Cryptography Project

ITRI 615

SC Fourie 24060984, W Swarts 24223468

2018

# Contents

[Contents 1](#_Toc514218990)

[Table of Figures 1](#_Toc514218991)

[Introduction 2](#_Toc514218992)

[Objectives 2](#_Toc514218993)

[Vigenѐre 2](#_Toc514218994)

[Background 2](#_Toc514218995)

[Explanation of code 3](#_Toc514218996)

[Source Code 5](#_Toc514218997)

[Transposition 6](#_Toc514218998)

[Background 6](#_Toc514218999)

[Explanation of code 7](#_Toc514219000)

[Encryption 7](#_Toc514219001)

[Decryption 7](#_Toc514219002)

[Source Code 8](#_Toc514219003)

[Vernam 9](#_Toc514219004)

[Background 9](#_Toc514219005)

[Explanation of the code 9](#_Toc514219006)

[Source code 10](#_Toc514219007)

[Own algorithm 11](#_Toc514219008)

[Explanation of the code 11](#_Toc514219009)

[Source code 12](#_Toc514219010)

[Conversion Code 13](#_Toc514219011)

[Using the graphical user interface 13](#_Toc514219012)

[References 15](#_Toc514219013)

# Table of Figures

[Figure 1: Traditional Vigenѐre Table () 3](#_Toc514219014)

[Figure 2: Vigenere Source Code 6](#_Toc514219015)

[Figure 3: Transposition Source Code 9](#_Toc514219016)

[Figure 4: The Xor process 10](#_Toc514219017)

[Figure 5: Source code for the Vernam cipher 11](#_Toc514219018)

[Figure 6: Source code for our own algorithm 13](#_Toc514219019)

[Figure 7: The main menu for the program 14](#_Toc514219020)

[Figure 8: Output generated using the Vernam algorithm 15](#_Toc514219021)

# Introduction

The purpose of the project is to design 4 different cipher algorithms based on the Vigenѐre cipher, Transposition cipher, Vernam cipher, and any cipher that we design. Thee algorithms should be fairly complex and should be able to encrypt any files or messages.

## Objectives

* Do research on the background of each cipher
* Design and code an algorithm for each cipher
* Explain algorithm
* Design own algorithm, can be based/combinations of other ciphers
* Design and implement GUI

# Vigenѐre

## Background

Traditionally the Vigenѐre Cipher uses a 26x26 table called the Vigenѐre table. The first row is consists out of the normal English alphabet, A-Z. The second row follows the first but starts at B, therefore B-A. The rest of the rows follows in the same manner. In our case, we used a 36x36 table which includes numbers, 0-9, as well. This makes it more complex as there can be more variety and numbers in the plaintext can also be encrypted, as well as using numbers in the keyword.

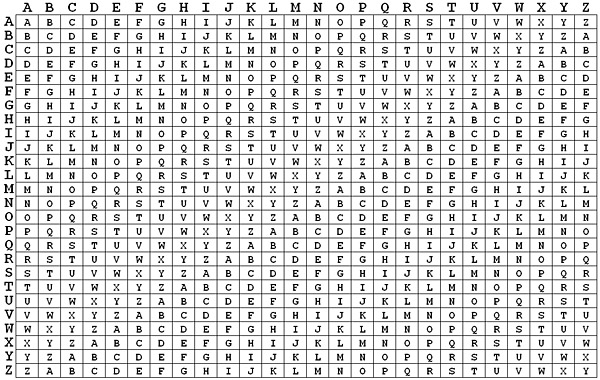


Figure 1: Traditional Vigenѐre Table ()

The Vigenѐre Cipher also makes use of a keyword. The keyword is repeated until it matches the length of the plaintext, e.g. if the keyword is **pass** and the plain text is **North West University** the result will be:

**North West University**

**passp assp asspasspas**

Traditionally all spaces and punctuation are removed, all letters are made uppercase and the plaintext is divided into blocks of five characters. This was a little difficult to implement as difficulties presented themselves when the message had to be decrypted again. Therefore we made all the letters uppercase but left the rest as is.

To encrypt the plaintext, use the plaintext letter in reference to the column heading and the keyword letter in reference to the row heading. Where these two letters intersect is the letter that should be used in the cipher text. Using the table above and the example, the first letter in the cipher text would be **C** as this is where **N** and **P** intersects.

