File Encoder Application v1.5 User Handbook

# Index

[0. Index 1](#_Toc63982685)

[1. Introduction 2](#_Toc63982686)

[2. Quick start 4](#_Toc63982687)

[2.1. Requirements 4](#_Toc63982688)

[2.2. Quick start 5](#_Toc63982689)

[3. User interface 6](#_Toc63982690)

[3.1. Main window 6](#_Toc63982691)

[3.2. File Menu 8](#_Toc63982692)

[3.3. Tools Menu 9](#_Toc63982693)

[3.4. View Menu 9](#_Toc63982694)

[3.5. Help Menu 9](#_Toc63982695)

[3.6. List of encrypting configurations 11](#_Toc63982696)

[3.6.1. Basic concepts 11](#_Toc63982697)

[3.6.2. The form 11](#_Toc63982698)

[3.7. Encrypting configuration 13](#_Toc63982699)

[3.7.1. Encrypting configuration parameters 13](#_Toc63982700)

[3.8. Application Settings 18](#_Toc63982701)

[3.8.1. Application configuration parameters for encryption 18](#_Toc63982702)

[3.8.2. Application configuration parameters for language 20](#_Toc63982703)

[3.8.3. Application configuration parameters for view 21](#_Toc63982704)

[3.9. Deletion of files 22](#_Toc63982705)

[3.10. About… 24](#_Toc63982706)

[3.11. License 25](#_Toc63982707)

[4. Encrypting via Command Line 26](#_Toc63982708)

[4.1. Scripts for Windows 26](#_Toc63982709)

[4.2. Scripts for Linux and Mac 26](#_Toc63982710)

[4.3. Passing Encrypting Configuration Parameters As Arguments 27](#_Toc63982711)

[5. Method of encryption used 30](#_Toc63982712)

[5.1. Header of the encrypted file 30](#_Toc63982713)

[5.2. Algorithm used for encryption 30](#_Toc63982714)

[5.3. Pseudorandom generator used 32](#_Toc63982715)

[5.4. Use of execution threads (parallelism) 34](#_Toc63982716)

[6. Types of encryption 37](#_Toc63982717)

[6.1. FileEncoderType-1 37](#_Toc63982718)

[6.2. FileEncoderType-2 37](#_Toc63982719)

[6.3. FileEncoderType-3 38](#_Toc63982720)

[6.4. FileEncoderType-4 39](#_Toc63982721)

[6.5. FileEncoderType-5 39](#_Toc63982722)

[6.6. FileEncoderType-6 40](#_Toc63982723)

[6.7. FileEncoderType-7 41](#_Toc63982724)

[6.8. FileEncoderType-8 42](#_Toc63982725)

# Introduction

 The File Encoder Application is an application for file encryption that allows you to do the following:

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

You can deal with encrypted and decrypted files (with .jfe file extension) within the GUI. .

In addition, the application engine can be invoked through the command line interface thereby obtaining the same functionality in terms of encrypting and decrypting files as the GUI.

This is a Java based application; therefore it is portable across multiple platforms and should remain fully compatible. In other words, it should be possible to encrypt a file in one platform and decrypting it within another.

This application has only been tested using Intel processors running on the following operating systems: Windows 8, Linux and OS-X. But it should also be compatible with any platform on which Java Virtual Machine (JRE) is installed.

Several encrypting configuration parameters exist which, when suitably adjusted, allows you to do the following:

* Increase encryption/decryption speed.
* Increase the robustness of the encryption.
* Minimize memory usage..

Unfortunately, not all of these factors can be optimized at once. There are trade-offs.

For example, increasing the robustness of the encryption by adjusting the value of the encrypting parameters will negatively affect the speed or your ability to minimize memory usage.

If you are not interested in learning about the parameters, and how to use them the application includes pre-set configurations based on the size of the file to be encrypted. This allows larger files to be encrypted at a faster configuration than smaller files.

Regardless of the parameters used for encryption, a strong password helps to improve its robustness. . Therefore, a password consisting of several characters, even a complete sentence, is highly recommended in order to make any attempt to break the encryption nearly impossible. For example, a strong password should consist of random capital and lowercase letters, numbers, and be between 30 to 40 characters long.

**New features in Version 1.2**

 This new version of the application came about due to a bug found in fileEncoderType 2. The bug was discovered while encryption texts on large files were being executed.

FileEncoderType has been created to replace fileEncoderType 2which fixes the bug found and increases the range of the numbers up to 256.

In addition,two new fileEncoderTypes (FileEncoderTypes 5 and 6) have been created and adapted for the encryption of large files. .

FileEncoderTypes 5 and 6 are much more difficult to break than their predecessors, but again there is a trade-off.

FileEncoderType 6 simplifies the encryption process which, in turn, renders it a little more susceptible to hacking than fileEncoderType 5 for large files. On the other hand, FileEncoderType 6 increases encryption speed considerably.

Using the new fileEncodertype (6)to encrypt small to medium-sized (less than 10 MB) is not recommended.

This application successfully passed a number of encryption tests for large files over 50 GB.

**New features in Version 1.3**

In this new version of the application, new encryption types have been created using several execution threads (parallelism), ideal for larger files.

* fileEncoderType 7.

This encryption type works in two ways:

* + Input and output files located on the same disk:
		- One dedicated thread is used to access the disk (the same thread is used for reading and writing).
		- One or more threads are used for encryption.
	+ Input and output files located on different disks:
		- One dedicated thread for reading the input file.
		- One dedicated thread to write the output file.
		- One or more threads are used for encryption.

Depending on the processing capability of the machine and the speed of the input and output disks, improvement in encryption speed will vary between devices.

In tests carried out (using the optimum number of threads and all system resources exclusively utilized by the application) the encryption speed increased by a factor of 2 for large files.

In addition, if this encryption type is used, and the input and output files are on different disks, speed will be subsequently increased by a factor of more than 3.

For more details on this new encryption type see the chart below.

***6.7-FileEncoderType-7***

* + Benchmark for encryption times of large files (may vary in different systems):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size | FileEncoderType | -No. of Threads | Different disks | Duration of encryption (s) | Duration of decryption (s) |
| 27,2 GB | 6 | (1) | no | 1631 | 1512 |
| 27,2 GB | 7 (new) | 3 | no | 704 | 886 |
| 27,2 GB | 7 (new) | 6 | yes | 455 | 669 |

* fileEncoderType 8.

This encryption type is like the 7, but doing the encryption dividing the encrypted file into volumes of a specific size.

For more information review***: 6.8-FileEncoderType-8***

**New features in Version 1.4**

A bad effect, which was produced in machines with high dpi (high density of pixels per inch) with Java-9 and above.

