

Assessed Coursework 1

SYMMETRIC ENCRYPTION

Jacob Cooper |H00251723 |Computer Network Security | 21/10/19

# Introduction

This is the first coursework for Computer Network Security (F20CN) on Symmetric Encryption. All work presented here is my own and any that isn’t has been given proper accreditation. Throughout this coursework I hope to understand the various methods used in Symmetric Encryption and how they are implemented so that I can have a greater understanding of how they work. Specifically I’m hoping it will deepen my understanding of the more complicated encryptions such as AES – 128 cipher and its modes of operation as well selecting a safe and robust IV for said encryptions. For task one I hope to understand the use of frequency analysis and how it shows Monoalphabetic Substitution Ciphers to be extremely vulnerable. For task 2 I hope to understand the application of a multitude of different ciphers and how effective they are. Task 3 seems to be the most complicated as such I’m hoping to learn a lot from it: I hope gain a further understanding of how AES-128 cipher and all its modes work as well as how they differ and which are more safe; I also hope to learn how a single-bit error affects the decryption of these encryptions; I would also like to learn about implementation of IV. For task four I would like to continue this learning about IV’s and how they play an important part in encryption integrity and how an attacker can exploit a poorly chosen IV. For task 5 I hope to understand how a brute force known-plain-text attack can be carried out (provided that a certain level of information is fulfilled) as well as how effective this strategy is.

# 2. Task 1

### 2.1 Conventions

One convention I did use was representing the cipher text with lowercase and the plain-text with upper case. This is to make it clear when deciphering which letters have been substituted in and are a part of the deciphered plain text and which haven't been substituted in and are a part of the original cipher text.

### 2.2 Frequency analysis

Using the first resource provided ( <http://www.richkni.co.uk/php/crypta/freq.php> ) I inputted the cipher text I was given into the resource and it provided me with the frequency of each of the characters in the cipher

Using the second resource ( <https://en.wikipedia.org/wiki/Frequency_analysis> ) its clear that ‘e’ is the most frequent letter in most English plain-text and ‘h’ is the most frequent letter in the cipher text so I hypothesize that ‘h’ is actually ‘e’. This is backed up by ‘the’ being a common trigram in English ( <https://en.wikipedia.org/wiki/Trigram> ) plain-text and ‘bph’ being the most common trigram in the cipher text. So we can say that b=T , p = H and h=E .

### 2.3 Substitution

To substitute the predicted letters into the cipher text I used the command

bash-4.2$ tr bpe THE

which is the command I will be using every time I substitute letters into the cipher. When ‘T’ ,‘H’ and ‘E’ were subbed in for ‘b’ , ‘p’ and ‘h’ there was no apparent problems and it opened up more possible exploration.

### 2.4 Completing the words

From this start we got from frequency analysis we can continue on to decode the whole text we do this by completing words such as ‘THjEE’ that appear and substituting the obvious missing letter (in this case ‘j’ must be ‘R’ ) into the cipher text. I then repeated this process until I had the full decrypted plain text (See Appendix 1) using this key:

abcdefghijklmnopqrstuvwxyz

XTVZKOCEARMSFWJHYGLUPQOBIN

In the command like so:

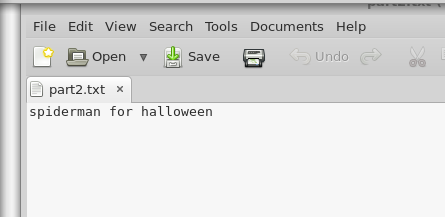
bash-4.2$ tr abcdefghijklmnopqrstuvwxyz XTVZKOCEARMSFWJHYGLUPQOBIN

# 3. Task 2

The three ciphers I used were AES(Advanced Encryption Standard) 128 bit, Blowfish and DES (Data Encryption Standard). The three modes I used were ECB(Electronic Codebook) , CBC (Cipherblock Chaining) and CFB(Cipher Feedback) on each of the ciphers.

Each of my screenshots follow the same pattern of showing the terminal with the command that has run on the left and the output in ‘cipher.txt’ on the right.

