File Encoder Application v1.0 user handbook

# Index

[0. Index 1](#_Toc418431195)

[1. Introduction 1](#_Toc418431196)

[2. Quick start 2](#_Toc418431197)

[3. Application operation 3](#_Toc418431198)

[3.1. Main window 3](#_Toc418431199)

[3.2. File Menu 5](#_Toc418431200)

[3.3. Tools Menu 5](#_Toc418431201)

[3.4. About Menu 6](#_Toc418431202)

[3.5. List of encrypting configurations 6](#_Toc418431203)

[3.5.1. Basic concepts 6](#_Toc418431204)

[3.5.2. The form 7](#_Toc418431205)

[3.6. Encrypting configuration 8](#_Toc418431206)

[3.6.1. Encrypting configuration parameters 8](#_Toc418431207)

[3.7. Application settings 11](#_Toc418431208)

[4. Encrypting via command line 13](#_Toc418431209)

[4.1. Scripts for Windows 14](#_Toc418431210)

[4.2. Scripts for Linux and Mac 14](#_Toc418431211)

[4.3. Passing encrypting configuration parameters as arguments 15](#_Toc418431212)

[5. Method of encryption used 15](#_Toc418431213)

[5.1. Header of the encrypted file 16](#_Toc418431214)

[5.2. Algorithm used for encryption 16](#_Toc418431215)

[5.3. Pseudorandom generator used 17](#_Toc418431216)

# Introduction

File Encoder Application is an application for file encryption. It allows you to:

* Encrypt a file.
* Decrypt a file (previously encrypted with this application).
* Open an encrypted file (by previously decrypting it)

You can operate with encrypted and decrypted files (with .jfe extension) through the application GUI.

Moreover, the application engine can be invoked through the command line interface, obtaining in this way the same functionality for encrypting and decrypting files as with the GUI. The only disadvantage of using the command line interface is that you have to enter the password as a visible input parameter, so a nosey person could easily peek.

The application has been developed in Java, so it is portable to different platforms while it should remain fully compatible. That is, it should be possible to encrypt a file in a particular platform and decrypt it in another.

The application has been tested only with Intel processors, but running on different Operative Systems: Windows 8, Linux and OS-X. It should thus be compatible with any platform on which the Java Virtual Machine (JRE) is installed.

There exist several encrypting configuration parameters which, suitably adjusted, allow you to:

* increase the speed of the encrypting/decrypting processes
* increase the robustness of the encryption
* minimize the use of memory.

Unfortunately, not all factors can be optimized at once. There are trade-offs. For example, if you want to increase the robustness of the encryption by adjusting the value of the encrypting parameters, this goes against increasing speed or minimizing memory usage.

If you are not willing to learn the meaning of the parameters and how to use them, the application comes with pre-set configurations based on the size of the file to be encrypted, which allow bigger files to be encrypted with a faster configuration than that of smaller files.

Regardless of the parameters used for encryption, a good password helps to improve the robustness of the encrypting. Therefore, it is strongly suggested to use a very long password (it may even be a complete sentence) in order to make any attempt to break the encryption by brute force tougher. For example, a strong password should have capital and lowercase letters, numbers and should not just be composed of dictionary words. If we then consider that the password can be very long, for example between 30 and 40 characters, the encrypting system gets much stronger.

# Quick start

Before starting, we should notice that the application uses the microphone to collect random bytes; therefore it is necessary to have a microphone plugged to the computer for the application to work properly. Otherwise the encrypting would be much weaker.

To start using the application, take the following steps:

1. Open the File Encoder Application by double clicking on the application’s binary: FileEncoderApplication.jar
2. Change the interface language to English: (Menu: *Herramientas*->*Idioma*->*EN*)
3. Open a new file window (Menu: *File*->*New*)
4. *File*. Chose the file that you want to encrypt (anything but .jfe extension) or to decrypt (file name with .jfe extension). You can click on the *"..."* button to open the file picker dialogue.

Please, take into account that in order to be able to encrypt or decrypt a file, the target encrypted or decrypted file should not exist. If the file exists, the operation will fail because the application will not overwrite the existing file.

1. *Password*. Enter the password for the new encrypted file, or that used to decrypt a file.
2. *Repeat password*. Enter the password again. Both passwords must match to be allowed to encrypt or decrypt a file.

After completing these steps, you can use the *Encrypt*, *Decrypt* or *Open encrypted file* buttons.