To decrypt the cipher text, the letter from the keyword is used as reference to row heading and the column heading is used to reference the plaintext letter. First find the keyword letter in the list of rows, then find the cipher text letter in the row of that keyword letter. The column heading of that cipher text letter is the plaintext letter. E.g. using **P** in to show the row in the table, we find **C** in the row of **P**, then see in which column the letter **C** is. We find the column heading to be **N**.

## Explanation of code

Firstly, the entire Vigenѐre table is not required, therefore, only a constant variable of type string is declared with the entire English alphabet as well as the numbers 0 – 9. Then three different functions are created, the first two simply for distinguishing between encryption and decryption; “***encryptMess*** ***(key, message)***” and “***decryptMess (key, message)***”. This was done because they follow very similar code with a slight difference that can be made inside the third function; “***messTranslation (key, message, func)***”.

In this method, the key and the message are both made to be upper case as all the values in the Vigenѐre table is uppercase. The key index is also created and assigned a value of 0 so that it starts at the beginning of the key. The process then loops through every character in the message; “***for sym in message***” which checks to see if the character is in the loop or not by using an if else statement. The process can be divided up by explaining the following code:

* “***index = VIGENERE\_TABLE.find(sym)***”: This looks up the location of the character in the Vigenѐre table. It is the equivalent of the column character when encrypting and the row character when decrypting. Using the example used in the background, we can see that the first character is **N** which means the original index is 13. If we want to decrypt, the first character would be **C**, which is at index 2.
* “***if func == 'enc':***

***index += VIGENERE\_TABLE.find(key[keyIndex])***

***elif func == 'dec':***

***index -= VIGENERE\_TABLE.find(key[keyIndex])***” :

This is the slight difference and why the first two functions are required. First of all, the index where this loop’s key character can be found in the Vigenѐre table is used to either increment the original index (when encrypting) or decrement it (when decrypting). As an example, if we use the password pass, the first loops key character would be **P** meaning the index of this character in the Vigenѐre table is 15. If we are encrypting this would be added to the original index of 13 giving 28. Decrypting would give a result of -13.

* “***index %= len(VIGENERE\_TABLE)***”: This creates a wrap-around for both positive numbers or negative numbers. It creates a virtual Vigenѐre table where each new row moves one step to the left. Using this code in the example, we can see that 28 mod 26 would give 2 and -13 mod 26 gives 13. Both these values are still in the range of the length of the Vigenѐre table.
* “***translation.append(VIGENERE\_TABLE[index])***”: appends the end result, whether encryption or decryption, with the respective characters at the calculated indexes in the Vigenѐre table. 2 would give **C** and 13 would give **N**. These are added to an array.
* The next few lines of code simply ensures that the key index is incremented, that the index restart at 0 when all characters have been used, and then to combine the entire array of characters into a string. This array is return to the calling process.

