Faculté sciences & sciences de l'ingénieur

Université Bretagne Sud

Stream Cipher Based on Linear Feedback Shift Register

Azamat Shirinshoev, Kamran Khan Arabzai

Cryptology

Gogniat Guy

October 2, 2024

Contents

1 Introduction

In today's digital world, ensuring the security of the information has become a critical requirement. Cryptography plays a key role in protecting the data from unauthorized access. Evolving from ancient times to the present day, cryptography has advanced through the implementation of various algorithms that ensure data security. The stream cipher is one such crucial cryptography technique. Stream ciphers relates to symmetric algorithms that process the data bit by bit, or byte by byte to make them highly efficient for encrypting continuous streams of data in real-time applications such as video streaming and telecommunication. In order to that, stream ciphers often use a special tool called a Linear Feedback Shift Register (LSFR). An LFSR is a shift register that shifts its bits while using feedback from some of its positions to determine the new input bit. This mechanism makes LFSRs ideal for lightweight encryption algorithms, such as stream ciphers, which require fast and efficient key stream generation.

1.1 Objective of the Lab

The objective of this lab is to implement a stream cipher based on Linear Feedback Shift Registers (LFSRs) and to evaluate its performance and security. By exploring the structure and behavior of LFSRs, we aim to understand how the generated key stream influences the overall encryption process.

1.2 Materials and Methods

- All experiments and analyses were conducted within a Jupyter Lab environment.
- Python was used as a primary programming language.

1.2.1 Software and Libraries

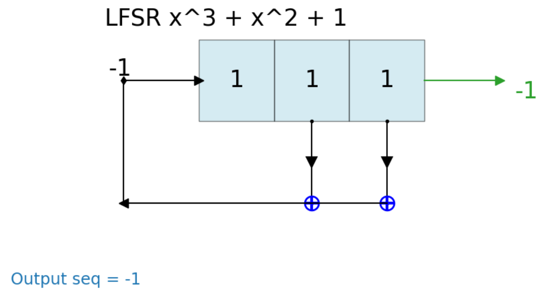
- **pylfsr:** This package was used to create, simulate and analyze LFSRs.
- **NumPy:** IIt helped us work with the bits and bytes like building blocks, making it easier to scramble and unscramble the secret messages during encryption and decryption.
- **Matplotlib.pyplot & pyplot:** These libraries were used to create visualization of the LFSR outputs, keystreams and other relevant data.
- **PIL(Pillow):** This library was used to capture the images and getting them ready to be turned to GIF animation.
- **io:** The **io** module helped to manage the process of saving the images and creating the final GIF file.

2 LFSR Implementation and Analysis

The simplicity, efficiency, and ability to produce long sequences make LFSR essential for secure communication. In this lab we explored how to implement LFSR using the pylfsr Python package and analyze their structure and behaviour.

To demonstrate, this code provides simple and clear image of a 3-bit LFSR with feedback polynomial `x^3 + x^2 + 1` using `pylfsr` package. It is initialized by state = [1, 1, 1] . Then the feedback polynomial `x^3 + x^2 + 1` is represented by the list fpoly = [3,2]. The exponents 3 and 2 shows that the feedback taps are on 3rd and 2nd position of the LFSR. These feedback taps determines how the feedback values are calculated. Basically, the feedback values is generated by XORing 3rd and 2nd registers and using this result to shift the the registers. Each step involves shifting all registers to the right and placing new feedback value in the leftmost register. To vizulaize it in the code L.Viz() method is used to visualize the LFSR. All in all, by using the pylfsr package we can simulate this behaviour and observe the generated sequence as well as the internal state transitions.

```
state = [1,1,1] # Initialisation state␣
↪is 1, 1, 1
fpoly = [3,2] # the LFSR feedback␣
↪polynomial
L = LFSR(fpoly=fpoly,initstate =state,␣
↪verbose=True)
L.Viz(title='LFSR x^3 + x^2 + 1')
```



2.1 Manual explanation of all the states of the LFSR

We begin with the initial state of the LFSR, which is [1, 1, 1]. This means all three registers start with a value of 1. The feedback is generated by XORing the values of the 3rd and 2nd registers. In each step, this feedback value is used to update the leftmost register, while the other bits shift to the right. At the same time, a keystream bit is generated from the rightmost register. The process repeats as the LFSR transitions through various states, cycling through the same sequence after a few steps. Below is the detailed breakdown:

Step	State	Feedback (XOR)	Keystream Bit
1	[1, 1, 1]	$1 \oplus 1 = 0$	1
2	[0, 1, 1]	$1 \oplus 1 = 0$	1
3	[0, 0, 1]	$1 \oplus 0 = 1$	1
4	[1, 0, 0]	$0 \oplus 0 = 0$	0
5	[0, 1, 0]	$0 \oplus 1 = 1$	0
6	[1, 0, 1]	$1 \oplus 0 = 1$	1
7	[1, 1, 0]	$0 \oplus 1 = 1$	0
8	[1, 1, 1]	$1 \oplus 1 = 0$	(Repeats)

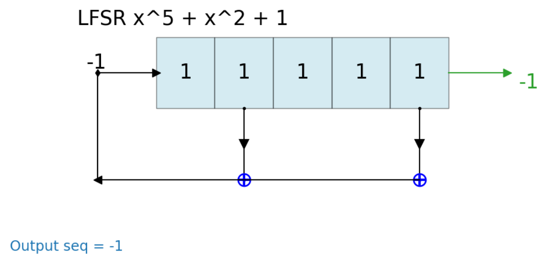
When we compare the keystream generated manually with the keystream produced by running the code, we observe that they are identical. Using the **L.runFullPeriod()** function, the code goes through the full period of the LFSR states and outputs the following:

```
S:  [1 1 1]
S:  [0 1 1]
S:  [0 0 1]
S:  [1 0 0]
S:  [0 1 0]
S:  [1 0 1]
S:  [1 1 0]
array([1, 1, 1, 0, 0, 1, 0])
```

2.2 Second exemple p(x) = x^5 + x^2 + 1

When testing for the 5-bit LFSR with feedback polynomial `x^5 + x^2 + 1`, we follow the same procedure as it was done for the above example. First, after executing the following code, we can analyze the structure of the LFSR and its initial state.

```
    L = LFSR()
    L.Viz(show=False,␣
↪show_labels=False,title='LFSR x^5␣
↪+ x^2 + 1')
```



From the image and the polynomial we know that the feedback taps are in 5th and 2nd positions of the LFSR. When perfoming the XOR (1 XOR 1) the leftmost register will update to 0 [0, 1, 1, 1, 1], and the keystream bit, which is the output of rightmost register, to 1. Meanwhile, the other bits shift to the right. The following table shows how we manually tested ten first states of the LFSR starting from the initial state.

Step	State	Feedback (XOR)	Keystream Bit
1	[1, 1, 1, 1, 1]	$1 \oplus 1 = 0$	1
2	[0, 1, 1, 1, 1]	$1 \oplus 1 = 0$	1
3	[0, 0, 1, 1, 1]	$1 \oplus 0 = 1$	1
4	[1, 0, 0, 1, 1]	$1 \oplus 0 = 1$	1
5	[1, 1, 0, 0, 1]	$1 \oplus 1 = 0$	1
6	[0, 1, 1, 0, 0]	$0 \oplus 1 = 1$	0
7	[1, 0, 1, 1, 0]	$0 \oplus 0 = 0$	0
8	[0, 1, 0, 1, 1]	$1 \oplus 1 = 0$	1
9	[0, 0, 1, 0, 1]	$1 \oplus 0 = 1$	1
10	[1, 0, 0, 1, 0]	$1 \oplus 1 = 0$	0

To make sure if our result [1111100110] is correct, by running the following code

```
L.runFullPeriod()
L.info()
```

we could compare and make sure that the first 10 bits of the keystream are identical.

