UNIWERSYTET KOMISJI EDUKACJI NARODOWEJ
W KRAKOWIE

Instytut Bezpieczeństwa i Informatyki

KRYPTOGRAFIA
(ćwiczenia laboratoryjne)

Ćwiczenie numer:
3

Imię i nazwisko: Grzegorz Golonka
Numer grupy: L2
Studia stacjonarne

Czas realizacji zajęć: 135 min
Temat ćwiczenia: Algorytm A5/1

1. Wstęp

Celem niniejszego ćwiczenia było zapoznanie się z teorią działania szyfratora strumieniowego A5/1, historycznie używanego w standardzie GSM, oraz jego praktyczna implementacja w języku Python. Zadanie obejmowało stworzenie programu zdolnego do szyfrowania i deszyfrowania plików graficznych w formacie BMP przy użyciu algorytmu A5/1 i 64-bitowego klucza. Poprawność implementacji została zweryfikowana przez zaszyfrowanie przykładowego obrazu, a następnie jego odszyfrowanie i porównanie wizualne z oryginałem. Cały proces, wraz z opisem implementacji i wynikami, został udokumentowany w niniejszym sprawozdaniu.

2. Wprowadzenie teoretyczne

A5/1 jest szyfratorem strumieniowym, co oznacza, że generuje sekwencję bitów (lub bajtów) zwaną strumieniem klucza (keystream), która jest następnie łączona (zazwyczaj za pomocą operacji XOR) z danymi wejściowymi (tekstem jawnym) w celu uzyskania szyfrogramu. Ten sam proces, z tym samym strumieniem klucza, jest używany do deszyfrowania (XOR szyfrogramu ze strumieniem klucza odtwarza tekst jawny).

Kluczowe cechy algorytmu A5/1:

- Klucz: 64-bitowy.
- Struktura: Opiera się na trzech rejestrach przesuwających liniowo ze sprzężeniem zwrotnym (LFSR) o różnych długościach:
    - R1: 19 bitów
    - R2: 22 bity
    - R3: 23 bity
- Inicjalizacja: Rejestry są inicjalizowane przy użyciu 64-bitowego klucza. Każdy bit klucza jest bezpośrednio ładowany do odpowiedniej komórki rejestrów.
- Mechanizm taktowania (Clocking): A5/1 wykorzystuje nieregularny mechanizm taktowania oparty na funkcji większościowej (majority function). W każdym kroku sprawdzane są bity taktujące (tzw. "clocking bits") w środkowych pozycjach każdego rejestru (R1[8], R2[10], R3[10]). Obliczana jest wartość większościowa tych trzech bitów. Tylko te rejestry, których bit taktujący jest zgodny z wartością większościową, są przesuwane (taktowane) w danym kroku.
- Sprzężenie zwrotne: Podczas taktowania, nowy bit wchodzący do rejestru (na pozycję 0) jest obliczany jako XOR wybranych bitów (tzw. "taps") z tego samego rejestru. Konkretne pozycje bitów używane do sprzężenia zwrotnego są stałe dla każdego rejestru.
- Generowanie strumienia klucza: Bit strumienia klucza jest generowany w każdym kroku jako XOR ostatnich bitów każdego z trzech rejestrów (R1[18] ^ R2[21] ^ R3[22]).

3. Opis implementacji

- **Inicjalizacja**

Klasa A51 odpowiada za całość logiki algorytmu A5/1, w tym inicjalizację rejestrów, ładowanie klucza, numeru ramki oraz nieregularne taktowanie zgodne z regułą większości.
Rejestry R1, R2 i R3 są zerowane, a następnie przez 64 takty ładowany jest 64-bitowy klucz, a przez kolejne 22 takty – numer ramki. W obu tych etapach wszystkie rejestry są taktowane jednocześnie.
Po załadowaniu danych następuje 100 ślepych taktów, zgodnych z regułą większości, których wynik nie jest wykorzystywany do generowania klucza.

```python
class A51:
    def __init__(self, key: int, frame_number: int = 0):
        self.key = key
        self.frame_number = frame_number
        self.R1 = np.zeros(19, dtype=np.uint8)
        self.R2 = np.zeros(22, dtype=np.uint8)
        self.R3 = np.zeros(23, dtype=np.uint8)
        self._initialize_registers()

    def _initialize_registers(self):
        self.R1[:] = 0
        self.R2[:] = 0
        self.R3[:] = 0

        for i in range(64):
            bit = (self.key >> i) & 1
            self._clock_all(bit)

        for i in range(22):
            bit = (self.frame_number >> i) & 1
            self._clock_all(bit)

        for _ in range(100):
            self._clock_majority()
```

