Goal

In this Lab, you will perform a correlation attack to break the Geffe generator, a stream cipher based on three LFSRs.

0. Before starting

Download the archive Lab2.zip.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_2/Lab2.zip
```

Unzip the file.

```
$ unzip Lab2.zip
```

You should also download the Lib files Lib_light.zip and unzip the `Lib_light` in the folder that contains the folder `Lab2`.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_2/Lib_light.zip
```

```
$ unzip Lib_light.zip
```

The folder `Lib_light` contains your file `buffer.c` with buffers and functions on buffers. You do not have to modify these files. You should only go in the `Lib_light folder` and execute the Makefile.

```
$ cd Lib_light
```

```
$ make clean; make
```

This creates a first version of the library `inf558.a`. See the documentation. Feel free to read it and use the functions you may need.

Then go in the Lab2 folder.

```
$ cd ../Lab2
```

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. Playing with bits

In this lab, we handle bytes represented as integers between 0 and 255. The best type for that is `unsigned char` which has been redefined for convenience as `uchar`.

We first make some basic exercises to handle bits. In a byte, i.e. in an `uchar`, bits are indexed from 0 to 7. They are ordered from right to left from the least significant one to the most significant one. That is, the byte $a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ represents the integer : $a_7.2^7 + a_6.2^6 + a_5.2^5 + a_4.2^4 + a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0$. For instance: 19 has binary representation : `00010011`.

In file `bits.c`, complete the functions:

- `uchar getBit(uchar t, int position)` returning the value of the bit number `position`.
- `uchar setBit(uchar t, int position, uchar value)` which returns an `uchar` obtained from `t` by setting bit number `position` at `value` (value is either 0 or 1).
- `int HammingWeightByte(const uchar c)` returns the number of 1's in a byte.
- `int HammingWeight(buffer_t *buf)` returning the number of 1's in a buffer.

The 'buffer' object: for this question and in the rest on this Lab you will be manipulating buffers. These objects are defined in the `inf558.a` library that you created in the `Lib_light` directory. Take some time to look at the `buffer.h` file to understand the structure and see the available functions. You will have to use some of these functions in the rest of the Lab. The name of the functions is usually explicit but you can read the documentation of the library if you want more details.

Optional question (only if you have time): complete the function

- `void oneTimePad(buffer_t *encrypted, buffer_t *msg, buffer_t *key)` implements Vernam's cipher by encrypting the plaintext `msg` with the key `key` and writes the ciphertext in the buffer `encrypted`. You may want to use the function `buffer_append_uchar` from `buffer.c`.

Tests: once you have completed the functions, you can test your code using:

```
$ make -f MakefileEx1
```

```
$ ./testEx1 <x>
```

where `x = 1, ....` corresponds to the function you wish to test

1. tests `getBit`;
2. tests `setBit`;
3. tests `HammingWeightByte`;
4. tests `HammingWeight`;
5. tests `oneTimePad`;

6. tests `oneTimePad` but is more funny than the previous ones;

You can also perform the complete list of tests as follows:

```
$ make -f MakefileEx1
```

```
$ make -f MakefileEx1 tests
```

Upload your file `bits.c` on VPL and evaluate your submission.

2. LFSR

An LFSR is represented as follows.

The stream $(s_n)_n$ is defined as follows:

- The first L bits (s_0, ..., s_{L-1}) are provided: they are called the *initial value* (or `IV`).
- The next bits are obtained thanks to the following linear relation:
  $$s_{t+L} = c_0 \bullet s_t \oplus \ldots \oplus c_{L-1} \bullet s_{t+L-1}$$
  where • stand for AND gates and $\oplus$ stand for XOR gates. The sequence of bits $c_0, \ldots, c_{L-1}$ is referred to as the *transition vector*

Caution. Even if the c_i's and s_i's are bits we will store them by blocks of eight bits in bytes i.e. into `uchars`. Therefore, in the file `LFSR.c`, the initial value will be represented as a `buffer_t` of `L/8` bytes. In particular, from now on, we always suppose that `L` is a multiple of 8. Similarly, the transition vector is also represented as a `buffer_t` of length `L/8`.

Play with your LFSR (nothing to program)

In the file `LFSR.c`, the function `void LFSR(buffer_t *stream, buffer_t *trans, buffer_t *IV, int stream_length)` provides an LFSR stream. Its inputs are

- `stream` the buffer in which the output of the LFSR will be written;
- `trans` the transition vector;
- `IV` the initial value;
- `length` the length of the stream we want to compute, i.e. the length of `stream` when the execution is done. **Caution** the variable `length` refers to as the number of **bytes** of the output stream. Hence, the output stream contains `8*length` bits.

You can test it using

```
$ make -f MakefileEx2
```

```
$ ./testEx2 1
```

Then you can observe the distribution of 0's and 1's in the output stream by using

```
$ ./testEx2 2
```

3. Exhaustive search for key recovery

In an LFSR, the value of the transition vector `trans` ($c_0, \ldots, c_{L-1}$) is usually known, whereas the initial value `IV` ($s_0, \ldots, s_{L-1}$) plays the role of the key and is kept secret.