Output Sequence: 11111001101001000010101110110001

3 LFSR Properties

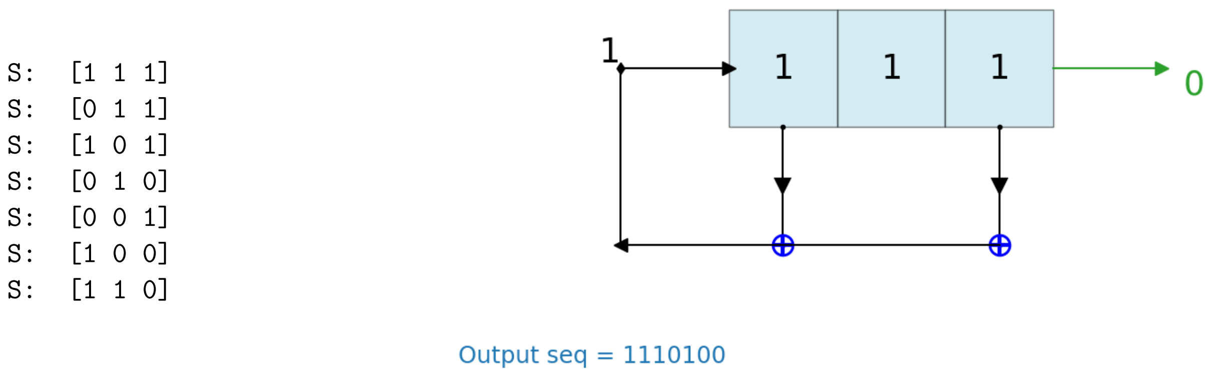
In this section, we discuss the essential properties of LFSR, focusing on primitive and non-primitive polynomials, and how these properties impact the security and performance of LFSR. Furthermore, it is important to mention the key properties of periodicity, balance, run length, and autocorrelation, highlighting their relevance in assessing the effectiveness of LFSRs in cryptographic applications.

3.1 Feedback Primitive Polynomials

A primitive polynomial is like a prime number for polynomials; it can't be factored into smaller polynomials using only 0s and 1s (in a field called GF(2)). This property is crucial for LFSRs because it guarantees the LFSR generates the longest possible sequence—a period of $2^n - 1$—before repeating, cycling through all possible states except the all-zero state, where n is the degree of the primitive polynomial.
For example, with a 3-bit LFSR, the expected period should be $2^3 - 1 = 7$. To test this, we select a 3-bit polynomial such as $x^3 + x + 1$ and then verify it using the following code:

```python
state = [1,1,1] # 3 registers and initialisation state is 1, 1, 1
fpoly = [3,1] # descrption of the polynomial x^3+x+1
L = LFSR(fpoly=fpoly,initstate =state, verbose=True)
L.runFullPeriod()
L.Viz(show=False, show_labels=False)
```

After running this code, we can confirm that this polynomial is indeed primitive, as the LFSR outputs 7 unique states before repeating, meeting the maximum period for a 3-bit LFSR.


```
S:  [1 1 1]
S:  [0 1 1]
S:  [1 0 1]
S:  [0 1 0]
S:  [0 0 1]
S:  [1 0 0]
S:  [1 1 0]
```

Output seq = 1110100

3.1.1 Simulation of Primitive Polynomial

For better understanding using Python specifically the `pylfsr` library we simulate the LFSR, which runs over 22 iterations, and visualize the states at each step. The final result is stored as a GIF for further analysis. The following steps summarize the key aspects of the simulation:

LFSR Initialization:

The LFSR is initialized with an initial state of $[1, 1, 1]$ and a feedback polynomial of $x^3 + x + 1$.

```python
# Initialize the LFSR
from pylfsr import LFSR
L = LFSR(initstate=[1,1,1], fpoly=[3,1])
```

LFSR State Visualization:

The visualization updates the plot for each LFSR state during 22 iterations. The state is updated using the `next()` function and is plotted dynamically.

```
# Visualization loop
for _ in range(22):
    ax.clear()            # Clear the previous plot
    L.Viz(ax=ax, title=r"LFSR x^3 + x + 1")
    fig.canvas.draw()
    fig.canvas.flush_events()
    L.next()            # Move to the next state
```

Saving the Visualization as GIF:

After simulating the LFSR and capturing its states, we store the results in a GIF file for later analysis.

```
# Save the visualized frames as a GIF
frames[0].save('lfsr_simulation.gif',
               save_all=True,
               append_images=frames[1:],
               duration=500,
               loop=0)
```

Here you can view the simulation of the above code: View the GIF animation

We can also simulate a larger LFSR with a higher-degree feedback polynomial. Below is an example using the feedback polynomial:

```
x^32 + x^28 + x^19 + x^18 + x^16 + x^14 + x^11 + x^10 + x^9 + x^6 + x^5 + 1
```

This example maintains the same structure as previous simulations, but the iteration range is adjusted to 35 to accommodate the 32nd-degree polynomial's full period. The resulting animation is available here:

View the GIF animation

3.1.2 Primitive vs. Non-Primitive Polynomials

Now that we've defined primitive polynomials, let's explore how they differ from non-primitive ones. Primitive polynomials are essential in generating maximal-length sequences in LFSRs, whereas non-primitive polynomials result in shorter, less optimal sequences. To illustrate this difference, we can use the function `get_fpolyList`(n) to obtain a list of primitive polynomials of degree n and test their primitiveness using our LFSR simulation code.

