2^{nd} Assignment Information Security

Francesco Segala 3521885 Manos Gionanidis 3542068

September 25, 2017

1 Exercise 5

The alternative Text that we extract from the plaintext *informationsecurity* with a Vernam cipher using the alternative key *tlftrffwmixor|{xbch* is: *knapsackbagssecrets*

2 Exercise 6

We implemented the Feistel cipher in Python 2.7 technology. Here is the code:

```
import hashlib
import struct

def get_passw():
    password = raw_input('Give me a password: ')
    firstPass = hashlib.sha256(password.encode()).hexdigest()
    secondPass = hashlib.sha256(firstPass.encode()).hexdigest()
    return firstPass+secondPass

def XOR(op1, op2):
    a=int(op1,16)
    b=int(op2,16)
    return hex(a^b)[2:]

# compute 1 step of the cipher
def do_round(left, right,key):
    tmp=right
    right=XOR(left,f(key,right))
    left=tmp
    return left,right

# enchipher or decipher a block
def encipher_block(plaintext_block,ks,encryption=True):
    plaintext_block=str2hex(plaintext_block)
    left=plaintext_block[:len(plaintext_block)/2]
    right=plaintext_block[len(plaintext_block)/2:]
    if not encryption:
        ks=ks[::-1]
    for k in ks:
        left ,right=do_round(left ,right ,k)
    return right+left

################### auxiliary functions ##############################
def blockify(plaintext ,blocksize):# plaintext -> block1 ,block2 ,...blockN
    lenght=len(plaintext)
    padding=0
    if lenght % blocksize != 0 :
        padding=blocksize - (lenght % blocksize)
    plaintext+=padding*"␣"
    return map(''.join, zip(*[iter(plaintext)]*blocksize))

def str2hex(string):
    hexa=""
    for i in string:
        hexa+=hex(ord(i))[2:]
    return hexa #return just the hexa string without 0x

def hex2str(hex_string):
    return ''.join(chr(int(hex_string[i:i+2], 16))\\
    \\ for i in range(0, len(hex_string), 2))

def f(key,oper):
    return key

def build_keys_stream(password):
    return map(''.join, zip(*[iter(password)]*8))
########################### main routine ##############################
def main_routine(plaintext):
    passw=get_passw()
    KS=build_keys_stream(passw)
    plaintext_blocked=blockify(plaintext ,8)
    cipher=""
    for block in plaintext_blocked:
        cipher+=encipher_block(block ,KS)
    cipher=hex2str(cipher)
    print"CIPHER", cipher
    ciphertext_blocked=blockify(cipher,8)
    res=""
    for cblock in reversed(ciphertext_blocked):
        res=hex2str(encipher_block(cblock,KS,False))+res
    print"\ndecription:", res, " len: ", len(res)

#main
main_routine("Example txt for The Feistel cipher")
```

Our code works perfectly with a long lenght key, however with a short lenght key we faced some problems with some charachters that are decrypted wrongly but unfortunately we didn't came up with this bug.
The text that we have to decrypt for extra points for this exercise is:

Feistel Cipher
From Wikipedia: *http : //en.wikipedia.org/wiki/Feistel_cipher* In cryptography, a Feistel cipher is a symmetric structure used in the
construction of block ciphers, named after the German-born physicist and cryptographer Horst Feistel who did pioneering research while working for IBM
(USA); it is also commonly known as a Feistel network.

A large proportion of block ciphers use the scheme, including the Data Encryption Standard (DES).
The Feistel structure has the advantage that encryption and decryption operations are very similar, even identical in some cases, requiring only a reversal of the key schedule.

Therefore, the size of the code or circuitry required to implement such a cipher is nearly halved.

