COMP2700 Lab 9 – Solutions

Exercise 1.

a) To extract the header, we can use 'head' command:

head -n 3 Tux.ppm > header.txt

To extract the bitmap (line 4 onwards in Tux.ppm):

tail -n +4 Tux.ppm > pixels.bin

To encrypt the pixel map, use openssl:

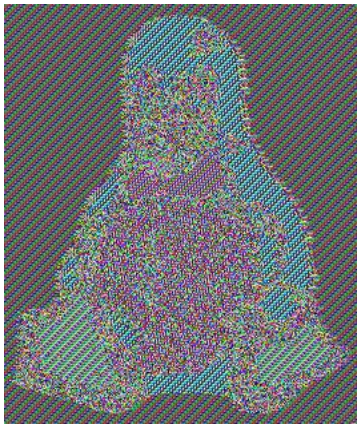
openssl enc -aes-128-ecb -nosalt -in pixels.bin -out epixels.bin

(Use any passwords you like; for this problem it does not matter which passwords you use.)

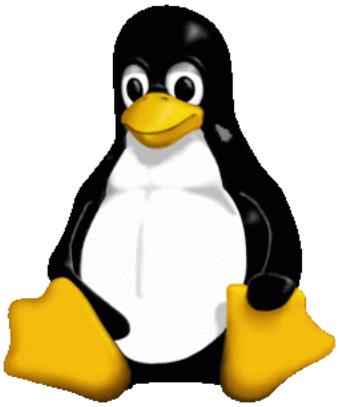
b) To view the encrypted pixels from a), first we need to add the appropriate header. We can simply use the header for the unencrypted image. Technically the size of the encrypted fixels file is slightly larger than the unencrypted one, due to the addition of paddings (to make the file size equal to a multiple of block size), but the additional pixels will generally be ignored by image file viewer. So we simply add the original header to the encrypted pixels:

cat header.txt epixels.bin > ecbTux.ppm

The image should look something like this:



Compare this to the original image:

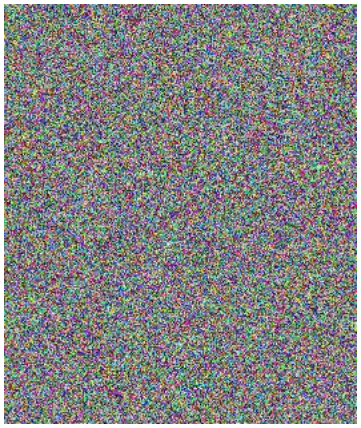


Observe that the pattern (penguin-shape) is still present in the encrypted image, even though individual pixel colors are changed.

c) To encrypt using CBC mode, we only need to change one command (the rest stay the same):

openssl enc -aes-128-cbc -nosalt -in pixels.bin -out epixels.bin

The encrypted image you get would look something like this:



Observe that the patterns in the original unencrypted file are completely masked.

Exercise 2.

a) We first create two files corresponding to the two transactions:

```
$ echo -n "CBA:82934681003:NAB:99203848881:AUD:120" > tr1.txt
$ echo -n "CBA:82934681003:ANZ:45200943921:AUD:3500" > tr2.txt
```

Then encrypt the files (using password “123456”):

```
$ openssl enc -aes-128-ecb -nosalt -in tr1.txt -out etr1.bin
$ openssl enc -aes-128-ecb -nosalt -in tr2.txt -out etr2.bin
```

The encrypted files for the transactions are respectively etr1.bin and etr2.bin.

b) To forge the transaction (from CBA account to NAB account) without knowing the encryption key and having access only to the two encrypted transactions (etr1.bin and etr2.bin) above, we need to substitute some encrypted blocks.

Since AES encrypts a 128-bit block (16 bytes) at a time, we know that the first two blocks of etr1.bin correspond to the plaintext:

CBA:82934681003:NAB:99203848881:

and the last block of etr2.bin corresponds to the plaintext

```
AUD:3500
```

Note that the plaintext in this case is only 8bytes (rather than 16 bytes), so openssl adds some 'padding' to the plaintext to make it 16 byte long. We shall not discuss the details of the padding here.

