Stream cipher based on Linear Feedback Shift Register

Applied cryptography

25 september 2024

Hussaini Mohmmad Qasim, Volonterio Luca

Contents

1 Objective

The basic objective of this lab is to understand and analyze the behavior of Linear Feedback Shift Registers (LFSRs) and their application in cryptographic systems. This lab focuses on examining the properties of LFSRs, particularly the difference between primitive and non-primitive polynomials, and their influence on the number of generated states and security in general. Furthermore, the lab explores the design and implementation of a simple stream cipher using LFSRs, providing insights into their practical use in secure communication.

2 LSFR description and Properties

A Linear Feedback Shift Register (LFSR) is a shift register used to generate a sequence of bits, usuallu used as a key stream for cryptographic applications. It operates by shifting its bits to the right at each step, and the new bit entering the leftmost position is calculated with a XOR operation of a few selected bits from the register, that are determined by the characteristic polynomial of the system.

The first tasks in this lab require to describe a particular LSFR configuration and try to compute its output manually. The first given LSFR 1 is made up of three registers and XOR gates, represented as the state array containing the initial state of the registers (IV), and the fpoly array specifies which terms of the polynomial goes to the feedback. At each clock cycle, the contents of the register shift one position to the right. The output bit from the last position is sent to the output stream and a new input bit is generated based on the XOR applied to certain bits of the register. The process continues for each subsequent clock cycle, generating a stream of output bits. After manually calculating the keystream for the given LFSR, we use a python library to compute them and compare them.

2.1 Python usage

In the following code, you can see a sample LFSR item being instantiated and computed using the $L.seq()$ method, that will print the keystream for the system.

```python
state = [1,1,1] # Initialization Vector
fpoly = [3,1] # feedback degree array
L = LFSR(initstate=state, fpoly=fpoly)
print(L.runFullCycle())
```

Comparing hand-calculated results with the function output we can say their results match.
We performed this task with two LSFR: $P(x) = x^3 + x^2 + 1$ and $P'(x) = x^5 + x^2 + 1$. The initialization vector is always an array of 1s. For the first polynomial, a sample output of a full cycle command s is provided.

```python
L.runFullCycle()
```

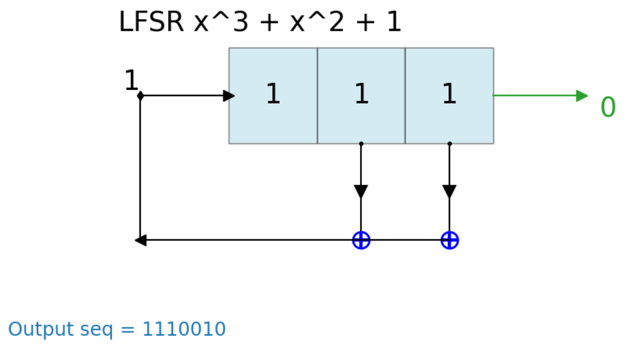


Figure 1: First LFSR

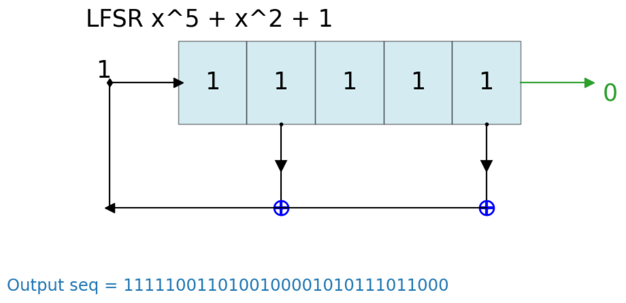


Figure 2: Second LFSR

returns

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

that is the keystream.

2.2 Primitive Polynomial

A polynomial is considered primitive if the sequence generated by the corresponding LFSR has a maximum possible length, known as the full period, which is is $2^n - 1$ where n is the degree of the polynomial i.e. the number of registers.

