Goal

In this Lab, you will

1. Find collisions on a (toy) hash function and experiment the birthday paradox;
2. Test the diffusion properties of AES;
3. Perform an attack on ciphertexts encrypted using AES in CBC mode with padding RFC2040.

0. Before starting

Download the file Lib.zip and extract it in the folder containing folders `Lib_light`, `Lab1` and `Lab2`

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/Lib.zip
```

```
$ unzip Lib.zip
```

Compile the library files.

```
$ cd Lib
```

```
$ make clean
```

```
$ make
```

```
$ cd ..
```

Then remove the former `Lib_light` directory.

```
$ rm -rf Lib_light
```

In `Lib`, we included some tool files such as `buffer.c` that you used last week and `bits.c` on which you worked in the previous lab. We also add some files to do crypto, namely : `aes.c` (an implementation by Brad Conte), `operating_modes.c` and `sha3.c`. Details are given further when

necessary.

Download the source files Lab3.zip, unzip them and move to this directory.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_3/Lab3.zip
```

```
$ unzip Lab3.zip
```

```
$ cd Lab3
```

You are now ready to code. Open your favorite source-code editor. If you are used to working with VS Code, this can be done with:

```
$ code .
```

1. Birthday paradox

A toy hash function is implemented in `easy_hash.c`. Your goal is to program the search for collisions on this function using a hash table. An implementation of hashtables is provided in `Lib/hashtable.{c,h}`. In the file `collisions.c`, you should complete the function `find_collision` such that it computes hash values for many random buffers of length 4 and waits for a collision. We ask that you print the correponding strings. Proceed as follows:

- Generate a hash table `H` and a table of buffers `tab` that you will use to store the buffers you already treated.
- Generate `imax` random buffers of length 4 (using function `buffer_random` in `Lib/Tools/buffer.c`). For the `i`-th one, compute its hash `h` and search in the hash table if this `h` has already been found using the function `hash_get`;
  - if not, put the pair (key, value) = `(h, i)` in the hash table using function `hash_put`.
  - Else, print the corresponding pair of buffers providing a collision using the `print_collision` function.

```
void print_collision(buffer_t *value1, buffer_t *value2)
```

- Finally, put the buffer at the `i`th entry of `tab`.

To test your program, simply try

```
$ make -f MakefileEx1
```

```
$ ./testEx1 <imax>
```

for increasing values of `imax`.

Upload your file `collisions.c` on VPL

2. AES, diffusion

In file `aes.c` in folder `Lib/Crypto` you have functions to perform AES encryption and decryption. For the moment we will only consider the case of encryption of one single block of 128 bits (i.e. 16 bytes).

Note. In this whole Lab we will only consider AES128, i.e. AES with 128 bits keys, i.e. 16 bytes keys.

The functions you can use are:

- `void aes_key_generation(buffer_t *key, int byte_length, uint seed)` fills in the buffer `key` with `byte_length` random bytes. In this Lab, `byte_length` will always be 16;
- `void aes_block_encrypt(buffer_t *out, buffer_t *in, buffer_t *key)` performs encryption and writes the ciphertext in buffer `out`;
- `void aes_block_decrypt(buffer_t *out, buffer_t *in, buffer_t *key)` performs decryption and writes the decrypted text in buffer `out`;
- `void aes_block_encrypt_few_rounds(buffer_t *out, buffer_t *in, buffer_t *key, int Nr)` performs encryption with `Nr` < 11 rounds and writes the ciphertext in buffer `out`. This last function will be useful in 2.2.

You can quickly test encryption and decryption using

```
$ make -f MakefileEx2
```

```
$ ./testEx2 0
```

that encrypts and decrypts the 16 bytes message "Crypto is great!".

2.1. Diffusion test

We want to test the diffusion property of the AES function. To do so, you are expected to complete the following functions in file `diffusion.c`

- `double diffusion_test_for_key(buffer_t *key, int nr_tests)`
- `double diffusion_test_for_msg(buffer_t *msg, int nr_tests)`

which work as follows.

The function `diffusion_test_for_key` takes as input a key and performs `nr_tests` times the following test:

- define a random value for the message `msg` using `buffer_random`;
- compute the AES encryption of the message with the key and write the result in buffer `encrypted`;
- draw a random integer `position` in [0, L [where L denotes the **bit size** of the key (different from the **byte size**) using the function `rand()`;
- flip the `position`-th bit of the key using the function `buffer_flip_bit` of `bits.c`, this yields `key2`;
- compute the AES encryption of the message with the new key after this bit flipping and save the result in buffer `encrypted2`;
- compute the number of bits where the two ciphertexts differ using the function `HammingDistance` in `bits.c`;
- return the average number of distinct positions of the ciphertexts over `nr_tests` tests.