So we just need to put together these three blocks: the first two blocks from etr1.bin and the third block from etr2.bin. This is achieved by the following commands:

```
$ dd if=etr1.bin of=prefix.bin bs=16 count=2
$ dd if=etr2.bin of=sum.bin bs=16 skip=2 count=1
$ cat prefix.bin sum.bin > etr3.bin
```

To confirm that we have got a valid cipher text, we can try to decrypt it:

```
$ openssl enc -d -aes-128-ecb -nosalt -in etr3.bin -out tr3.txt
```

We can then confirm that tr3.bin decrypts to the desired plaintext:

```
$ cat tr3.txt
CBA:82934681003:NAB:99203848881:AUD:3500
```

Exercise 3.

In OFB mode, the key stream generated is determined completely by IV and the key. If the same IV and key are used to encrypt different plaintexts, confidentiality may be compromised. Suppose m is an n-block plaintext, ie.,

$m = x_1 x_2 \cdots x_n$ where each x_i is a block.

The OFB mode will generate n block key stream $s = s_1 s_2 \cdots s_n$ and the ciphertext is obtained by XOR-ing m and s:

$y = m \oplus s = (x_1 \oplus s_1) \cdots (x_n \oplus s_n) = y_1 \cdots y_n$

where each $y_j = x_j \oplus s_j$ for every $j \in \{1, \dots, n\}$.

Suppose the same IV and k are used to encrypt another n block message u, and suppose that y and m are known to the attacker. Then the attacker can recover the key stream by $s = y \oplus m$ and use it to decrypt u by $u \oplus s$.

The above strategy can be used to decrypt ofb2.enc without knowing the key or the IV. Simply xor ofb1.txt and ofb1.enc to obtain the key stream, and xor the key stream with ofb2.enc.
So to ensure security of OFB, each IV can only be used once, for the same key. (Alternatively, use different key to sign different messages, but this may not be practical).

Exercise 4.

Recall that encryption/decryption in counter mode is done as follows:

$$\textbf{\textit{Encryption}}: y_i = e_k(IV||CTR_i) \oplus x_i, \quad i \geq 1$$
$$\textbf{\textit{Decryption}}: x_i = e_k(IV||CTR_i) \oplus y_i, \quad i \geq 1$$

That is, to encrypt, the block encryption of the concatenation of IV and a counter is xor-ed with the plaintext. This means the concatenation $(IV \,||CTR_i)$ needs to be exactly 128-bit long.

The counter has to encrypt 1 TiB of data without repeating itself. Since each counter is used to encrypt a 128-bit block (16 bytes = 2^4 bytes), we need at least 40-4 = 36 bits to ensure no counter is repeated. This yields an IV of maximum size of 92=128−36bits.

Exercise 5.

a. The last byte of the block will always contains the value of r. In this case it is $r = 01$. So we 'unpad' the padded message by removing 1 byte from the end of the block, resulting in the original message: 0a11d34488220011f100aabb330101.
b. The proposed padding scheme (by padding the block with byte 00) can create ambiguity when unpadding a padded message. For example, the following (padded) message:

   0a11d34488220011f100aabb330000

   can be the result of padding either of the following messages:
   - 0a11d34488220011f100aabb33, or
   - 0a11d34488220011f100aabb3300.

Exercise 6.

The decryption of a CBC-encrypted file is defined by $x_i = d_K(y_i) \oplus y_{i-1}$, when $i > 0$, and $x_0 = d_{K(y_0)} \oplus IV$. Since we know the key *K* and the pair (x_0, y_0) (from the second file), the unknown IV can easily be obtained by converting the equation:

$$IV = d_K(y_0) \oplus x_0$$

Once we know the IV, the first (unidentified) file can be decrypted using this IV and the key K.

Exercise 7.