2.2.1 Verifying Primality of Polynomial

In this part of the lab, the goal is to Verify that the polynomial $x^3 + x^1 + 1$ is primitive, and analyze the number of iterations in the sequence. Also, we have to simulate the LFSR operation and observe its output. The degree of the polynomial is three and expected maximum length sequence is 7 unique iterations. The simulation is run for 21 iterations to observe the state transitions and the output sequence. Since the polynomial is expected to be primitive, we should see a repeating cycle of 7 distinct states.
Here is the simulation code and its output:

```
state = [1,1,1]
fpoly = [3,1]
```

```python
L = LFSR(initstate=state, fpoly=fpoly)

print('count \t state \t\toutbit \t seq')
print('-'*50)
for _ in range(21):
    print(L.count, L.state, '', L.outbit, L.seq, sep='\t')
    L.next()
print('-'*50)
print('Output: ', L.seq)
```

```
count    state    outbit   seq
--------------------------------------------------
0 [1 1 1]   -1  [-1]
1 [0 1 1]   1 [1]
2 [1 0 1]   1 [1 1]
3 [0 1 0]   1 [1 1 1]
4 [0 0 1]   0 [1 1 1 0]
5 [1 0 0]   1 [1 1 1 0 1]
6 [1 1 0]   0 [1 1 1 0 1 0]
7 [1 1 1]   0 [1 1 1 0 1 0 0]
8 [0 1 1]   1 [1 1 1 0 1 0 0 1]
9 [1 0 1]   1 [1 1 1 0 1 0 0 1 1]
...
20  [1 1 0]   0 [1 1 1 0 1 0 0 1 1 1 0 1 0 0 1 1 1 0 1 0]
--------------------------------------------------
Output:  [1 1 1 0 1 0 0 1 1 1 0 1 0 0 1 1 1 0 1 0 0]
```

The LFSR runs for 21 iterations, starting with the initial state *[1, 1, 1]*. After every 7 iterations, the key repeats, confirming that the LFSR is generating a maximal-length sequence.
Since the degree of the polynomial is 3, the maximal length sequence should be $2^3 - 1 = 7$. This is verified as the states cycle every 7 iterations, proving that the polynomial $x^3 + x^1 + 1$ is primitive.

2.3 Constructing Primitive Polynomials and Non-Primitive Polynomials

In this part of the lab, we are asked to construct both a primitive and a non-primitive polynomial for $m = 4$ and observe their differences in terms of states and security impact.

The used LFSR livrary provides a $get\_fpolyList(n)$ which provides a non-exhaustive list of primitive polynomial for the degree n.

```python
L.get_fpolyList(6)
```

that produces

```
[[6, 1], [6, 5, 2, 1], [6, 5, 3, 2]]
```

which means that for degree 6 there are three primitive polynomials: x^6+x+1, $x^6+x^5+x^2+x+1$ and $x^6+x^5+x^3+x^2+1$

Primitive Polynomial:

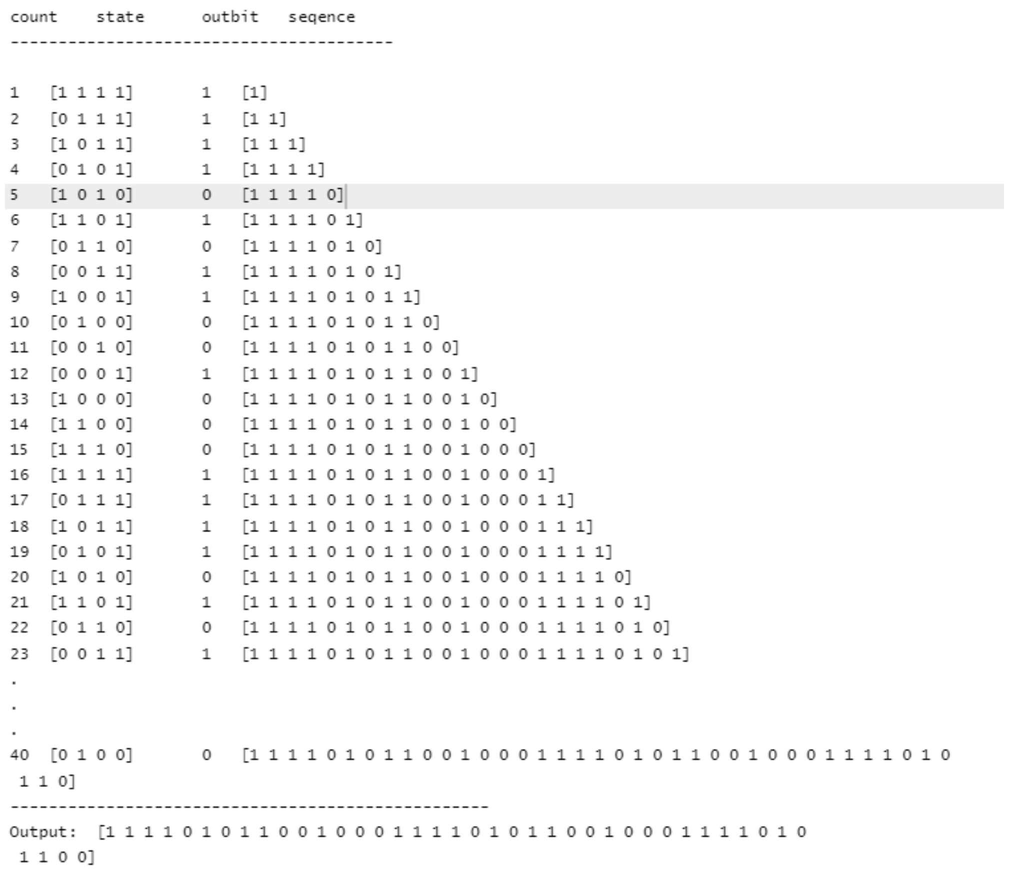
```
Polynomial: f(x)=x^4+x+1
Initial state: [1, 1, 1, 1]
Feedback: [4, 1]
Sequence generated:
[1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0] (15-bit sequence)
```

Non-Primitive Polynomial:

```
Polynomial: f(x)=x^4+x^3+x^2+x+1
Initial state: [1, 1, 1, 1]
Feedback: [4, 3, 2, 1]
Sequence generated:
[1, 1, 1, 1, 0] (5-bit sequence)
```

Number of States:

The test performed using the LFSR with the primitive polynomial $P(x) = x^4+x+1$ generated the following states and outputs for 40 cycles:



```
count    state      outbit   seqence
----------------------------------------

1   [1 1 1 1]      1   [1]
2   [0 1 1 1]      1   [1 1]
3   [1 0 1 1]      1   [1 1 1]
4   [0 1 0 1]      1   [1 1 1 1]
5   [1 0 1 0]      0   [1 1 1 1 0]
6   [1 1 0 1]      1   [1 1 1 1 0 1]
7   [0 1 1 0]      0   [1 1 1 1 0 1 0]
8   [0 0 1 1]      1   [1 1 1 1 0 1 0 1]
9   [1 0 0 1]      1   [1 1 1 1 0 1 0 1 1]
10  [0 1 0 0]      0   [1 1 1 1 0 1 0 1 1 0]
11  [0 0 1 0]      0   [1 1 1 1 0 1 0 1 1 0 0]
12  [0 0 0 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1]
13  [1 0 0 0]      0   [1 1 1 1 0 1 0 1 1 0 0 1 0]
14  [1 1 0 0]      0   [1 1 1 1 0 1 0 1 1 0 0 1 0 0]
15  [1 1 1 0]      0   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0]
16  [1 1 1 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1]
17  [0 1 1 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1]
18  [1 0 1 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1]
19  [0 1 0 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1]
20  [1 0 1 0]      0   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0]
21  [1 1 0 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1]
22  [0 1 1 0]      0   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0]
23  [0 0 1 1]      1   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 1]
.
.
.
40  [0 1 0 0]      0   [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0
 1 1 0]
------------------------------------------------
Output:  [1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0
 1 1 0 0]
```

The sequence generated from this primitive polynomial has a periodic length of 15 bits, as expected, before it starts repeating. For a non-primitive polynomial, the number of different states would be less than 15, thus limiting the sequence to a shorter periodic cycle.

3 Properties of LFSR Test

In this part of the lab, we are going to describe all the properties provided by the $L.test\_properties()$ function and compare it between a primitive and non primitive polynomial.