For each of the encryptions I used the following as the plain-text ( I named it “part2.txt”)



### 3.1 AES-128-CBC

A screenshot of a social media post

Description automatically generated

Here is the encryption using AES-128 ( indicated by the ‘-aes-128’ part of the above command) in the CBC mode (indicated by the ‘-cbc’ part of the command directly after the ‘-aes-128’ part)

### 3.2 AES-128-CFB

A screenshot of a social media post

Description automatically generated

Here is the encryption using AES-128 (indicated by the ‘-aes-128’ part of the above command) in CFB mode (indicated by the ‘-cfb’ part of the command directly after the ‘-aes-128’ part)

### 3.3 AES-128-ECB

A screenshot of a social media post

Description automatically generated

Here is the encryption using AES-128 (indicated by the ‘-aes-128’ part of the above command) in ECB mode (indicated by the ‘-ecb’ part of the command directly after the ‘-aes-128’ part)

### 3.4 Blowfish-CBC

A screenshot of a cell phone

Description automatically generated

Here is the encryption using Blowfish(indicated by the ‘-bf’ part of the above command) in CBC mode (indicated by the ‘-cbc’ part of the command directly after the ‘-bf’ part)

### 3.5 Blowfish-CFB

A screenshot of a social media post

Description automatically generated

Here is the encryption using Blowfish(indicated by the ‘-bf’ part of the above command) in CFB mode (indicated by the ‘-cfb’ part of the command directly after the ‘-bf’ part)

### 3.6 Blowfish-ECB

A screenshot of a cell phone

Description automatically generated

Here is the encryption using Blowfish(indicated by the ‘-bf’ part of the above command) in ECB mode (indicated by the ‘-ecb’ part of the command directly after the ‘-bf’ part)

### 3.7 DES-CBC

A screenshot of a cell phone

Description automatically generated

Here is the encryption using DES(indicated by the ‘-des’ part of the above command) in CBC mode (indicated by the ‘-cbc’ part of the command directly after the ‘-des’ part)

### 3.8 DES-CFB

A screenshot of a cell phone

Description automatically generated

Here is the encryption using DES(indicated by the ‘-des’ part of the above command) in CFB mode (indicated by the ‘-cfb’ part of the command directly after the ‘-des’ part)

### 3.9 DES-ECB

A screenshot of a cell phone

Description automatically generated

Here is the encryption using DES(indicated by the ‘-des’ part of the above command) in ECB mode (indicated by the ‘-ecb’ part of the command directly after the ‘-des’ part)

# 4. Task 3

## 4.1 Hypothesis

### 4.1.1 ECB

For ECB I hypothesize that all but the block in which the corrupt bit is held will be recoverable. This is because of the formula:

cj = Ek(xj)

This means that each block of plaintext is enciphered independently and so since there is no link between the enciphering of each block of plaintext then only the original corrupt block will be irrecoverable

### 4.1.2 CBC

For CBC I hypothesize that the corrupted block and all blocks after it will be Irrecoverable. This is because each cipher-text block uses the previous cipher-text block in its encryption and decryption as in the formula:

c0:cj = Ek(xj XOR cj-1)

In the task the 4th block is corrupted and so the 5th block would have to use this corrupted block in its decryption and so it would also be affected and then the 6th one will be affected because of this and so on meaning everything past the 3rd block will be irrecoverable.

### 4.1.3 CFB

For CFB I hypothesize that it will have the exact same problem as CBC. This is because of the formula:

Cj = Ek(cj-1) XOR xj

Like the example before the 5th block will be affected by the corrupted 4th block and the 6th block will be affected by the 5th block and so on, meaning that all after the 3rd block will be irrecoverable

### 4.1.4 OFB

For OFB I hypothesize that only the one bit with the corruption will be irrecoverable. This is because of the formula:

cj = Oj XOR xj

Oj = Ej(Oj-1)

Since the ecryption function is not applied directly to the plain-text instead its applied to a vector this means that only the one changed bit will affect only one vector which in turn will only affect one bit on decryption.

## 4.2 Encoding

I used the AES-128 cipher in 4 different modes(ECB, CBC,CFB, OFB), with an initial text called ‘part3.txt’ included in Appendix 2. I will only show the encoding of the file and not the output as this will be shown in section 4.3.

### 4.2.1 ECB

A screenshot of a computer

Description automatically generated

Here I am running the command in ECB mode and storing the output in ecb.bin.

### 4.2.2 CBC

A screenshot of a computer

Description automatically generated

Here I am running the command in CBC mode and storing the output in cbc.bin.

### 4.2.3 CFB

A screenshot of a computer

Description automatically generated

Here I am running the command in CFB mode and storing the output in cfb.bin.

### 4.2.4 OFB

A screenshot of a computer

Description automatically generated

Here I am running the command in OFB mode and storing the output in ofb.bin.

