CSC3621 Cryptography - Exercise 3

Aim - To understand the working of a One Time Pad and practice a Two-Time Pad

Introduction:

A One Time Pad is unbreakable when used correctly. A One Time Pad is a completely random key which is used to encrypt a message of the same length.

The One Time Pad is perfectly secure, however it has practical issues.

The technique uses modular arithmetic to encrypt the bits of the message with the key.

The bits are XOR'd then the corresponding bits make the cipher text.

Implementation:

To implement such an algorithm, I first had to understand how the arithmetic worked. I then had to understand how the system reads characters and use Java to manage the process.

Encrypt

My encryption works my taking in a plain text String in ASCII form and a byte array created by placing the hexadecimal key through the hexToByteArray() method taken from Feng Hao's OTPAttack.java file.

The encryption takes the plain text and converts it to bytes and stores in a byte array. The method then completes a check to ensure that the message length is the same as the key. If it is not then it returns an error.

```
public String encrypt(String pText, byte[] key) {
    /*
    *Converts plain text to bytes using getBytes() and takes key as string and converts to byte array
    *(using hexStringToByteArray) function and Xor's the bytes, then rebuilds the string using
    *bytesToHex function.
    */
    final byte[] plain = pText.getBytes();
    final byte[] keyByte = key;

    if (plain.length > keyByte.length) {

        System.err.println("Message length and key length not equal");
        System.exit(0);

    }

    final byte[] encrypted = new byte[plain.length];

    for (int i = 0; i < plain.length; i++) {
        encrypted[i] = (byte) ((plain[i] ^ keyByte[i]));
    }
    String e = bytesToHex(encrypted);

    return e;
}
```

A new array is created if the lengths are equal and the plain text/key is iterated through and each byte is XOR'd. The bytes are then converted to a hexadecimal form using Feng Hao's bytesToHex() function taken from the OTPAttack.java file. It is then stored as a String and returned.

Decrypt

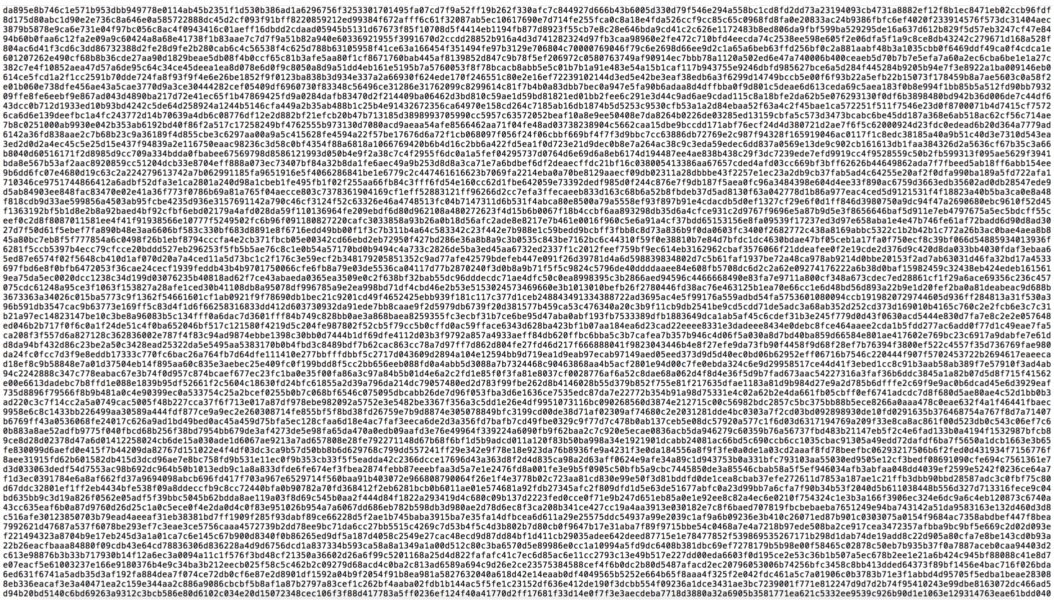
The decryption does the same process however uses the Cipher text to XOR with instead of the plain text.

```java
public String decrypt(String cText, byte[] key) {
    /*
    *Converts cipher text to bytes and takes in key bytes (using hexStringToByteArray) function
    *and Xor's the bytes, then rebuilds string of decrypted text
    *
    */

    final byte[] cipher = hexStringToByteArray(cText);
    final byte[] keyByte = key;
    final byte[] decrypted = new byte[cipher.length];
    if (cipher.length > keyByte.length) {

        System.err.println("Message length and key length not equal");
        System.exit(0);

    }

    for (int i = 0; i < cipher.length; i++) {
        decrypted[i] = (byte) ((cipher[i] ^ keyByte[i]));
    }

    String e = new String(decrypted);

    return e;
}
```

Generator

I wrote into the program a function that allows the creation of a text file of 1mb in size including 500000 characters. This is a One time pad generator and is completely random. The generator is used to generate a secure random one time pad and writes it to a file.