3.1 Evaluating primitive polynomial

This experiment evaluated an LFSR using a primitive polynomial $P(x) = x^5 + x^2 + 1$ represented as *[5, 2]* and initial state *[1, 1, 1, 1, 0]*. The LFSR passed key tests:

- *Periodicity*: Achieved the expected period of 31, confirming maximal length.
- *Balance Property*: Balanced output with 16 ones and 15 zeros.
- *Run-Length Property*: Proper run-length distribution, indicating randomness.
- *Autocorrelation*: Noise-like autocorrelation, validating low similarity with shifted versions.

All tests confirmed the LFSR's primitive polynomial, which ensures high-quality pseudo-random sequence generation. The periodicity test determines if the sequence has a maximal period, specifically ((2 by the power of n)-1) represents the degree of the polynomial. The balance test ensures that the sequence contains approximately equal numbers of 0s and 1s. The run-length array counts the number of runs (i.e., consecutive sequences of identical bits) for each length. For example, [8,4,2,1,1] means that there are 8 runs of length 1, 4 runs of length 2, etc.
Finally, the auto-correlation test compares portions of the sequence with shifted versions of itself. It measures how similar the sequence is to a shifted version of itself, helping to evaluate its randomness and structure.

3.2 Evaluating non-primitive polynomial

This experiment evaluated an LFSR using a non-primitive polynomial $P(x) = x^5 + x^3 + x^2 + 1$ represented as *[5, 3, 2]* and initial state *[1, 1, 1, 1, 0]*. The LFSR did not pass key tests:

- *Periodicity*: Did not achieve the expected period of 31.
- *Balance Property*: Unbalanced output with 17 ones and 14 zeros.
- *Run-Length Property*: Vulnerable run-length distribution, indicating high predictability.

- *Autocorrelation*: High autocorrelation in certain points, validating high similarity with shifted versions of the signal.

All test confirmed the theshis that primitive polynomial produce better sequence for cryptography applications. Non-primitive polynomials are less suitable for LFSR use because they produce shorter and less balanced sequences. They often have an imbalance in the number of 1s and 0s, too many runs of length 1, and a predictable autocorrelation pattern, which significantly reduces their security compared to primitive polynomials.

4 Building a Toy Stream Cipher Using LFSR

We aim to build a simple stream cipher using two Linear Feedback Shift Registers (LFSRs) with primitive polynomials. The output from each LFSR is combined using the XOR operation to create the keystream for encryption. On the sender's side (Alice), we use this keystream to encrypt a message. On the receiver's side (Bob), the same keystream is used to decrypt the message and verify its correctness.

4.1 Constructing a Stream Cipher Using Two LFSRs

In this part of the lab, we built our own stream cipher combining two LFSR with primitive polynomials. The output of the two LFSRs are combined with a Xor operation.
1- On the sender side (ALICE Side) encrypt a message using the generated keystream.
2- Then, on the receiver side (BOB Side) decrypt the message and check if the original message is retrieved.
Property tests and their importance periodicity: the expected period is (2 by the power of M) 1 = 65335, which confirms that the LFSR cycles through all possible states except the all-zero state.
Code explanation:

```
    string_to_binary(text): #It converts the input message fro string to binary.
    binary_to_string(binary_string): # It converts the input from binary to string.
    xor(a, b): #It performs the XOR operation between a and b.
    xor_string_poly(s, p): #It encrypts and decrypts using the XOR operation.
```

Alice does the following operation on its side:
Alice sets the LFSR state to all 1s and prepares to encrypt the message. Then, Alice converts the plaintext message into its binary form using *string_to_binary*() and then XORs each bit of the binary message with the LFSR keystream using *xor_string_poly*(). The result is the encrypted message.
On the other hand, On the Bob's side: Bob also sets the LFSR to the same initial state as Alice and prepares to decrypt the received encrypted message. After that, Bob uses the same LFSR keystream to XOR the encrypted message and retrieves the original binary message. Using *binary_to_string*(), Bob converts the binary message back into readable text.

```python
# LSFT initialization
degree = 16
fpoly = L.get_fpolyList(degree)[0]
period_length = pow(2,degree) - 1
warm_up = degree * 4 + 1
state = [1] * degree
poly = LFSR(initstate=state,fpoly=fpoly)
# ALICE
poly.set_state([1]*degree)
poly.runKCycle(warm_up)
message = "CIAO COME STAI"
binary_message = string_to_binary(message)
print(binary_message)
encrypted_message = xor_string_poly(binary_message, poly)
print("ALICE: ", message)


#BOB
poly.set_state([1]*degree)
poly.runKCycle(warm_up)
decrypted_message = xor_string_poly(encrypted_message, poly)
text_message = binary_to_string(decrypted_message)
print("BOB: ", encrypted_message)
print("Decrypted: ", text_message)
```