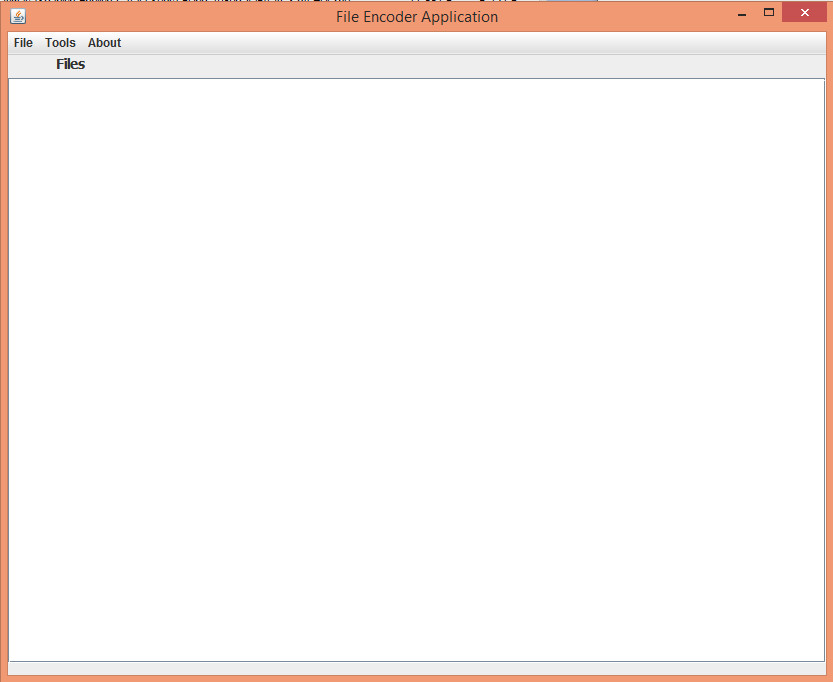
# Application operation

In this chapter we will see how to encrypt, decrypt and open encrypted files.

The most common options to perform these actions will be shown too.

## Main window

When you run the GUI, the following window shows up:

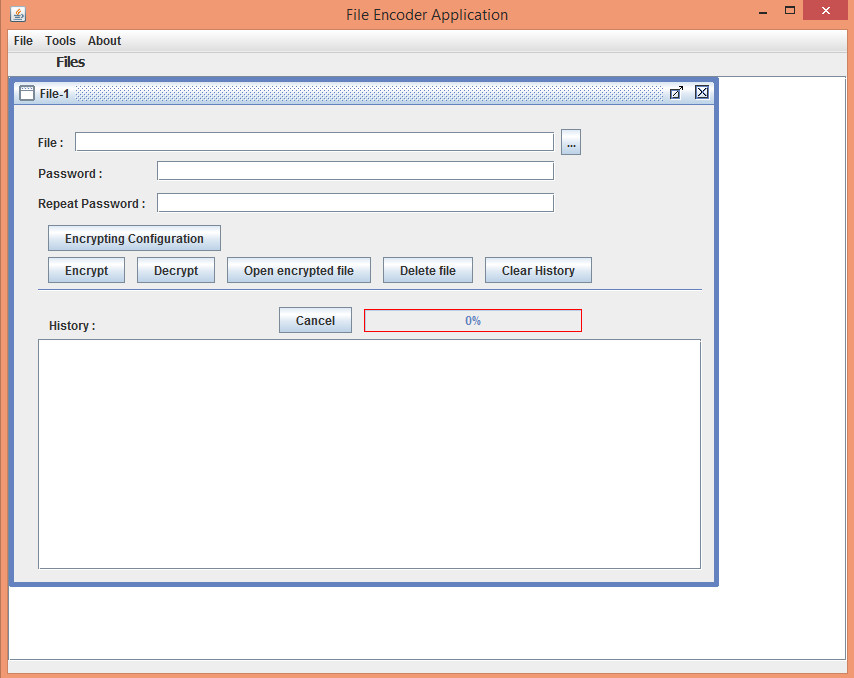


It consists of a main menu and a big main area where the MDI windows of the application will come up.

The area that holds the file windows has a pop-up menu which is shown by right clicking on it.

There is a single option in this menu: *New*. Clicking on this option will make a new File window appear. You can open as many file windows as you need.

A screenshot of the file window can be seen in the next picture:



There are some parameters which have to be provided before accessing the functionality.

* *File*. This text box contains the file name on which the actions will be done.

The file may be either non-encrypted (with any extension but .jfe) or encrypted (with extension .jfe).

These are some options to fill it up:

* + Clicking the *"..."* button. A dialog is then opened which allows you to browse the folders in the hard disks and select the file you want to work with.
  + Entering the file name through the keyboard (this can be useful when you have already got the file name written in the text field, and you want to switch between the encrypted and decrypted file, just by adding or removing the .jfe extension)
* *Password* and *Repeat password*. You must write the password to be used in these two text fields.

Both text fields must contain the exact same password. This can avoid some trouble, especially if you type the wrong password when encrypting a file. If there was not a second text field, you could mistakenly encrypt a file with an unknown password, and then it would be very difficult for you to decrypt the file.

Obviously, the password used for decrypting must be the same one used for encrypting.

Once the previous parameters have been filled up, all the functionality (encrypting, decrypting and open) can be used.

You can access the following functionalities from the file window by clicking on the buttons below:

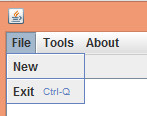
* *Encrypt*, to encrypt the file.
* *Decrypt, to* decrypt the file.
* *Open encrypted file*, to decrypts and open the file using the application pre-set in the system for that file type.
* Modify the *Encrypting configuration*. In this form you can modify the encrypting parameters. See section: 3.6-Encrypting configuration.
* *Delete file*, to delete the file.
* *Clear history*, to clear the history’s text area.
* *Cancel*, tocancel the encrypting or decrypting.

There is a text area (*History*) which shows the status and result of the actions taken on the files.

