Advanced Programming of Cryptographic Methods

Project Report

Shamir's Secret Sharing Secure Storage System

William Riccardo Duro
Anna Giacomello
Nicola Smaniotto

December 22, 2023

Contents

1 Description

The project aims to create a system for secure file storage. It should allow users to freely copy the encrypted data on multiple devices (thus ensuring availability), while having some assurance that no single compromised device can decipher it on its own.

2 Requirements

2.1 Functional Requirements

Using our program, a user should be able to:

- Encrypt a file, to protect it from an attacker
- Copy the encrypted file on all their devices, to have it always accessible
- Set a threshold of devices that are needed to recover the file
- New devices can be added later, from anywhere
- Outdated or compromised shares can be excluded from the system.

An attacker with only a partial knowledge of the secrets,in other words an amount of shares below the threshold, should not be able to decrypt the file.

2.2 Security Requirements

To encrypt the file we chose AES, for its speed and ubiquity. Moreover, through the JCA we can use any implementation available on the system, even hardware ones, without having to change our code.

The cipher works in Galois Counter Mode, which provides also authenticity to the archive, being tagged with a specific phrase. The key is a randomly generated 256 bit string. It is the highest key length the cipher allows, and should resist to attacks for many years in the future even with computational power increases. The key is split into shares with Shamir's Secret Sharing.

The security of the scheme comes from the fact that the polynomial is chosen randomly. This means that it is impossible to predict the values of the shares without knowing the secret. Even if the attacker knows the values of t shares, they still cannot reconstruct the secret without knowing the values of the other shares.

About Shamir's Secret Sharing:

- The security is based on the properties of polynomial interpolation.
- The security does not depend on any assumptions about the computational power of the attacker. This means that even if the attacker has access to the most powerful computers in the world, they still cannot break the scheme.
- The security of the system is not weakened by any partial information the attacker may have obtained. As long as it is not enough to recover the secret, it is as if the attacker knows nothing.

For this reasons Shamir's Secret Sharing is said to be unconditionally secure.

3 Technical Details

3.1 Architecture

The main components of the project have been split into different Java packages, which provide respectively:

- `ssssss`: The main entrypoint, with command line or graphical interface.
- `constants`: Constants for the program.
- `finitefield`: Arithmetical operations in a finite field.
- `polynomial`: Polynomial generation, evaluation, and interpolation.
- `point`: Helper package for polynomials.
- `secretsharing`: Methods and data structures to perform the secret sharing.

3.2 Implementation

3.2.1 Language choice

The code has been written in standard Java 8, with no dependency on external libraries. The main portions of the code have been split in different Java packages.

We have chosen Java because it is object-oriented, which allowed us to more easily group related pieces of code in the same class. Moreover, the presence of the JCA freed us from depending on a specific implementation of cryptographic primitives, which are instead provided by the standard library itself.

3.2.2 Shamir's Secret Sharing

The Shamir's Secret Sharing polynomials are defined on finite fields of the form $GF(2^w)$. The characteristic of the field is 2, which allows for efficient bitwise implementation of the operations. The elements in the field are represented as binary sequences of coefficients in variables of type `int`. The degree w can therefore be any natural number up to 32. Lower values create smaller fields and make the operations faster, but waste space inside the Java integers. Moreover, it also determines the total amount of shares that can be generated. We chose 32, as in this way we use the whole variables and the user can generate more than 4 billion shares, which are more than enough for all intended use cases.

3.3 Code Structure

The source code is stored in the `src/main/java` folder, as this is the default project structure used by Apache Maven[1]. The main entrypoint for the program is in `src/main/java/ssssss/App.java` and the rest of the code is divided between the packages.

[1] `https://maven.apache.org/`

- `ssssss.App`: The main entrypoint, it parses command line options or starts the GUI.
- `constants.Constants`: Constants for the program, set here and imported elsewhere.
- `finitefield.FiniteField`: Finite field class, its instance performs operations in a specific field.
- `polynomial.Polynomial`: Generates random polynomials for Shamir's secret sharing algorithm, and interpolates points.
- `point.Point`: Stores two integers as coordinates of a point in 2D space.
- `secretsharing.Share`: Data structure for the individual shares.
- `secretsharing.SecretSharing`: Main secret sharing class, splits byte arrays in shares and reconstructs the secrets.