The function `diffusion_test_for_msg` does the same but flips a bit of the plaintext instead of the key.

Test your functions with

```
$ make -f MakefileEx2
```

```
$ ./testEx2 1
```

and

```
$ ./testEx2 2
```

Give an interpretation of these results. If you cannot, ask your professor.

2.2. Diffusion test with various number of rounds

The number of rounds is a fundamental variable to perform an optimal diffusion. To observe it, fill in function `double diffusion_test_nr_rounds(buffer_t *msg, uint seed, int Nr, int nr_tests)` which is very similar to `diffusion_test_for_msg` but using `aes_block_encrypt_few_rounds` instead of `aes_block_encrypt` and performs `Nr` rounds for AES encryption.

Test your function with :

```
$ ./testEx2 3
```

Upload your file `diffusion.c` on VPL

3. CBC operating mode, padding and an attack from S. Vaudenay

Introduction

The raw use of AES for encrypting blocks is unsafe for several reasons explained during the lecture. Hence we use operating modes. In this exercise we will focus on Cipher Block Chaining (CBC) mode. This mode is described in slide number 27/28 of the lecture slides and implemented in `operating_modes.c`.

More precisely, the file `operating_modes.c` contains functions `aes_raw_CBC_encrypt` and `aes_raw_CBC_decrypt` that performs encryption and decryption in CBC mode.

Last block padding RFC2040. For this operating mode to be possible, the number of bits (resp. bytes) you encrypt should be a multiple of 128 (resp. 16). Since it does not hold for any plaintext, one should add extra characters to the last block to obtain a length that is a multiple of 128 bits. This operation is called *padding*. There are several methods to perform such a padding. One of them, called RFC2040 proceeds as follows.

- If the last block misses a bytes to be complete (for instance if the last block contains 11 bytes, it misses 5 bytes to be complete), then you complete it by appending a additional bytes whose value is a. For instance, if the last block has 11 bytes
  $m_0\ m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8\ m_9\ m_{10}$,
  it is padded as
  $m_0\ m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8\ m_9\ m_{10}\ 5\ 5\ 5\ 5\ 5$.
- Else, if the last block is complete, then you append an additional block containing only `16`'s.

To extract the plaintext from the padded text, it suffices to read the last byte a and remove the a last bytes.

In an article published in 2002, Serge Vaudenay proved that the padding RFC2040 is unsecure when using CBC mode. The goal of this exercise is to program a part of this attack.

An oracle

Suppose that you have an oracle which does the following: given a ciphertext, returns `true` if it is valid (i.e. if it is the CBC encryption of a plaintext with RFC2040 padding) and returns `false`

otherwise. Such an oracle is provided in file `attack_RFC2040.c` by the `oracle` function.

```
$ int oracle(buffer_t *encrypted, buffer_t *key)
```

In the real world, such an oracle can be simulated since, if a ciphertext is invalid, the receiver could send an error message or ask you to resend it.

A FUNNY PROPERTY

Suppose the ciphertext is composed of the following 16 byte blocks : $e_0\ e_1\ \dots\ e_n$ (e_0 is the initial value `IV`) and the corresponding plaintext is $p_1\ \dots\ p_n$. Let `i` be an integer, *0 < i <= n*. Suppose then that we replace the block e_{i-1} in the ciphertext by $e_{i-1} \oplus s$ for some 16 byte word `s`. Prove that the corresponding decrypted text will have $p_i \oplus s$ as its `i`-th block. Write the proof in the file `funny.txt` and upload it on VPL.

VAUDENAY'S ATTACK

Using the previous property and the oracle, one can build an attack as follows.

FINDING THE PADDING POSITION

Given an encrypted message, we know that the last block contains some padding. Using the property above, let us find at which position in the block the padding starts.

Consider the block e_{n-1} of the ciphertext. It is composed of 16 bytes: $e_{n-1} = e_{n-1,0} \cdots e_{n-1,15}$.

Xor $e_{n-1,0}$ with the value 1 and ask the oracle. If the oracle returns `false`, this means that the padding was starting at the 0-th byte of the block. Else, if the oracle returns `true`, keep going: Xor $e_{n-1,1}$ with 1, ask the oracle, etc. This way, you can identify the position of the first byte of the padding.

Complete the function

```
$ int get_padding_position(buffer_t *encrypted, buffer_t *key)
```

that performs this search and returns the position in `{0..15}` of the first byte of padding.

You can test your function with

```
$ make -f MakefileEx3
```

```
$ ./testEx3 2
```

Note that instead of Xoring with 1 you can Xor with any othe value. Why is that? Explain this in `funny.txt` and upload it once more.