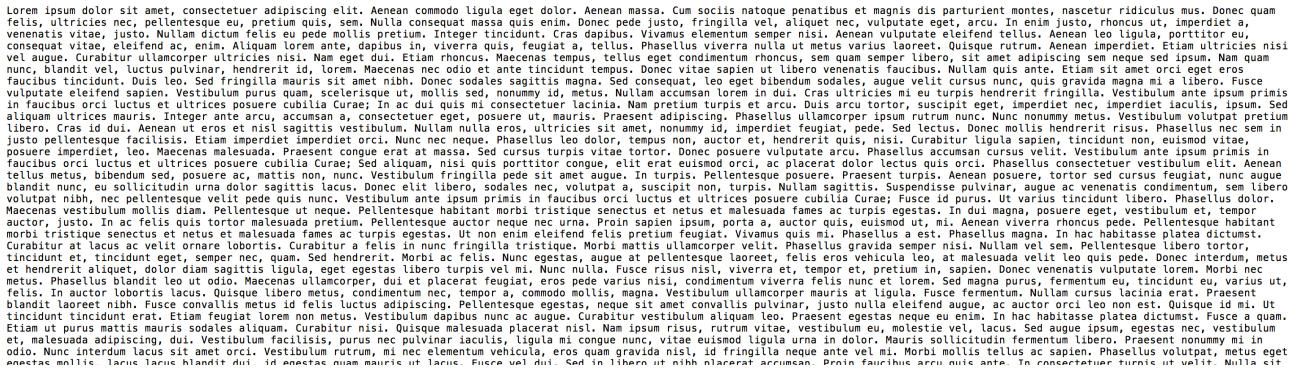
This is to be used for testing purposes and I have used it against a large 500000 character file with latin english. This is to test the ability to encrypt and decrypt large files but also to test the difference when using the algorithm with different distributions to normal english text.



When encrypted the file looks like this, completely random hexadecimals.

The encryption has converted the hexadecimals to bytes and XOR'd them then converted the new bytes back to hex format to generate this.

Decrypted the file is latin english as such:



The distribution unlike the previous two algorithms has very little affect as a frequency is not really important.

Tests

As part of the analysis I wanted to test another piece of text. I used the generator to create a one time pad to be the same length as “Supercalifragilisticexpialidocious”

The One time pad is
“360a4129135a53415310777c642d46ea64377a1f464c7402027f5a34564c6a636e44"

Using this one time pad I was able to encrypt the Plain text above and outputted the following.

This result is the encryption and decryption of the plain text.

```
360a4129135a53415310777c642d46ea64377a1f464c7402027f5a34564c6a636e44
Supercalifragilisticexpialidocious
```

I also tested other plain text and cipher texts into the functions as outputted below:

```
Encryption of plainText: 28b14ab7ecc33ea157b539ea426c5e9def0d81627eed498809c17ef9404cc5

Encryption of plainText3: 360a4129135a53415310777c642d46ea64377a1f464c7402027f5a34564c6a636e44

Decryption of cipherText3: Supercalifragilisticexpialidocious

Decryption of cipherText: Every cloud has a silver lining
```

XOR Spaces

As part of the analysis is was important to test the effects of XORing a space with a character.

I wrote a function to test this and then tried with different cases.

I found that XORing a space with a character of lowercase results in the character in uppercase and vice versa.

The affect is reversible.

```
XORing spaces tests:
A
a
B
b
```

```
System.out.println("XORing spaces tests: ");
otp.xorSpaceTest("a");
otp.xorSpaceTest("A");
otp.xorSpaceTest("b");
otp.xorSpaceTest("B");
System.out.println("\n");
```

These are the results from XORing the characters with a space. This shows that the character case is inverted when XORed with the space.

Two time Pad

Using this knowledge I was able to run the OTPAttack.java and from the result I could see where spaces were in the resulting output because of the common similarities. This meant that I could deduce where some letter were. This has been achieved by XORing each cipher text.

“ - o u - h a v - - d - n - - y - u r - w - - k

The text shown above is what I deduced and from that I can see that the message is most likely “ you have done your work”. This is because where the spaces are assumed the characters are all uppercase and are similar at various points. Therefore XORing the space would make them lower case of the same character leaving us with the message above.

Conclusion

I found this task extremely challenging to program as I am still quite an inexperienced programmer. Because of this I chose to manually do the two time pad attack using Feng Hao’s OTPAttack.java. This allowed me to see the result and piece together manually the result. I understand this is not programmed however I understand the theory behind the algorithm and understand that it is not possible to retrieve the whole message as there are some spaces where there are too many possibilities which would need to be brute force attacked and on a large scale this would be extremely difficult to complete if not “impossible”.

References

OTPAttack.java Feng Hao