When relevant, the source files have a corresponding JUnit 5[2] test class, stored in `src/test/java`.
For the variable naming we were inspired by Oracle's naming conventions[3], with the exception of packages since we didn't prepend an organization name.
All code classes, attributes and methods have been documented using both Javadoc syntax and normal comments.

We now present a detailed description of all the classes and the corresponding methods.

3.3.1 constants.Constants

This class has been created to provide a centralized source of global parameters for the whole project. It only contains some public attributes.

Attributes

- `DEGREE`: The degree of the polynomial used to define the finite field
- `POLYNOMIAL`: The polynomial itself

The polynomial is an element of $\mathbb{F}_2[x]$ encoded as a string of bits that lists its (binary) coefficients. In our code we chose $x^{32} + x^{15} + x^9 + x^7 + x^4 + x^3 + 1$, the Conway polynomial of degree 32.

We have hard-coded these parameters in the source itself because a user should not have any reason to change them. Choosing different parameters would most likely impact the performance of the program, but not its security. Still, setting a non-irreducible polynomial or the wrong degree would disrupt the correct operation of the software.

[2] `https://junit.org/junit5/`

[3] `https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html`

3.3.2 finitefield.FiniteField

Instances of define finite fields of characteristic 2 and of some degree, determined by the defining polynomial. They implement the main arithmetical operations that were needed in the program.

Elements of an abstract finite field are residue classes of polynomials, we identify each one by its representative of minimal degree. Its coefficients are stored inside Java `int` variables.

Attributes

- `DEFINING_POLYNOMIAL`**:** The defining polynomial, encoded as in Constants.
- `DEGREE`**:** The degree of the polynomial. It is determined automatically.

Constructor The field is defined only by the polynomial, the provided encoding is then bit shifted to determine its degree.

add, subtract Due to our representation of the elements of the field, addition and subtraction can be performed by bitwise XOR of the Java integers. Moreover, in characteristic 2 the operations are the same.

multiply First, a normal polynomial multiplication is performed in a Java `long` to avoid overflows. For this we simply check the bits in `a` and if they are set we add a multiple of `b` to the temporary variable.

After the multiplication is done we obtain its representative of minimal degree by zeroing out the top 32 bits. We can do this by shifting the defining polynomial and XORing it with the bits we want to remove. The resulting bitstring is cast to an `int`, to make it compatible with the other class methods.

divide To calculate the ratio of two elements, we multiply the first by the multiplicative inverse of the second. An exception is thrown if the second element cannot be inverted.

modInverse To invert an element of the field we use the extended Euclidean algorithm with the parameter `a` and the defining polynomial. If the element is 0, an exception is thrown, otherwise we already know that the GCD will be 1, as in a field any nonzero element is invertible.

In this way, by Bézout's identity, we obtain two coefficients s and t such that: $1 = \mathtt{a} \times s + p \times t = \mathtt{a} \times s \bmod p$. So the coefficient s is the multiplicative inverse of `a`. The other coefficient is not needed, so the code does not store it.

getRandom, getSecureRandom It is useful to generate elements of the field, for example as coefficients of our secret sharing polynomials. Since the code expects the leftmost unused bits of the `int` variables to be empty, a simple Java integer chosen at random may not be an element of the field.

For this reason these methods were added, which generate integer variables, but ensure that they are acceptable elements of the field by zeroing the top bits with a mask (that depends on the degree of the defining polynomial).

3.3.3 polynomial.Polynomial

This class represents polynomials over a finite field and defines various operations on polynomials.

Attributes

- `FINITEFIELD`: this attribute represents the finite field over which the polynomial operates. It is set through the constructor and is final, meaning it cannot be changed after the object is constructed.
- `COEFFICIENTS`: stores the coefficients of the polynomial as an array of integers. The coefficients are set through the constructor and are also final.