Finding the value of the last byte

When you know the position of the first byte of the padding, you know how the plaintext is padded. Suppose it is padded with the value `5`, you can proceed as follows. Consider the 5 last bytes of e_{n-1} in the ciphertext and Xor them with $5 \oplus 6$. Then, consider the byte number 10 (the 6th one starting from the right) of the block $e_{n-1} = e_{n-1,0} \cdots e_{n-1,15}$ of the ciphertext. For any $x \in \{0..255\}$, Xor this byte with `a` and ask the oracle. When the oracle returns true, it means that the actual value of the last byte of the plaintext is $x \oplus 6$. Hence we can find the value of the plaintext of the byte.

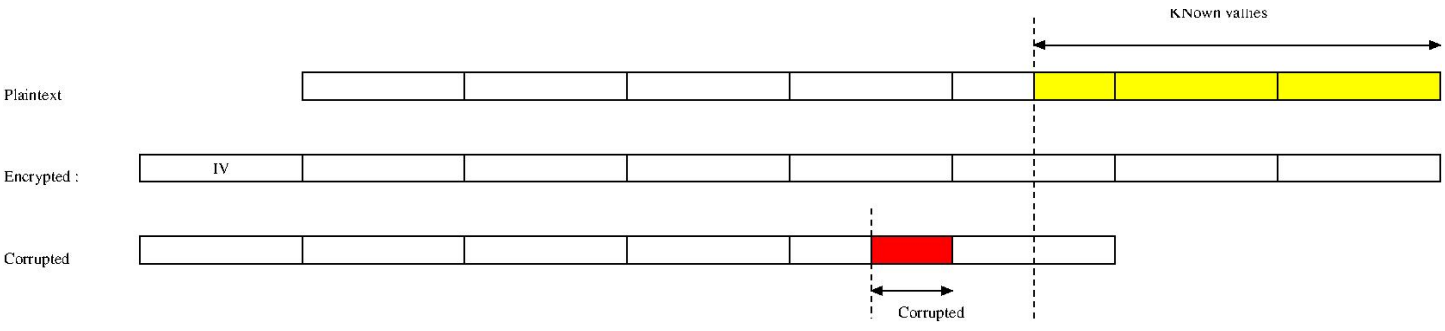
Why is this true? Give your answer in `funny.txt` an upload it again.

Preparing the corrupted ciphertext

In the general case, we know certain bytes of the padded plaintext (in yellow on the figure). They are stored in a buffer `decrypted` which has the same length as the padded plaintext and whose last bytes are those of the padded plaintext which we already know. Next, we create a buffer `corrupted` from the encrypted text by modifying some bytes in order to guess a new byte of the plaintext. Corrupted bytes appear in red on the figure.

Suppose that we know the value of all bytes indexed from position `known_positions` in the plaintext. If we were to cut the plaintext at the position `known_positions`, let `a` be the integer that would be used for padding.

- Define a `corrupted` buffer, whose values are equal to those of the encrypted text `encrypted`. The position `known_positions` of the plaintext should correspond correspond to the last block in `corrupted`, so the `corrupted` buffer might be shorter than the `encrypted` one (be careful, remember that the plaintext is one block shorter).
- In the penultimate block of `corrupted`, change the value of the `a` rightmost bytes by Xoring the known value and `a+1`.
- In the same block, corrupt the `(a+1)`-th byte (from the right), by Xoring `a+1`.



Example

```
Plaintext =
[070, 100, 049, 041, 100, 134, 237, 156, 215, 031, 194, 007, 037, 072, 032, 162]

Key =
[231, 055, 082, 144, 096, 231, 016, 169, 062, 151, 024, 221, 062, 041, 065, 142]
```

```
IV =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 232, 197, 197, 251, 090]

Encrypted =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 232, 197, 197, 251, 090]
[110, 250, 126, 196, 136, 200, 151, 123, 050, 061, 061, 104, 137, 033, 031, 023]
```

Imaging that we know the value of the plaintext starting from byte 12 (ie you know the last 4 bytes `037, 072, 032, 162`). We want to find the value of byte 11. We prepare the corrupted text as follows:

```
Corrupted =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 237, 229, 136, 222, 253]
[110, 250, 126, 196, 136, 200, 151, 123, 050, 061, 061, 104, 137, 033, 031, 023]
```

Explainations: For the last byte, we know that the plaintext is `162` and we want to turn it into a `005`. To do this we Xor `162^005=167` to the last byte of the previous block, which becomes `090^167=253`. We do the same for the three previous bytes. Concerning the byte that we want to guess (the 5-th starting from the end), we Xor `005` to the corresponding byte in the previous block, which becomes `232^005=237`. If we send this to the oracle, it will decrypt to the following value:

```
Decrypted (oracle) =
[070, 100, 049, 041, 100, 134, 237, 156, 215, 031, 194, 002, 005, 005, 005, 005]
=> the oracle returns FALSE.
```

