SIC Lab
Symmetric Cryptography

version 1.0

Authors: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version

1 Cipher Modes

Your task is to implement two modes of operation for the AES-128 block cipher: the Electronic Codebook (ECB) mode and the Cipher Block Chaining (CBC) mode.

- **AES-128 Implementation**: You should use an implementation of AES-128 block cipher. Remember that AES-128 operates on 16-byte blocks and uses a key of 16 bytes.
- **ECB Mode**: In ECB mode, each block of plaintext is encrypted separately. This means that if two identical blocks of plaintext are encrypted with the same key, they will produce identical blocks of ciphertext.
- **CBC Mode**: In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This means that each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.
- **Validation**: After implementing ECB, and CBC, validate your implementation against the test vectors provided in file `aes128-test-vectors.txt`.

Your implementation should produce the same output as the test vectors for the same input. Start by encrypting the plaintext and compare the result with the ciphertext, then decrypt the result and compare it with the expected plaintext (the original one). Note that the values in the file are in hexadecimal.

You can use as a starting point the `ecb.c` and `cbc.c` files; the code in these files is used by `aes128-test-modes.c`; otherwise, you can produce your own code. Along with `aes128-test-modes.c` there is an implementation of `AES-128` built on top of Intel AES instruction set (in `aes-ni.c`) and another one built on top of Openssl (in `aes-sw.c`). Inspect both files. For making your life easier, a Makefile is provided to be used with the `make` command.

NOTE : If your CPU do not support the Intel AES instruction set, remove from `Makefile` all the lines that generate code that depends on the AES-NI instructions (`aes-ni.o` and `aes1280-test-modes-ni`).

You can check the availability of Intel AES instruction set with the command:

```
$ grep -c aes /proc/cpuinfo # if it prints some number greater than 0 the AES instruction set is available
```

If you want to use `python` you need to install Python3 Cryptography module. The module can be installed using:

```
$ sudo apt update
$ sudo apt install python3-cryptography
```

If you want to use `c` you need to install some tools to compile the code:

```
$ sudo apt update
$ sudo apt install build-essential libssl-dev
```

Then you can execute:

```
$ make
```

2 Padding

A block cipher requires input blocks of a fixed size that equals the algorithm block size. However, its improbable that a file to encrypt as a number of bytes that is multiple of the block size of the algorithm to be used, i.e., frequently, the number of bytes that remain for the last block do not equals the block size. To solve this problem, extra bytes are added to have a block with the correct size. These extra bytes are then removed when in the decryption operation.

There are several standards for padding. PKCS#7 is one of them. Your task is to implement two functions: one for adding padding to a given plaintext according to the PKCS#7 standard, and another for removing the padding from a given ciphertext.

- **Add Padding**: Write a function that adds padding to an arbitrary `plaintext` in order to make it a valid input for the encrypt operation.
- **Remove Padding**: Write a function that removes the padding of the result of a decrypt operation. The function should verify that the padding is correct.
- **Validation**: Validate your implementation against the tests provided in file `aes128-test-padding.c`. For ECB abd CBC, use the test vectors provided in file `aes128-test-padding.txt`.

Note: Take special care when the size of the input data is a multiple of the block size. In such cases, PKCS#7 padding requires that a full block of padding be added. This is to ensure that the padding can always be unambiguously removed.

You can either use as a starting point the `aes128-test-padding.c` or produce your own code.

3 Symmetric key generation

Before a cipher is used, it is required the generation of proper arguments. These arguments are the key, the cipher mode, and potentially the IV. The cipher mode is chosen at design time, and the IV should always be a large random number (size similar to the block size or key) that is never repeated. In the previous examples we have been using known keys and IVs in order to validate the implementations. This lab will discuss the creation of random keys and IVs.

The key can be obtained from good sources of random numbers, or generated from other primitive material such as a password. When choosing a password, it is imperative to transform the user text into a key of the correct complexity. While there are many methods, we will consider the Password Based Key Derivation Function 2 (PBKDF2), which takes a key, a random value named salt, a digest algorithm, and a number of iterations (you should use several thousands). The algorithm will iterate the digest algorithm in a chain starting in the concatenation of the salt and key, for the specified number of iterations. Using SHA2, the result is at least 256 bit, which can be used as key and IV, two values of 128 bit.

Your task is to enable the user to provide a password that will be used to generate random values for keys and IVs.

- Construct a function that generates 256 bits according to PBKDF2-HMAC-SHA256, i.e. with SHA256 as digest function used in the derivation (done in `pbkdf2-kickoff.c`).