There is also a progress bar which shows advance on the ongoing action, and a tag comes up to its right when the encryption/decryption is completed that shows the time taken by the action in milliseconds.

## File Menu

The *File* menu looks like this:

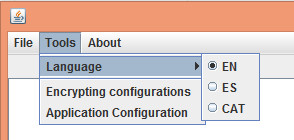


This menu has two options:

* *New*. When you click on this option, a new file window is created as seen in the previous section.
* *Exit*. It quits the application.

## Tools Menu

The *Tools* menu looks like this:



This menu has three options:

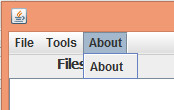
* *Language*. When you choose this submenu, the available languages are shown. You can then select the language of the application interface.

By default the next languages are available:

* *EN*, English
* *ES,* Spanish
* CAT, Catalan
* *Encrypting configurations*. A form comes up which allows you to modify the encrypting configuration to be used according to the size of the file to be encrypted. See point: 3.5- List of encrypting configurations
* *Application settings*. A form comes up which allows changing the application’s configuration parameters. See point: 3.7- Application settings

## About Menu

The *About* menu looks like this:



This menu has one only option:

* *About*. A window comes up with greetings, the application data and the contact data.

## List of encrypting configurations

This form allows creating, modifying and deleting the encrypting configurations, or to recover the default List of encrypting configurations.

### Basic concepts

As said in the introduction of this handbook, there are some parameters which affect the speed, the amount of memory used and the robustness of the encrypting. See point: 3.6.1-Encrypting configuration parameters

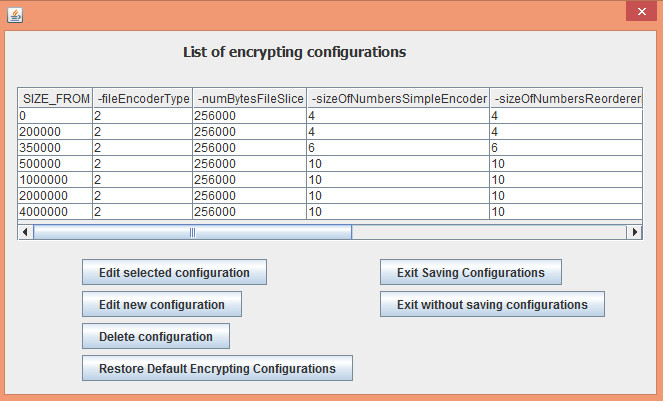
Since the encrypting parameters affect mainly the encrypting and decrypting speed, it seems reasonable to think that the encrypting configuration to be used could be calculated based on the actual file size.

To facilitate things to the end user, the File Encoder Application has predefined encrypting configurations enabled for the different ranges of file sizes, and it is able to choose automatically one encrypting configuration or another as per this criterion.

This form allows editing the encrypting configurations as defined for each range of file size.

### The form

The form looks like this:



In this form there is a table which summarizes the encrypting configurations, sorted from smallest to largest file size. Each row corresponds to one such encrypting configuration, associated to a particular file size range. You can select only a row at a time.

That encrypting configuration is applied to the file size ranging from the value of the column SIZE\_FROM (in bytes) in that row up to the value of the same column in the next row (corresponding to the next encrypting configuration), or up to an unlimited file size if no subsequent row exists.

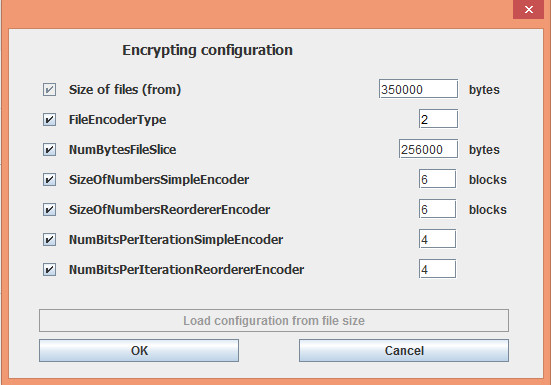
There are also some buttons which provide the form with its functionality:

* *Edit chosen configuration*. It opens the form which allows editing the encrypting configuration parameters of the selected row. See point: 3.6-Encrypting configuration
* *Edit new configuration*. It opens the form which allows creating a new encrypting configuration from scratch. It is mandatory to include the field SIZE\_FROM which will make it possible to define the file size range to which this encrypting configuration will apply.
* *Delete configuration*. It deletes the selected encrypting configuration from the table.
* *Restore Default Encrypting Configurations*. It undoes all the changes which have been made to the file size range encrypting configurations, bringing back the original list of encrypting configurations (the one which came with the application).
* *Save configurations & exit*. It exits the form after saving the changes made in the encrypting configurations.
* *Discard configurations & exit*. It exits the form discarding all changes made in the encrypting configurations’ file size range.

## Encrypting configuration

This form allows you to edit the parameters of a particular encrypting configuration, either to be applied directly on a file to be encrypted (from within a file window) or to edit the parameters for the encrypting configuration of a particular file size range.

The form looks like this:



This form can be reached either through a file window (section 3.1- Main window) or through the form for the *List of encrypting configurations* (section 3.5.2-The form).