Obtaining Primitive Polynomials
First, we retrieve the list of primitive polynomials for a given degree. For example, let's obtain the primitive polynomials of degree 4:

```
L = LFSR()
# list of 4-bit feedback polynomials
L.get_fpolyList(4)
```

```
[[4, 1]]
```

This output indicates that [4,1] is a primitive polynomial of degree 4.

Checking Primitiveness and Non-Primitiveness of a Polynomial The numbers of different states are not identical when using a primitive polynomial versus a non-primitive polynomial. When we use non-primitive polynomial, the shorter cycle length and fewer states reduce the unpredictability of the output sequence, which weakens the security of applications like stream ciphers or pseudorandom number generators.

In contrast, a primitive polynomial maximizes the unpredictability by generating the largest possible number of unique states, making the system more resistant to attacks.

Explanation

The code checks whether a given polynomial is primitive by running the LFSR through all possible states and counting the number of unique states before repeating. A polynomial is considered primitive if it generates $2^n - 1$ unique states, where n is the number of registers. The function $lfsr\_run$ outputs the sequence of bits and the number of unique states generated.

Relevant Code

```python
def lfsr_next(state, fpoly):
    feedback = 0
    for pos in fpoly:
        feedback ^= state[-pos]

    new_state = np.zeros_like(
  state)
    new_state[0] = feedback
    new_state[1:] = state[:-1]
    return new_state

def lfsr_run(initstate, fpoly):
    state = np.array(initstate)
    states_seen = set()
    count = 0
    while tuple(state) not in
  states_seen:
        states_seen.add(tuple(
  state))
        state = lfsr_next(state,
  fpoly)
        count += 1
    is_primitive = len(states_seen
  ) == (2 ** len(initstate) - 1)
```

The rest of the code remains unchanged; however, we must initialize the polynomial we are checking. Below is how we checked for degree 4 polynomials.

Primitive Polynomial Example Initialization

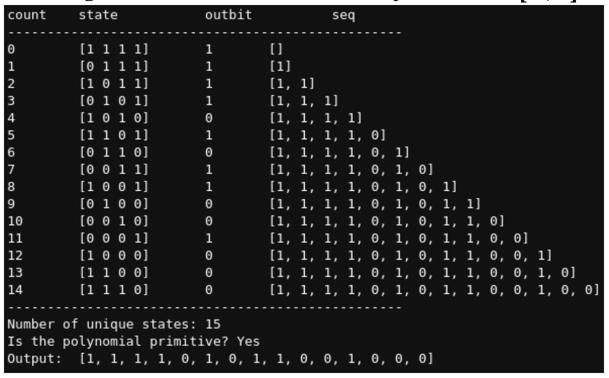
```python
initstate = [1, 1, 1, 1]  #
   Initial state
fpoly = [4, 1]              #
   Primitive polynomial
```

Non-Primitive Polynomial Example Initialization

```python
initstate = [1, 1, 1, 1]  #
   Initial state
fpoly = [4, 2]              # Non-
   primitive polynomial
```

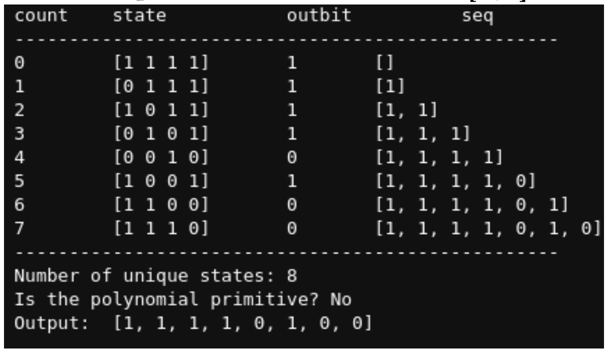
Following the initialization, we run the LFSR for both polynomials and capture the output, highlighting the differences in unique states and sequences generated. The table below presents the output from both the primitive polynomial $[4, 1]$ and the non-primitive polynomial $[4, 2]$ side by side for comparison:

Output for Primitive Polynomial [4,1]



```
count    state          outbit         seq
-----------------------------------------------
0        [1 1 1 1]      1       []
1        [0 1 1 1]      1       [1]
2        [1 0 1 1]      1       [1, 1]
3        [0 1 0 1]      1       [1, 1, 1]
4        [1 0 1 0]      0       [1, 1, 1, 1]
5        [1 1 0 1]      1       [1, 1, 1, 1, 0]
6        [0 1 1 0]      0       [1, 1, 1, 1, 0, 1]
7        [0 0 1 1]      1       [1, 1, 1, 1, 0, 1, 0]
8        [1 0 0 1]      1       [1, 1, 1, 1, 0, 1, 0, 1]
9        [0 1 0 0]      0       [1, 1, 1, 1, 0, 1, 0, 1, 1]
10       [0 0 1 0]      0       [1, 1, 1, 1, 0, 1, 0, 1, 1, 0]
11       [0 0 0 1]      1       [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0]
12       [1 0 0 0]      0       [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1]
13       [1 1 0 0]      0       [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0]
14       [1 1 1 0]      0       [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0]
-----------------------------------------------
Number of unique states: 15
Is the polynomial primitive? Yes
Output:  [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0]
```

Output for Non-Primitive [4,2]



```
count    state          outbit         seq
-----------------------------------------------
0        [1 1 1 1]      1       []
1        [0 1 1 1]      1       [1]
2        [1 0 1 1]      1       [1, 1]
3        [0 1 0 1]      1       [1, 1, 1]
4        [0 0 1 0]      0       [1, 1, 1, 1]
5        [1 0 0 1]      1       [1, 1, 1, 1, 0]
6        [1 1 0 0]      0       [1, 1, 1, 1, 0, 1]
7        [1 1 1 0]      0       [1, 1, 1, 1, 0, 1, 0]
-----------------------------------------------
Number of unique states: 8
Is the polynomial primitive? No
Output:  [1, 1, 1, 1, 0, 1, 0, 0]
```

3.2 Properties Comparison

3.2.1 Periodicity

- **Primitive Polynomial**: The expected period is $2^M - 1$, where M is the number of registers in the LFSR. A primitive polynomial generates the maximum number of unique states.
  - Example: For the polynomial $[5, 2]$, the period is 31 (Pass).
- **Non-Primitive Polynomial**: The period will be shorter than $2^M - 1$, leading to a less effective output sequence.
  - Example: For the polynomial $[5, 1]$, the period is shorter than 31 (Fail).

3.2.2 Balance Property

- **Primitive Polynomial**: The number of 1s equals the number of 0s plus one. This balance ensures uniform randomness.
  - Example: For $[5, 2]$, there are 16 ones and 15 zeros (Pass).
- **Non-Primitive Polynomial**: This property may not hold, leading to an uneven distribution of 1s and 0s.
  - Example: For $[5, 1]$, there are 17 ones and 14 zeros (Fail).

3.2.3 Runlength Property

- **Primitive Polynomial**: The run-lengths (consecutive sequences of 1s or 0s) follow a predictable pattern. This ensures uniform randomness.
  - Example: The run-lengths for $[5, 2]$ are $[4, 2, 1, 1]$ (Pass).
- **Non-Primitive Polynomial**: The run-lengths are irregular, reducing randomness.
  - Example: The run-lengths for $[5, 1]$ are $[10, 2, 1, 1, 2]$ (Fail).

3.2.4 Autocorrelation Property

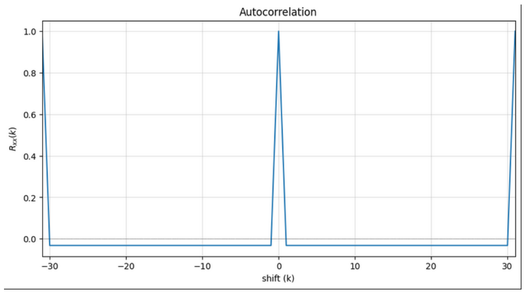
- **Primitive Polynomial**: The autocorrelation is noise-like, with 1 at $k = 0$ and close to $-\frac{1}{M}$ everywhere else, ensuring hard-to-predict sequences.
  - Example: For $[5, 2]$, the autocorrelation function is approximately $[1, -0.032, -0.032, \dots]$ (Pass).
- **Non-Primitive Polynomial**: The autocorrelation is irregular, making the sequence more predictable.
  - Example: For $[5, 1]$, the autocorrelation has irregular peaks (Fail).