Given the output stream of an LFSR and its transition vector, a key recovery attack consists in finding the initial value `IV`. This is what we want to achieve here. One way to do that is ... bruteforce: we try all possible `IV` values, compute the stream corresponding to this initial value and see if it matches our stream.

For this, we will need to iterate on all possible values of `IV`. We start by writing an auxiliary function for this. Then we perform the exhaustive search.

In file `LFSR.c`,

1. complete the function `increment_buffer(buffer_t *buf)`. This function increments the integer represented by the buffer. We use the following convention:
   - the buffer has length `buf->length` and hence represents an integer of `8*buf->length` bits;
   - the most significant bits are on the left. Therefore, a buffer `buf` of length 3 such that `buf->tab = [a, b, c]` represents the integer `c + b * 256 + a * 256`^{2}
   - the length of the buffer is fixed. No need to extend it when you reach the largest integer represented by the buffer!

2. complete the function

```
$       bourrinate_IV(buffer_t *searched_IV, buffer_t *trans, buffer_t *stream)
```

which performs the exhaustive search as follows:

- `stream` is the output stream of our LFSR whose initial value is unknown;
- `trans` is its transition vector.
- First initialise `searched_IV` as a buffer of length `trans->length` and whose bytes are all equal to `0`.
- Compare the output stream of an LFSR of transition vector `trans` and initial value `searched_IV` with `stream` using the function `buffer_equality`.
- If the streams are equal, then you are done, stop the search. Else increment your buffer using `increment_buffer` and try again.

Tests: once you have completed the functions, you can test your code using:

```
$ make -f MakefileEx3
```

```
$ ./testEx3 0
```

Then you can try again with

```
$ ./testEx3 u
```

where `u` is an integer in `{0..255}`. The value `u` will be the first byte of the searched IV (the key). Because we start the search at zero, the larger the value of `u`, the longer the search. We suggest to test the cases `u = 1, 2, 4, 8, 16`. Indeed, You can evaluate the time using the function `time` of the shell:

```
$ time ./testEx3 u
```

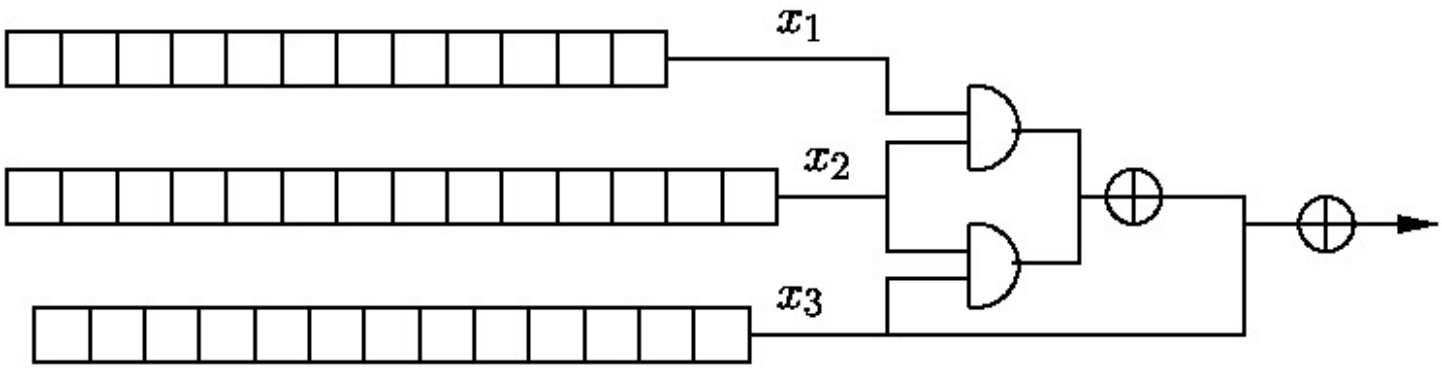
Upload your file `LFSR.c` on VPL and evaluate your submission.

4. The Geffe generator

The Geffe generator is a stream cipher obtained by combining three LFSR. At eack clock cycle the LFSRs outputs are x_1, x_2 and x_3 and the Geffe stream cipher outputs:

$$F(x_1, x_2, x_3) = x_1.x_2 \oplus x_2.x_3 \oplus x_3.$$

This stream cipher is represented by the following picture.



4.1 Design

In file `Geffe.c`, complete the function

```
$   void Geffe(buffer_t *output, buffer_t *s1, buffer_t *s2, buffer_t *s3)
```

that takes as input the output streams `s1`, `s2`, `s3` of the LFSR's and writes the output of Geffe's cipher into the buffer `output`. You may use the `buffer_append_uchar` function.

