Name: Salah Mahamod

Student ID: 20152428
Unit: ISEC2000 Fundamental Concepts of Cryptography
Date: 22/04/2022
Assessment: Assignment 1, weighs 25% of the final mark

Assignment 1 Report

Table of Contents

Question 1 – Attack a cipher

How did I implement the Letter Analysis Attack?

Step-by-step implementation

- I read the cipher text file that was part of the command line arguments to execute the python script.
- I created a file, that has the same name as the file obtained from the command line, but it has "LA_DECRYPTED_" as a prefix, this will have the letters substituted post letter frequency analysis.
- I read the cipher text file, character by character and store the number of occurrences of characters A – Z and a – z, only.
- The frequency of letters (as a percentage, relative to other letters) is outputted to the screen.
- The letters are arranged by their frequency.
- The file is re-read as capital letters, and the characters are now substituted with letters that much that rank.
  For e.g., the letter 'O' occurs the most in plain English after 'E', 'T' and 'A' so the fourth most occurring letter in the cipher text is replaced with capital 'O' and written to the newly created file for the recovered plain text.

Manual Post-Processing of the decrypted cipher.txt file

The letter 'V' is the most occurring at 12.736% so it is safe to assume it is 'E'.

The first line has the word "lack" so it's safe to assume that this corresponds with "qjpn" as well.

There's a line that says "24 rouni", the assumption is that "rouni" should be "hours" as that's what would usually follow 24, hence; O and U are substituted correctly and H(≠R), R(≠N) and S(≠I) aren't: "hours" = "ezril"

Following "24 rouni" there's "a dag" which obviously means "a day" 'D' = 'Y' and 'D' ≠ 'G'.

There are a lot of words that resemble "to" and when this is substituted into the first word "oev" -> "the", when using the values, we have already. So "oz" = "to".

There is an "eveh" which should correspond to 'even', so the 'w' was incorrectly subbed to 'h' when it should've been an 'n'.

Firstline contains 'hontrenh' when this is corrected with what I have so far -> 'northern', which proves that.

D is subbed correctly as I saw the phrase "wet cold" which makes sense, therefore 's' = 'd'.

There is a line with "140 dewneei betpeeh" which sounds like "140 degrees between", this is true from looking at what I have gathered so far. So, 'x' ≠ 'p' and 'x' ≠ 'w'.

That's all I can manually post-process without having to automate the substitution of the corrected values. Top row contains cipher text characters, second row illustrates their correct substitution, the third row shows what the incorrect substitution was.

V	Q	J	P	N	E	Z	R	I	L	D	O	M	W	S	X										
E	L	A	C	K	H	O	U	R	S	Y	T	B	N	D	W										
					R			N	I	G			H		P										

The substitution table obtained by letter frequency analysis

Letter	**Frequency in plain English**	**Present in cipher.txt**
E	12.60 %	4.692%
T	9.37 %	2.311 %
A	8.34 %	0.097 %
O	7.70 %	8.583 %
N	6.80 %	0.824%
I	6.71 %	7.37 %
H	6.11 %	6.821 %
S	6.11 %	3.313 %
R	5.68 %	2.796 %
L	4.24 %	7.273 %
D	4.14 %	1.697 %
U	2.85 %	1.196 %
C	2.73 %	1.891 %
M	2.53 %	1.681 %
W	2.34 %	6.594 %
Y	2.04 %	2.538 %
F	2.03 %	0.178 %
G	1.92 %	0.259%
P	1.66 %	2.651 %
B	1.54 %	2.344 %
V	1.06 %	12.736 %
K	0.87 %	0.113 %
J	0.23 %	8.308 %
X	0.20 %	1.681 %
Q	0.09 %	4.364 %
Z	0.06 %	7.419 %

Check references for source.

How did I implement the Brute-Force Attack?