Feistel networks were first seen commercially in IBM's Lucifer cipher, designed by Horst Feistel and Don Coppersmith in 1973. Feistel networks gained respectability when the U.S. Federal Government adopted the DES (a cipher based on Lucifer, with changes made by the NSA). Like other components of the DES, the iterative nature of the Feistel construction makes implementing the cryptosystem in hardware easier (particularly on the hardware available at the time of DES's design)

3 Exercise 7

For this exercise we computed the superincreasing knapsack sequence and then we choose 2 coprime numbers N and M. After this we formulate our Public Key s.t. :

```
GK=[89, 356, 534, 1068, 2225, 4450, 9167, 18245,
  36401, 72980, 41023, 81868, 59066, 13106, 26212, 52513]
```

after that the encripted message that the TA send us was:

The Battle of Ypres took place during the First World War, in the general area of the Belgian city of Ypres, where the German and the Allied armies (Belgian, French, British Expeditionary Force and Canadian Expeditionary Force) clashed. There were hundreds of thousands of casualties. The term "Battle of Ypres" could mean all the fighting that occurred in that area. But the "Battle of Ypres" could refer more specifically to any one of five battles which have been separately identified and named (and which themselves can be subdivided into smaller named battles).
Finally we design our code in such a way that if someone wanted to encrypt a zero-byte message he can do it.
We overcome to this issues by adding some padding-bytes at the end of the text during the encryption process, (it works also for odd-lenght strings).

If you want to see the code we used for this exercise go to : $https : //github.com/FrancescoSegala/infosec/tree/2 - assignme$

4 Exercise 8

Here is the pseudocode description:

Algorithm 1 Cipher Block chaining- encryption

```
procedure E( K , T )( )
    InitializationVector(IV)
#otherwise we can use a starting variable. V or SV is a block of
#bits that is used by several modes to randomize the encryption
#and hence to produce distinct ciphertexts even if the same
#plaintext is encrypted multiple times, without the need for a
#slower re-keying process
    read allThePlainText
#reading all the plain text just to check if a padding is needed
    Padding(allThePlainText)
#Handling the plain text because a block cipher works on units of a fixed size
#but messages come in a variety lenghts. There are many techniques for padding
#we consider this one to add null bytes to the plain text
    read plainText
#read the first block
    y = (plainText) xor (IV)
    cipherText = blockCipherEncryption(K,y)
#main encryption procedure
#first encrypted block
    cipherText
#continue for the other blocks
    for $x$ in range(len(allThePlainText)/n)-1):  do
#n is the number of the bytes that consist the block
#for example if the text is 32 bytes and i want to read block of 8 bytes
#so 32/8 = 4 so i am going to read 4 blocks minus 1 because the first
#iteration took place outside the loop using the IV so i am starting from the next block
        read plainText
#reading the next plain text block
#so the procedure now is the same for all the other blocks
        y = (plainText) xor (cipherText)
#using the previous cipherText
        cipherText = blockCipherEncryption(K,y)
#main encryption procedure
        cipherText
#next encrypted block
```

Algorithm 2 Cipher Block chaining - decryption

```
procedure D( K , C )( )
    read cipherText
#read the first block of the cipherText
    y = blockCipherDecryption(K,cipherText)
#main procedure of decryption
    plainText = (y) xor (IV)
#same IV as in the encryption
#first iteration take place outside the loop because i use the IV
    previousCipherText = cipherText
    for $x$ in range((len(allTheCipherText)/n)-1): do
#the procedure for all the blocks
        read cipherText
#read the next encrypted block
        y = blockCipherDecryption(K,cipherText)
#main procedure for decryption
        plainText = (y) xor (previousCipherText)
#using the previous encrypted block
        previousCipherText = cipherText
#prepare the next input of the XOR gate
        plainText
#is the decrypted block
```