3.2.5 Side-by-Side Comparison of Test Results

Primitive Polynomial: [5, 2]

Test Results

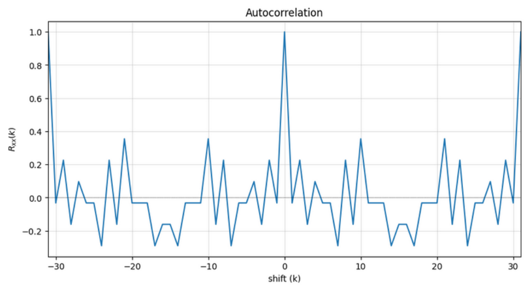
- **Periodicity:** 31 (Pass)
- **Balance:** 16 ones, 15 zeros (Pass)
- **Runlength:** [4, 2, 1, 1] (Pass)
- **Autocorrelation:** Noise-like (Pass)



Non-Primitive Polynomial: [5, 1]

Test Results

- **Periodicity:** Less than 31 (Fail)
- **Balance:** 17 ones, 14 zeros (Fail)
- **Runlength:** [10, 2, 1, 1, 2] (Fail)
- **Autocorrelation:** Predictable (Fail)



4 Toy Stream Cipher Design

This part explains how to design a simple stream cipher by combining two Linear Feedback Shift Registers (LFSRs) using primitive polynomials. The output of the two LFSRs is combined using an XOR operation, which is then used to encrypt and decrypt a message. Also, how to encrypt the content of the files and to provide the ciphertext that is stored in a new file. Where the receiver side can read the ciphertext and decrypt the content of the file to get the plaintext.

4.1 Two LFSRs with XOR

This process is demonstrated with Python code, and the roles of both Alice (the sender) and Bob (the receiver) are illustrated.

4.1.1 Keystream Generation

We generate the keystream by combining the output of two LFSRs using XOR. The function `build_keystream` takes as input the primitive polynomials and initial states of the LFSRs, and outputs a keystream.

```python
def build_keystream(lfsr1_poly, lfsr1_initstate, lfsr2_poly,
    lfsr2_initstate, keystream_length):
    """Generates a keystream using two LFSRs and XOR."""
    L1 = LFSR(fpoly=lfsr1_poly, initstate=lfsr1_initstate)
    L2 = LFSR(fpoly=lfsr2_poly, initstate=lfsr2_initstate)
    keystream = []

    # Generate bits using both LFSRs and combine them using XOR
    for _ in range(keystream_length):
        bit1 = L1.next()
        bit2 = L2.next()
        keystream.append(bit1 ^ bit2)  # XOR operation

    return np.array(keystream)
```

The function initializes two LFSRs and in each loop, generates one bit from each LFSR. These bits are combined using XOR, and the resulting keystream is returned as a NumPy array.

4.1.2 Encryption and Decryption

The `encrypt_decrypt` function is responsible for both encrypting and decrypting a message by XORing it with the keystream.

```python
def encrypt_decrypt(message, keystream):
    """Encrypts/decrypts a message using a keystream."""
    encrypted_message = message ^ keystream  # XOR message with keystream
    decrypted_message = encrypted_message ^ keystream  # XOR again to
    decrypt

    return encrypted_message, decrypted_message
```

Encryption and decryption are both performed by XORing the message with the keystream. Since XOR is symmetric, applying the operation twice with the same keystream results in the original message.

4.1.3 Alice's Side (Encryption)

On Alice's side, the message is converted into a bitstream, and the keystream is generated. The message is then encrypted using the keystream.

```python
message = "Bonjour, Bob!".encode("utf-8")  # Convert string to bytes
message_bits = np.unpackbits(np.frombuffer(message, dtype=np.uint8))

# Generate keystream using two LFSRs
keystream = build_keystream(lfsr1_poly, lfsr1_initstate, lfsr2_poly,
    lfsr2_initstate, len(message_bits))

# Encrypt the message
encrypted_message, _ = encrypt_decrypt(message_bits, keystream)
```

The message is converted into binary format, and the keystream is generated. The message is then encrypted by XORing it with the keystream.