There is a button to load the default encrypting configuration based on the file size. This will only be enabled if the form was reached through a file window. When accessed via the form for the *List of encrypting configurations*, this button is disabled.

Moreover, this button only works when it is enabled and provided there is a file configured in the file window. Otherwise, the application will do nothing when the button is clicked.

### Encrypting configuration parameters

The following parameters can be configured in this form:

* *File size (from)*. This is not a configuration parameter properly speaking and it is only enabled when the form has been reached through the form *List of encrypting configurations*.

This is because, in this case, it is used to create or edit an encrypting configuration which will be applied to a range of file sizes. *File size (from)* will indicate the beginning of the file size range in bytes. The end of the range to which the encrypting configuration of this form will be applied depends on the parameter *File size (from)* of the next row in the list of encrypting configurations.

If the form is reached via the file window, the encrypting configuration which you can edit in this form, once accepted, would be used in the next encryption of the file window, and depending on the encrypting priorities explained in section 3.7- Application settings.

* *FileEncoderType*. This parameter configures the kind of FileEncoder that will be used for encrypting. It is a numeric value that in version 1.0 may take the value either 1 or 2.
* 1 (deprecated). A FileEncoder class which uses a chaotic pseudorandom generator based on the BigDecimal Java class to make the calculations.

This class is used to do the encrypting operations at a very low level and it is possible that a change in its internal implementation could break backwards compatibility.

Therefore it was decided to create a new variable precision numerical class which made the application less dependent of the Java implementation.

* 2 (recommended). This is the preferred value for this version of the application. It will make the FileEncoder class use a chaotic pseudorandom generator based on a proprietary variable precision numerical class to make the calculations (BigLong).

This class (BigLong) allows working with variable precision numbers, but it is not generic. It is fully optimized to work with the numerical range used by the chaotic pseudorandom generator. It stores the number information in an array of long elements. Each long element in the array uses only the 31 lowest bits to store the number information. This is so for reasons of optimization.

When the term "block" is used in the following encrypting parameters applicable to this FileEncoderType, a long element of the array is meant (that is, the 31 lowest bits).

* *NumBytesFileSlice*. This encrypting configuration parameter contains the number of bytes in the slice used for encrypting.

For big files it is not feasible to load the whole file in memory and then make the calculations for encrypting, because the amount of memory needed is between 40 and 100 times greater than the size of the file being encrypted.

For example, if we are encrypting a file with a size of 100 MiB, the memory used to encrypt this file would be around 10 GiB. This amount of memory is not available in many current PCs.

This parameter enables the file content to be divided into slices. For each slice, the slice content will be read into a byte array first of all, the encrypting process will then be applied, and finally the encrypted content of the slice will be written into the encrypted file. To encrypt the next slice it will not be necessary to keep the previous slices in memory, therefore not so much memory will be required as if the whole file was to be encrypted at once.

Decreasing this parameter makes the application’s memory usage decrease. However, it would not be a good idea to decrease this parameter too much, because the encryption of the whole file could be broken if the encryption of the first slice is hacked.

So, to make it more difficult to break this encryption, this parameter should as high as possible, depending on the memory available for the application.

Still, it should be taken into account that if we set this parameter to a very high value, a machine fitted with less memory might not be able to decrypt this file.

* *SizeOfNumbersSimpleEncoder*. This encrypting configuration parameter contains the size of the numbers used by the chaotic pseudorandom generator.

See point: 5.2-Algorithm used for encryption

The *SizeOfNumbersSimpleEncoder* parameter refers to the size of numbers used by the chaotic pseudorandom generator used in the first step of encrypting (XOR).

The base unit of this parameter is slightly different for FileEncoderType 1 and 2.

* For FileEncoderType 1, the units of this parameter are bytes.
* For FileEncoderType 2, the units of this parameter are blocks of 31 bits. In this case it is equal to the length of the array of longs which store the number information. (Each long element of the array stores 31 bits of the number).
* *SizeOfNumbersReordererEncoder*. This encrypting configuration parameter contains the size of the numbers used by the chaotic pseudorandom generator in the second step of the encrypting (reordering).

The base unit of this parameter is slightly different for FileEncoderType 1 and 2.

* For FileEncoderType 1, the units of this parameter are bytes.
* For FileEncoderType 2, the units of this parameter are blocks of 31 bits. In this case it is equal to the length of the array of longs which store the number information. (Each long element of the array stores 31 bits of the number).
* *NumBitsPerIterationSimpleEncoder*. This encrypting configuration parameter contains the number of bits that the pseudorandom generator returns in every iteration.

This parameter applies to the pseudorandom generator used in the first step (XOR).

This parameter impacts sharply on the computation time of the encryption. It is suggested using a very small value of this parameter to encrypt small files, in order to make the encryption stronger. For big files, it would be better to use high values of this parameter, otherwise the encrypting time would take too long and it would be a hard wait every time you have to encrypt/decrypt the file.

The values accepted by FileEncoderType 1 and 2 are slightly different:

* For FileEncoderType 1, this parameter can accept the values 1, 2 or 4.