Constructor The class has a constructor that takes an array of coefficients and a `FiniteField` object as parameters. It initializes the `FINITEFIELD` and `COEFFICIENTS` fields.

getRandomPoly This method generates a random polynomial of a specified degree, with a given y-intercept, over the specified finite field.

evaluate This method evaluates the polynomial for a given x-value using the Horner's method.

interpolate `interpolate` uses Lagrange interpolation to generate a polynomial that passes through a given set of points. It uses a `HashSet<Point>` (`X_Set`) to check if the same point is passed multiple times. It then calculates the interpolated value for a given x using the Lagrange interpolation formula. The result is the sum of terms calculated for each unique point in the `X_Set`.

getYIntercept There are two versions of this method, as it is overloaded:

- `getYIntercept(Point[] points, FiniteField f)`: this is a static method that calculates the y-intercept of the polynomial using the `interpolate` method with $x = 0$.
- `getYIntercept()`: this method returns the y-intercept of the polynomial directly from its coefficients.

3.3.4 point.Point

The point class is used to store a pair of elements of the finite field (therefore, `ints`) as coordinates. We also implemented the methods needed to compare them.

Attributes

- `X:` The x coordinate of the point
- `Y:` The y coordinate of the point

equals The method to compare points, overrides the standard Java one provided by `Object`. Two points are the same if and only if both of their coordinates are the same.

hashCode Converts a point into a Java integer. This must be overridden to comply with the Java `hashCode` contract[4].
The equality methods have been written to use Java data structures like `HashSet`.

3.3.5 secretsharing.Share

The class Share has been created in order to represent the shares as their own entity.

Attributes The fields of this class are the following:

- `X:` this `int` value represents the x coordinate of the share
- `YS:` this array of `int` contains the evaluations of all the secret sharing polynomials at point x

While we could have used different x coordinates for each polynomial, we use the same one to obtain a more compact share. Therefore each share has exactly one x coordinate, while the number of y coordinates depends on the length of the secret and the degree of the finite field we're currently operating on.
Here we provide an insight on all the methods of the class.

Constructor The constructor of the class takes as inputs the `x` coordinate and the array of corresponding `y` coordinates. These parameters are assigned, respectively, to the fields `X` and `YS`.

parseShare This method parses a share that was previously converted to a String in order to reconstruct the corresponding `Share` object.
The method takes as inputs the following parameters:

- `s`: the String to parse in order to retrieve the share

The method raises `IllegalArgumentException` in case the length of the String is not of the adequate length to be parsed.
Since each hexadecimal digit can hold 4 bits, the number of digits `n_digits` necessary to hold a value of the finite field is calculated as the ceiling of the division between the degree of the field and 4. `IllegalArgumentException` is thrown if `n_digits` is 0 or if the length of the string is not a multiple of `n_digits`.

The String `s` is then iterated: the first *n_digits* are used to reconstruct the `X` coordinate of the share by using the `Integer.parseUnsignedInt` method.
The rest of the String is divided in blocks of `n_digits` block to reconstruct the elements of `YS` in an similar way as done for the `X` coordinate.
The method then returns the share generated with the reconstructed fields.

[4]`https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--`

toString This method is responsible for the conversion to String of the share. No arguments are taken as inputs.
The method calculates the number of hexadecimal digits necessary to hold a value of the finite field by computing the ceiling of the division between the degree of the field and 4. This is done because a hexadecimal digit is capable of representing 4 bits.
The `X` coordinate is then formatted by invoking `String.format("%0" + n_digits + "x", X)`. This String is then concatenated with the formatted representation of each element of `YS`.
The resulting String is then returned.

equals We decided to override the `equals` method of the `Object` class in order to be able to compare several `Share` objects.
The parameters taken as input are:

- `other`: the object to be compared

If one of the compared objects is `null`, or if the compared objects do not belong to the same class the returned value is `false.`
If the length of `YS` is different in the two objects, the returned value is `false`.
This method returns true if the `X` values are the same for both shares and if all the elements of `YS` are the same and are in the same order.

hashCode This method overrides the `hashCode` method of the `Object` class and returns the hash of the components of the share.

update This method is used to update a share and takes as input the following arguments:

- `u`: the update share to combine with the share
- `FIELD`: the finite field we are currently operating on

The method adds together (by using the `add` method of the `FiniteField` class) the corresponding pairs ot `y` coordinates in the `YS` arrays of the two shares.
Thanks to the update share `u` being generated by using polynomials with y intercept 0, the update operation does not change the value of the secret.

3.3.6 secretsharing.SecretSharing

This class manages the mechanism in charge of generating and updating the shares, as well as the retrieval of the secret starting from the shares.

Attributes The attributes of this class are:

- `FIELD`: the finite field on which we are working. It is stated as final since it never changes during the execution.