Java-9 adds an automatic treatment for screens having a high density of pixels per inch.

This new treatment was behaving badly together with the functionality of zooming that my applications have.

To avoid this bad effect, that new Java-9 functionality has been disabled in my applications.

**New features in Version 1.5**

Platform v1.4 is used, which adds the search for a new version during the startup of the application.

# Quick start

## Requirements

To execute the application is necessary to have the java version 8 or later environment installed (Java Runtime Environment).

With this, the file .jar in folder ./binary/ can be opened.

## Quick start

Before starting, it should be noted that the application uses a microphone to collect random bytes. Therefore, in order for the application to work properly it is necessary to have the microphone plugged into the computer.

In this version of the application, this part has been improved, so if the microphone is disconnected, random numbers are generated using the generator for seeds that Java provides.

To start the application, take the following steps:

1. Open the File Encoder Application by double clicking on the application’s binary file : jfe.v1.3.jar
2. Change the user 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 (any filename) or to decrypt (any file name with the .jfe extension). You can click on the *"..."* button to open the file picker dialog.

Please note that in order to encrypt or decrypt a file, the target file should not exist. If this file exists, the operation will fail because the application will not overwrite the existing file.

1. *Enter a password.* Enter the password created for the newly encrypted file, or the password used to decrypt the file.
2. *Re-type password*. Enter the password again. The first and second passwords must match in order to be allowed to encrypt or decrypt a file.

After completing these steps, the *Encrypt*, *Decrypt* or *Open encrypted file* buttons will be activated

# User interface

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

The most common options for performing these actions will be detailed as well.

##  Main window

When you run the GUI application, the following window will pop up:



This window consists of a main menu and a large area where the MDI windows of the application will be displayed.

Click on the *New* option within the File menu to create a new file window. You can open as many file windows as needed.

A screenshot of the file window is displayed below:



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

* *Decrypted file name*. This text box contains the decrypted file name.

*Encrypted file name.* This text box contains the decrypted file name (with the .jfe extension).

There are some options on what to enter:

* + Clicking on the *"..."* button will open a dialog box which will allow you to browse folders within in the hard drives) and select the file you wish to access.
	+ Typing in the file name (this is particularly useful when the file name is already visible within the text field, and you want to edit the name of the file)
* *Enter and re-type password*. You must enter the password to be used within both text fields.

Using the same password for both text fields will help to prevent any issues, especially if you enter the wrong password when encrypting a file. If the second text field didn´t exist, you could mistakenly encrypt the file with an unfamiliar password, making it very difficult for you to retrieve the right password and decrypt the file.

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

* Checkbox: “Input and output are on different devices”. Configures to use the parameter of the encryption “-encDecOnDifferentDrives” which enables the application to use exclusive threads for reading and writing for fileEncoderTypes 7 and 8.
* Checkbox: “auto” commands the application to automatically calculate the value of the previous checkbox (the application takes different drives into account when input and output files are located on different partitions).

Once previous parameters are filled, you can encrypt, decrypt, and open encrypted files.

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 decrypt and open the file using the pre-set system application for that file type.
* Modify the *Encrypting configuration*: Can be used to modify encrypting parameters. See section: ***3.7-Encrypting configuration*** .
* *Delete file<:Next to* the text components that contain the names of encrypted and decrypted files, there are icons featuring a cross. These buttons allow files that appear in the matching text box to be deleted. See point:

***3.9-Deletion of files***

* *Clear history*: - To clear the history within the text area.

*Cancel*: *To* cancel the encryption/decryption processes or delete files.

The text area located within (*History*) shows the statuses and results of actions taken within the files.

There is also a progress bar which details the status of any on-going action. At the end of the encryption/decryption process, a tag displaying the total time (in milliseconds) taken will be displayed on the right

## File Menu

The *File* menu looks like this:



This file menu has two options:

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

## Tools Menu

The *Tools* menu looks like this:



This menu has three options:

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

By default the languages available are:

* *EN*, English
* *ES,* Spanish
* CAT, Catalan
* *Encrypting configurations*. A form pops up which allows you to modify the encrypting configuration to be used according to the size of the file to be encrypted. See point: ***3.6- List of encrypting configurations***
* *Application settings*. Another form pops up which allows you to change the application’s configuration parameters. See point: ***3.8- Application Settings***

## View Menu

The *View* menu looks like this:



This menu only has one option:

* *Zoom* allows you to choose the window size within the GUI. The percentages displayed indicate the size of components in comparison to the default size (100%).

##  Help Menu

The *Help* menu looks like this:



This menu has two options:

* *Help*. Opens this help menu.
* *Search for a new version*. This option searches the server to see if there is a new version of the application.
* *What’s new*. This option displays the document the explains what is new in this version.
* *License*. Displays the license that was accepted the first time the application was executed. **See section: *3.11-License***
* *About*… This windows display greeting, information about the application, and my contact details. See point: ***3.10-About…***

##  List of encrypting configurations

This form allows configurations for encrypting to be created, modified, and deleted. It also allows you to reset to default encrypting configurations.

### Basic concepts

As stated in the introduction of this handbook, there are some parameters which affect the speed, the amount of memory used, and the robustness of the encryption. . See point: ***3.7.1-Encrypting configuration parameters***

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

To order to facilitate this process for the end user, the File Encoder Application has enabled predefined encrypting configurations based on file size. It is also able to automatically choose one encrypting configuration over another based on this criterion.

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

### The form

The form looks like this:



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

The encryption configuration would apply to the range of file sizes from the SIZE\_FROM field of the byte encryption configuration itself, to the SIZE\_FROM field of the next encryption configuration (the next row), or unlimited if there is no subsequent

Below are buttons that provide the form with its functionality:

* *Revert button*. This button reverses all changes that have been made to the encrypting configurations based on file size range, and it restores the default encrypting configuration list.
* *Accept button (apply)*. This button allows you to exit the form after saving changes made to the encrypting configurations.
* *Cancel button*. This button allows you to exit the form and discard all changes made to the encrypting configurations based on file size range.

In addition to the options mentioned above when you right click the table, the following pop-up menu will appear allowing you to select individual configurations on which to carry out tasks.



* *Edit chosen configuration*. This opens a form which allows editing of the encrypting configuration parameters of the selected row. See point: ***3.7-Encrypting configuration***
* *Edit new configuration*. This opens a form which allows for the creation of a new encrypting configuration from scratch. It is mandatory to include the field SIZE\_FROM. This makes it possible to define the file size range to which this encrypting configuration will apply.
* *Delete configuration*. This allows deletion of any encrypting configuration selected from the table.