Example (in the case where the number of known bytes is a multiple of 16)

```
Plaintext =
[070, 100, 049, 041, 100, 134, 237, 156, 215, 031, 194, 007, 037, 072, 032, 162]
[196, 168, 090, 235, 011, 032, 065, 073, 079, 139, 241, 248, 205, 048, 241, 019]

Key =
[231, 055, 082, 144, 096, 231, 016, 169, 062, 151, 024, 221, 062, 041, 065, 142]

IV =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 232, 197, 197, 251, 090]

Encrypted =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 232, 197, 197, 251, 090]
[110, 250, 126, 196, 136, 200, 151, 123, 050, 061, 061, 104, 137, 033, 031, 023]
[015, 127, 022, 176, 043, 020, 148, 222, 203, 005, 144, 202, 063, 061, 192, 208]
```

Imaging that we know the value of the plaintext starting from byte 16 of the plaintext (ie you know the last block). We want to find the value of byte 15. We prepare the corrupted text as follows:

```
Corrupted (prepared) =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 232, 197, 197, 251, 091]
```

```
[110, 250, 126, 196, 136, 200, 151, 123, 050, 061, 061, 104, 137, 033, 031, 023]
```

Explainations: We pretend that the plaintext is one block shorter. We Xor `001` to the last byte of the penultimate block of the corrupted ciphertext which becomes `090^001=091`. If we send this to the oracle, it will decrypt to the following value:

```
Decrypted (oracle) =
[070, 100, 049, 041, 100, 134, 237, 156, 215, 031, 194, 007, 037, 072, 032, 163]
=> the oracle returns FALSE because this is not correctly padded.
```

Complete the function `prepare` which fills in this `corrupted` buffer. Be carefull, you should also consider the case where `a` is a multiple of 16.

Caution! As you see, the buffers do not have the same lengths since the ciphertext starts with the IV. This makes things a bit cumbersome.

You can test your function with

```
$ make -f MakefileEx3
```

```
$ ./testEx3 3
```

Guessing a new byte

We are ready to guess a new byte of the plaintext using our corrupted text. For this, we define the function `find_last_byte`.

This function take as input the `corrupted` buffer (we can suppose that this vector has been properly prepared using the `prepare` function) and the integer corresponding to the `position` (indexed as in the ecrypted text) that we want to guess. Proceed as follow:

- For all possible values `x`, take your buffer `corrupted` and replace the position corresponding to the byte you wish to guess (offset by the block length because `position` corresponds to the index in `encrypted`) by this byte Xored with `x`.
- Then call the decryption oracle with `corrupted`. If the oracle returns true, then you got the guessed value : `x` else, increment the value of `x`.
- Note that `x` is a `uchar`, thus there are at most 256 tries to perform.

Example

We continue with the values of the previous example. Starting from the corrupted vector (from the `prepare` function), we iterate on all possible values of the byte. We replace `237` by `(237`

`^ x)` for all values `x` from `0` to `255`. We consult the oracle. For the value `x = 007` (and only this value), we obtain:

```
Corrupted =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 234, 229, 136, 222, 253]
[110, 250, 126, 196, 136, 200, 151, 123, 050, 061, 061, 104, 137, 033, 031, 023]

Decrypted (oracle) =
[070, 100, 049, 041, 100, 134, 237, 156, 215, 031, 194, 005, 005, 005, 005, 005]
=> the oracle returns TRUE.
```

For all other values of `x`, the oracle returns `FALSE`. Hence, we deduce that the value of the byte in the plaintext is `007`.

Example (in the case where the number of known bytes is a multiple of 16)

We continue with the values of the previous example. Starting from the corrupted vector (from the `prepare` function), we iterate on all possible values of the byte. We replace `091` by `(091 ^ x)` for all values `x` from `0` to `255`. We consult the oracle.

```
Corrupted =
Corrupted (prepared) =
[080, 107, 078, 091, 140, 143, 164, 219, 027, 149, 211, 232, 197, 197, 251, 091^x]
[110, 250, 126, 196, 136, 200, 151, 123, 050, 061, 061, 104, 137, 033, 031, 023]

Decrypted (oracle) =
[070, 100, 049, 041, 100, 134, 237, 156, 215, 031, 194, 007, 037, 072, 032, 163^x]
```

The oracle will return `TRUE` if and only if `x==162`.

Complete the function `find_last_byte`. You can test your function with

```
$ make -f MakefileEx3
```

```
$ ./testEx3 4
```

The full attack

Finally, complete the function `full_attack` that permits to decrypt completely the message. You can test your function with

```
$ make -f MakefileEx3
```

```
$ ./testEx3 5
```

Upload your file `attack_RFC2040.c` on VPL