## 4.3 Editing files

I will show the output of the above commands before and after the 55th byte has been edited to have only a 1 bit error. I did this using the GHex software on the linux software, I took the hex value given for the 55th byte and converted to binary using the hex to binary converter found here : <https://www.binaryhexconverter.com/hex-to-binary-converter>. Then to change it, I selected a hex value next to the one in the 55th byte ( e.g. if it was ‘7C’ I would select either ‘7B’ or ‘7D’)we’ll call this ‘x’. When selecting ‘x’ its equivalent binary has to be one bit different to the binary value for the hex value of the 55th byte( E.g. if the 55th byte value were ‘07’ I couldn’t choose ‘08’ for ‘x’ as it would change 4 bits but I could choose ‘06’ as it only changes 1 bit).

### 4.3.1 ECB

A screenshot of a social media post

Description automatically generated

This is the unedited file of the ECB mode encryption, the 55th byte, as you can see, has a hex value of ‘06’ which converts to ‘0000 0110’ in binary

A screenshot of a social media post

Description automatically generated

This is the ECB mode encryption file after editing I changed the hex value from ‘06’ to ‘07’ as the difference is only the 8th bit (‘0000 0110’ compared to ‘0000 0111’)

### 4.3.2 CBC

A screenshot of a social media post

Description automatically generated

This is the unedited file of the CBC mode encryption, the 55th byte, as you can see, has a hex value of ‘9E’ which converts to ‘1001 1110’ in binary

A screenshot of a social media post

Description automatically generated

This is the CBC mode encryption file after editing I changed the hex value from ‘9E’ to ‘9F’ as the difference is only the 8th bit (‘1001 1110’ compared to ‘1001 1111’)

### 4.3.3 CFB

A screenshot of a social media post

Description automatically generated

This is the unedited file of the CFB mode encryption, the 55th byte, as you can see, has a hex value of ‘52’ which converts to ‘0101 0010’ in binary

A screenshot of a social media post

Description automatically generated

This is the CFB mode encryption file after editing I changed the hex value from ‘52’ to ‘53’ as the difference is only the 8th bit (‘0101 0010’ compared to ‘0101 0011’)

### 4.3.4 OFB

A screenshot of a social media post

Description automatically generated

This is the unedited file of the OFB mode encryption, the 55th byte, as you can see, has a hex value of ‘7D’ which converts to ‘0111 1101’ in binary

A screenshot of a social media post

Description automatically generated

This is the OFB mode encryption file after editing I changed the hex value from ‘7D’ to ‘7C’ as the difference is only the 8th bit (‘111 1101’ compared to ‘0111 1100’)

## 4.4 Decoding

### 4.4.1 ECB

### 4.4.2 CBC

### 4.4.3 CFB

### 4.4.4 OFB

## 4.5 Evaluation

### 4.5.1 ECB

### 4.5.2 CBC

### 4.5.3 CFB

### 4.5.4 OFB

# 5. Task 4

## 5.1 The Maths

To start we need to look at the OFB equation which is:

Cj = Oj XOR xj

Oj = Ek(Oj−1)

Where ‘j’ is the block number, ‘c’ is the cipher text, ‘x’ is the plain text , ‘k’ is the key and ‘E’ is the encryption used.

From this we can say that a full cipher (c ) of an entire plaintext (p) is

c = O XOR p

Where for every block ‘j’

Oj = Ek(Oj−1)

We can rearrange the former equation to get this:

p = O XOR c

So since we are trying to figure out the value of the second plaintext(p(2)) from the value of its cipher (c(2)) we can put it into the equation to get:

p(2) = O(2) XOR c(2)

Note that we don’t use subscript 2 here as we aren’t referring to blocks , instead we use (2) as we are referring to entire plaintext and entire cipher text.

We know c(2) so to figure out p(2) we need to know O(2) .

We know a different cipher text (c(1)) and its deciphered plaintext (p(1)) so we can figure out O(1) like so by rearranging the above equation to get:

O(1) = p(1) XOR c(1)

Since the keys and IV’s were the same for both these ciphers, it is true that O(1) equals O(2) and as such we can substitute the equation for O(1) in the place of O(2) like so:

p(2) = O(2) XOR c(2)

O(1) = p(1) XOR c(1)

O(2) =O(1)

Therefore:

p(2) = (p(1) XOR c(1)) XOR c(2)