Step-by-step implementation

- I read the cipher text file that was part of the command line arguments to execute the python script and stored it as a string.
- I created a file, that has the same name as the file obtained from the command line, but it has "BF_DECRYPTED_" as a prefix, this will have the decrypted 'recovered' plaintext for every combination for 'a' and 'b' keys, which are parameters for the affine cipher.
- The 'b' parameter is obtained from a normal loop that loops 26 times, since this is the affine cipher, the 'a' parameter needs to have a common divisor of 1 with 26 (as this will indicate whether the inverse of this encryption i.e., decryption can be applied), only 12 numbers (1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23 and 25) can be used for the 'a' parameter. A loop will iterate over these values, which inside the 26 times loop. These parameters will be passed on along with the string containing cipher text to the decrypt function.
- The ***decrypt*** function decrypts the ciphertext using the 'a' and 'b' arguments as parameters for the affine cipher. The decimal value of each character in between A-Z and a-z is subtracted by 65 or 97, multiplied by the 'a' parameter and then subtracted by 'b', the remainder of this when it is divided by 26 is then added to 65 or 97. The 65 or 97 (its use depends on whether the character is between A-Z or a-z) is subtracted so the decimal value is now in between 0 and 26 and is added so the decimal value is in between 65 and 90 or 97 and 122. Each decrypted character is appended to a string containing the other decrypted characters and is returned when there are no more characters to be decrypted left
- The loops that call the ***decrypt*** function annotate each new loop in the 'recovered' plaintext file, indicating which 'a' and 'b' parameters are used for readability.

The key found by the brute-force attack

The keys are **a = 9** and **b = 3**.

Question 2 – Implement Data Encryption Standard (DES)

General overview of my implementation

- User is asked to input a key, this key will be chopped or padded (equal padding strategy) to a 64-bit key.
- This key is permuted using the permuted choice 1 (PC -1) permutation resulting in a 56-bit key. The ***permutation*** function handles all permutations regardless of length by taking the bits (as a string), the permutation table array and the length of the resulting permutation as arguments. It will re-arrange the string using the order provided in the permutation table array.
- File is read using the ***readFileAsBinary*** function, which returns the entire file as bits into a string.
- The ***encypt*** function implements DES encryption, takes the 56-bit key and file as a binary string as arguments, and the resulting bits are converted to hexadecimals using the ***binaryToHex*** function. The ***encrypt*** function does the following:
  - Uses PKCS5 padding, i.e., the remaining bytes are padded by using the value of the remaining bytes in hexadecimal, convert them to binary and add them a number of times depending on the initial value. So, if there were 6 bytes out of 8, the remaining 2 bytes are padded by converting 02 hexadecimal to binary -> 00000010 and adds this 2 times. If the last block has 64 bits, then 8 bytes containing 8 repeated as binary are appended.
  - Each 64 bits of the plain text are treated as blocks and are permuted using the initial permutation table, put through the feistal network with the use of ***feistalNetwork*** function, and the result is permuted again using the final permutation (which the inverse of the initial permutation), these encrypted blocks are appended to one another and returned.
- The ***feistalNetwork*** function puts the 64-bit block and 56-bit key through 16 rounds (loops) of DES and does the following:
  - For each loop it calls the **scheduleKey** function, which takes the latest key and round number as arguments and does a left shift on each of the two halves of the latest key in **leftShift** function 1 or 2 times depending on the round number. For rounds 1, 2, 9 and 16 it shifts each of the halves leftmost bit to the right and for the others the two leftmost are shifted.
  - Key for the round obtained from ***scheduleKey*** is now permuted using permuted choice 2 (PC – 2) permutation, which reduces the keys length from 56 bits to 48 bits.
  - The 64-bit block is split into two halves and the right is now passed to the ***fFunction*** function as an argument alongside the permuted key.
  - The resulting 32 bits are XORed (using the ***<u>xor</u>*** function) with the left, and this is now set as the new 'right' part of the final block.
  - The initial 32 bits on the right of the initial block are set as the first 32 bits (i.e., to the left) of the final block, the final 64-bit block is returned to ***encrypt***.
- The **fFunction** function which takes the right side of the 64-bit block and the 48-bit key does the following:
  - Expands the 32-bit right side of the block to 48 bits using the expansion permutation and XORs it with key.
  - The result is passed on to the ***sbox*** function which reduces the length from 48 bits to 32.
  - The 32-bit result is permuted using the f function permutation table and the resulting 32 bits are returned to ***fiestalNetwork***.
- The ***<u>sbox</u>*** function which reduces the 48 bits to 32 bits by:
  - Using each 6 bits to get 4 bits by using the first and last bit, convert them to decimal from binary and used to access row of the sbox table. Similarly, the 2nd to 5th bits are converted to decimal and used to access the column of the sbox table. There are 8, 6-bit iterations and each iteration have their own sbox table.

