Back to TOC

2.11 ESTABLISHING SECURE COMMUNICATIONS FOR FUN (BUT NOT FOR PROFIT)

This section has two goals:

- To demonstrate that if all that you want is to establish a medium-strength secure communication link between yourself and a buddy, you may be able to get by without having to resort to the full-strength crypto systems that we will be studying in later lectures.

- To introduce you to my **BitVector** module. You will be using this module for several homework assignments throughout this course.

The `BitVector` module comes with both a Python and a Perl implementation. However, if you are not multilingual in your scripting capabilities, it is sufficient if you become familiar with either the Python version or the Perl version of the module. Note that the scripts shown in this section only provide a brief introduction to the module. Please also spend some time going though the API of the module that you will find at the following links:

Python: `https://engineering.purdue.edu/kak/dist/BitVector-3.5.0.html`

Perl: `https://metacpan.org/pod/Algorithm::BitVector`

So here we go:

- Fundamentally, the encryption/decryption logic in the scripts shown in this section is based on the following properties of XOR operations on bit blocks. Assuming that A, B, and C are bit arrays, and that $\oplus$ denotes the XOR operator, we can write

$$\begin{aligned} [A \oplus B] \oplus C &= A \oplus [B \oplus C] \\ A \oplus A &= 0 \\ A \oplus 0 &= A \end{aligned}$$

- More precisely, the Python and Perl encryption/decryption scripts in this section are based on **differential XORing** of bit blocks. Differential XORing means that, as a file is scanned in blocks of bits, the output produced for each block is made a function of the output for the previous block.

- Differential XORing destroys any repetitive patterns in the messages to be encrypted and makes it more difficult to break encryption by statistical analysis.

- The encryption/decryption scripts presented in this section require a key and a passphrase. While the user is prompted for the key in lines (J) through (M), the passphrase is placed directly in the scripts in line (C). In more secure versions of the scripts, the passphrase would also be kept confidential by the parties using the scripts.

- Since differential XORing means that the output for the current block must depend on the output that was produced for the previous block, that raises the question of what to do for the first bit block in a file. Typically, this problem is solved by using an initialization vector (IV) for the differential XORing needed for the first bit block in a file. We derive the needed initialization vector from the passphrase in lines (F) through (I).

- For the purpose of encryption or decryption, the file involved is scanned in bit blocks, with each block being of size `BLOCKSIZE`. For encryption, this is done in line (V) of the script shown next. Since the size of a file in bits may not be an integral multiple of `BLOCKSIZE`, we add an appropriate number of null bytes to the bytes extracted by the last call in line (V). This step is implemented in lines (W) and (X) of the encryption script that follows.

- For encryption, each bit block read from the message file is first XORed with the key in line (Y), and then, in line (Z), with the output produced for the previous bit block. The step in line (Z) constitutes differential XORing.

- If you make the value of `BLOCKSIZE` sufficiently large and keep both the encryption key and the passphrase as secrets, it will be very, very difficult for an adversary to break the encryption — especially if you also keep the logic of the code confidential.

- The implementation shown below is made fairly compact by the use of the `BitVector` module. [**This would be a good time to become familiar with the `BitVector` module by going through its API. You'll be using this module in several homework assignments dealing with cryptography and hashing.**]

```
#!/usr/bin/env python

###  EncryptForFun.py
###  Avi Kak  (kak@purdue.edu)
###  January 21, 2014, modified January 11, 2016

###  Medium strength encryption/decryption for secure message exchange
###  for fun.

###  Based on differential XORing of bit blocks.  Differential XORing
###  destroys any repetitive patterns in the messages to be encrypted and
###  makes it more difficult to break encryption by statistical
###  analysis. Differential XORing needs an Initialization Vector that is
###  derived from a pass phrase in the script shown below.  The security
###  level of this script can be taken to full strength by using 3DES or
###  AES for encrypting the bit blocks produced by differential XORing.

###  Call syntax:
###
###       EncryptForFun.py  message_file.txt  output.txt
###
###  The encrypted output is deposited in the file `output.txt'

import sys
from BitVector import *                                                    #(A)

if len(sys.argv) is not 3:                                                 #(B)
    sys.exit('''Needs two command-line arguments, one for '''
             '''the message file and the other for the '''
             '''encrypted output file''')
```

```
PassPhrase = "Hopes and dreams of a million years"                          #(C)

BLOCKSIZE = 64                                                              #(D)
numbytes = BLOCKSIZE // 8                                                   #(E)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)                                  #(F)
for i in range(0,len(PassPhrase) // numbytes):                              #(G)
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]                         #(H)
    bv_iv ^= BitVector( textstring = textstr )                              #(I)

# Get key from user:
key = None
if sys.version_info[0] == 3:                                                #(J)
    key = input("\nEnter key: ")                                            #(K)
else:
    key = raw_input("\nEnter key: ")                                        #(L)
key = key.strip()                                                           #(M)

# Reduce the key to a bit array of size BLOCKSIZE:
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)                                 #(N)
for i in range(0,len(key) // numbytes):                                     #(O)
    keyblock = key[i*numbytes:(i+1)*numbytes]                               #(P)
    key_bv ^= BitVector( textstring = keyblock )                            #(Q)

# Create a bitvector for storing the ciphertext bit array:
msg_encrypted_bv = BitVector( size = 0 )                                    #(R)

# Carry out differential XORing of bit blocks and encryption:
previous_block = bv_iv                                                      #(S)
bv = BitVector( filename = sys.argv[1] )                                    #(T)
while (bv.more_to_read):                                                    #(U)
    bv_read = bv.read_bits_from_file(BLOCKSIZE)                             #(V)
    if len(bv_read) < BLOCKSIZE:                                            #(W)
        bv_read += BitVector(size = (BLOCKSIZE - len(bv_read)))             #(X)
    bv_read ^= key_bv                                                       #(Y)
    bv_read ^= previous_block                                               #(Z)
    previous_block = bv_read.deep_copy()                                    #(a)
    msg_encrypted_bv += bv_read                                             #(b)

# Convert the encrypted bitvector into a hex string:
outputhex = msg_encrypted_bv.get_hex_string_from_bitvector()                #(c)

# Write ciphertext bitvector to the output file:
FILEOUT = open(sys.argv[2], 'w')                                            #(d)
FILEOUT.write(outputhex)                                                    #(e)
FILEOUT.close()                                                             #(f)
```