## 5.2 Using the Equation

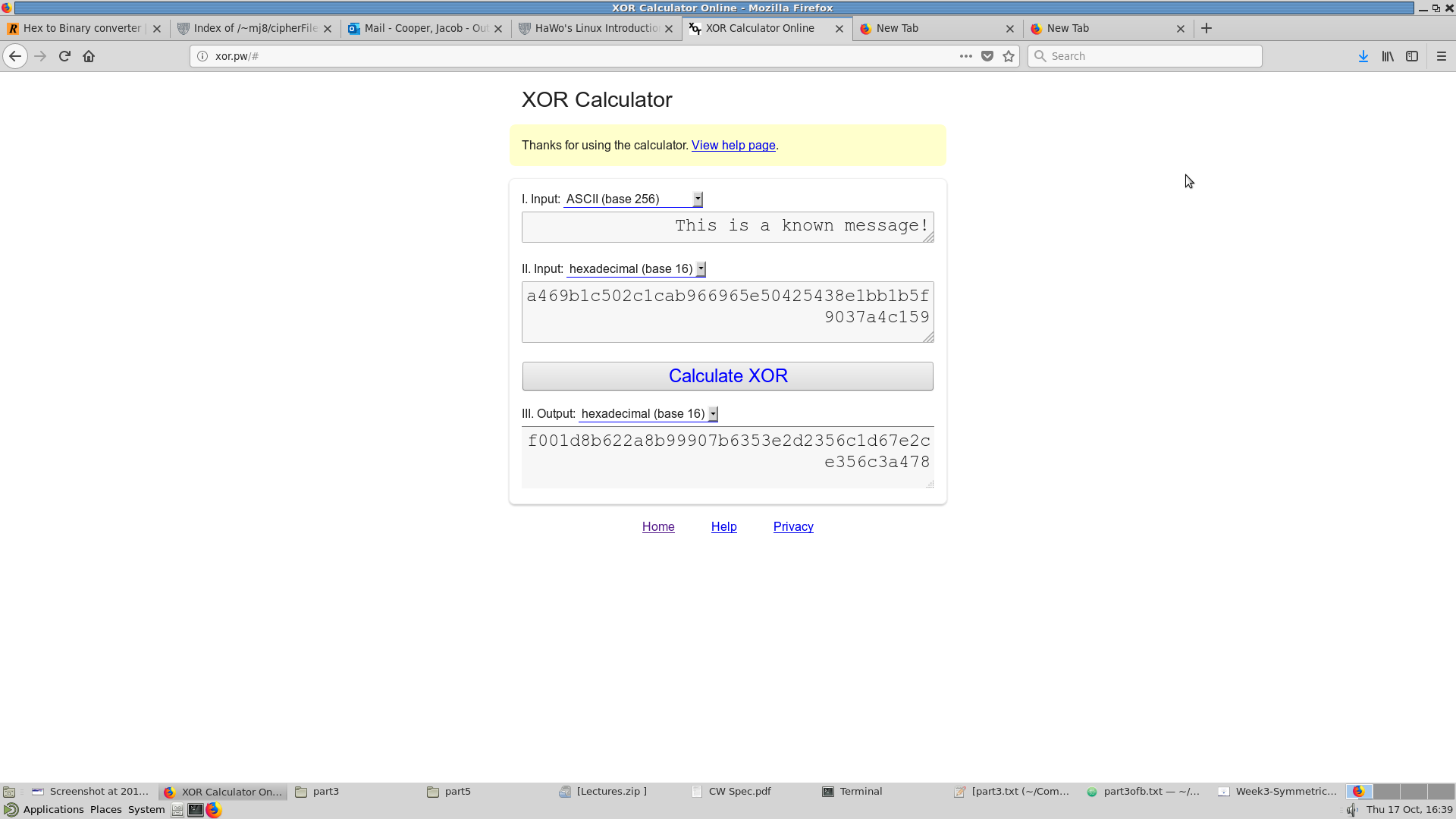
Now all we need to do is calculate p(1) XOR c(1) and XOR the result of that with c(2). We can do this by using the link provided (

p(1) = This is a known message!