A value of 4 produces the fastest encryptions; however it is not recommended to use it with values of *SizeOfNumbersSimpleEncoder* below 16 bytes, since in that case we would obtain numbers between 0 and 15 in every iteration of the pseudorandom generator that would not be equiprobable.

For numbers starting at a size of 16 bytes, a value of 4 for this parameter could be used, although it would be better with numbers of size 32 bytes.

* For FileEncoderType 2, this parameter can accept the values 1, 2 and 4, or even multiples of 8 up to 64 (64 is the maximum value). Not every value is allowed.

For a value of 4 it is suggested not using a number size (as configured on *SizeOfNumbersSimpleEncoder*) lesser than 5 blocks, although it would be better for the number size to be greater than or equal to 8 blocks.

For values greater than 4 (multiples of 8 from 8 up to 64), the configured value has to meet this condition:

*NumBitsPerIterationSimpleEncoder*  <= 3 x *SizeOfNumbersSimpleEncoder*

* *NumBitsPerIterationReordererEncoder*. This encrypting configuration parameter contains the number of bits that the pseudorandom generator returns in every iteration. This parameter applies to the pseudorandom generator used in the second step (reordering).

This parameter impacts sharply on the computation time of the encryption. It is suggested using a very small value of this parameter to encrypt small files, in order to make the encryption stronger. For big files, it would be better to use high values of this parameter, otherwise the encrypting time would take too long and it would be a hard wait every time you have to encrypt/decrypt the file.

The values accepted by FileEncoderType 1 and 2 are slightly different:

* For FileEncoderType 1, this parameter can accept the values 1, 2 or 4.

A value of 4 produces the fastest encryptions; however it is not recommended to use it with values of *SizeOfNumbersReordererEncoder* below 16 bytes, since in that case we would obtain numbers between 0 and 15 in every iteration of the pseudorandom generator that would not be equiprobable.

For numbers starting at a size of 16 bytes, a value of 4 for this parameter could be used, although it would be better with numbers of size 32 bytes.

* For FileEncoderType 2, this parameter can accept the values 1, 2 and 4, or even multiples of 8 up to 64 (64 is the maximum value). Not every value is allowed.

For a value of 4 it is suggested not using a number size (as configured on *SizeOfNumbersReordererEncoder*) lesser than 5 blocks, although it would be better for the number size to be greater than or equal to 8 blocks.

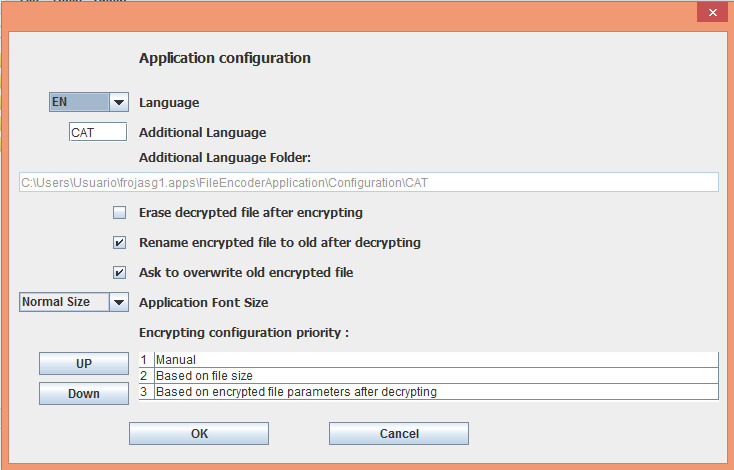
For values greater than 4 (multiples of 8 from 8 up to 64), the configured value has to meet this condition:

*NumBitsPerIterationReordererEncoder* <= 3 x *SizeOfNumbersReordererEncoder*

## Application settings

This form allows you to configure the application parameters.

The form looks like this:



The parameters which can be configured are the following:

* *Language*. It is the language of the interface texts of the application.

The languages that can be chosen are the following:

* *EN*. English
* *ES*. Spanish
* *Additional language*. By default it is CAT (Catalan)
* *Additional Language*. It is the additional language that appears when you want to change the language.

If you want to set up a language for which there are no interface texts in the application, you may change the additional language to your own language and translate the files to be found in the folder that will be created when you click the OK button.

The files to be translated will be copied into the folder specified in the *Additional Language Folder* text field.

The format of these text files is that of Java properties. If you are unfamiliar with this format, note that the file has a title and a variable number of configuration labels following next. Every label has its own value, similar to these:

# TITLE

# xxxxxxxxxx

LABEL1=text1

LABEL2=text2

...

The labels must not be modified and the texts should be modified depending on the translation in the language of choice.

If you create a language file different from the ones available in the application, you can send it to me (frojasg1@hotmail.com) and I will incorporate it to the next releases of the application.

* *Erase decrypted file after encrypting*. This check box allows you to set up whether after the encryption of a file you want to erase the decrypted file automatically.

The deletion is not just a standard one, the file is erased by writing "0"-bytes in every byte of the file. After that, a normal deletion is done.

This way it is impossible to recover the decrypted file unless you decrypt the encrypted file in the usual way, by using the password.