## Source Code

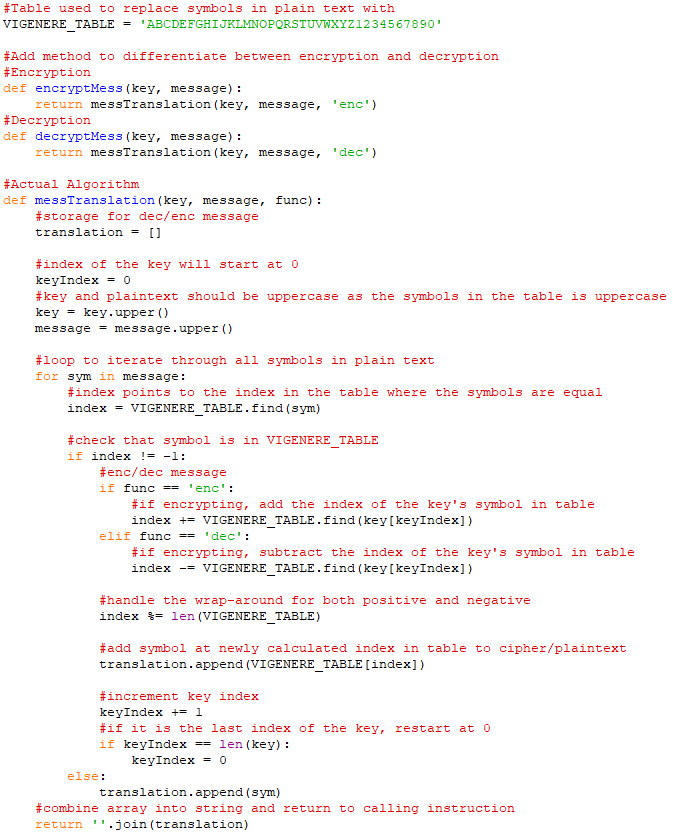


Figure 2: Vigenere Source Code

# Transposition

## Background

In this cipher, the characters of a plain text is shifted in some systematic pattern to create the cipher text (). Easily remembered mnemonics are used in the manual Transposition ciphers, such as the “rail fence” cipher. In this example, the characters of the plain text is alternated between rows, and the rows are then read sequentially to form one string.

WE ARE AT WAR

Would be processed like this:

W A R T A

E R A W R

WARTAERAWR

This is, however, the simplest example of route ciphers () which were used in the early days of cryptography. A more general method would be to write the characters in prearranged order, such as a matrix. The arrangement is agreed upon by both sender and receiver beforehand. The geometric array, starting points, and the routes taken are kept secret by a key. The key can usually be anything and used in any way, depending on the arrangement and routes of the algorithm. Using the previous phrase as example with a 3x4 matrix and a key of lat we find

L A T

2 1 3

W E A

R E A

T W A

R … …

The cipher text would be EEWWRTRAAA.

To decrypt the Transposition cipher the cipher text is entered into the matrix according to the encryption route and read according to original order of entry (). The key hides the order and therefore is essential for decryption.

The method we used simply uses the key as an indication of how many columns are in the matrix. The characters of the plain text is then shuffled between these indexes one by one. The cipher text is created by combing these columns starting at column one.

## Explanation of code

Both encryption and decryption takes in a key and a string as parameters. The character values of the key is essentially converted to integers and summed which is then used as the key.

### Encryption

The following code are key parts of the code and will be explained:

* “***cText = [''] \* key***”: This defines an array which gives the sense of a grid with as many columns as the value of the key. Each ‘column’ will exist out of a string after the encryption, that is all the characters added to the ‘column’. For example, if the key is 8, the grid will have 8 columns.
* This is followed by the definition of the key index, converting the message to string (in case it is numerical), and defining the length of the message. The key index starts at zero and will span to the specified number.
* “***for sym in mess:***

***cText[cIndex] += sym***”:

Looping through the message, manipulating each character individually. The symbol is appended to the ‘***cText***’ array, the location in the array is specified by the key index. For example, the first character in the message will be added to “***cText[0]***” and the second character at “***cText[1]***”.

* “***cIndex += 1***

***if cIndex == key:***

***cIndex = 0***”:

The key index is incremented after each loop and restarts at zero when the key number is reached. For instance if the key is 8, the index will revert to 0 when it reaches 8.

* The last part simply combines the “***cText***” array into a single string and returns it to the calling process.

### Decryption

The following code are key parts of the code and will be explained:

* “***columns = int(math.ceil(len(cText) / key))***”:” In order to decrypt a transposition cipher we need to work out a different grid. The calculation is done here, where the length of the cipher text is divided by the key and the result is rounded up. For example, if the length of the cipher text is 30 and the key is 8, we get a result of 3.75 which is rounded up to 4. This represents the ‘columns’ or the size of the array.
* Next an array is defined to the size of the columns defined above. And the amount of required empty spaces, called shaded spaces, is calculated by subtracting the length of the cipher text from the product of the ‘columns’ and the key. These spaces fill the last column from the bottom and should not be filled by any values.
* “***for char in cText:***

***mess[col] += char***

***col += 1***”:

This is the start of the loop. Each character in the loop is added to the specified ‘column’ in the array. The ‘column’ is specified by a variable ‘***col***’ defined before the loop. This value is incremented after each loop.

* The next if statement is used to revert the ‘**col**’ variable to 0 when it equals the ‘**column**’ value or if a shaded space is found.
* The last line of code combines the array into a single string and returns it to the calling process.