## Encrypting configuration

This form allows you to edit the parameters of a particular encrypting configuration, either to be applied directly to the file being 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 either be reached through a file window (section ***3.1 - Main window***

) or through the *List of encrypting configurations* form (section ***3.6.2-The form***).

There is a button to load the default encrypting configuration based on file size. It is only enabled if the form is accessed via a file window. This button is disabled when accessed via *List of encrypting configurations* form.

Furthermore, r, this button only works when it is enabled and there is a file configured in the file window. Otherwise, the application will not run 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 in the proper sense of the word, and it is only enabled when 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 top of the file size range in bytes. The bottom of the range, to which the encrypting configuration of this form will be applied, depends on the parameter *File size (from)* of the following row within the list of encrypting configurations.

If the form is accessed via the file window the encrypting configuration, which you can edit in this form once accepted, would be used in the following encryption of the file window depending on the encrypting priorities explained in section ***3.8- Application Settings***.

* *FileEncoderType*. This parameter configures the kind of FileEncoder that will be used for encrypting. It is a numeric value that within version 1.0 may have a value of1 or 2.
* 1 (NOT RECOMMENDED). 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 encrypting operations at a very low level, and it is possible that a change in its internal implementation could break backward compatibility.

As a result, it was decided to create a new variable precision numerical class which would make the application less dependent on Java implementation.

* 2 (DEPRECATED). This fileEncoderType has a bug which renders it deprecated. It results in the FileEncoder class using a chaotic pseudorandom generator based on a proprietary variable precision numerical class to make calculations (BigLong).

This class (BigLong) allows the user to work with variable precision numbers, but it is not generic, and 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 only uses the 31lowest bits to store number information (for optimization reasons).

When the term "block" is used in the following encrypting parameters applicable to this FileEncoderType, it refers to a long element of the array (in other words, the 31 lowest bits)

This type of encoder uses a XOR encryption combined with a pseudorandom reordering of bytes of every one of the slices in which the file to be encrypted is divided.

* 3 (DEPRECATED). This FileEncoderType is identical to the previous one mentioned above with the difference being y that the size of the numbers increases its possible range. This fileEncoderType allows users to work with sizes up to 256.

This FileEncoderType has a bug which also renders it deprecated like the previously mentioned FileEncoderType above.

* 4. This fileEncodertype is identical to the previous one mentioned but the difference is that the bug has been fixed.

5. This fileEncoderType is identical to the previous one mentioned, but in addition to the XOR encryption being combined with the pseudorandom reordering of bytes (as observed in all previous FileEncoderType, it adds a particular feature. All slices of the file to be encrypted are re-ordered. This makes the encryption much less susceptible to being hacked. This type of encryption is particularly good for large files.

* 6. (NOT RECOMMENDED) This fileEncoderType is identical to the previously mentioned one, but the pseudorandom reordering of bytes has been removed in order to increase encryption/decryption speed.

As a consequence this fileEncoderType is much faster than fileEncoderType 5, and should be much more difficult to be hacked than fileEncoder4 for large files.

This fileEncoderType is not recommended for small to medium-sized files of less than 10 MB.

Therefore it is more appropriate to use fileEncoderType 7 as it can be notably faster.

* 7. This fileEncoderType is very similar to the previous one mentioned, but with the added feature of using several execution threads.

It is highly recommended for large files (greater than 100 MB where you will be able to appreciate the difference in speed.

* 8. This fileEncoderType is identical to the previous one mentioned but with the added feature of being about to split the encrypted file into volumes smaller than a configurable amount of bytes.
* *NumBytesFileSlice*. This encrypting configuration parameter contains the number of bytes in each slice used for encrypting.

For large files, loading the entire file into memory, and then making calculations for encrypting is not feasible. This is because the amount of memory needed is 100 to 200 times greater than the size of the file being encrypted.

For example, if we were encrypting a 100 MiB file we would need around 10 GiB of memory. .Of course, this amount of memory is not available in most PCs nowadays.

This parameter enables the file content to be divided into slices. First, for each slice, content is read in an array of bytes,, then the encrypting is applied. Finally, the encrypted content of the slice is written into the encrypted file. It is not necessary to store the previous slices within memory in order to encrypt the next slice. Therefore not as much memory would be required as if the whole file was to be encrypted at once.

Decreasing this parameter decreases the application’s memory usage. 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, in order to make this encryption more difficult to break, this parameter should as high as possible, depending on the memory available for the application.

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

When encrypting, if a value of 0 is assigned to this parameter, the file will not be divided into slices (as in the case of using a single slice that includes all bytes of the 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 the numbers used by the chaotic pseudorandom generator in the first step of encrypting (XOR).

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

* For FileEncoderType 1, the units of this parameter are in bytes.
* For FileEncoderType 2, 3, 4, 5, 6, 7 and 8 the units of this parameter are in blocks of 31 bits. In this case it is equal to the number of long elements in the array which stores information of the number (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 encryption process (reordering).

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

* For FileEncoderType 1, the units of this parameter are in bytes.
* For FileEncoderType 2, 3, 4 and 5 the units of this parameter are in blocks of 31 bits. In this case it is equal to the number of long elements in the array which stores information of the number l (Each long element of the array stores 31 bits of the number).
* FileEncoderType 6, 7 and 8 do not use this parameter.
* *NumBitsPerIterationSimpleEncoder*. This encrypting configuration parameter contains the number of bits the pseudorandom generator returns within each y iteration.

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

This parameter dramatically affects computation time for the encryption. Using a very small value for this parameter to encrypt small files is recommended. For large files using high values for the parameter is better. If not, encryption time would be take too long, and this would be the case every time you needed to encrypt/decrypt the file..

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

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

A value of 4 produces the fastest encryptions; however it is not recommended to use the value with values lower than 16 bytes for the SizeOfNumbersSimpleEncoder because we would obtain numbers between 0 and 15 for every iteration of the pseudorandom generator the possible values would not have the same probability

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 at a size of 32 bytes.

* For FileEncoderType 2, 3, 4, 5, 6, 7 and 8, this parameter can accept the values of 1, 2 and 4, or even multiples of 8 up to 64 (64 at maximum). Not every value is allowed.

It is recommended not to use a number (as configured on *SizeOfNumbersSimpleEncoder*) of less than 5 blocks for a value of 4. However, 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 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 the pseudorandom generator returns in every iteration. This parameter applies to the pseudorandom generator used in the second step (reordering).

This parameter sharply impacts the computation time of the encryption. . In order to make the encryption stronger, using a very small value for this parameter to encrypt smaller files is recommended.

For large files, it is better to use high values for this parameter. If not, the encryption time would take too long, and this would be the case every time you had to encrypt/decrypt the file.