* *Rename encrypted file to .old after decrypting*. This check box allows you to set up whether you want to keep the encrypted file that has been just decrypted by renaming it as *file.old* after the decryption. This way you can have a backup copy if something unexpected happens.
* *Ask to overwrite .old encrypted file*. In case the previous option was checked, this option allows you to set up whether you want to be asked to confirm the overwriting of an encrypted .old file with the new encrypted file now being renamed to .old
* *Application Font Size*. This parameter allows you to change the size of the Fonts used in the application.

It has two options:

* *Normal size*. In this case the fonts used in the screen are the original ones.
* *Large size*. In this case the fonts used in the screen are 1.4 times bigger than the original ones.
* *Encrypting configuration priority*. The priority order of the encrypting configuration can be established in this table.

By default the options have this priority:

* (1) - *Manual*. It refers to an encrypting configuration which is applied manually; that is, by editing the parameters by hand in the corresponding form.

It seems logical to place it as the first option. If it were not so, it would be possible to make a modification of the encrypting configuration and to accept it, but this configuration might not be applied when you perform the encryption.

* (2) - *Based on file size*. This option refers to applying an encrypting configuration automatically from the list of encrypting configurations which are established depending on file size. See point: 3.5- List of encrypting configurations
* (3) - *Based on encrypting file parameters got after decrypting*. It means using the encrypting configuration from the last decrypted file.

The way to configure this priority order is by selecting on the table the row with the option whose priority you want to change, and click then the buttons *Up* or *Down* to increase or to decrease its priority.

# Encrypting via command line

Besides being able to encrypt and decrypt through the application GUI, it is possible to invoke the encrypting and decrypting functions from the command line.

This is not much advisable because to encrypt/decrypt this way, you’d have to write the password down along with the command. Somebody could find it out, for instance by looking into the command history or even by looking into the processes running in the system. It is also possible that the password would appear on the screen, so this is not very safe.

However, it is possible to do it.

There is a folder (\_scripts) containing some scripts created to facilitate the task.

There are two versions of the scripts. One of them is for Windows (\*.bat scripts) and another can be used either in Linux or Mac (\*.sh scripts).

## Scripts for Windows

The Windows version of the scripts is located in the folder:

*...\\_scripts\windows*

The available scripts are the following ones:

* *command.interface.FileEncoder.bat*
* *decodeFile.bat*
* *encodeFile.bat*
* *example.decodeFile.bat*
* *example.encodeFile.bat*

Invoke these scripts from the same folder in which the scripts are.

The first script contains the call to the Java application, collecting the arguments which are passed and adding them to the arguments for the Java application.

The second script contains the commands for decrypting an encrypted file. It takes as arguments the name of the encrypted file and the password.

The third script contains the commands for encrypting a file. It takes as arguments the file name and the password.

The fourth script is an example of how to invoke the script which does the encryption.

The fifth script is an example of how to invoke the script which does the decryption.

## Scripts for Linux and Mac

The Linux and Mac version of the scripts is located in the folder:

*.../\_scripts/Mac.or.Linux*

The scripts are initially compressed in the file *scripts.tar.gz*.

This file has to be decompressed to be able to work with the scripts. For this, type the next command from the folder where the file scripts.tar.gz is:

*tar -xvzf scripts.tar.gz*

Once the file is decompressed we can see the following files:

* *command.interface.FileEncoder.sh*
* *decodeFile.sh*
* *encodeFile.sh*
* *example.decodeFile.sh*
* *example.encodeFile.sh*

Invoke these scripts from the same folder in which the scripts are.

The first script contains the call to the Java application, collecting the arguments which are passed and adding them to the arguments for the Java application.

The second script contains the commands for decrypting an encrypted file. It takes as arguments the name of the encrypted file and the password.

The third script contains the commands for encrypting a file. It takes as arguments the file name and the password.

The fourth script is an example of how to invoke the script which does the encryption.

The fifth script is an example of how to invoke the script which does the decryption.

## Passing encrypting configuration parameters as arguments

The main script is this one:

*command.interface.FileEncoder*

It is used to invoke the Java application. The main parameters are the ones described in section 3.6.1-Encrypting configuration parameters

Here is a summary of the parameters:

* *-password* followed by a string with the password
* -*encodedFileName* followed by a string with the encoded file name
* *-decodedFileName* followed by a string with the decoded file name
* -*encode* or -*decode* to choose between encryption and decryption
* *-fileEncoderType* followed by the id of the FileEncoderType (1, deprecated; or 2) (only for encrypting)
* *-sizeOfNumbersSimpleEncoder* followed by the size of the numbers for step 1 (XOR) (only for encrypting)
* *-sizeOfNumbersReordererEncoder* followed by the size of the numbers for step 2 (reordering) (only for encrypting)
* *-numberOfBitsPerIterationSimpleEncoder* followed by the number of bits to be returned by the pseudorandom generator in every iteration for step 1 (XOR) (only for encrypting)
* *-numberOfBitsPerIterationReordererEncoder* followed by the number of bits to be returned by the pseudorandom generator in every iteration for step 2 (reordering) (only for encrypting)
* *-numBytesFileSlice* followed by the number of bytes in every slice. It is recommended to use a high value for this parameter. The memory used by the process will be bounded by roughly 100 hundred times the slice size (only for encrypting)
* *-useFileSizeForEncryptingParams.* If this parameter is present, all the parameters of the encryption (defined by the previous 5 parameters) will be calculated based on the size of the file to be encrypted, and these 5 previous parameters will be overwritten.