## Source Code

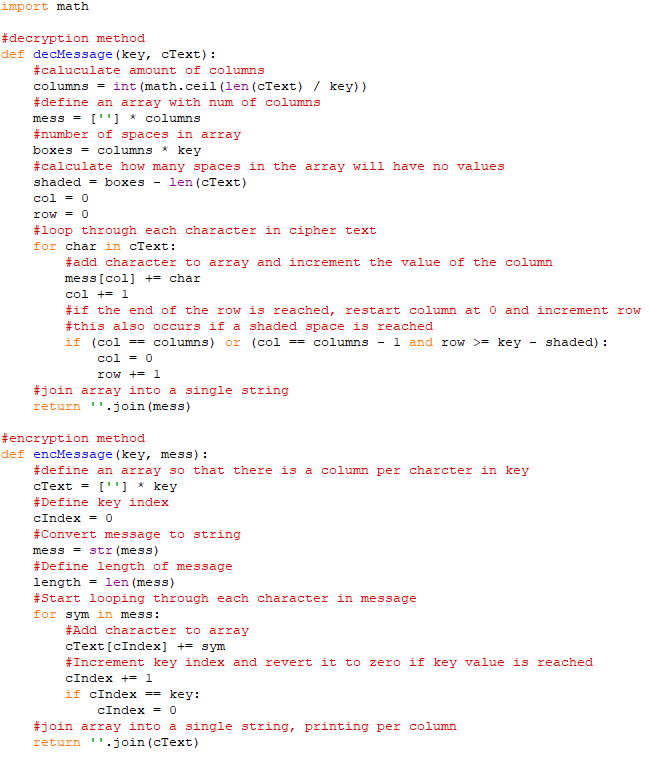


Figure 3: Transposition Source Code

# Vernam

## Background

With the Vernam cipher, the cipher text is calculated by adding the key to the plaintext by calculating the exclusive-or (Xor) of each individual characters of the key and plaintext. This generates a pseudo-random ciphertext.

To decrypt the above ciphertext, the Xor is calculated again with the key. This generates the original plaintext, due to the nature of the Xor process.

As an example, let C be the cipertex, P the plaintext, K the key and f() the Xor process:



Figure 4: The Xor process

## Explanation of the code

Both encryption and decryption takes in a key and a string as parameters. The character values of the key is essentially converted to integers and summed which is then used as the key. Should a key not be given, a random key is generated.

**Encryption**

The following functions explain the key principles behind the encryption program:

* **Chr(ord(char) ^ ord(key[i])) :**

This part of the algorithm calculates the bit-wise exclusive-or of each individual character of the key and the plaintext.

* The result of the above operation is then cast to a character and appended to a returning string.

**Decryption**

The encryption function (**givevernam()**) is executed with the ciphertext and key as input, generating the plaintext in a returning string.

## Source code



Figure 5: Source code for the Vernam cipher

# Own algorithm

Our own algorithm incorporates substitution, transposition and confusion principles to encrypt the plaintext.

First the key and the plaintext is combined by using a Xor. This implements the principle behind the Vernam cipher. Each key is then shifted by a value k, which is computed by calculating the sum of the byte value of each character in the key. Finally, the cipher text is then generated by implementing the transposition algorithm with k as the key.

The result of the above sequence is a ciphertext which is substituted, shifted and shuffled, with some elements that are based on the key, without using the actual key value explicitly.

## Explanation of the code

**Encryption**

* **“for char in key:**

**k += ord(char)”:** This calculates the k value, as mentioned above.

* **“vernem = vernOb.giveVernam(msg, key)” :**

This re-uses the Vernem algorithm to calculate the Xor value from the key and the plaintext.

* **for i in range(0, len(msgNum)):**

**cipher += chr(msgNum[i]+k)**

**cipher = Transposition.encMessage(k, cipher):**

The above shifts the cipher by k bytes and transposes the result into a cipher text.

**Decryption**

The cipher is decrypted by applying the encryption algorithm in reverse order and shifting the cipher in the reverse of what was applied.

