Cryptography Project 3

João Pedro Lourenço

December 2020

Exercise 1: Finding the key

This exercise was solved with Python, and the code can be found after the explanation.

The program's logic is as follows:

- For each connection polynomial we have, the program iterates over all possible starting states, and generates the sequence from that initial state. The iteration is done with the `get_possible_sequences` method, and it uses the `lfsr` method to generate a sequence for each initial state.

  Besides the sequence, the program also stores the initial state that lead to that sequence, and the similarity of the sequence with the given encrypted sequence. This similarity is calculated by estimating $p*$ with the *hamming distance*. When plotting the points on a scatter plot, it's very easy to see which one stands out the most (x-axis contains the state in decimal, and y-axis contains the similarity value):

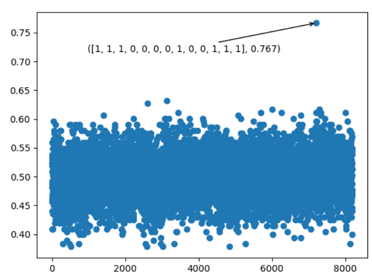

Figure 1: Plotting similarities with the first LFSR

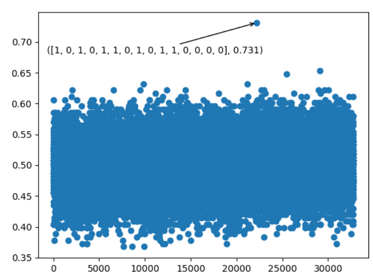

Figure 2: Plotting similarities with the second LFSR

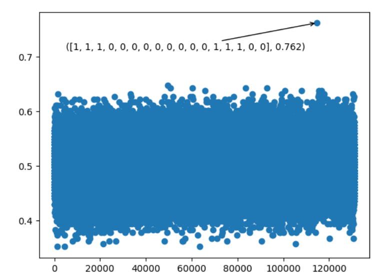

Figure 3: Plotting similarities with the third LFSR

  The initial states that generate the most similar sequence are joined together to make the key:

$$K = (1110000100111, 101011010110000, 11100000000011100)$$

  And these have probabilities 0.767, 0.731, and 0.762, respectively.

- The program also validates that these states generate the provided sequence, by following the rules highlighted in the project description.

The program

```python
from tqdm import tqdm
from dataclasses import dataclass
import matplotlib.pyplot as plt


with open("input_sequence.txt", "r") as seq:
    encrypted_sequence = [int(x) for x in seq.readline()]


@dataclass
class Sequence:
    """Class for storing info about a given sequence"""

    sequence: list[int]
    similarity: float
    initial_state: list[int]


def lfsr(polynomial, initial_state, base, length):

    # linear behaviour - logic borrowed from project 2
    internal_register = initial_state.copy()
    sequence = []
    for i in range(length):
        new_register_value = 0
        for coefficient, content in zip(polynomial, internal_register):
            new_register_value += -coefficient * content

        internal_register.append(new_register_value % base)
        sequence.append(internal_register.pop(0))

    return sequence


def correlation(sequence1, sequence2):
    differing_positions = 0
    for sequence1_digit, sequence2_digit in zip(sequence1, sequence2):
        if sequence1_digit != sequence2_digit:
            differing_positions += 1
    return 1 - differing_positions / len(sequence1)


def get_possible_sequences(polynomial):
    sequences = []
    for i in tqdm(range(2 ** len(polynomial))):
        # Convert to binary with polynomial length digits, and then to int,
        # and save it on a list
        initial_state = [int(x) for x in format(i, f"0{len(polynomial)}b")]

        sequence = lfsr(polynomial, initial_state, 2, len(encrypted_sequence))
        similarity = abs(correlation(sequence, encrypted_sequence))

        sequences.append(Sequence(sequence, similarity, initial_state))

```

```python
    return sequences


def plot_sequences(sequences, max, state, max_index, number):
    x_coordinates = list(range(len(sequences)))
    y_coordinates = [sequence.similarity for sequence in sequences]

    plt.scatter(x_coordinates, y_coordinates)
    plt.annotate(
        f"({state}, {max})",
        xy=(max_index, max),
        xytext=(max_index / 2, max - 0.05),
        ha="center",
        arrowprops=dict(arrowstyle="->", lw=1),
    )
    plt.savefig(f"lfsr_{number}.png")
    plt.close()


def validate(generated_sequences, states):
    generated_keystream = []

    print("Verifying combined LFSR sequence...")
    for output_symbol1, output_symbol2, output_symbol3 in zip(
        generated_sequences[0], generated_sequences[1], generated_sequences[2]
    ):
        outputs = [output_symbol1, output_symbol2, output_symbol3]
        if outputs.count(1) <= 1:
            generated_keystream.append(0)
        else:
            generated_keystream.append(1)

    if generated_keystream == encrypted_sequence:
        print("Generated sequence is equal to given sequence! Keys are: ")
        for s in states:
            print(str(s))
        return True
    else:
        print(
            "Could not generate an equal sequence. Closest was the following sequence and keys:"
        )
        print("    * Sequence: " + str(generated_keystream))
        print("    * Keys: " + str(states))

        return False


def main():

    # Coefficients - left is higher exponent
    connection_polynomials = [
        [1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1],
        [1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0],
        [1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0],
    ]

```

```python
    states = []
    generated_sequences = []
    plot = False

    for index, polynomial in enumerate(connection_polynomials):
        print(f"\nGenerating all sequences for polynomial {polynomial}")
        sequences = get_possible_sequences(polynomial)

        # Find sequence with higher correlation with the encrypted sequence
        most_similar_sequence = max(sequences, key=lambda s: s.similarity)

        if plot:
            plot_sequences(
                sequences,
                round(most_similar_sequence.similarity, 3),
                most_similar_sequence.initial_state,
                sequences.index(most_similar_sequence),
                index,
            )

        # Save the "best" sequence, and the initial state that leads to it
        generated_sequences.append(most_similar_sequence.sequence)
        states.append(most_similar_sequence.initial_state)

    # Verify that this generates the provided keystream
    validate(generated_sequences, states)


main()
```

Exercise 2 - Exhaustive key search

For each LFSR, we have to search $2^{\text{length of LFSR}}$ states, or $2^{13}+2^{15}+2^{17}$ in total, which we assume takes T seconds. We only need to go through the possible states independently because we perform a correlation - if we didn't, we would have to test all possible states combinations, which would result in $2^{13} \times 2^{15} \times 2^{17} = 2^{45}$. Therefore, the time testing these combinations would take is:

$$\frac{2^{45}}{2^{13}+2^{15}+2^{17}} T \text{ seconds} \approx 2.0452T \times 10^{8} \text{ seconds} = 6.485T \text{ years}$$

As an example, my computer (with an Intel i5 7200U) took 86 seconds to iterate over all the states (and generate all possible sequences). This means it would take 557.71 years to perform the exhaustive key search - a very significant difference.