HY-457
Assignment 1

Assigned: 12/3
Tutorial: 12/3
Due: 9/4

Office Hours: 16/3, 18/3, 23/3, 20/3, 27/3, 1/4, 6/4, 8/4 14:00-16:00 at https://bbb.tbm.tudelft.nl/b/eva-fbi-q3j-fp2

Introduction

In this assignment you are going to implement from scratch, using C, a simple cryptographic library named cs457_crypto. The cryptographic library will provide four basic but fundamental cryptographic algorithms, (i) One-time pad, (ii) Caesar's cipher, (iii) Playfair cipher, (iv) Affine cipher and (v) Feistel cipher. The main purpose of this assignment is to offer you the opportunity to get familiar with the implementation and internals of such simple ciphers and help you understand the development decisions and tricks that developers have to deal with when implementing security critical functions that, at first, seem trivial to develop.

The cs457_crypto library will consist of two files. The cs457_crypto.h, which contains the C function declarations and any macro definitions you think are important and the cs457_crypto.c file, containing the implementation of the above algorithms.

One-time pad

The One-Time-Pad (OTP) algorithm is a very simple but yet very strong algorithm in the sense that it can not be cracked even with quantum computers. The algorithm pairs each plaintext with a random secret key (also referred to as a one-time pad). Specifically, each bit or byte/character of the plaintext is combined with the corresponding bit or byte/character from the random secret key. One-time pad requires that the secret key is of the same size or longer than the plaintext.

Implementation details:
In order to generate a random secret key you will use a pseudorandom generator, such as /dev/urandom. The pseudorandom generator will read N random characters from /dev/urandom, where N is the number of bytes/characters found in the plaintext. Then, the algorithm will encrypt each byte/character of the plaintext by XOR-ing it with the corresponding byte/character of the random secret key.

Since /dev/urandom will return a new random value upon each read, you will first need to generate an appropriate sized random secret key and store it in memory in order to successfully decrypt the encrypted message. For this functionality you can develop your own separate function or macro. Also, since the usage of /dev/urandom is our suggested pseudorandom generator, you are advised to use a Linux-based system for the development and testing of the OTP algorithm. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as the random secret key and should return the result of the operation. Special characters, such as “!”, “@”, “*”, etc. that are not part of the english alphabet should be skipped as if the character set only consists of numbers 0-9 followed by uppercase characters A-Z and lowercase characters a-z. The same applies for all the rest of the printable and non-printable ASCII characters such as “\n”, “\t”, “\0” etc. Notice that XOR-ing specific characters together might result in non-printable characters or even “\0”. For this reason you should think around this problem when handling and printing any results.

Caesar's cipher

This technique is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each byte/character of the plaintext is replaced by a byte/character found at some fixed number of positions down the alphabet/ASCII set. For example, given the ASCII character set, a shift of 3 will replace the letter “A” of the plaintext with the letter “D” at the ciphertext. Also, a shift of 4 will encrypt the plaintext “hello” as “lipps”. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as the random secret key and should return the result of the operation.

Implementation details:

The implementation should support numbers, uppercase and lowercase characters. Special characters, such as “!”, “@”, “*”, etc. that are not part of the english alphabet should be skipped as if the character set only consists of numbers 0-9 followed by uppercase characters A-Z and lowercase characters a-z. The same applies for all the rest of the printable and non-printable ASCII characters such as “\n”, “\t”, “\0” etc. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as a positive number indicating the number of shifted positions and should return the result of the operation.

Playfair cipher

The Playfair cipher encrypts pairs of letters (digraphs), instead of single letters as is the case with simpler substitution ciphers such as the Caesar Cipher. The key element of this cipher is a 5x5 grid that represents the key. In order to create the table, the letters of the key are placed in the grid, from left to right beginning from the first row. Then the rest of the alphabet's letters are inserted in the grid alphabetically. Each letter is placed once in the grid. Since the 5x5 grid can only hold 25 characters J is usually omitted and treated as 'I'. For example the key grid for the pass phrase "HELLO WORLD" will create the following key grid[1]:

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

During the encryption the plaintext is broken in groups of two letters. If the letters of the group are the same the second letter is replaced with 'X'. If the number of the letters in the plaintext is odd, the last letters are grouped with an 'X' character. For example, "WILL ATTACK AT DAWN" will result in:

"WI L**X** AT TA CK AT DA WN"

For each group of letters:

If the letters appear on the same row of the key grid, they will be replaced with the letters to their immediate right respectively (wrapping around to the left side of the row if a letter in the original pair was on the right side of the row). For example, **"DA"** will be encrypted to "**AB**".

If the letters appear on the same column of the key grid, they will be replaced with the letters immediately below respectively (wrapping around to the top side of the column if a letter in the original pair was on the bottom side of the column). For example, "**LX**" will be encrypted to "**AL**".

If the letters are not on the same row or column, they will be replaced with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original

[1] Note that each letter in the key is used only once

pair. The order is important – the first letter of the encrypted pair is the one that lies on the same row as the first letter of the plaintext pair. For example "**WI**" will be encrypted to "**LM**", "**AT**" to "**CQ**", "**CK**" to "**BM**", etc.

The ciphertext of the above scenario will be, "LM AL CQ QC BM CQ AB HT".
In order to decrypt the message the *inverse* of the encryption rules is used.

Implementation details

The characters that can be found in the alphabet should only be the uppercase characters A-Z, thus lowercase characters a-z, digits 0-9 or any other ASCII characters should not be used. You should omit character 'J' from the key grid and replace any occurence in the plaintext with character 'I'. If the number of letters in the plaintext is odd, the last letter of the plaintext must be grouped with an 'X' character. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as the keyphrase and should return the result of the operation.