# Method of encryption used

The algorithms of encryption and decryption are very similar, and identical in processing time (symmetrical encryption).

The following section will show a summary of the encryption algorithm used in the application.

## Header of the encrypted file

A hash of the encryption password (SHA-256) is calculated returning a key of 256 bits which we will call key-1. The size of this key-1 is always the same.

If we used the pseudorandom generators with certain sets of configured parameters, maybe a key of 256 bits would not be optimal. The optimal size of the key to be used to initialize the pseudorandom generators for the encryption is calculated and we will call it size-2.

An amount of size-2 random bytes is calculated from the microphone source making up key-2, which will be used for the encryption. **It is important to have the microphone connected in order to obtain random bytes**. If the microphone is not connected, these random bytes are calculated via the SecureRandom class, with a seed of at least one nearly random byte. This is not that much, so it would be much more preferable to connect the microphone to obtain the random bytes.

Key-2 is used to encrypt the file.

Key-1 is used to encrypt key-2.

The encrypted key-2, the parameters of the encrypting configuration (unencrypted), and a hash (MD5, encrypted) of the original file will be written in the header of the encrypted file. These data will be used for decrypting the file. The MD5 hash will be used to check if the decrypting process has been successful.

## Algorithm used for encryption

This application performs the encryption of the files in slices; that is, by dividing the file into smaller parts. To encrypt each one of these slices, the next steps are followed:

* XOR stile encryption of the slice. In this stage of the encryption, the application generates pseudorandom bytes and does a XOR with the original data. This way, an encrypted slice is obtained whose bytes are quite random.

However, if we finished the encryption here, it would be relatively fast to try to break this encryption, provided some unencrypted bytes from the beginning of the file were known. This is the case for many file types which can be opened with an application (for example Word or Excel) because this file types generally have some kind of header, therefore making mere XOR encryption weaker.

It is for this reason that the application takes a second encryption step.

* Pseudorandom reordering of the bytes within the slice. In this stage of the encryption, the application generates pseudorandom positions (of bytes), where the bytes of the slice are to be moved to, thus re-sorting every byte in the slice.

Hence, to obtain the initial bytes of the decrypted file, it is necessary to decrypt a complete slice. So it is then more difficult to break the encryption because it is costlier in terms of computation time to decrypt a whole slice completely (whose size is configurable and 256.000 bytes by default) than to decrypt the first 100 or 1000 bytes and to compare them with a valid header of this type of file.

The previous steps (XOR and reordering) use chaotic pseudorandom generation to perform their tasks.

In the current implementation, these pseudorandom generators are characterized by the size of the numbers to be used in the generator and the number of pseudorandom bits to be returned in every iteration.

Depending on these two parameters, the pseudorandom generator calculates the optimum number of bytes that the key should have in order to initialize it.

The greater the number size is, the more optimal the key size to initialize it, and the longer the computation time for every iteration will be. That is to say, the greater the number size, the greater the time of encryption/decryption, and the stronger the encryption will be.

The smaller the number of bits returned in every iteration is, the greater the effect of number size over computation time will be.

The ideal configuration would be one that has a strong encryption and that is fast to perform. To achieve this goal, we could use a large number size (stronger) with a high number of bits per iteration (faster), combining it with big slice sizes (much stronger, but it could use too much memory). For small files you should use a very small number of bits to be returned per iteration (for instance: 1), and a large number size. The size of the slice should be at least the size of the file.

## Pseudorandom generator used

The pseudorandom generator used in this application is a pseudorandom generator based on chaotic sequences. It is a very simple one and its implementation is very easy.

The description of this pseudorandom generator was obtained from the book:

"Secuencias pseudoaleatorias para telecomunicaciones" Ediciones UPC (1996) Ernesto J. Forner Cruselles and José L. Melús Moreno

I thank them for them for having written such an interesting book.

Here follows the translation of a relevant part of the book:

"

*This generator, proposed by M. Romera, I. Jiménez y J. Negrillo [ROM90], is based also in the generation of pseudorandom sequences through the use of chaotic functions.*

*[ROM90] ROMERA, M.; JIMENEZ, I. NEGRILLO, J. Generación de Secuencias Cifrantes Mediante*

*Funciones Caóticas. I Reunión Española de Criptología. Mallorca, 1990.*

*...*

*The chaotic sequence of this generator (for Real numbers) is defined by:*

*Xo = 0*

*Xi+1 = Xi2 + K*

*If -2 <= K <= 0.25, then the numbers of the sequence are bounded: -2 <= Xi <= 2*

*Some sequences are cyclically convergent (periods 1, 2, 3 ...) and others are chaotic.*

*When K is very near to -2, as for instance K = -1.99999XXXXXXXXXXX, the series obtained from the Mandelbrot iteration are nearly always chaotic.*

*...*

*The real sequence obtained can be easily converted to a binary pseudorandom sequence, by taking the sign of every number of the series (the distribution is symmetric in respect to zero), or by applying a parity criterion to the digits of every number in the series.*

"

Based on this generator of chaotic sequences, the generator used in the application, has the following features:

-1 <= Xo  <= 1

K = -1.99999999xxxxxxxxxxxxxxxxxxxxxxxx

To initialize the pseudorandom generator we need a seed (a set of bytes) which will be used to initialize Xo and the variable part of K.