Please note that the LSFR is ”warmed up” for a fixed number of cycles meaning that that part of the keystream is discarded. The original message is converted to binary and successfully encrypted. The encrypted binary message is different from the original binary string. Bob successfully decrypted the message, recovering the original plaintext.
We chose a primitive polynomial of order 16 for our LFSRm in order to have a really long period. We then convert the string to binary and XOR it with the keystream.

4.2 File Encryption and Decryption Using LFSR-Based Stream Cipher

In this part of the lab, we used the previous stream cipher, on the sender side, we wrote a program that reads a file and encrypts the content of the file to provide a ciphertext that is stored in a new file. On the receiver side, we read the ciphertext from the new file and decrypt the content of the file to retrieve the plaintext.
We extended the previous stream cipher implementation to encrypt and decrypt the contents of a file. The goal is to develop two programs: one for the sender (Alice) to encrypt the file and another for the receiver (Bob) to decrypt the ciphertext and retrieve the original plaintext. This method demonstrates how the LFSR-based stream cipher can be applied to file encryption.

```python
def read_file(filename):
    "Read the text in 'filename' and convert it into a simple string."
    res = []
    with open(filename, 'r', encoding='ascii') as f:
        for l in f:
            line = l.strip()  # remove newline characters
            if line:  # avoid empty lines
                res.append(line)
    return ' '.join(res)
```

Function: Alice's Encryption The function alice() is responsible for encrypting the file content.

```python
def alice(read_file_name, write_file_name):
    poly.set_state([1]*degree)  # Initialize LFSR state
    poly.runKCycle(warm_up)  # Warm up LFSR to prevent initial state prediction
    message = read_file(read_file_name)  # Read plaintext file
    binary_message = string_to_binary(message)  # Convert plaintext to binary
    encrypted_message = xor_string_poly(binary_message, poly)  # Encrypt using LFSR
    keystream
    with open(write_file_name, 'w') as fo:  # Write encrypted message to a new file
        fo.write(encrypted_message)
```

Function: Bob's Decryption The function bob() is responsible for reading the encrypted file and decrypting it.

```python
def bob(read_file_name):
    poly.set_state([1]*degree)  # Re-initialize LFSR to the same state
    poly.runKCycle(warm_up) #Warm up the LFSR
    encrypted_message = read_file(read_file_name) #Read ciphertext from the file
    decrypted_message = xor_string_poly(encrypted_message, poly) # Decrypt the message using
     LFSR keystream
    text_message = binary_to_string(decrypted_message) #Convert decrypted binary message to
    plaintext
    return text_message
```

This code demonstrated how to extend the LFSR-based stream cipher to encrypt and decrypt the contents of a file. By using the same initialization and LFSR parameters on both the sender and receiver sides, Alice and Bob were able to securely communicate using a file-based encryption process. The stream cipher offers a simple yet effective method for securing file content, ensuring that the original message is successfully recovered after decryption.

5 Conclusions

In this lab, we explored the behavior and applications of Linear Feedback Shift Registers (LFSRs), focusing on their cryptographic properties and practical use in stream ciphers. By analyzing the properties of both primitive and non-primitive polynomials, we observed how the periodicity, balance, and randomness of LFSR-generated sequences impact their security, with primitive polynomials offering maximum-length sequences and higher security.
Finally, we successfully implemented an LFSR-based stream cipher, demonstrating its effectiveness in encrypting and decrypting messages and files through the XOR operation.

Despite the simplicity and efficiency of LFSRs, their linear nature suggests the need for additional complexity, such as combining multiple LFSRs, to enhance security in cryptographic systems.