c(1) = a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

c(2) = bf73bcd3509299d566c35b5d450337e1bb175f903fafc159

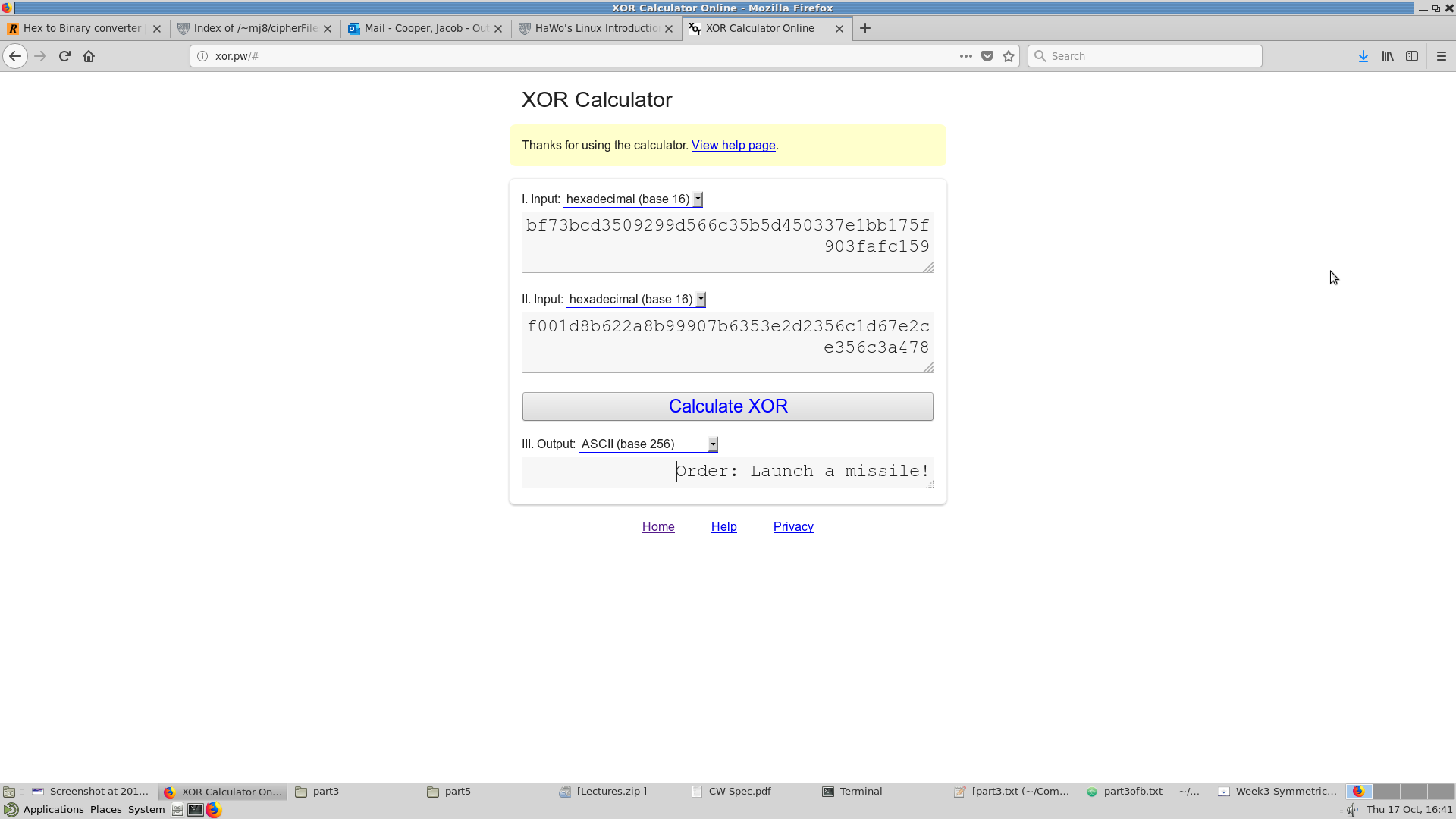
### 5.2.1 Finding p(1) XOR c(1)



Here you can see p(1) XOR c(1) being calculated and output in the bottom box which we will use for the next part

### 5.2.2 Finding p(2)

Now that we have p(1) XOR c(1) and we know c(2) all we need to do is XOR them together to get p(2)



Here we can see p(2) being calculated by XORing c(2) (seen in the top box) with the result from p(1) XOR c(2) (seen in the middle box) . From this we can see that:

p(2) = Order: Launch a missile!

# 6. Task 5

I used the file line-count.sh that we were provided as a base to start. I cut most of the code from the file as it wasn’t necessary but what I did use was the while loop which loops through and reads each line of a given input file, I also kept the ‘exec’ commands and their closing , as well as the count variable. Other than that, the rest of the code was written by me. You can see my script in Appendix 3.

First off I started by adding 3 arguments to my script naming them ‘dictionary’ , ‘cipher’ and ‘plain’ respectively. These will be used for inputting the dictionary , cipher and plaintext into the script.

Then in the while loop I added an openssl command that encrypted the file in the ‘plain’ variable with the word currently being read from the file in the ‘dictionary’ variable and then outputs this to a file called out.txt (this file will be made by this command if not already present).

# 7. Summary