Both Xi and K must be numbers of variable precision allowing a very high precision.

If we used the Java basic number types, the seed range for the highest precision type (double) would be very limited and the encryption could be broken by brute force, based on this pseudorandom generator.

The options we have are:

* To use the Java class BigDecimal (used in FileEncoderType=1, deprecated).
* Develop a new class which can manage numbers of variable precision (used in FileEncoderType=2).

The pseudorandom bits derived from the chaotic sequence are obtained in these ways:

* When the number of bits produced in every iteration is 1, 2 or 4 (FileEncoderType 1 and 2):

The "1" bits of Xi are counted, and their remainder is calculated (2 if the number of bits per iteration was 1, 4 if it was 2 and 16 if it was 4).

This is the strongest version of the pseudorandom generator (most costly in computation time and least dependent on the particular value of Xi). The pseudorandom bits of every iteration of the chaotic sequence are obtained in this way.

* If 1 bit per iteration is returned, there is no limit on the number size because the possible values produced (0 and 1) are always equiprobable regardless of the number of bits used.
* If 2 bits per iteration are returned, it is better to use number sizes equal to or greater than 16 bits. Because if using, for instance, numbers of size 1 byte (8 bits) when calculating the remainder of the number of "1" bits divided into 4 (resulting 0, 1, 2 or 3), these values would not be equiprobable. But if we use numbers of size 2 bytes (16 bits) the resulting number (0, 1, 2 or 3) would be almost equiprobable, so it could be used.
* If 4 bits per iteration are returned, it is better to use number sizes equal to or greater than 32 bytes (256 bits)

In the following table you can see the likelihood of every possible value depending on the number size used when returning 4 bits per iteration:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Value | numbers of 16 bits | numbers of 32 bits | numbers of 64 bits | numbers of 128 bits | numbers of 192 bits | numbers of 256 bits |
| 0 | 0,00305176% | 13,99499346% | 9,93997199% | 7,29371224% | 6,55137753% | 6,33706399% |
| 1 | 0,02441406% | 13,17175925% | 9,64196694% | 7,21415681% | 6,52843586% | 6,33043664% |
| 2 | 0,18310547% | 10,97647697% | 8,80338313% | 6,98766512% | 6,46310389% | 6,31156353% |
| 3 | 0,85449219% | 8,08803737% | 7,57614083% | 6,64887061% | 6,36532878% | 6,28331793% |
| 4 | 2,77709961% | 5,25798649% | 6,17124246% | 6,24950378% | 6,24999687% | 6,24999998% |
| 5 | 6,66503906% | 3,00877392% | 4,81247919% | 5,85042763% | 6,13466680% | 6,21668205% |
| 6 | 12,21923828% | 1,52314156% | 3,69661687% | 5,51233488% | 6,03689611% | 6,18843647% |
| 7 | 17,45605469% | 0,73142946% | 2,96941304% | 5,28654495% | 5,97156857% | 6,16956339% |
| 8 | 19,63806152% | 0,48979651% | 2,71754309% | 5,20728020% | 5,94862872% | 6,16293605% |
| 9 | 17,45605469% | 0,73142946% | 2,96941304% | 5,28654495% | 5,97156857% | 6,16956339% |
| 10 | 12,21923828% | 1,52314156% | 3,69661687% | 5,51233488% | 6,03689611% | 6,18843647% |
| 11 | 6,66503906% | 3,00877392% | 4,81247919% | 5,85042763% | 6,13466680% | 6,21668205% |
| 12 | 2,77709961% | 5,25798649% | 6,17124246% | 6,24950378% | 6,24999687% | 6,24999998% |
| 13 | 0,85449219% | 8,08803737% | 7,57614083% | 6,64887061% | 6,36532878% | 6,28331793% |
| 14 | 0,18310547% | 10,97647697% | 8,80338313% | 6,98766512% | 6,46310389% | 6,31156353% |
| 15 | 0,02441406% | 13,17175925% | 9,64196694% | 7,21415681% | 6,52843586% | 6,33043664% |

As shown in this table, if the size of the numbers is small, the likelihood of the values for the bits resulting from the iteration will not be equiprobable enough to be used for the encrypting.

* When the number of bits produced per iteration is a multiple of 8 between 8 and 64 (both included) (FileEncoderType 2):

In this case, all the bytes of Xi are to be used as if they were pseudorandom numbers. This is not exact but it runs faster. Using this method, it is very fast to obtain the bits returned from the iteration. It is enough to take the bytes directly from Xi. Besides, this method allows taking a large number of bits per iteration.

The following condition has to be fulfilled by the number of bits returned per iteration and the number size measured in blocks (blocks of 31 bits; every block is stored in a long which is stored in array of longs of variable length):

NumberOfBitsPerIteration / 8 <= 2 x NumberOfBlocksPerNumber

At most 3 bytes of each block are taken.