COMP2700 Lab 9 – Encryption Mode

In this lab we will look at different encryption modes, i.e., how a block cipher can be used to encrypt data that is larger than the block size of the cipher, and security issues arising from improper uses of encryption modes.

Some of the exercises in this lab are selected from Paar & Pelzl's "Understanding Cryptography" (Chapter 5).

In addition to the pycryptodome library, we will also use the **openssl** command to perform AES encryption/decryption and the Linux **dd** command to perform bit blocks copying. Both utilities should be available in any standard linux installation, so feel free to use your own linux installation if you'd like.

To encrypt with AES 128 bit in ECB mode, use the following:

```
openssl enc -aes-128-ecb -nosalt -in input-file -out output-file
```

This will prompt you for the password that is used to generate the AES key. This command encrypts input-file using AES 128 bit in ECB mode, and outputs the result in output-file. The command for decryption is almost identical, with an addition of the '-d' option:

```
openssl enc -d -aes-128-ecb -nosalt -in input-file -out output-file
```

To encrypt using CBC mode, simply replace -aes-128-ecb with -aes-128-cbc .

Note: in CBC mode, the option -nosalt causes the same IV to be used for every encryption. DO NOT use this option in practice as it opens up various security issues. This option is used in this lab for testing purpose only.

We can use dd command to copy blocks of bytes. We will explain the syntax of this command with an example:

dd if=input.bin **of**=output.bin **bs**=16 **skip**=2 **count**=2

This command copies 2 blocks of bytes (specified in the option *count=2*) from the input file (*if=input.bin*), each block consisting of 16bytes (*bs=16*), skipping the first two blocks (*skip=2*), and outputs the result to output.bin (*of=output.bin*). You can tailor the parameters to suit your particular needs. For this particular lab, we will work only with block size of 128 bit (16 bytes), so we will use the option bs=16.

Exercise 1 and Exercise 3 require additional files. These can be downloaded from the Wattle page for this lab (see file lab-9-files.zip), or simply 'wget' it from the following URL:

wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab9_files.zip

For Exercise 3, we will use the PyCryptodome library in python; specifically its strxor module, to perform the XOR operation on byte arrays.

Exercise 1. In this question, we will encrypt the famous Tux penguin logo for Linux, that is provided in the course site for this tutorial (file *Tux.ppm*). We will use a file format called PPM: see

https://en.wikipedia.org/wiki/Netpbm_format#PPM_example

There are two kinds of PPM formats: one in which the color (RGB code) of each individual pixel is encoded using ASCII characters (P3 type), and the other in which the color is encoded using binary representation (P6 type). We will use the latter. The header of a P6-type PPM format consists of three lines of texts: the first line contains the word 'P6'; the second line contains two numbers specifying the width and the height of the image in pixels, and the third line is the maximum value for each color (this can range from 1 – 65535). Typical values for the third line are either 255 or 65535. For example, the file Tux.ppm included in this lab has the following header

```
P6
265 314
255
```

That is, it is a P6 type PPM image file, of size 265x314 pixels and each colour is represented by a value less or equal to 255. What follows the header is then just a bitmap of the pixels, where each pixel is represented by three values (RGB colour) of the pixel. If the max value of each colour is less or equal to 255, each colour value is represented by a byte. If it is greater than 255, then it is represented by 2 bytes.

a) Encrypt the image content (the pixel map, without the header) of Tux.ppm using AES-128 bit in ECB mode. How do you extract only the pixels (without the header)? What commands do you use?
b) Try to view the encrypted content from a) as an image. How do you do that? What do you observe?
c) Now repeat the above using AES 128 bit, but in CBC mode. What do you observe in the output image?

Exercise 2. The purpose of this question is to illustrate the danger of ECB mode when it is used to encrypt a message longer than one block, where the meaning of the message may be dependent on multiple blocks.

Consider the following (hypothetical) protocol for fund transfer between banks. A core part of the protocol uses the following message format to encode transactions:

AAA:XXXXXXXXXXX:BBB:YYYYYYYYYYY:CCC:<amount to be transferred>

The message contains 6 colon-separated fields:

- AAA: three letter code of the sender bank.
- XXXXXXXXXXX: 11 digit account number of the sending account.
- BBB: three letter code of the receiving bank.
- YYYYYYYYYYY: 11 digit account number of the receiving account.
- CCC: three letter code for currency of the transfer.
- The last field is a variable length field containing the amount to be transferred.

For example, the following message:

CBA:11122233344:ANZ:01234567890:AUD:100

contains an instruction to transfer AUD 100 from the account 11122233344 at CBA to account 01234567890 at ANZ.

Suppose the banks use AES 128 encryption in ECB mode to encrypt transactions.

a) Use the openssl tool to encrypt the following two transactions. Save each transaction in a file and encrypt the file using AES in ECB mode. You can use any password to generate the encryption key.

   CBA:82934681003:NAB:99203848881:AUD:120
   CBA:82934681003:ANZ:45200943921:AUD:3500

b) Using only the encrypted messages from a), create another encrypted message for the following transaction:

   CBA:82934681003:NAB:99203848881:AUD:3500

   (*Note: you are not allowed to use the encryption key in a) to do this, i.e., think of a scenario where the attacker managed to intercept the ciphertexts in a) and the attacker wants to forge another transfer. Use the dd command to help you manipulate ciphertexts*).