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

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

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

For numbers with size greater or equal than 16 bytes, a value of 4 for this parameter is allowed, although it would be better with numbers greater or equal than 32 bytes.

* For FileEncoderType 2, 3, 4 and 5, this parameter can accept the values of 1, 2 and 4, or even multiples of 8 up to 64 (64 at maximum). Not every value is allowed.

It is suggested not to use a number (as configured on *SizeOfNumbersSimpleEncoder*) of less than 5 blocks for a value of 4. However, 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 up to 64), the configured value has to meet this condition:

*NumBitsPerIterationReordererEncoder* <= 3 x *SizeOfNumbersReordererEncoder*

* FileEncoderType 6, 7 and 8 do not use this parameter.
* *NumOfThreads*. It is the number of encryption threads that will be attempted to be used in case the encrypted and the decrypted file are on the same disk (for fileEncoderType 7 and 8).

The “optimum” value can be configured, which indicates that prior to the encryption process, the optimum number of threads will be calculated. This is an experimental feature, and if possible, it is recommended to use concrete numbers to configure this parameter.

The total amount of threads to be used in this case will be the value of this parameter plus an additional thread that will be used for reading and writing the disk.

* *NumOfThreadsForDifferentDrives*. It is the number of encryption threads that will be attempted to be used in case the encrypted and the decrypted files are on different drives (for fileEncoderType 7 and 8).

The “optimum” value can be configured, which indicates that prior to the encryption process, the optimum number of threads will be calculated. This is an experimental feature, and if possible, it is recommended to use concrete numbers to configure this parameter.

The total amount of threads to be used in this case will be the value of this parameter plus an additional thread that will be used for reading and writing the disk.

* *MaxTotalNumOfThreads*. This is the limit on the total amount of threads that the application will use for encryption, including threads for reading and writing files.

You can use the “max” value to indicate the maximum number of threads that your processor supports (this is related to the number of cores your processor possesses).

* *SizeInBytesForVolumes*. This is the maximum number of bytes each volume, in which the encrypted file will be split, can have.

##  Application Settings

This form allows you to configure the application parameters.

The form has several tabs that are shown below.

### Application configuration parameters for encryption

See tab below:



The parameters which can be configured are the following:

* *Erase decrypted file after encrypting*. This check box allows you to select whether or not you want to automatically erase the decrypted file after encryption.

The deletion is not just a standard one, the file is erased by writing "0" for the value of all bytesof the file. After this, a normal deletion is done.

This way it is impossible to recover the decrypted file unless you decrypt it in the standard way by using a password.*Rename encrypted file to .old after decrypting*. This check box allows you to select whether or not you want to keep the encrypted file after it has been decrypted by renaming it as *file.old.*. This way you have a backup copy in case something unexpected happens.

* *Ask to overwrite .old encrypted file*. In the event that the previous check box option has been 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
* *Encrypting configuration priority*. The order of priority of the encrypting configuration can be established in this table.

By default the options have this order of priority:

* (1) - *Manual*. This option refers to a manually applied encrypting configuration whose parameters have been edited 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 the automatic application of an encrypting configuration from the list of encrypting configurations which have been established based upon file size. See point: ***3.6***- ***List of encrypting configurations*** .
* (3) - *Based on encrypting file parameters got after decrypting*. This refers tousing the encrypting configuration from the last decrypted file.

In order to reconfigure this order of priority select the row on the table whose priority you want to change, then click on the *Up* or *Down* buttons to increase or decrease its priority.

* Always ask to switch obsolete fileEncoderTypes.

This application version, some fileEncoderTypes have been found to be deprecated, due to a bug found in them.

This check box allows the application to ask the user what to do if one of these fileEncoderTypes is found while encrypting a file.

If this check box is unchecked then the following check box will be enabled.

* Always switch obsolete fileEncoderTypes.

If the previous check box is checked, this option will allow the determining what automatic behaviour to use when deprecated fileEncoderTypes are found.

If this check box is checked (always swtich obsolete fileEncoderTypes), when fileEncoderType 2 or 3 are found, they will be changed by fileEncoderType 4.

If this check box is unchecked, then the application will leave the fileEncoderType without changing it, ignoring the it is a deprecated one. (This can be useful if you want to decrypt the file with a previous version of the application, which only had fileEncoderTypes 1 and 2).

### Application configuration parameters for language

See tab below:



You can configure the following parameters:

* *Language*. This is the interface text language of the application.

The languages available are the following:

* *EN*. English
* *ES*. Spanish
* *Additional language*. CAT (Catalan) is the default additional language.
* *Locale (associated to language)*. The application will use Java Locale for this language.

The application uses it to format the numeric strings.

* *Web language*. Indicates the language that is going to be used with the server in the new version inquiries.
* *Additional Language*. The additional language is what appears when you want to change the language.

For this new language you will have to choose the locale to be used.

If you want to set up a language in which there are no interface texts, you can add your own language by translating files found in the folder that will be created when you click the OK button.

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

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

# TITLE

# xxxxxxxxxx

LABEL1=text1

LABEL2=text2

...

The labels must not be modified, but texts should be modified depending on the language of choice.

In addition, there are also some files in RTF (Rich Text Format)which will have to translated using an RTF editor such as MS Office Word for example.

If you create a translation for a language currently not available, you can send it to me via email at (frojasg1@hotmail.com), and I will be sure incorporate it into new releases.

### Application configuration parameters for view

See tab below:



* Application zoom. This parameter allows you to modify window size the within the application, giving you the possibility to choose the small, medium, or large.

## Deletion of files

If output already exists, file encryption/decryption is not permitted. In other words, the application does not allow files to be overwritten during the encryption and decryption processes.

For this reason, an user-friendly option for deleting files has been enabled.

To open the dialog box for deleting files, you only need to click the X button associated to the file you want to delete:



If the selected file does not exist, then a message will appear in the text box under ¨History¨

If the file exists, then a dialog box giving you the option to choose which deletion method you prefer will appear:



At this point, you have three options:

* *Secure deletion*: This method prevents the recovery of the deleted file under any circumstances by rewriting all

 all the bytes of the file with zeroes and finally deleting in an a traditional manner

This form of file deletion takes more time so processing time is displayed along the option to cancel at any time. *Normal deletion*: This method performs normal file deletion.

* *Cancel*: This option allows the cancellation of the operation, in case you selected file deletion by mistake.

Once you have chosen a deletion method, the application checks to see if the selected file exists (for a matched file we understand the encrypted and the decrypted file This is to avoid the accidental deletion of a file that will not be able to via encryption or decryption.

