Data Security and Privacy [106568]

Lab 1: Correlation attacks in stream ciphers

In this assignment we will experiment with performing a correlation attack on a stream cipher. More precisely, we will study a nonlinear stream-cipher key generator that combines the outputs of three linear-feedback shift registers (LFSRs), as shown in Figure 1.

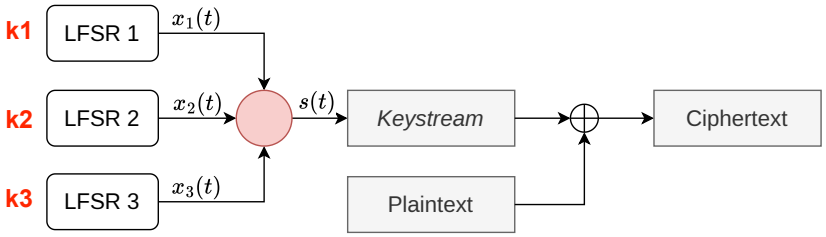

Figure 1: Our stream cipher

At time t, the three LFSRs output the bits $x_1(t)$, $x_2(t)$ and $x_3(t)$. The keystream bit $s(t)$ is computed as follows:

This is the same as saying when x1 = 0 then i get x3.
s= x2 if x1=1
s= x3 if x1=0

$$s(t) = \big(x_1(t) \wedge x_2(t)\big) \oplus \big(\neg x_1(t) \wedge x_3(t)\big). \quad (1)$$

Here $\wedge$ denotes logical AND, $\oplus$ denotes XOR, and $\neg$ denotes NOT. This nonlinear combination aims to produce a more complex keystream. The initial states of the three LFSRs constitute the cipher key.

1 Important information

For this lab you must submit the answers to the exercises and the code used to complete them. The code can be a Python file or a Jupyter notebook. You can, for example, deliver the report as a PDF and include or point to the required code (Python files or notebook).

- The only accepted file formats are PDF, text, Python (`.py`) files, or Jupyter notebooks.
- Each submitted file must clearly indicate the names of the group members and their corresponding NIUs.
- The written answers must be concise and precise. Avoid overwriting and bloated answers. Favor correctness, clarity, precision, and brevity.

Files must be submitted through the Moodle Classroom via the dedicated submission. Only one submission per work group is required.

⚠ Remember that an individual in-person validation of the lab is required during the final session. You will need to answer questions about your work and the solutions you provided. If you do not successfully pass this validation, you will not receive a grade for the lab.

2 Preliminaries

In this section we will describe the code provided with the lab in the file `scipher.py`, and some related aspects.

2.1 Representation and encoding

Convert text → binary via UTF-8.

Encrypt binary → random bits (not readable text).

Encode encrypted bits → hexadecimal (safe, printable).

We will encrypt text. Text can be represented in binary using its UTF-8 encoding (using 8, 16, 24, or 32 bits unequivocally), which is convenient in Python. Once encrypted, the resulting bits usually cannot be interpreted as UTF-8 characters, since arbitrary bit sequences may not correspond to valid printable characters. To avoid this, encrypted data will be encoded using the hexadecimal representation of the binary string. As a summary, we will deal with the following encodings:

- *bits*: a list of integers (0 or 1). This is the basic internal representation for binary data (for example, keystream bits).
- *text*: UTF-8 encoding of text. Each character is represented by one or more bytes. We will us it to represent the plaintext.
- *hex*: the hexadecimal string corresponding to a given binary string (*bits*). We use it to represent the ciphertext.

You might have noticed that this is by no means the best approach. It is hard to think of a less efficient way to work with binary numbers than using such lists of integers. But we are here to learn cryptography, so we have opted for this approach for clarity.

2.2 `scipher.py`: LFSR and Cipher classes

You can find the implementation of the stream cipher in the `scipher.py` file. It is implemented as two main classes: `LFSR` and `Cipher`, and includes some helper functions.

- **LFSR class**: represents a linear-feedback shift register. It has two main attributes: its initial state and its feedback (or characteristic) polynomial.
  - `state`: A list of bits (0/1). The leftmost bit (`state[0]`) is the output.

    It's 'backwards', that's why we pick the leftmost as the output.
  - `poly`: A list of binary coefficients that defines the feedback function (`poly[0]` is the constant term / lowest degree coefficient). For example, the polynomial $x^9 + x^4 + x^3 + x + 1$ is encoded as `[1, 1, 0, 1, 1, 0, 0, 0, 0, 1]`.

    1, x, x2, x3, x4, x5, x6, x7, x8, x9
  - **Methods:**
    - `clock()`: advances the LFSR by one step and returns the output bit.

  Example: generating a keystream of length 10:

```
import scipher

initial_state = [1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1]
polynomial = [1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1]

lfsr = scipher.LFSR(initial_state, polynomial)
keystream = [lfsr.clock() for _ in range(10)]
```

- **Cipher class**: implements the cipher that combines three LFSRs as described above. The polynomials for the three LFSRs are:

  $$\text{LFSR1} : x^{20} + x^3 + 1$$
  $$\text{LFSR2} : x^{19} + x^5 + x^2 + x + 1$$
  $$\text{LFSR3} : x^{21} + x^2 + 1$$

  To instantiate the class, you only need the initial state of the LFSRs, the `key_bits`.