We are now going to provide an insight on all the methods of the class.

splitSecret `SplitSecret` is the method that is in charge of creating and returning the shares based on the secret passed as a parameter.
The parameters received by this function are the following:

- `secret`: this is the information we have to distribute among the shares. It is given as an array of bytes as this is usually a secret key (in our case of an AES encryption) the length of which exceeds by a considerable amount the 32 bits of an `int` - or even the 64 bits of a `long` - numeric type.
- `shareNumber`: this is the number of shares that the user wishes to generate
- `threshold`: this is the number of shares the user wants to be the minimum amount to retrieve the secret. This implies that $shareNumber \geq threshold$. Setting a threshold different from the number of shares is useful in the case the user distributes the shares among several devices but wishes to be able to retrieve the secret with a number of shares lower than the total number of shares.

The method performs the following operations.
First of all, the secret has to be split in pieces of the adequate amount of bytes. The number of bytes is calculated as `Constants.DEGREE / 8`. In this way we compute the number of bytes that can be passed to the field by performing the integer division between the degree of the finite field and 8 (number of bytes in a bit). It is worth reminding that the degree of the finite field is set to be at most 32, so an integer is always going to be capable to hold the information.

Then, the number of blocks in which the secret is going to be split is calculated, as the ceiling of the division between the length of the array of bytes representing the secret and the number of bytes per block, which has been just calculated.

The method then asks the finite field to generate a number of `x` coordinates equal to the amount of shares asked by the user. The shares are generated in a secure random way, and we make sure no duplicates are present by adding the coordinates to an `HashSet` and generating new `x` coordinates until the wanted size of the set is reached. There is also a control that ensures that no $x = 0$ is accepted, as this would expose the secret in the corresponding share.

The function then enters a `for` loop that has the purpose of splitting the secret in blocks of bytes coherently with what stated above.
The amount of bytes selected at each iteration is converted to an `int` by left-shifting each byte of a multiple of 8 positions, depending on the byte considered, and by progressively adding the new-found value with the previous ones.

For each portion of the secret, a random polynomial with the secret as the `y` intercept is generated. The polynomial is evaluated in the `x` values generated previously. The evaluation of the polynomial in all the `x` coordinates is going to constitute a column in a matrix. Since this process is going to be repeated for all the blocks of secret, the matrix will have dimensions $n \times m$, where

- n: number of `x` coordinates

- m number of blocks (or polynomials)

The choice to populate the matrix in this seemingly inefficient way (column major) was chosen so that at the end each row would correspond to the evaluation of all the polynomials in a certain `x` coordinate, that is, the `YS` field of a share.

It was then possible to compose the shares by taking, one by one, the `x` coordinates and the corresponding row of the matrix. The cost of this operation is linear in the number of rows, since we inspect the matrix by row and use each row as the array `YS` to construct the shares along with the corresponding `x` coordinate.

The shares are finally inserted in an array and returned.

retrieveSecret `RetrieveSecret` is the method in charge of receiving an array of shares and recomposing the secret. It takes as input the following parameters:

- `shares`: the array of shares to manipulate to retrieve the secret

The function begins with a `for` loop: for each element of `YS` of a given share, a `Point` object is constructed by taking the *i-th* element of `YS` and the `X` of the same share. All the points generated from all the *i-th* point of each share are put in an array; we can observe that this array contains all the points belonging to the same polynomial. The array is then given to the `interpolate` method of the `Polynomial class` in order to retrieve the y intercept of the polynomial, i.e. the *i-th* block of secret.
All the blocks of secret are put in order inside an array of `int`, called `secretIntArray`; the only thing left to do is now to rearrange the information retrieved from the polynomials into an array of bytes.