Affine cipher

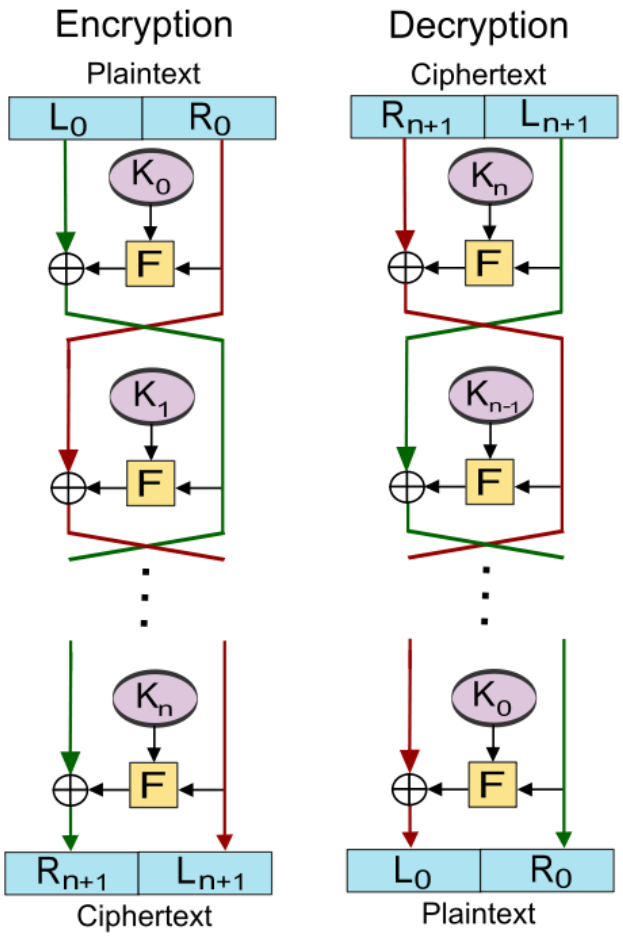
The affine cipher is a type of monoalphabetic substitution cipher where each letter is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter. Each letter is encrypted with the function f(x) = (ax + b) mod m, where "a" is a constant, "b" is the magnitude of the shift and "x" is the letter to encrypt. The formula used, means that each letter encrypts to a single letter, thus, the cipher is essentially a standard substitution cipher with a rule governing which letter goes to which.
In order to decipher a letter affine cipher uses a function in the form of D(x) = a^-1 * (x - b) mod m, where a^-1 is the modular multiplicative inverse of a mod m. The multiplicative inverse of *a* only exists if *a* and *m* are coprime. Hence without the restriction on *a*, decryption might not be possible. The letter x denotes the encrypted letter.

Implementation details
The characters that can be found in the alphabet should only be the uppercase characters A-Z, thus all lowercase characters should be converted into uppercase. Digits [0-9] and special characters should be omitted before the encryption scheme.
For the cipher function you should use the values: a = 11, b = 19 and m = 26.

Feistel cipher

In cryptography a Feistel cipher is a symmetric structure used in the construction of block ciphers. In a Feistel cipher, encryption and decryption are very similar operations, and both

consist of iteratively running a function called a "round function" a fixed number of times. A Feistel network uses a *round function*, a function which takes two inputs, a data block and a subkey, and returns one output the same size as the data block. In each round, the round function is run on half of the data to be encrypted and its output is XORed with the other half of the data. This is repeated a fixed number of times, and the final output is the encrypted data.



In the figure above you see the encryption and decryption operations. In order to encrypt a plaintext we first separate it in blocks of size S. Each block is then splitted in half. We encrypt the left block by XOR-ing it with the output of the round function in each step. We do not apply any kind of operations in the right block. Lastly we reverse the order of the blocks (meaning the left goes right and the right goes left) and we repeat for n iterations.
Decryption works in the opposite way.

Implementation details

For your implementation you should use a block of 64 bits (S = 64) and split the block into two equal blocks of 32 bits. In each round you should generate a pseudo-random key and supply it as one of the two parameters of the round function. The round function is in the form of **F(K_i, R_i) = (R_i * K_i) mod (2^32)**. “K_i” refers to the key generated in iteration “i” and “R_i” refers

to the right block of the current iteration “i”. In order to decrypt the cipher you need to use the same keys you created in the encryption phase so you need to store them when you run the encryption phase. You could use /dev/urandom to generate the keys.In case a block is not 64 bits you should use padding. The number of rounds should be 8 (n = 8).

Important notes

1. You need to submit the hy457_crypto.h , the hy457_crypto.c, a Makefile that compiles the library, a test file that utilizes all the implemented function and demonstrates their correct usage and a Readme file that explains your implementation or unimplemented parts. You should place all these files in a folder named <login>_assign1. For example, if your login is csd9999 the folder should be named csd9999_assign1.
2. If you implement more helper functions or macros, explain their functionality in the Readme file.
3. This assignment should be implemented using C on Linux-based machines.
4. You could use a different way to produce random keys than the /dev/urandom but you need to ensure that the produced keys are produced correctly.
5. Follow the steps described above and do an incremental implementation. This will be very helpful in order to debug the various parts before putting them all together in the test file.
6. You can use the course’s mailing list for questions. However, read the previous emails first since your question might have already been answered.
7. Do not send private messages with questions to the TAs, since other students might have the same question and everyone deserves the answer.
8. Do not send code snippets or pieces of your implementation to the mailing list when asking a question.
9. Submitted code will be tested for plagiarism using plagiarism-detection software.