5 Exercise 9

Stream cipher are based on generating an 'infinite' cryptographic keystream, and using that to encrypt one bit at a tie , whereas block ciphers work on larger chunks of data at a time, often combining blocks for additional security. So stream ciphers are typically faster than block ciphers, but that has it's own price. Block ciphers typically require more memory, since they work on larger chunks of data and often have "carry over" from previous blocks, whereas since stream ciphers work on only a few bits at a time they have relatively low memory requirements (and therefore cheaper to implement in limited scenarios such as embedded devices, firmware, and esp. hardware). Stream ciphers are more difficult to implement correctly, and prone to weaknesses based on usage - since the principles are similar to one-time pad, the keystream has very strict requirements. On the other hand, that's usually the tricky part, and can be offloaded to e.g. an external box. Because block ciphers encrypt a whole block at a time (and furthermore have "feedback" modes which are most recommended), they are more susceptible to noise in transmission, that is if you mess up one part of the data, all the rest is probably unrecoverable. Whereas with stream ciphers bytes are individually encrypted with no connection to other chunks of data (in most ciphers/modes), and often have support for interruptions on the line. Also, stream ciphers do not provide integrity protection or authentication, whereas some block ciphers (depending on mode) can provide integrity protection, in addition to confidentiality. Furthermore block cipher are more useful when the amount of data is pre-known - such as a file, data files , or request , protocols, such as HTTP . So these are the reasons that block ciphers tend to be preferred over stream ciphers.

This is our code for exercise 9:

```
import binascii
import hashlib

#function to check if the input charachter represents and integer or not
# accordingly returns True , False
def RepresentsInt(s):
    try:
        int(s)
        return True
    except ValueError:
        return False


#fills a list with only bytes, so only  0,1 characherts are been passed
def onlyBytes(listA):
        listB =[]
        for x in listA:
                if not(RepresentsInt(x)):
                        pass
                else:
                        listB.append(x)
        return listB

 #key-scheduling algorithm
def KSA(key):
        keylength = len(key)
        #initialize S
        S = range(256)
        j = 0
        for i in range(256):
                j = (j + S[i] + key[i % keylength]) % 256
                S[i], S[j] = S[j], S[i]  # swap
        return S
```

```
#stream generator
#Pseudo-random generation algorithm (PRGA)
def PRGA(S,text):
        i = 0
        j = 0
        k=[]
        #first 256 calls and the encrypt/decrypt
        for x in range(256):
                i = (i + 1) % 256
                j = (j + S[i]) % 256
                S[i], S[j] = S[j], S[i]  # swap
                K = S[(S[i] + S[j]) % 256]
        for x in range(len(text)):
                i = (i + 1) % 256
                j = (j + S[i]) % 256
                S[i], S[j] = S[j], S[i]  # swap
                K = S[(S[i] + S[j]) % 256]
                k.append(K)
        return k

#convert text
def convertText(s):
        return [ord(c) for c in s]

def initialize(text,key):
        print 'The text for decryption/encryption is: ',text
  print ' , with the key:  ',key,'\n'
        key  = convertText(key)
        text = convertText(text)
        S = KSA(key)
        random=[]
        random = PRGA(S,text)
        cipherText=[]
        for x in range(len(text)):
                cipherText.append(chr(text[x]^random[x]))#XOR logical gat
        result = ''.join(cipherText)
        print 'The text after RC4 procedure is: ' ,result,'\n'
        return result

#read the input encrypted text from a file
def decryptUniversity():
        f=open('/home/manos/Groningen/Assignment2Crypto/rc4(3).enc','r')
        textHex = f.read()
        print textHex ,'\n'
        textHex1 = hashlib.sha256(textHex).hexdigest()
        print 'sha256 is: ' ,textHex1,'\n'
        return textHex

#main procedure
def main():
        text = decryptUniversity()
        key = raw_input('Give me the key: ')
        result = initialize(text,key)

main()
```

This is the decrypted text:

> But I don’t want to go among mad people,’ said Alice. ’Oh, you can’t help that,’ said the cat. ’We’re all mad here.’
> This quote is from ‘Alice in Wonderland’ by Lewis Carroll