The way this is done is by performing the inverse process as the one used in 3.3.6 to divide the secret in blocks: the method calculates the amount of bytes necessary to hold the information in each element of `secretIntArray`. Then, each element of `secretIntArray` is progressively bit-shifted of 8 to the right in order to obtain the bytes composing the number. These bytes are inserted - in an order coherent with the one with which the bytes were given in 3.3.6 - into an array of bytes called `secretByteArray`, which is then returned.

guessThreshold `guessThreshold` is the method used to guess the threshold, meaning the minimum number of shares to retrieve the secret.
This method takes the following parameters as inputs:

- `shares`: the shares used to guess the threshold

The method follows this procedure.
The shares are used to retrieve the secret, which is memorized in a `byte` array. The number of shares used to retrieve the secret is stored in a variable called `t`. Then, the method retrieves the secret by resorting to an increasingly lower number of shares by progressively decreasing the value of `t`. If the secret stays the same as the one found at the previous iteration, `t` is decreased, otherwise it means that we have hit the number for which the secret is altered. The method returns $t + 1$, as this is the last value for which the secret was still coherent with the ones found in the previous iterations.

generateUpdates This method is used to generate updates for the desired shares. This feature is useful in case one or more shares have been compromise, thus it is safer to generate new shares that, combined with the old, still valid ones, create updated shares; in this way, the compromised shares are useless to combine with other shares in order to retrieve the secret; they are, therefore, excluded.

This function has been overloaded to take as an input different parameters:

- `generateUpdates(Share[] old)`
  - `old`: the old shares that need to be updated
- `generateUpdates(int[] xCoords, int nPolynomials, int threshold)`
  - `xCoords`: the `X` coordinates of the shares to update
  - `nPolynomials`: the number of polynomials contained in each share
  - `threshold`: the minimum number of shares needed to retrieve the secret, i.e. the degree of the polynomials plus 1.

Since the first calls the latter inside its body, we will start describing the first one and then move on to the description of the second one.

The method constructs the array of `x` coordinates by extracting them from the input shares.
It then finds the threshold by invoking `guessThreshold` (see 3.3.6).
Finally, it invokes the method `generateUpdates(int[] xCoords, int nPolynomials, int threshold)` by providing the variables extracted above as inputs.

We will now analyze the behavior of `generateUpdates(int[] xCoords, int nPolynomials, int threshold)` in greater detail.
The method generates `nPolynomial` polynomials by invoking `getRandomPoly` of the `Polynomial` class. The peculiarity of the generated polynomials is that they all have 0 as the y intercept. The reason for this will be clarified ir 3.3.5.
Each polynomial gets evaluated in all the `x` coordinates, thus obtaining the corresponding `y` coordinates. These coordinates are then inserted as a column of a matrix. This matrix has dimension $n \times m$, where:

- n: number of `x` coordinates
- m: number of polynomials

At the end of the loop, the matrix contains in each row the corresponding `y` coordinates to one `x` coordinate; these `y` coordinates are the result of the evaluations of all the polynomials in said `x`, which means that each row constitutes the `YS` component of the new shares.
The new shares are constructed and returned. It is worth noting that these shares still need to be combined with the old ones in order to obtain the updated ones.

updateShare This method is used to combine an old share with the update in order to obtain the updated share.
This method takes as inputs the following parameters:

- `share`: the share to update
- `update`: the new share to combine with the old one

This method invokes the method `update` of the `Share` class and returns the updated share.

generateMoreShares This method is responsible for adding new shares on top of existing ones.
The method has been overloaded to take different inputs:

- `generateMoreShares(Share[] shares, int amount)`: this method takes as inputs:
  - `shares`: the existing shares
  - `amount`: the number of new shares to generate
- `public static Share[] generateMoreShares(Share[] shares, int amount, int[] X_values)`: in addition to the inputs taken by the other version of the method, this one takes:
  - `X_values`: the `x` coordinates which cannot be used to create new shares

Apart from some checks on the `x` coordinates in the case of the second version of the method, the functioning of the two methods is pretty similar, so a single description will be provided for both.

The method inserts the `X` values of each of the existing shares in an `HashSet`; in the case of the second version of the method, the elements of `X_values` are also added to the set. This allows us to generate new `x` coordinates for the new shares without risking using duplicates values of `x`.

The method then generates a number of `amount` `x` coordinates by invoking `FIELD.getSecureRandom` and avoids collisions by adding them to the aforementioned `HashSet` and checking whether the value has actually been added.
Then, for each new `x` coordinate, the corresponding `YS` array is constructed: the *k-th* element is taken from the `YS` arrays of all the existing shares and used to create an array of points by pairing each one with the respective `X` coordinate. Then the *k-th* element of the `YS` array of the new share is computed by interpolating the points to retrieve the polynomial and by evaluating the polynomial in the new x coordinate (both these operations are performed by the `interpolate` method of the `Polynomial` class).

