# IDATT2503 - Cryptography Assigment 3

Edvard Berdal Eek

October 2025

## Task 1

I did not find the lecture slides/notes any helpful in understanding LSFR, but after asking a LLM my understanding is that it in this case works as follows:

$$s_i = (z_i, z_{i+1}, z_{i+2}, z_{i+3})$$
$$s_{i+1} = (z_{i+1}, z_{i+2}, z_{i+3}, (z_i + z_{i+1} + z_{i+2} + z_{i+3}) \bmod 2)$$
$$...$$

Shifting to the left and appending the new $z$ value on the right side. And when a task asks for the periods, we look for how many iterations it takes before the sequence repeats.

a) Given the LFSR $z_{i+4} = z_i + z_{i+1} + z_{i+2} + z_{i+3} (\bmod 2)$, $C = (1, 1, 1, 1)$, what are the periods using the keys

1 $K = 1000$

$$s_0 : (1, 0, 0, 0) \to s_1 : (0, 0, 0, (1 + 0 + 0 + 0) \bmod 2) \to s_2 : (0, 0, 1, (0 + 0 + 0 + 1) \bmod 2)$$
$$s_3 : (0, 1, 1, (0{+}0{+}1{+}1) \bmod 2) \to s_4 : (1, 1, 0, (0{+}1{+}1{+}0) \bmod 2) \to s_5 : (1, 0, 0, (1{+}1{+}0{+}0) \bmod 2)$$

$$s_5 : (1, 0, 0, 0) = s_0 : (1, 0, 0, 0)$$

It took 5 iterations for the sequence to repeat therefore there are 5 periods.

2 $K = 0011$

$$s_0 : (0, 0, 1, 1) \to s_1 : (0, 1, 1, 0) \to s_2 : (1, 1, 0, 0)$$
$$s_3 : (1, 0, 0, 0) \to s_4 : (0, 0, 0, 1) \to s_5 : (0, 0, 1, 1)$$

$$s_5 = s_0$$

Again it took 5 iterations, so 5 periods.

3 $K = 1111$

$$s_0 : (1,1,1,1) \rightarrow s_1 : (1,1,1,0) \rightarrow s_2 : (1,1,0,1)$$
$$s_3 : (1,0,1,1) \rightarrow s_4 : (0,1,1,1) \rightarrow s_5 : (1,1,1,1)$$

$$s_5 = s_0$$

Again it took 5 iterations, so 5 periods.

b) What are the periods with the same keys using the following LFSR $z_{i+4} = z_i + z_{i+3} (\mathrm{mod}\, 2)$?

For this task i implemented LSFR in Python and outputted 30 generated $z$ values.

```
def next_bit(self):
        feedback_bit = 0
        for tap in self.taps:
            feedback_bit ^= (self.state >> (tap - 1)) & 1

        self.state = ((self.state << 1) | feedback_bit) & ((1 << self
            .num_bits) - 1)
        return feedback_bit
```

This goes through each tap position and does XOR with previous tap results making the *feedback_bit* 0 if the sum of ones is even and 1 if the sum is odd. Then it shifts the current state to the left and appends the final *feedback_bit* with an OR bitoperator. Finaly, it does an AND bitoperation with a number with the same bit length and only containing ones to prevent the state from growing beyond its original number of bits.

```
-- K = 1000: --
Map: {'1000': 1, '0001': 1, '0011': 1, '0111': 1, '1111': 0, '1110': 1, '1101': 0, '1010': 1, '0101': 1,
'1011': 0, '0110': 0, '1100': 1, '1001': 0, '0010': 0, '0100': 0}
Sequence: [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]
Periods: 15

-- K = 0011: --
Map: {'0011': 1, '0111': 1, '1111': 0, '1110': 1, '1101': 0, '1010': 1, '0101': 1, '1011': 0, '0110': 0,
'1100': 1, '1001': 0, '0010': 0, '0100': 0, '1000': 1, '0001': 1}
Sequence: [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]
Periods: 15

-- K = 1111: --
Map: {'1111': 0, '1110': 1, '1101': 0, '1010': 1, '0101': 1, '1011': 0, '0110': 0, '1100': 1, '1001': 0,
'0010': 0, '0100': 0, '1000': 1, '0001': 1, '0011': 1, '0111': 1}
Sequence: [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]
Periods: 15
```

Figure 1: LSFR results

We can see that map has 15 entries and that we have a repeating sequence after 15 iterations for all keys.

## Task 2

I implemented the HMAC in python with the following functions:

```
def midsquare_hash(x):
    squared = x * x
    squared &= 0xFF
    middle = (squared >> 2) & 0b1111
    return middle
```

*midsquare_hash* squares the message $x$ and does an AND bitoperation with $0\text{x}FF = 256 = 0\text{b}\,11111111$ to extract the remainder $x^2 \bmod 2^8$, as in the last 8 bits. It then shifts this two to the right and and does another bit AND operation with $0\text{b}\,1111$ to extract the last four bits, as in the middle four bits.

```
def hmac(message, key, ipad, opad):
    inner_input = (key ^ ipad) ^ message
    inner_hash = midsquare_hash(inner_input)

    outer_input = (key ^ opad) ^ inner_hash
    outer_hash = midsquare_hash(outer_input)

    return outer_hash
```

This function follows the definition of the HMAC from the lecture slides by applying the *midsquare_hash* and utilizing the ipad and opad.

```
-- a) --
HMAC for 0110: 0000

-- b) --
Calculated HMAC for 0111: 1001
Is message authentic? False
```

Figure 2: HMAC ouput

## Task 3

I implemented CBC-MAC in Python with the following functions:

```
def e_caesar(x):
    return (x + 3) & 0xFF

def cbc_mac(message_blocks):
    C = 0
    for block in message_blocks:
        C = e_caesar(block ^ C)
    return C
```

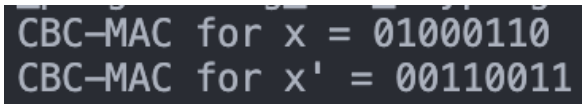


Figure 3: CBC-MAC output

# Task 4

```
def ctr_keystream_bytes(nonce, sbox, count=4):
    out = []
    for ctr in range(count):
        idx = (nonce + ctr) & 0xFF
        out.append(sbox[idx])
    return out
```

a) Using the nonce = 01100101, write down the first 4 bytes produced with counter values 0,1,2,3.


Figure 4: Keystream block cipher CTR-mode

b) What is the period of the key-stream?

As we run mod $2^8$ we can get 256 distinct ouputs, as in the period of the keystream is 256.

c) Can the computation of the keystream be easily paral- lelized?

Because the result of one computation is not dependent on the result of another, the computation of the keystream can easily be parallelized.

d) How could information about known plaintexts-ciphertext pairs be used to infer information about the key used?

In CTR-mode a ciphertext block is $C = P \oplus KS$. If you know a plaintext block $P$ and a ciphertext $C$ you can recover its keystream block $KS$: $KS = P \oplus C$.

Known plaintexts can therefore reveal keystream bytes at the corresponding counters. If the block cipher or keyspace is weak/small, that information can be exploited to recover the key. However with a strong cipher (AES) it only reveals keystream bytes (which is expected), not the key.