4.1.4 Bob's Side (Decryption)

On Bob's side, the encrypted message is decrypted using the same keystream, and the original message is recovered.

```python
# Decrypt the message
decrypted_message, _ = encrypt_decrypt(encrypted_message, keystream)
decrypted_bytes = np.packbits(decrypted_message).tobytes()
decrypted_string = decrypted_bytes.decode("utf-8", errors="ignore")  #
    Handle potential padding
```

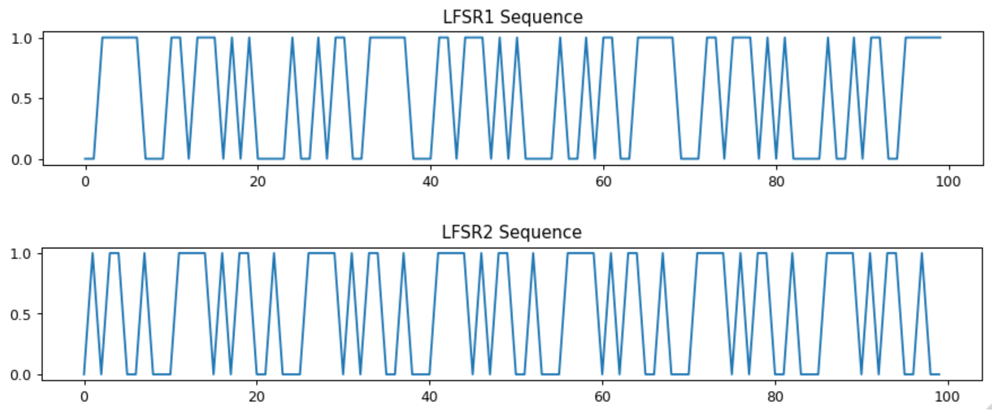
Decryption is performed by XORing the encrypted message with the same keystream. After that, the decrypted bits are converted back into readable string format.

4.1.5 Visualizing LFSR Output

We can visualize the output of the two LFSRs to see how they evolve over time. Below is the code for plotting the LFSR sequences for both registers.

```python
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
plt.title("LFSR1 Sequence")
plt.plot(LFSR(fpoly=lfsr1_poly, initstate=lfsr1_initstate).runKCycle(100))
     # Plot LFSR1 sequence

plt.subplot(2, 1, 2)
plt.title("LFSR2 Sequence")
plt.plot(LFSR(fpoly=lfsr2_poly, initstate=lfsr2_initstate).runKCycle(100))
     # Plot LFSR2 sequence
plt.tight_layout()
plt.show()
```



This code generates a plot showing the evolution of the LFSR sequences for 100 cycles, providing insights into their behavior.

4.1.6 Example Output

```
Original Message: Bonjour, Bob!
Encrypted Message (bits): [0 0 1 0 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 1 0 1
1 1 0 0 0 0 1 0 0 0 1 1 0 1 1 1 1 1 1 0 1 0
1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 1 0 1 1 1 ]
Decrypted Message: Bonjour, Bob!
```

The original message is successfully encrypted and decrypted, demonstrating the functionality of the stream cipher using two LFSRs.

4.2 File Encryption and Decryption using LFSR-based Stream Cipher

In this section, we extend the use of the stream cipher designed with two LFSRs. The task is to encrypt and decrypt files. On the sender's side, the content of a file is read and encrypted using a keystream. The ciphertext is then stored in a new file. On the receiver's side, the ciphertext is read from the new file, decrypted, and the plaintext is retrieved. The steps below show how we can achieve this using Python.

4.2.1 Keystream Generation

Just like before, the keystream is generated by combining the output of two LFSRs with XOR.

```python
def build_keystream(lfsr1_poly, lfsr1_initstate, lfsr2_poly,
  lfsr2_initstate, keystream_length):
    """Generates a keystream of specified length."""
    L1 = LFSR(fpoly=lfsr1_poly, initstate=lfsr1_initstate)
    L2 = LFSR(fpoly=lfsr2_poly, initstate=lfsr2_initstate)
    keystream = []

    for _ in range(keystream_length):
        keystream.append(L1.next() ^ L2.next())  # XOR operation

    return np.array(keystream)
```

This function takes in the primitive polynomials and initial states of both LFSRs and generates a keystream of the desired length.