The following warning message is shown :



Essentially ,the alert is requesting confirmation to delete the file.

Once deletion of the file has been confirmed the selected actionswill be executed on all associated files. For example, (if the file you wish to delete is an encrypted file within fileEncoderType-8, which consists of multiple volumes, then all assocated volumes would be deleted at the same time.).

## About…

This displays a window that includes a summary of the new features introduced in this version and greetings.

See below:



## License

Within Inside the About menu, there is the License option, which shows you the license you just accepted the first time you run the application.

Located within the About Menu is the License option which display the license you have accepted after

See below:



# Encrypting via Command Line

In addition to encrypting and decrypting via the GUI application, it is possible to initiate these functions from the command line.

The password for encryption can be passed through a parameter of the scripts. This is not advisable because in order to encrypt/decrypt this way, you’d have to write the password down along with the command. This could result in the encryption being hacked as it would be possible to discover the password simply by looking within the command history or into processes running within the system. would appear on the screen, so this is not very safe way to use the scripts.

It is advisable not to pass the password as a parameter of the invocation to the scripts.. This way, the application will ask by the keyboard the value for password and its confirmation.

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

There are two versions of the scripts available for Windows (\*.bat scripts) and Linux or Mac (\*.sh scripts) respectively.

## Scripts for Windows

The Windows version of the scripts is located in the folder named*...\\_scripts\windows*

The scripts available are the following:

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

Invoke these scripts within the same folder in which they are located.

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 commands for decrypting an encrypted file. It takes as arguments the following: Name of the encrypted file and password.

The third script contains 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 that does the encryption.

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

The password parameter has been made optional within all the scripts, therefore if the password is not included during invocation of the application, the password will be requested.

## Scripts for Linux and Mac

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

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

These scripts are initially compressed into the file *scripts.tar.gz*.

It is necessary to decompress this file in order to be able to work with the scripts. To do this, type the next command from the folder where the file scripts.tar.gz is located:

*tar -xvzf scripts.tar.gz*

Once the file is decompressed we will 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 located.

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 following: Name of the encrypted file and password.

The third script contains the commands for encrypting a file. It takes as arguments the following: File Name and password.

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

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

 The password parameter has been made optional within all the scripts, therefore if the password is not included during the invocation of the application, the password will be requested

## Passing Encrypting Configuration Parameters As Arguments

The main script is the following:

*command.interface.FileEncoder*

This script is used to invoke the Java application. The main parameters are described in Section ***3.7.1***-***Encrypting configuration parameters***

Here is a summary of the parameters:

* *-password* is followed by a string with the password. It is optional, and if the password is not provided, the password and password confirmation will be requested.
* -*encodedFileName* is followed by a string with the encoded file name
* *-decodedFileName* is followed by a string with the decoded file name
* -*encode* or -*decode* is used to select between encryption and decryption
* *-fileEncoderType* Parameter followed by the id of the FileEncoderType (1, deprecated; or 2) (only when encrypting encryption)
* *-sizeOfNumbersSimpleEncoder* is followed by the size of the numbers for step 1 (XOR) (only when encrypting)
* *-sizeOfNumbersReordererEncoder* is followed by the size of the numbers for step 2 (reordering) (only when encrypting)
* *-numberOfBitsPerIterationSimpleEncoder is* followed by the number of bits to be returned by the pseudorandom generator in every iteration for step 1 (XOR) (only when encrypting)
* *-numberOfBitsPerIterationReordererEncoder* is followed by the number of bits to be returned by the pseudorandom generator in every iteration for step 2 (reordering) (only when encrypting)
* *-numBytesFileSlice is* followed by the number of bytes in every slice. It is recommended to use a high value for this parameter. The memory used in the process will be bounded by roughly 100 hundred times the slice size (only when encrypting)
* *-useFileSizeForEncryptingParams.* If this parameter is present, all the parameters of the encryption (defined by the 5 previous parameters) will be calculated based on the size of the file to be encrypted, and all previous parameters will be overwritten.
* *-encDecOnDifferentDrives.* This parameter only applies in FileEncoderType-7 and 8.

If this parameter is present (it never has an associated value)this means that the encrypted and decrypted files are on different drives, and that the application will use dedicated threads for reading and writing files (one thread for every task).

If the parameter is not present, then one only dedicated thread will be used for both reading and writing of the files.

* *-numOfThreads* This parameter is only valid for fileEncoderType 7 and 8 and communicates the number of encryption threads that will be used (in this case parameter -encDecOnDifferentDrives does not appear).

See point ***5.4-Use of execution threads (parallelism)***about use of threads.

“optimum” value:

This parameter can configure the ´´optimum¨ value that will communicate to the application estimated optimum number of threads through the formula…. This parameter is in beta.

This parameter can be configured with the “optimum” value (which is in experimental phase) that will tell the application to estimate the optimum number of threads through the formula detailed in ***5.4-Use of execution threads (parallelism)*** point).

* *-numOfThreadsForDifferentDrives* This parameter will be valid only for the fileEncoderType 7 and 8 and communicates the number of encryption threads that will be used (in this case the parameter -encDecOnDifferentDrives appears).

This parameter takes values larger than those of the previous parameter.

See point ***5.4-Use of execution threads (parallelism)*** about use of threads.

“optimum” value:

This parameter can be configured with the “optimum” value (which is in experimental phase) that will tell the application to estimate the optimum number of threads through the formula detailed in ***5.4-Use of execution threads (parallelism)*** point).

* *-maxTotalNumOfThreads* This parameter is only for the fileEncoderType 7 and 8 and is used to limit the total number of threads used by the application.

In case that number is reached, the number of encryption threads will be lowered.

See point ***5.4-Use of execution threads (parallelism)*** about use of threads.

This parameter can be configured with the “max” value which indicates the maximum number of threads the processor can have.

* -sizeInBytesForVolumes. This parameter is valid only for FileEncoderType-8 and is used to configure the maximum length each volume in which the encrypted file is split into can reach.

# Method of encryption used

The algorithms for encryption and decryption are very similar and possess identical processing times (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 pseudorandom generators with particular sets of configured parameters, it’s possible 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.

Size-2 random bytes are calculated from the microphone source making up key-2, which will be used for the encryption.

If the microphone is not available, random bytes will be obtained using the generateSeed function of the SecureRandom class.

This method to collect random bytes might be slower than the microphone, so it is advisable to keep the microphone connected.

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. (This means that in addition to encryption/decryption, the application has to re-read the decrypted file to calculate the MD5 hash).

This data will be used to decrypt the file. The MD5 hash will be used to check if the decrypting process has been successful.