We need two pairs of plaintext-ciphertext that correspond to two consecutive blocks in OFB mode, i.e., (x_i, y_i) and (x_{i+1}, y_{i+1}) where

$$y_i = s_i \oplus x_i$$

$$y_{i+1} = s_{i+1} \oplus x_{i+1}$$

From (x_i, y_i), we can compute $s_i = y_i \oplus x_i$, and from (x_{i+1}, y_{i+}) we can compute $s_{i+1} = y_{i+1} \oplus x_{i+1}$.

Since $s_{i+1} = e_k(s_i)$, we can use the plaintext-ciphertext pair (s_i, s_{i+1}) to brute force the block cipher that corresponds to the function e_k . The knowledge of IV is not required to perform the brute force attack on the block cipher.

Given two pairs (x_i, y_i) of plaintext and ciphertext, the i-th and i+1-th output S_i and S_{i+1} of the underlying block cipher (eg, AES) can be computed.

$$y_i = s_i \oplus x_i$$

Exercise 8.

Instructor notes:

- *This exercise uses advanced concepts that are not part of the assessable material; the provided solution below is for self-study only and will not feature in any assessments for this course.*
- *This exercise was taken from Paar & Pelzl's Understanding Crytography, Problem 5.13 (Chapter 5). Paar & Pelzl released a solution to this exercise (see https://www.crypto-textbook.com/index.php) that appears to be wrong. The solution below is the instructor's own solution.*

This question is related to a concept called known-plaintext unicity distance[1] – i.e., how many plaintext-ciphertext pairs are needed in a brute-force search so that one can determine with a high degree of confidence that a particular key candidate is the correct key. If the key space is larger than the block size, for example for AES-192, then two keys may encrypt a given input to the same output. For instance, for AES-192, for any x and y, there are on average 2^{64} candidate keys that satisfy $e_k(x) = y$. Among these candidates is the true key that was used to encrypt x, which we call the target key – the other keys are called false keys.

For a block cipher with a key length of K bits and block size n, the expected number of false keys that can encrypt t plaintext-ciphertext pairs $(x_1, y_1), \dots, (x_t, y_t)$ is given by the formula

[1] See A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, Chapter 7.

$$2^{K-t\cdot n}$$

(see Theorem 5.2.1 in *Understanding Cryptography*, page 137). So to ensure we guess the correct target key with a high degree of confidence, we want to minimise the number of expected false keys to be less than 1, preferably as close to 0 as practical. For AES-128, $K = 128$ and $n = 128$. So for $t = 2$ (i.e., two plaintext-ciphertext pairs), the expected number of false keys is $2^{128-2\cdot 128} = 2^{-128}$, so two pairs of plaintext-ciphertexts suffice. That means that if a key candidate encrypts correctly two given plaintext-ciphertext pairs, it is extremely unlikely that it is a false key.

Similarly for AES-192, we need at least two plaintext-ciphertext pairs ($t = 2$), so that the expected number of false keys falls below 1: $2^{192-2\cdot 128} = 2^{-64}$. For AES-256, we need at least three pairs ($t = 3$), to get an expected number of false keys below 1: $2^{256-3\cdot 128} = 2^{-128}$.

1. **已知明文对的重要性**:
   - 若密钥空间大于块大小，则可能存在多个密钥生成相同的密文。
   - 需要足够多的明文-密文对来排除所有错误密钥，以唯一确定正确密钥。
2. **计算公式**:
   - 设 AES 的密钥长度为 `K`，块大小为 `n`。
   - 期望错误密钥的数量:
     $$2^{K-t\cdot n}$$
   - 目标是使期望的错误密钥数量小于 1，以提高唯一性。
3. **不同 AES 密钥长度的计算**:
   - **AES-128**: `K = 128`, `n = 128`, 需要 `t = 2` 对明文-密文对。
   - **AES-192**: `K = 192`, `n = 128`, 需要 `t = 2` 对明文-密文对。
   - **AES-256**: `K = 256`, `n = 128`, 需要 `t = 3` 对明文-密文对。