  - `key_bits`: a tuple with the three initial states of the LFSRs, each defined as a list of bits.
  - **Methods:**
    - `keystream_bits(n)`: generate n keystream bits.
    - `encrypt_to_hex(text)`: encrypt a UTF-8 string and return the ciphertext in *hex*.
    - `decrypt_from_hex(hex_text)`: decrypt a ciphertext in *hex* back to a UTF-8 string.

  Example: encryption/decryption with the `Cipher` class.

```python
import scipher

key_bits = (
    # LFSR1 initial state (20 bits)
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1],
    # LFSR2 initial state (19 bits)
    [0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0],
    # LFSR3 initial state (21 bits)
    [0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0],
)

cipher = scipher.Cipher(key_bits)

ct_hex = cipher.encrypt_to_hex("Hello DSP!")
print("Ciphertext:", ct_hex)
pt = cipher.decrypt_from_hex(ct_hex)
print("Decrypted:", pt)
```

Initial states of the LFSRs

- **Helper function**: the module `scipher.py` also contains some helper functions:
  - `read_string_from_file(file_path)`: given a file, return a string with its contents.
  - `hex_to_bits(hex_str)`: convert an hexadecimal string (*hex* ciphertext) to a list of bits.
  - `str_to_bits(text)`: convert an string (plaintext) to a list of bits.

2.3 Goal of the assignment

The goal of the assignment is to perform a cryptanalysis of the stream cipher by exploiting correlations between the outputs of the LFSRs and the keystream. We will perform a known-plaintext attack.

We assume the cryptosystem uses the same key during an entire day. For each message the system resets the keystream (the LFSRs are initialized to their initial states). We know the first message sent each day is a song used to cheer up the users. Today we could guess that song and thus we have both the *plaintext* and its corresponding *ciphertext*. Later we observe an encrypted *secret* message. Our goal is to recover that day's keys so we can decrypt the secret message.

The files you need are:

- `NN-known-plaintext.txt`: plaintext of the song sent in the morning.
- `NN-known-ciphertext.txt`: intercepted ciphertext, which corresponds to the previous song.
- `NN-secret-ciphertext.txt`: intercepted secret message to be decrypted.

The files can be downloaded from the URLs listed in Table 1, where `NN` is your lab group number (if you are group 2, then $NN = 02$).

http://deic.uab.cat/~gnavarro/dsp/NN-known-plaintext.txt
http://deic.uab.cat/~gnavarro/dsp/NN-known-ciphertext.txt
http://deic.uab.cat/~gnavarro/dsp/NN-secret-ciphertext.txt

Table 1: Intercepted messages

A naive approach would be to brute-force the key (i.e., try all possible initial states for the three LFSRs) and check which one encrypts the `known-plaintext` into the `known-ciphertext`. This requires checking an average of 2^{56} possible keys. We will instead use a more efficient approach: **attack each LFSR separately using statistical correlations**.

3 Exercises

We will perform the attack step by step. Please follow the exercises, and answer the points and indications of the boxed parts.

3.1 Exercise 1: Find potential correlations

We want to analyze our stream cipher looking for possible vulnerabilities. Determine whether there is a correlation between the **individual** outputs of the LFSRs and the keystream. That is, determine if there is a correlation between any of the LFSRs separately and the keystream.

(a) Explain how you find these possible correlations analytically.
(b) Explain how you validate them empirically.
(c) Provide or identify the code used to do it.

Note: As a result, you should have found that the output of at least one LFSR has a correlation with the keystream!.

3.2 Exercise 2: Recover the keystream of the known ciphertext

Attempt to reconstruct the maximal portion of the keystream employed to encrypt `NN-known-ciphertext.txt`.

(a) Explain how you recovered the keystream.
(b) Provide or identify the code used to do it.

3.3 Exercise 3: Find the initial state of a correlated LFSR

If an LFSR output is correlated with the keystream, find its initial state.

(a) Explain how you find such an initial state and show the result.
(b) How many bits from the keystream did you use? What are the implications of using more or less bits from the keystream to find such correlations?
(c) Provide or identify the code used to do it.

3.4 Exercise 4: Define a strategy for a non-correlated LFSR

If an LFSR's output is not correlated to the keystream, define how you can find its initial state. ❗ This strategy should improve on a brute-force search by leveraging the information obtained from correlated LFSR(s).

(a) Explain how you can find the initial state for the LFSR(s) that are not correlated.

3.5 Exercise 5: Find the key

Find the key of the stream cipher.

(a) Find the key for the stream cipher using the previous results and strategies.
(b) Provide or identify the code used to do it.
(c) Give an estimate of the computation time required to find it.

Note: on a relatively modern laptop, the whole process (ex.3 and ex.5) typically takes around 3-4 minutes.

3.6 Exercise 6: Decrypt the secret message

Decrypt the secret message `NN-secret-ciphertext.txt`. with the key obtained earlier. Success or failure should be obvious.

(a) Retrieve and show the secret message.
(b) Provide or identify the code used to do it.