In the case of FileEncoderTypes 5, 6, 7 and 8, it will be necessary to store the length of the decrypted file also, for the decrypted file to be correctly obtained in the decryption process.

In the case of FileEncoderTypes 7 and 8 the number of threads will also be stored in the header.

In the case of FileEncoderTypes 8 the maximum length of volumes will be stored as well.

Prior to decryption, FileEncoderTypes 5, 6, 7 and 8, due to special features, must create the entire decrypted file using zeroes. This is why for these types of encryptions the decryption process takes a little more time.

##  Algorithm used for encryption

This application performs the file encryption process in slices essentially dividing files into smaller parts. In order to encrypt each slice the following steps are taken:

* Step 1: XOR stile encryption of the slice. At this stage of encryption, the application generates pseudorandom bytes and performs a XOR with the original data. From this an encrypted slice is create whose bytes are quite random.

However, if we ended the process at this stage, the encryption would be relatively easy to hack provided that some unencrypted bytes from the creation of the file were known. This is the often the case for file types that can be accessed through an application such as MS Word or Excel. This is because these file types generally have some kind of header which in turn makes mere XOR encryption weaker.

For this reason the application requires a second encryption step.

Pseudorandom reordering of bytes within the slice. At this stage of the encryption process, pseudorandom positions (in bytes) are generated. Essentially, every byte within the slice is reordered Consequently, in order to obtain the initial bytes of the decrypted file, it becomes necessary to decrypt the entire slice. Therefore, breaking the encryption becomes more difficult in terms of time and effort. Completely decrypting a slice is more costly in terms of computation time than decrypting the first 100 to 1000 bytes of the slice and comparing them to the file´s header.

FileEncoderTypes 6, 7 and 8 remove this step, so as a result they are much faster than other FileEncoderTypes.

* Pseudorandom reordering of slices. This encryption step is only applicable to FileEncoderTypes 5, 6, 7 and 8.

This step makes the encryption of very large files very difficult to break. This is because slices have been randomly ordered .As a result half of the slices would have to be decrypted before the first slice (which contains the header of the decrypted file which can have a known format, and allow determining if a password generated by force brute is good or not) could be decrypted.

If the file is very large hacking it would not be worth the time or effort because identifying the correct password would be very time-consuming even with a super computer.

This feature makes FileEncoderTypes 5, 6, 7 and 8 the most difficultto break for big files.

In the previous points: XOR stile encryption and pseudorandom reordering of slices use chaotic pseudorandom generation to perform their tasks.

Within the current implementation, these pseudorandom generators are characterized by the size of the numbers - used in the generator and the number of pseudorandom bits returned within each iteration.

The pseudorandom generator calculates the optimum number of bytes that the key should have in order to initialize it based upon these two parameters.

The larger the ranges of the numbers are, , the more optimal the key size to initialize it . This also results in longer computation times for each iteration In other words, the greater the range of numbers and the longer the encryption/decryption process the stronger the encryption will be.

Furthermore, the smaller the number of bits returned within each iteration, the greater the effect of the range of numbers on computation time.

The ideal configuration would be strongly encrypted and fast to perform. To achieve this, we could use a large range of numbers (stronger encryption) with a high number of bits per iteration (faster performance), and combine it with large slice sizes which would result in much stronger encryption, but would require too much memory. For smaller files you should use a very small bits per iteration number (for example 1) and a large range of numbers The size of the slice should be at least equal to the size of the file.

 Examples of optimal configurations:

* For small files:
* FileEncoderType: 4
* Range of numbers: 256
* Bits per iteration: 1
* Slice size: 0
* For large files (larger than 100 MB):
* FileEncoderType: 7
* Range of numbers: 10
* Bits per iteration: 1
* Slice size: ( at least equal to the size of the file / 10, at a maximum of 16 MB).
* Number of threads: 3 (for input and output files located on the same drive).
* Number of threads: 5 (for input and output files located on different drives).

## Pseudorandom generator used

The pseudorandom generator used in this application is based on chaotic sequences. It is very basic and easy to implement.

The components of the pseudorandom generator were referenced from the following source:

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

I would like to personally thank the authors for writing such an interesting book.

The following is a translation of an excerpt from the book:

"

*The generator, proposed by M. Romera, I. Jiménez and J. Negrillo [ROM90], is also based upon 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 for this generator (in 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 ...) while others are chaotic.*

*When the value of K is very close to -2, for example K = -1.99999XXXXXXXXXXX, the series obtained from the Mandelbrot set is nearly always chaotic.*

*...*

*The real sequence that is obtained can easily be converted into a binary pseudorandom sequence. This can be achieved 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 can be hacked, based on this pseudorandom generator.

The options available are the following:

* Use Java class BigDecimal which isused in FileEncoderType=1,butnot recommended.
* Develop a new class which would be capable of managing numbers of variable precision (used in FileEncoderType=2 and 3 (DEPRECATED) and fileEncoderType=4 (recommended for small files), 5(recommended for medium-sized files), 6 (not recommended) and 7 and 8 (recommended for large files) ).

Pseudorandom bits derived from the chaotic sequence are obtained in the following ways:

* For all FileEncoderTypes, when the number of bits produced in each iteration is 1, 2 or 4:

The "1" bits of Xi are counted, and the remainder is calculated (2 if the number of bits per iteration was 1, 4 if the number of bits per iteration was 2, and finally16 if the number of bits per iteration was 4).

This is the most secure version of the pseudorandom generator (it requires the most computation time, and it is least dependent on the particular value of Xi). Pseudorandom bits within each iteration within 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 numbers equal to or greater than 16 bits because if 1 byte (8 bits) were used, when calculating the remainder the "1" bits would be divided into 4 resulting in values of 0, 1, 2 or 3 which would not be equiprobable. On the other hand, if we used 2 bytes (16 bits) the resulting value (0, 1, 2 or 3) would almost be equiprobable, therefore it could be used.
* If 4 bits per iteration are returned, it is better to use numbers equal to or greater than 32 bytes (256 bits)

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Value | numbersof 16 bits | numbersof 32 bits | numbersof 64 bits | numbersof 128 bits | numbersof 192 bits | numbersof 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 demonstrated above, 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 the values of 8 and 64 (both included) (FileEncoderType 2, 3, 4, 5, 6, 7 and 8):

In this case, all the bytes of value Xi are to be used as if they were pseudorandom numbers. This is not exact but it runs much faster. Using this method quickly obtains 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):

Number of Bits per Iteration / 8 <= 2 x Number of Blocks per Number

At most 3 bytes of each block are taken.

## Use of execution threads (parallelism)

This feature is only present in fileEncoderType 7 and 8.

These types of encryption add support to multiple execution threads.