- These 6 bits will point to 1 out of 8 sbox tables, and the decimal value accessed using the acquired 'row' and 'column' is converted to a binary string of 8 bits.
- The 6 bits converted to 8 bits in every iteration are appended to one another and the resulting 48 bits are returned to ***fFunction***.

- The ***decrypt*** function used for decryption works similar to the ***encrypt*** function, however it has its own **decFeistalNetwork**, **decScheduleKey** and **rightShift** functions which work similarly to their counterpart functions however their differences are:
  - The ***rightShift*** function shifts the rightmost bits i.e., 0, 1 or 2 bits, instead of 1 or 2 for **leftShift** to the left (this is the reversal of left shift) depending on the round. This is because the 16th key generated by ***scheduleKey*** is the 1st key generated by ***decScheduleKey***, so round 1's key shouldn't be right shifted.
  - The ***decScheduleKey*** calls the ***rightShift*** function instead of ***leftShift*** as the reverse key generation used in encryption is necessary.
  - The ***decFeistalNetwork*** calls the ***decScheduleKey*** function instead of ***scheduleKey***.
- Finally, the string stored from reading the file is encrypted using the 56-bit key and the resulting ciphertext in binary is converted to hexadecimals (for readability) using ***binaryToHex*** function and stored in a text file with the prefix of "ENCRYPTED_HEX_".
- For decryption this ciphertext file that had just been created is read, the hexadecimals are converted to bits using the ***hexToBinary*** function and are passed on to the ***decrypt*** function along with the 56-bit key.
- And again, the resulting bits are converted to back to characters using the ***binaryToString*** function and stored in a text file with the prefix of "RECOVERED_".

Answers to questions in the Assignment Specification

(a) Have you successfully recovered the plaintext? What are the lessons you learned, and difficulties you met, in the process of implementing DES?

Yes, I have successfully recovered the plaintext.

I have learnt a lot about coding in python as I have been learning as I went along. This is evident in my use of functions in the DES.py script vs. the 0 functions used in LetterAnalysis.py.

I have consolidated my understanding on DES as grasping every piece was critical in implementing it. I didn't know much about decryption, just assumed it to be the 'reverse' of encryption but after reading the Understanding Cryptography book by Christof Paar, I now, appreciate how DES was designed. The key used in the f function for round number X for decryption must be the same as the used in round number 16-X, which makes sense when you look at it symmetrically.

Surprisingly, I learnt more about the conversion from decimal -> binary -> hexadecimal and how to automate its computation.

Furthermore, I also learnt a lot about software design and testing. One of the difficulties I faced was after implementing the key scheduling, f function, the fiestal networks and the xor operation, the recovered text file couldn't be opened as the characters weren't supported. I had to test each component separately, as I was dealing with bits! There's no way to know for certain where the program failed, so print statements were useless. I overcame this by, dividing and conquering, making more functions that had a simpler task and consolidating my understanding on DES.

(b) What will happen if the key is initialised as all 0-bits?

The file would still be encrypted. However, this will remove the confusion operation from DES. This is because the key, which is used when XORing the 48-bit extended right-side of the block in the fiestal round. The key of all 0-bits would not provide any confusion here as XORing any bit with 0 preserves its initial value. So, it would be plain diffusion.

Furthermore, key scheduling would be useless as left shifting would be identical, which also means that decryption would literally be the same as encrypting the file twice, as the art of decryption was the reversal of key generation, but with all 0 bits the encryption algorithm can invert itself.

References

Code with B. 2020. *DES Encryption padding scheme*. YouTube video, 08:48. https://www.youtube.com/watch?v=JdM1FVckObU

Trost Stefan. 2017. *Alphabet and Character Frequency: English.* Stefan Trost Media. https://www.sttmedia.com/characterfrequency-english

Paar, Christof. and Pelzl, Jan. 2009. *Understanding Cryptography*. Leuven: Springer