 February 2024.