* Use of threads
	+ When encrypted and decrypted files are located on the same drive:
		- One dedicated thread is used to centralize input/output to and from files. This thread feeds the encryption threads by reading slices from the input file. It is also in charge of writing the result or outcome in the output file.

A set of encryption threads

* + When encrypted and decrypted files are located on different drives:
		- One dedicated thread is used to centralize the reading of the input file. This thread feeds the encryption threads by reading slices of the input file
		- One dedicated thread is used to centralize the writing on or of the output file.

This thread receives the slices that are to be written and writes them in parallel to the reading of the input file.

* + - A set of encryption threads
* Encryption thread features

Every encryption thread uses its own encryption/decryption encryptor/decryptor ,and every one of these threads are characterized by an index that ranges from zero to the total number of encryption threads used.

Every encryption thread will receive, in order, every slice with a slice location congruent to the threading module index or the total number of encryption threads. (In other words, slices will be distributed, in order, for every thread of encryption.

Every encryption thread uses the encryption/decryption process to encrypt/decrypt slices assigned to the input file in order.

* Optimum value for number of encryption threads
	+ In the event of there being only one dedicated thread for input and output files (located on the same drive).

Theoretically, the minimum number of encryption threads necessary to minimize total processing time can be calculated using this following formula.

$$optimum=ceil\left( \frac{T\_{p}}{\left(T\_{i}+T\_{o}\right)}\right)$$

Where: $optimum$ is the optimum number of encryption threads to be used

 $T\_{p}$ Represents the processing time (pure encryption) of a slice

 $T\_{i}$ Represents the time used to read of a slice.

 $T\_{o}$ Represents the time used to write of a slice.

* + - The value of $T\_{p}$ is dependent on the following two parameters of encryption: The size of the numbers to be used by the pseudorandom generator and the number of bits obtained per iteration). Obviously, $T\_{p}$ is also dependent on CPU power.
		- The values for $T\_{i}$ and $T\_{o}$ are dependent on the storage drive where the input file is located and where the output file will be written. . These values are also dependent on whether the drive is being used, at this moment, by another application or if the drive is fragmented, etc.

* + In the event of separate input and output threads, one for reading the input file and another for writing the output file (in this case located on different drives).

The optimum number of threads can be calculated using the following formula: :

$$optimum=ceil\left( \frac{T\_{p}}{max\left(T\_{i}, T\_{o}\right)}\right)$$

Because the input and output threads on the disk are separated (working in parallel to each other) the total time it takes for a slice to be read and written can be calculated as the maximum between these two values (which in any case, would be less than the sum of both).

This means that, in this case, the optimum number of threads will be greater than or equal to the case of combined input and output on the same drive.

* Theoretical limitations:

The total number of threads of encryption should not be exceed the number of threads that the processor supports.

The total amount of threads is the sum of the encryption threads plus the input/output dedicated threads.

To automatically limit the number of threads used you can use the following parameter: -maxTotalNumOfThreads.

The number of threads that the processor supports is usually a multiple of the number of cores that the processor has.

For example, the processor in my PC is an i7 (4 core), but due to hyper-threading the number of simultaneous threads that the processor can manage is 8.

After calculating the optimum number of encryption threads, that result must be limited to the theoretical limit we just saw.

This is because we could might have, a system with a very fast hard drive (very short input and output times) and a very slow processing time as the result of a very slow processor.

In that case, we would have a very large optimum value that could be larger than the maximum number of threads that is recommended (based upon the capacity of our processor).

In that case, despite having a very fast hard drive, we would have a bottleneck in the processor that would prevent us from using the optimum number of threads calculated.

* The fastest option

If the files are located on different drives, higher efficiency and CPU use can be achieved, resulting in faster encryption (if the CPU is powerful enough).

If you are looking for faster encryption speeds placing input and output files on different drives is ideal

* Memory used

Another point to take into account is the amount of memory used by the application.

The higher the number of threads, the more memory will be used by the application.

That is why a parameter that limits this number has been introduced.

# Types of encryption

The FileEncoderApplication application allows for encryption e based on a group of different types of encryption which has been increased with every update of the application.

This allows you to maintain backward compatibility, making it possible to add new functionalities by including new types of encryption.

In this chapter we will detail the different types of encryption available in this version of the application.

## FileEncoderType-1

This FileEncoderType, which is **not recommended**, is based on a chaotic pseudorandom generator that internally uses numbers in BigDecimal format.

The disadvantage with this pseudorandom generator is that it is closely linked to Java implementation, making its translation to other possible programming languages more difficult

In addition, by using those types of numbers control is lost, and it could happen that the implementation of that class between different versions of Java could change breaking the compatibility of the application executed over different versions of JRE.

That is why the use of this type of encryption is not recommended.

However, because of backwards compatibility it still exists.

The features for this type of encryption are the following:

* Slice-based.
* Every slice is encoded with an XOR based encryption.
* Every slice reorders its bytes internally, to make the decryption process longer.

Java class BigDecimal based chaotic pseudorandom generator.

* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of BigDecimal numbers (in bytes) make up part of the pseudorandom generator for XOR (range-> 1-127).
	+ SizeOfNumbersReordererEncoder. Size of BigDecimal numbers (in bytes) make up part of the pseudorandom generator for reordering (range-> 1-127).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.
	+ NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.

## FileEncoderType-2

**This** FileEncoderType, which is now outdated, worked with a chaotic pseudorandom generator that internally used numbers in BigLong format. It became BigLong\_deprecated after an Error in that class had been located which appeared from time to time but almost always when encrypting big files).

FileEncoderType-2 has been preserved in order to maintain backward compatibility, making the decryption of files of this encryption type from previous versions of the application possible.

The features for this type of encryption are the following:

* Slice-based. Based on slices.
* Every slice is encoded with an XOR based encryption.
* Every slice reorders its bytes internally, to make the decryption process longer.
* Chaotic pseudorandom generator based on its own BigLong\_deprecated class (which **has a severe error**).
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of BigLong\_deprecated numbers (in blocks of 31 bits) that are part of the pseudorandom generator for XOR (range-> 1-127).
	+ SizeOfNumbersReordererEncoder. Size of BigLong\_deprecated numbers (in blocks of 31 bits) that are part of the pseudorandom generator for reordering (range-> 1-127).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator i
	+ NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.

## FileEncoderType-3

This FileEncoderType, which is now outdated, is identical to the previous one mentioned above, but the size of numbers has been extended to take advantage of the sign bit of the byte dedicated to store that value.

FileEncoderType-3 has been preserved in order to maintain backward compatibility, making the decryption of files of this encryption type from previous versions of the application possible.