4.2.2 File Encryption

The `encrypt_file` function reads the contents of the file, converts it into bits, XORs it with the keystream, and writes the encrypted bits into a new file.

```python
def encrypt_file(input_filename, output_filename, keystream):
    """Encrypts a file using a keystream."""
    with open(input_filename, "rb") as infile, open(output_filename, "wb")
     as outfile:
        file_content = infile.read()
        file_bits = np.unpackbits(np.frombuffer(file_content, dtype=np.
  uint8))

        # Encrypting only up to file length
        encrypted_bits = file_bits ^ keystream[:len(file_bits)]
        encrypted_bytes = np.packbits(encrypted_bits).tobytes()
        outfile.write(encrypted_bytes)
```

This function reads the file in binary mode, converts the content into bits, encrypts the bits using the keystream, and stores the encrypted data in a new file.

4.2.3 File Decryption

The `decrypt_file` function works in reverse. It reads the encrypted file, decrypts it using the keystream, and writes the decrypted content back to a new file.

```python
def decrypt_file(input_filename, output_filename, keystream):
    """Decrypts a file using a keystream."""
    with open(input_filename, "rb") as infile, open(output_filename, "wb")
     as outfile:
        encrypted_content = infile.read()
        encrypted_bits = np.unpackbits(np.frombuffer(encrypted_content,
  dtype=np.uint8))

```

```
        # Decrypting only up to file length
        decrypted_bits = encrypted_bits ^ keystream[:len(encrypted_bits)]
        decrypted_bytes = np.packbits(decrypted_bits).tobytes()
        outfile.write(decrypted_bytes)
```

Similar to encryption, this function reads the encrypted content in binary, XORs it with the keystream, and writes the decrypted data into a new file.

4.2.4 Main Program

The main part of the program, where both encryption and decryption are performed, is shown below. It defines the filenames and LFSR parameters, generates the keystream, and performs file encryption and decryption.

```
# LFSR parameters
lfsr1_poly = [5, 3]
lfsr1_initstate = [1, 1, 1, 0, 0]
lfsr2_poly = [4, 1]
lfsr2_initstate = [1, 0, 1, 0]

input_filename = "plaintext.txt"
output_filename_encrypted = "ciphertext.bin"
output_filename_decrypted = "decrypted.txt"

# Alice (Encryption)
with open(input_filename, "rb") as f:
    file_size = len(f.read())  # Get file size in bytes
keystream_length = file_size * 8  # Keystream length in bits
keystream = build_keystream(lfsr1_poly, lfsr1_initstate, lfsr2_poly,
   lfsr2_initstate, keystream_length)

# Encrypt the file
encrypt_file(input_filename, output_filename_encrypted, keystream)

# Bob (Decryption)
decrypt_file(output_filename_encrypted, output_filename_decrypted,
   keystream)

# Verification: Checking if the original and decrypted files are identical
with open(input_filename, "rb") as original_file, open(
   output_filename_decrypted, "rb") as decrypted_file:
    if original_file.read() == decrypted_file.read():
        print("Decryption successful: Original and decrypted files match."
   )
    else:
        print("Decryption failed: Original and decrypted files do not
   match.")
```

This part of the code reads the input file `plaintext.txt`, encrypts it to `ciphertext.bin`, and then decrypts it to `decrypted.txt`. Finally, the program checks whether the original and decrypted files match.

4.2.5 Conclusion

This program successfully implements file encryption and decryption using a stream cipher based on two LFSRs. The full code and files can be accessed using the following links:

- `plaintext.txt`: `plaintext.txt`
- `ciphertext.bin`: `ciphertext.bin`
- `decrypted.txt`: `decrypted.txt`

References

References

[1] GeeksforGeeks. Stream ciphers. https://www.geeksforgeeks.org/stream-ciphers/, 2024. Accessed: 2024-09-28.

[2] Christof Paar. Introduction to cryptography. https://www.youtube.com/@introductiontocryptography4223, 2024. Accessed: 2024-09-28.

[3] Y. Pelzl and C. Paar. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2009.