Exercise 3. For this question, you are given three files: ofb1.txt, ofb1.enc and ofb2.enc. (These files can be downloaded from the Wattle page for this lab.)

The file ofb1.enc was obtained by encrypting ofb1.txt in OFB mode using some key k and some IV value I. The file ofb2.enc is obtained by encrypting another file, using OFB mode with the same key k and IV value I. Show how you can decrypt ofb2.enc without knowing the key or the IV value.

Python library Crypto.Util.strxor

To solve Exercise 3, we will use the Pycryptodome library (specifically the module Crypto.Util.strxor) to perform XOR on two byte strings.

To read a binary file as a byte array, you can use the 'open' function, to open a file, and use the read function to read into a byte array. Don't forget to use the 'close' function to close the file when you're done.

Here is an example of a python interactive session to read the content of 'file1' (treated as a binary file) into a byte array arr1.

```
>>> f=open("file1","rb")
>>> arr1=f.read()
>>> arr1
b'abcd'
>>> f.close()
>>>
```

The function open("file1","rb") opens "file1" as a binary file for reading. The value of byte array arr1 in this case consist of the ASCII codes of each of the character in the string 'abcd'. (In Python, you can treat a string as a byte array. Such an array is usually displayed with a prefix `b`, e.g., `b'abcd'`.)

To XOR a byte array with another byte array, we use the strxor function. Here is an example of an XOR operation; continuing our previous example (so the value of arr1 is the byte array `b'abcd'`).

```
>>> arr2=b'\x00\x00\x00\x00'
>>> arr2
b'\x00\x00\x00\x00'
>>> from Crypto.Util.strxor import *
>>> strxor(arr1,arr2)
b'abcd'
>>> arr3=b'\xff\xff\xff\xff'
>>> strxor(arr1,arr3)
b'\x9e\x9d\x9c\x9b'
>>> arr4=b'\x20\x20\x20\x20'
>>> strxor(arr1,arr4)
b'ABCD'
>>>
```

In this example, we first declare a variable arr2, and initialise it to 4 zero bytes; here we use the hex value of the bytes, each byte is represented as '\xHH' where HH is the hex value of the byte.

The strxor function is in the module Crypto.Util.strxor, which must be imported before we can use it. We show here three examples of strxor, between arr1 and each of arr2, arr3 and arr4. Notice that arr4 is a 4-byte array, where each element contains the byte value of the space ' ' character (\x20).

Exercise 4. We are using AES in counter mode (CTR) for encrypting a hard disk with 1 TiB (= 2^{40} bytes, approx. 1.1TB) of capacity. What is the maximum length of the IV?

Extension Exercises (Optional)

Exercise 5. Recall that block ciphers such as AES takes as input a block of fixed size, e.g., in AES it is 128 bits or 16 bytes. If the input to the AES is not exactly 16 byte long, padding bytes need to be added. A common padding scheme used with block ciphers and some encryption modes, notably CBC mode, is the PKCS#7 padding[1] scheme. This scheme works as follows: suppose the block size of the cipher is N bytes. If the last block of the input m is of length $(N - r)$ bytes, then add the byte value of r to m until the last block of m is N byte long. For example, if N = 16 byte (as in AES) and m is the following byte string (in HEX notation):

000102030405060708090a0b0c0d0e0faabbccddeeff

---

[1] The specification of PKCS#7 can be found here https://www.ietf.org/rfc/rfc2315.txt

The message m in this case is 22-byte long, it needs additional 10 bytes to make its length a multiple of block size (16 bytes). In this case, $r = 10$ (or in HEX, it's 0a), so we pad the byte string m with 10 bytes, each byte containing the value r (0a in HEX):

000102030405060708090a0b0c0d0e0faabbccddeeff0a0a0a0a0a0a0a0a0a0a

Note that if the length of the message m is already a multiple of N, an additional block is added (each byte of that block contains the byte value of N).

In the following questions, assume that N=16 byte.

a. Suppose the following block is a result of PKCS#7 padding.
   0a11d34488220011f100aabb33010101
   What is the original message before the padding?
b. Consider the following very simple padding scheme. If the input message m is not a multiple of N, add byte 00 to m until its length becomes a multiple of N. For example, if m is
   00112233445566778899aabbcc
   then the resulting padded message is
   00112233445566778899aabbcc000000
   What could be a potential issue with this naïve padding scheme?

Exercise 6 (*). In a company, all files which are sent on the network are automatically encrypted by using AES-128 in CBC mode. A fixed key is used, and the IV is changed once per day. The network encryption is file-based, so that the IV is used at the beginning of every file.

You managed to obtain the fixed AES-128 key, but do not know the recent IV. Today, you were able to eavesdrop two different files, one with unidentified content and one which is known to be an automatically generated temporary file and only contains the value 0xFF. Briefly describe how it is possible to obtain the unknown IV and how you are able to determine the content of the unknown file.

Exercise 7 (*). Keeping the IV secret in OFB mode does not make an exhaustive key search more complex. Describe how we can perform a brute-force attack with unknown IV. What are the requirements regarding plaintext and ciphertext?

Exercise 8 (*). Imagine that aliens — rather than abducting earthlings and performing strange experiments on them — drop a computer on planet Earth that is particularly suited for AES key searches. In fact, it is so powerful that we can search through 128, 192 and 256 key bits in a matter of days. Provide guidelines for the number of plaintext– ciphertext pairs the aliens need so that they can rule out false keys with a reasonable likelihood.