The features for this type of encryption are the following:

* Slice-based.
* Every slice is encoded with an XOR based encryption.
* Every slice reorders its bytes internally, to make the decryption process longer.
* Chaotic pseudorandom generator based on its own BigLong\_deprecated class (which **has a severe error**).
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of BigLong\_deprecated numbers (in blocks of 31 bits) that are part of the pseudorandom generator for XOR (range-> 1-256).
	+ SizeOfNumbersReordererEncoder. Size of BigLong\_deprecated numbers (in blocks of 31 bits) that are part of the pseudorandom generator for reordering (range-> 1-256).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.

NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.

## FileEncoderType-4

FileEncoderType-4 is **especially recommended for** **small files**.

 It is identical to the previous one mentioned abovebut w the error in the type of numbers has been solved.

The features for this type of encryption are the following:

* Slice-based.
* Every slice is encoded with an XOR based encryption.
* Every slice reorders its bytes internally, to make the decryption process longer.
* Chaotic pseudorandom generator based on its own BigLong Java class. For the moment no error has been found.
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of BigLong numbers (in blocks of 31 bits)make up part of the pseudorandom generator for XOR (range-> 1-256).
	+ SizeOfNumbersReordererEncoder. Size of BigLong numbers (in blocks of 31 bits) make up part of the pseudorandom generator for reordering (range-> 1-256).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.
	+ NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.

## FileEncoderType-5

This FileEncoderType is not recommended. ( It created for large files, but with a small slice size, which is not recommendable, as the head of the drive goes crazy and could break down).

The features of this type of encryption are the following:

* Sliced-based.
* Every slice is encoded with an XOR based encryption.
* Every slice reorders its bytes internally, to make the decryption process longer.
* Chaotic pseudorandom generator based on its own Java class BigLong. For the moment no error has been found.
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of the BigLong numbers (in blocks of 31 bits) that make up part of the pseudorandom generator for XOR (range-> 1-256).
	+ SizeOfNumbersReordererEncoder. Size of BigLong numbers (in blocks of 31 bits) make up part of the pseudorandom generator for reordering (range-> 1-256).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.
	+ NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.

## FileEncoderType-6

This FileEncoderType is **recommended for very large files** within in computers with little **RAM memory** available).

It is a copy of the previously mentioned fileEncoderType, but removes the internal reordering of bytes for each slice.

It is used for large files, because by removing the internal reordering of bytes for each slice, speed increased tremendously.

Furthermore, in order to make this encryption more difficult to break a pseudo-random reordering of the input file slices is carried out.

This means that on average) half of the slices of the encrypted file would have to be decrypted before getting to the initial slice of the decrypted file.

Furthermore, this means that slices can be larger (since the internal reordering of the bytes for each slice does not require so much memory), preventing the issue with the head of the hard drive mentioned earlier.

Nevertheless, for this version of the application, newer, faster encryption types have been created (which make use of several threads), specially designed to encrypt large files. As a result, using FileEncoderType-6 is only recommended when you have a small amount of RAM memory is available within the system.

The features of this type of encryption are the following:

* Slice-based.
* Every slice is encoded with an XOR based encryption.
* The positions of slices are reordered in a pseudo-random order.
* Chaotic pseudorandom generator based on its own BigLong Java class. For the moment no error has been found.
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of the BigLong numbers (in blocks of 31 bits) makes up part of the pseudorandom generator for XOR (range-> 1-256).

NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.

## FileEncoderType-7

This FileEncoderType is **recommended for very large files**.

If your computer has a small amount of RAM memory available, you should configure the number of encryption threads to 1.

It is a copy of the previously mentioned FileEncoderType-6, but by using multiple threads faster encryption speeds are possible

The features for this type of encryption are the following:

* Sliced based.
* Every slice is encoded with an XOR based encryption
* Every slice reorders its bytes internally, to make the decryption process longer.
* Chaotic pseudorandom generator based on Java class BigLong. For the moment no error has been found.
* Several threads are used in parallel:
	+ When encrypted and decrypted files are located on the same drive:
		- A common dedicated thread for reading and writing the disk.
		- One or more threads for encrypting.
	+ When encrypted and decrypted files are located on different drives:
		- A dedicated thread for reading the input file.
		- A dedicated thread for writing the output file
		- One or more threads for encrypting.
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of BigLong numbers (in blocks of 31 bits) make up part of the pseudorandom generator for XOR (range-> 1-256).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.
	+ NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.
	+ NumOfThreads. Number of encryption threads in case both encrypted and decrypted files are located on the same drive.
	+ NumOfThreadsForDifferentDrives. Number of encryption threads in case encrypted and decrypted files are located on different drives.
	+ MaxTotalNumberOfThreads. Limiton the total number of threads to be used (includes threads for encryption, and threads for input and output files input).

## FileEncoderType-8

This FileEncoderType **recommended for very large files**. It is especially recommended **when the drive** where the encrypted file will be written **has a limit on the maximum size of a file**.

FileEncoderType-8 is a copy of the previously mentioned one, but adds the possibility of spliting the encrypted file in smaller volumes, whose maximum sizes can be set by parameter.

The features of this type of encryption are the following:

* Slice-based.
* Every slice is encoded with an XOR based encryption.
* Every slice reorders its bytes internally, to make the decryption process longer.
* Chaotic pseudorandom generator based on Java class BigLong. For the moment no error has been found.
* Several threads are used in parallel:
	+ When encrypted and decrypted files are located on the same drive:
		- A common dedicated thread for reading and writing the disk.
		- One or more threads for encrypting.
	+ When encrypted and decrypted files are located on different drives:
		- A dedicated thread for reading the input file.
		- A dedicated thread for writing the output file
		- One or more threads for encrypting.
* Parameters:
	+ NumBytesFileSlice. Size of slices in bytes.
	+ SizeOfNumbersSimpleEncoder. Size of BigLong numbers (in blocks of 31 bits) make up part of the pseudorandom generator for XOR (range-> 1-256).
	+ NumBitsPerIterationSimpleEncoder. Number of bits gained from every iteration of the XOR pseudorandom generator.
	+ NumBitsPerIterationReordererEncoder. Number of bits gained from every iteration of the pseudorandom generator for reordering.
	+ NumOfThreads. Number of encryption threads in case both encrypted and decrypted files are located on the same drive.
	+ NumOfThreadsForDifferentDrives. Number of encryption threads in case encrypted and decrypted files are located on different drives.
	+ MaxTotalNumberOfThreads. Limit on the total number of threads to be used (includes threads for encryption and threads for input and output files).
	+ SizeInBytesForVolumes. Maximum size in bytes for every of the volumes in which the encrypted file will be split when encrypting.