You can test it with `testEx4.c`. Commands:

```
$ make -f MakefileEx4
```

```
$ ./testEx4 1
```

should print:

```
***************** Testing Geffe cipher ********************

LFSR1 output :   [ 1 0 1 0 1 0 1 1 | 0 1 0 1 1 1 1 0 | 1 1 0 0 1 1 1 1 | 0 0 1 0 0 1 0 0 | 0 0 0 0 0 1 1 1 ]
LFSR2 output :   [ 0 0 0 0 0 0 1 1 | 0 1 1 1 0 1 1 0 | 1 0 0 1 1 1 1 0 | 0 1 1 1 1 0 0 0 | 0 1 0 0 1 0 0 0 ]
LFSR3 output :   [ 0 0 0 0 0 0 0 1 | 0 0 0 0 1 1 1 0 | 0 0 0 0 0 1 0 0 | 1 1 0 0 0 0 1 1 | 0 1 0 0 1 1 1 0 ]
Geffe's output : [ 0 0 0 0 0 0 1 1 | 0 1 0 1 1 1 1 0 | 1 0 0 0 1 1 1 0 | 1 0 1 0 0 0 1 1 | 0 0 0 0 0 1 1 0 ]

[OK]
```

Upload your file `Geffe.c` on VPL and evaluate your submission.

4.2. Siegenthaler's attack on the stream cipher

Observation

Here is two observations that we will use to break Geffe's generator. You will prove later that these statements are true.

1. Prob($z = x_1$) = Prob($z = x_3$) = 3/4.

2. If $x_1 = 1$ and $x_3 = 0$, then $z = x_2$.

Note. The transition vectors of the LFSR's are supposed to be known... which is reasonable: when using combined stream ciphers, the secret key is the *seeds*, i.e. the initial values (`IV`) of each LFSR's. This is not the same problem as when you use a single LFSR, for which both seed and transition vectors must be kept secret (take a minute to be sure to understand why).

Outline of the attack

Using this observation, here is how you will proceed to attack the generator (each step is detailed in the next session).

1. Perform an exhaustive search on the initial value of the first LFSR as follows. For each initial value, generate the corresponding stream and compare it with the output stream of the Geffe cipher. Thanks to observation 1, if both sequences are correlated (about 75% of bits in common), then you have guessed the initial value of the first cipher.

2. Proceed similarly to find the initial value of the third LFSR.

3. Now, the output streams of the first and third LFSR are known. You will use the second observation to find the initial value of the second LFSR. From the knowledge of the transition polynomial, you could deduce its initial value by solving a linear system, but this method would be too cumbersome to program, thus we suggest to find the initial value of the second LFSR by a third exhaustive search and by comparing the output stream with the known values of the outputs of the second LFSR.

Programming

- Complete function

```
$ double correlation(buffer_t *s1, buffer_t *s2)
```

computing the correlation between the two buffers, i.e. the number of bit positions where they match divided by the total number of bits (buffers are supposed to have the same length).
You can test it with:

```
$ ./testEx4 2
```

- Complete function

```
$ void searchIV(buffer_t *IV_candidate, buffer_t *stream, buffer_t *trans, double threshold)
```

which performs an exhaustive search in a similar manner than `bourrinate_IV` but stops when `IV_candidate` is the IV of an LFSR with transition vector `trans` whose output stream has correlation above `threshold` with the stream `stream`.
You can test it:

```
$ ./testEx4 3
```

- Complete function

```
$ void positions(buffer_t *output, buffer_t *s1, buffer_t *s3)
```

which fills in the `ouptut` buffer with a sequence of bytes whose bits equal to 1 correspond to positions where `s1` equals 1 and `s3` equal 0.
You can test the function :

```
$ ./testEx4 4
```

- Complete the function

```
$ int match_at(buffer_t *s, buffer_t *s1, buffer_t *pos)
```

which returns
  - 1 if, for all positions where `pos` has a bit equal to 1, `s` and `s1` match at this position (or, in mathematical terms, `for all i, pos[i]==1 => s[i]==s1[i]`);
  - 0 otherwise.

Test with

```
$ ./testEx4 5
```

- Complete the function

```
$ void search_with_match(buffer_t *IV_candidate, buffer_t *stream, buffer_t *trans, buffer_t *pos)
```

which performs an exhaustive search on the initial values of an LFSR with transition vector `trans`, in order to find the initial value of a cipher whose output matches with `stream` at the positions suggested by `pos`. It should be written in the same spirit as `bourrinate_IV` and `search_IV`.
Test with

```
$ ./testEx4 6
```

- You are finally ready to perform the attack! Complete the function

```
$ void attack(buffer_t *IV_candidate1, buffer_t *IV_candidate2, buffer_t *IV_candidate3,
            buffer_t *stream, buffer_t *trans1, buffer_t *trans2, buffer_t *trans3,
            double threshold);
```

which performs the complete attack (as explained in the outline) by using the previous functions.
Test with

```
$ ./testEx4 7
```

If it succeeds, test with a more difficult challenge (this may take some time).

```
$ ./testEx4 8
```

Some questions:

1. Let z be the output of the cipher. Prove that
$$\text{Prob}(z = x_1) = \text{Prob}(z = x_3) = 3/4.$$
2. What is the complexity of a brute force attack on Geffe's generator in terms of the lengths l_1, l_2, l_3 of the LFSR's?
3. What is the complexity of the above attack?
4. What is the complexity of the attack when using linear algebra to guess the second LFSR?
5. Compared to a brute force attack, what is the interest of your attack when the length of the three LFSR's is about 30 bits?

Write your answers in the file `Geffe.txt` on VPL.