- Use those 256 bits as two 128 bit values, the key and IV, and integrate this change with the previous cipher modes.

You can either use as a starting point the `pbkdf2-kickoff.c` or produce your own code. The Makefile already is prepared to compile this file.

Note Try different number of iterations and measure this impact.

4 File encryption

Create a function to encrypt the contents of a file, whose name should be provided by the user. The key should be generated according to the previous task. The user must provide (as program parameters or by request or any other suitable method):

- a password;
- the mode of the encryption algorithm, according to the previous tasks;
- the name of the file to encrypt; and
- the name of the file to store the cryptogram.

You can also use stdin and stdout for input and output, which enable you to used your program in a shell pipeline. This allows you to specify inout and output files as redirection parameters, instead of program parameters.

If you need to save multiple fields to the same file (e.g., salt and cryptogram), save these as fixed length fields to the beginning of the output file. An alternative is to use Base64 to convert the objects to text (`base64`), and then use a delimiter such as `\n`.

Take in consideration that data may need to be encrypted in blocks, and the last block may require padding, according to the previous tasks.

Questions:

- Can you determine the structure of the plaintext from the cryptogram?
- Compare the lengths of the plaintext and the cryptogram?

5 File decryption

Alter your program by adding a function that decrypts a user file. For this functionality, the user must provide an extra parameter, to tell whether it should encrypt or decrypt the input.

Take care of removing the padding (if present!)

Questions:

- Is padding visible in the decrypted text?
- What happens if the cryptogram is truncated?
- What happens if the cryptogram lacks some bytes in the beginning?

6 Patterns

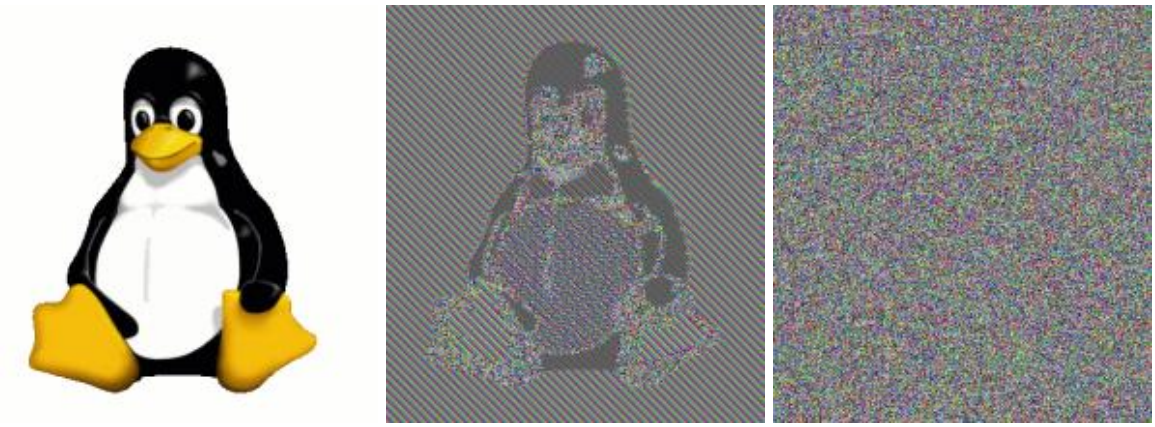
In this exercise we will analyze the impact of ECB and CBC cipher modes in the reproduction of patterns in the original document into the encrypted document. For this, we are going to encrypt an image in the bit map (BMP) file format, after which we are going to visualize the contents of the obtained encrypted file and compare it with the original image. In order we can visualize the contents of the encrypted file we must replace the first 54 bytes with the first 54 bytes from the original file (these bytes constitute the header of BMP formatted files, which is necessary so the file can be recognized as a BMP formatted file).

Use the program you developed in the previous sections to encrypt the file `Tux.bmp` using the ECB cipher mode. Using the `dd` application copy the copy the first 54 bytes from the original image file into the first 54 bytes of the encrypted file:

```
$ dd if=Tux.bmp of=Tux_enc.bmp ibs=1 count=54 conv=notrunc
```

Using a program to visualize images, open the original image and the encrypted image and compare them. What do you observe?

Repeat all the above operations, now using the CBC cipher mode instead of ECB cipher mode. Then, compare the original image with the obtained encrypted image.



An image and the typical result result of ECB and CBC cipher mode.

(Images by Larry Ewing)