## Source code

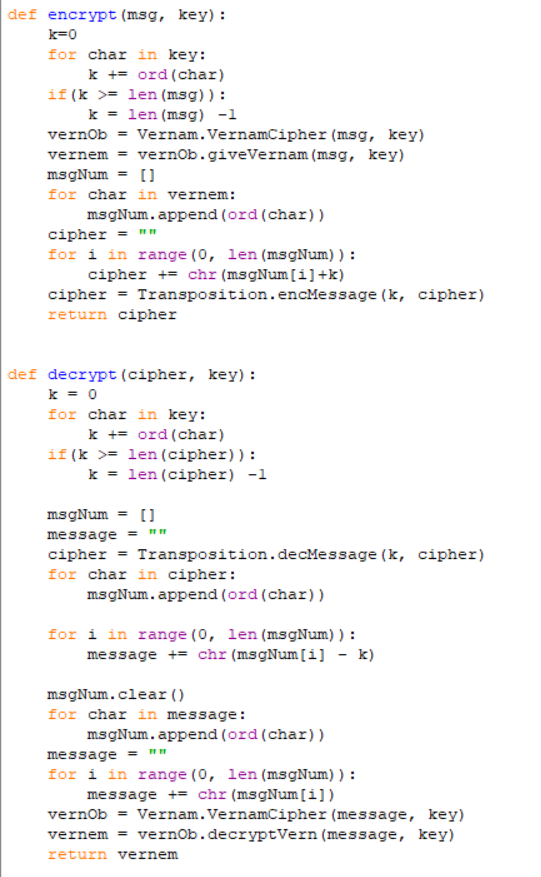
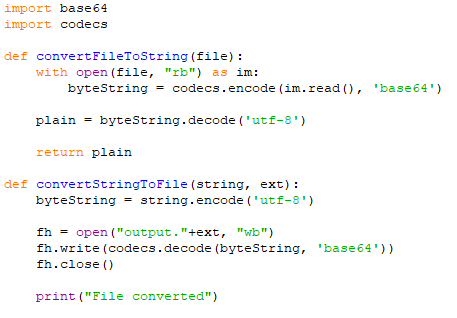


Figure 6: Source code for our own algorithm

# Conversion Code

In order to encrypt non-string files such as images, the file needs to encode to base 64 and decoded from utf-8. This gives am ASCII format, in other words a string which can be encrypted. This process is reversed once the file is decrypted in order to restore the original file.

The following is the source code for the conversion.



# Using the graphical user interface

First make sure that all the necessary python dependencies are installed. This can be done by running the “installs.bat file”. Next open the file called “GUI.py’ in the \_bin directory. This will open the interface for the program.

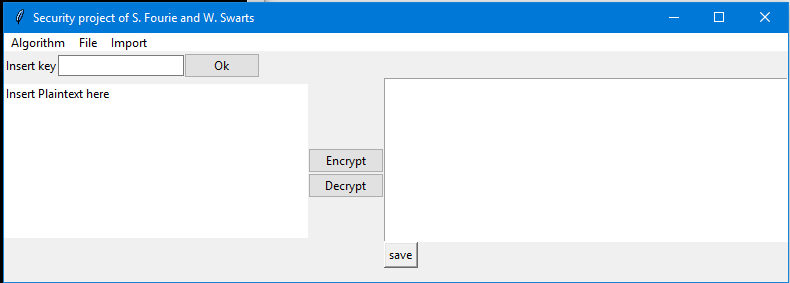


Figure 7: The main menu for the program

Select the algorithm to be used from the “Algorithms” Menu. Next type in the key into the key field and press the “Ok” button. Insert the plaintext into the left input box and click the “Encrypt” button to generate the ciphertext in the right input box. If you are satisfied with the cipher text, press the save button, which will open a file input dialog. Select the directory and click on “save’.

Files can be loaded from the “File” menu, which opens a dialog box in which the file can be selected. Images are also supported by using the Import menu. The encrypted image can then be located in the root directory of the GUI.py file.

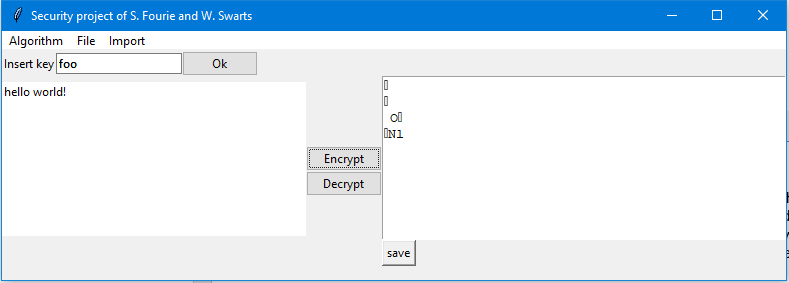


Figure 8: Output generated using the Vernam algorithm

# References