- Note that a very important feature of the script shown above is that the ciphertext it outputs consists only of printable characters. This is ensured by calling `get_hex_string_from_bitvector()` in line (c) near the end of the script. This call translates each byte of the ciphertext into two printable hex characters.

- The decryption script, shown below, uses the same properties of the XOR operator as stated at the beginning of this section to recover the original message from the encrypted output.

- The reader may wish to compare the decryption logic in the loop in lines (U) through (b) of the script shown below with the encryption logic shown in lines (S) through (b) of the script above.

```
#!/usr/bin/env python

###  DecryptForFun.py
###  Avi Kak  (kak@purdue.edu)
###  January 21, 2014, modified January 11, 2016

###  Medium strength encryption/decryption for secure message exchange
###  for fun.

###  Based on differential XORing of bit blocks.  Differential XORing
###  destroys any repetitive patterns in the messages to be ecrypted and
###  makes it more difficult to break encryption by statistical
###  analysis. Differential XORing needs an Initialization Vector that is
###  derived from a pass phrase in the script shown below.  The security
###  level of this script can be taken to full strength by using 3DES or
###  AES for encrypting the bit blocks produced by differential XORing.

###  Call syntax:
###
```

```
###        DecryptForFun.py  encrypted_file.txt  recover.txt
###
###  The decrypted output is deposited in the file ‘recover.txt’

import sys
from BitVector import *                                                       #(A)

if len(sys.argv) is not 3:                                                    #(B)
    sys.exit('''Needs two command-line arguments, one for '''
             '''the encrypted file and the other for the '''
             '''decrypted output file''')

PassPhrase = "Hopes and dreams of a million years"                            #(C)

BLOCKSIZE = 64                                                                #(D)
numbytes = BLOCKSIZE // 8                                                     #(E)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)                                    #(F)
for i in range(0,len(PassPhrase) // numbytes):                                #(G)
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]                           #(H)
    bv_iv ^= BitVector( textstring = textstr )                                #(I)

# Create a bitvector from the ciphertext hex string:
FILEIN = open(sys.argv[1])                                                    #(J)
encrypted_bv = BitVector( hexstring = FILEIN.read() )                         #(K)

# Get key from user:
key = None
if sys.version_info[0] == 3:                                                  #(L)
    key = input("\nEnter key: ")                                              #(M)
else:
    key = raw_input("\nEnter key: ")                                          #(N)
key = key.strip()                                                             #(O)

# Reduce the key to a bit array of size BLOCKSIZE:
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)                                   #(P)
for i in range(0,len(key) // numbytes):                                       #(Q)
    keyblock = key[i*numbytes:(i+1)*numbytes]                                 #(R)
    key_bv ^= BitVector( textstring = keyblock )                              #(S)

# Create a bitvector for storing the decrypted plaintext bit array:
msg_decrypted_bv = BitVector( size = 0 )                                      #(T)

# Carry out differential XORing of bit blocks and decryption:
previous_decrypted_block = bv_iv                                              #(U)
for i in range(0, len(encrypted_bv) // BLOCKSIZE):                            #(V)
    bv = encrypted_bv[i*BLOCKSIZE:(i+1)*BLOCKSIZE]                            #(W)
    temp = bv.deep_copy()                                                     #(X)
    bv ^=  previous_decrypted_block                                           #(Y)
    previous_decrypted_block = temp                                           #(Z)
```

```
        bv ^=  key_bv                                                   #(a)
        msg_decrypted_bv += bv                                          #(b)

    # Extract plaintext from the decrypted bitvector:
    outputtext = msg_decrypted_bv.get_text_from_bitvector()             #(c)

    # Write plaintext to the output file:
    FILEOUT = open(sys.argv[2], 'w')                                    #(d)
    FILEOUT.write(outputtext)                                           #(e)
    FILEOUT.close()                                                     #(f)
```

- To exercise these scripts, enter some text in a file and let's call this file **`message.txt`**. Now you can call the encrypt script by

```
EncryptForFun.py  message.txt  output.txt
```

The script will place the encrypted output, in the form of a hex string, in the file **`output.txt`**. Subsequently, you can call

```
DecryptForFun.py  output.txt   recover.txt
```

to recover the original message from the encrypted output produced by the first script.

- If you'd rather use Python 3, you can invoke these scripts as

```
python3  EncryptForFun.py  message.txt  output.txt

python3  DecryptForFun.py  output.txt   recover.txt
```

- What follows are the Perl versions of the two Python script shown above. For at least those of you who would like to be proficient in both Perl and Python, it would be educational to