Once the share has been computed, it is added to the array of new shares, which is returned once all the new shares have been generated.

4 Security Considerations

Shamir's Secret Sharing offers strong protection against unauthorized access. There are certain security considerations to keep in mind when employing the scheme.

- Choosing an Appropriate Threshold. The threshold parameter, which determines the minimum number of shares required to reconstruct the secret, plays a crucial role in SSS security. A higher threshold provides greater security by making it more difficult for an adversary to obtain enough shares to compromise the secret. However, it also increases the risk of data loss if multiple shares are lost or compromised. It's essential to carefully select a threshold that balances security and resilience against data loss based on the specific application and risk tolerance.

- Protecting Share Recipients. The security of SSS relies on the confidentiality of individual shares. If any share is compromised, it could potentially be used to reconstruct the secret, even if not all shares are available. Therefore, it's critical to ensure that share recipients handle shares securely and avoid storing them on insecure devices or sharing them with unauthorized individuals.

- Secure Generation of Shares. The moment of greatest vulnerability is when the shares have just been generated. The user, in this moment, has all of them in the same machine. It is of utmost importance that they are rapidly and securely moved to the respective devices.

- Mitigating Human Error and Social Engineering. Human error and social engineering attacks can pose significant threats to SSSSSS security. Care must be taken to educate share recipients about secure handling of shares and prevent social engineering tactics that could manipulate individuals into revealing shares or compromising their security.

- Protecting Against Physical Tampering. In physical access scenarios, it's important to consider physical tampering threats. Shares stored on physical media, such as paper or USB drives, could be vulnerable to physical attacks that could reveal the secret. For this reason we implemented a way to update shares when some are deemed compromised.

- Monitoring Share Activity and Detecting Compromises. Implementing mechanisms to monitor share activity and detect potential compromises can provide proactive protection against threats. This could involve logging share access, tracking share movement, and detecting signs of unauthorized access or modification.

5 Known Limitations

Code quality We have written automated checks for all the main packages that compose our project. These tests provide some assurance that the code behaves as expected, but due to time constraints we could not check the exact code coverage. Moreover, these checks have been written by hand based on expected pitfalls or as regression tests for edge cases for which we noticed some kind of failure. They are therefore by no means comprehensive.

Project security While the mathematical security property of the scheme are known, our code is vulnerable to some side-channel attacks. For example when the decrypted bytes are written to a file, there may be recoverable traces of it even after the user "deletes" it, especially on modern COW file systems.

Data transmission The encrypted file can be transmitted over any channel without risk of it being compromised, however special care must be taken during the movement of shares. In this project we have only considered the protection of data at rest, in all our analysis we assume that the user adequately handles the shares during their transfer to other devices. Creating a secure channel for their transmission was outside our scope.

6 Instructions for Installation and Execution

Compilation The repository contains a `Makefile` that provides common utilities to easily compile and test the code. A user will probably receive the precompiled jar file, which can be executed as usual with `java -jar <file>`. Developers can build it themselves with `make jar`, which will create the output file `target/ssssss-${VERSION}.jar`.

The `Makefile` also provides a `check` target to execute automated tests, and a `docs` target that compiles the javadoc in `target/site/apidocs`. This report can itself be compiled with `make report`.

The project only uses libraries included in the standard Java installation, an end user does not need to install any dependency. Developers would need the Apache Maven build system, which will download JUnit 5 to run tests.

Usage These are the command line parameters accepted by the program:

- `encrypt` To encrypt a file and split the key in shares. Requires the file name and threshold as parameters, optionally a specific number of shares may be requested.
- `decrypt` To decrypt a file by recovering the secret key from the shares. Requires a file name, optionally a text file containing the shares. If the shares are not provided as a file, it asks for them from standard input.
- `update` To update some shares. Optionally accepts a text file containing the shares. If the shares are not provided as a file, it asks for them from standard input.

- `newShares` To generate new shares for the same secret. Requires an amount of shares to generate, optionally a text file containing some shares. If the shares are not provided as a file, it asks for them from standard input.

- `newSharesX` Same as the previous one, but also accepts an additional comma separated list of integers that cannot become x coordinates for new shares.

Alternatively, if it is started without parameters a simple GUI is started, which provides the same functions in a more intuitive way.