- **Mechanizm taktowania**

Metoda `_clock_majority` implementuje jeden krok zgodny z regułą większości. Na podstawie bitów taktujących z każdego rejestru (R1[8], R2[10], R3[10]) obliczana jest wartość większościowa. Tylko te rejestry, których bit taktujący się z nią zgadza, są przesuwane i aktualizowane.

```
    def _clock_all(self, input_bit):
        self.R1 = np.roll(self.R1, 1)
        self.R1[0] = self.R1[13] ^ self.R1[16] ^ self.R1[17] ^ self.R1[18] ^ input_bit

        self.R2 = np.roll(self.R2, 1)
        self.R2[0] = self.R2[20] ^ self.R2[21] ^ input_bit

        self.R3 = np.roll(self.R3, 1)
        self.R3[0] = self.R3[7] ^ self.R3[20] ^ self.R3[21] ^ self.R3[22] ^ input_bit

    def _clock_majority(self):
        m = majority(self.R1[8], self.R2[10], self.R3[10])
        if self.R1[8] == m:
            fb = self.R1[13] ^ self.R1[16] ^ self.R1[17] ^ self.R1[18]
            self.R1 = np.roll(self.R1, 1)
            self.R1[0] = fb
        if self.R2[10] == m:
            fb = self.R2[20] ^ self.R2[21]
            self.R2 = np.roll(self.R2, 1)
            self.R2[0] = fb
        if self.R3[10] == m:
            fb = self.R3[7] ^ self.R3[20] ^ self.R3[21] ^ self.R3[22]
            self.R3 = np.roll(self.R3, 1)
            self.R3[0] = fb
```

- Generowanie strumienia klucza

Metoda `get_keystream` generuje strumień klucza o zadanej długości w bajtach. Każdy bit generowany jest jako XOR z wyjściowych bitów rejestrów (R1[18], R2[21], R3[22]). Dla uproszczenia, bit `0` jest mapowany na bajt `0x00`, a bit `1` na `0xFF`, co odpowiada wartościom typowym dla obrazów binarnych.

```
def get_keystream(self, length: int) -> np.ndarray:
    keystream = np.zeros(length, dtype=np.uint8)
    for i in range(length):
        keystream[i] = (self.R1[18] ^ self.R2[21] ^ self.R3[22]) * 255
        self._clock_majority()
    return keystream
```

- Obsługa obrazów BMP

Funkcja `process_image` odpowiada za przetwarzanie plików graficznych BMP. Obraz jest wczytywany i konwertowany do RGB, następnie przekształcany do tablicy `numpy`. Na podstawie

przekazanego klucza i numeru ramki tworzony jest obiekt `A51`, który generuje strumień klucza i wykonuje XOR z danymi obrazu. Wynik zapisywany jest jako nowy plik BMP.

```python
def process_image(input_path, output_path, key, frame_number=0):
    img = Image.open(input_path).convert("RGB")
    img_data = np.array(img)

    cipher = A51(key, frame_number)
    result_data = cipher.encrypt_decrypt(img_data)

    Image.fromarray(result_data).save(output_path)

# Przykład użycia
process_image('input2.bmp', 'encrypted.bmp', 0x123456789ABCDEF)
process_image('encrypted.bmp', 'decrypted.bmp', 0x123456789ABCDEF)
```

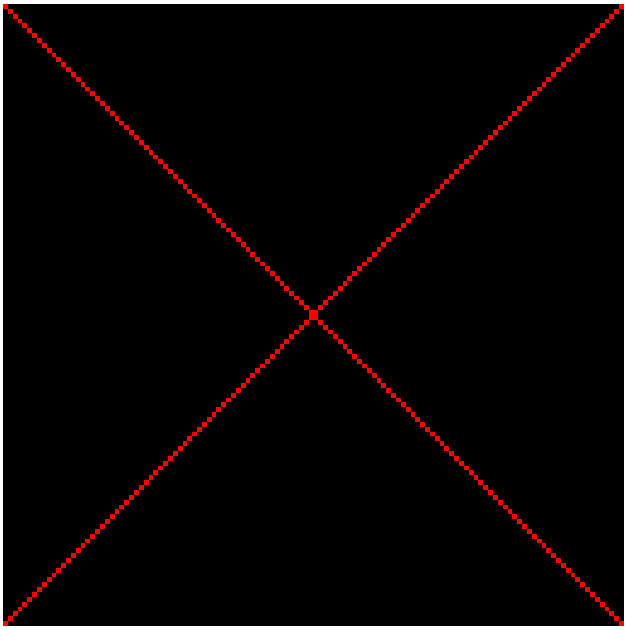
4. Testowanie i wyniki

Do przetestowania implementacji wykorzystano fragment kodu znajdujący się na końcu skryptu. Proces testowy przebiegał następująco:

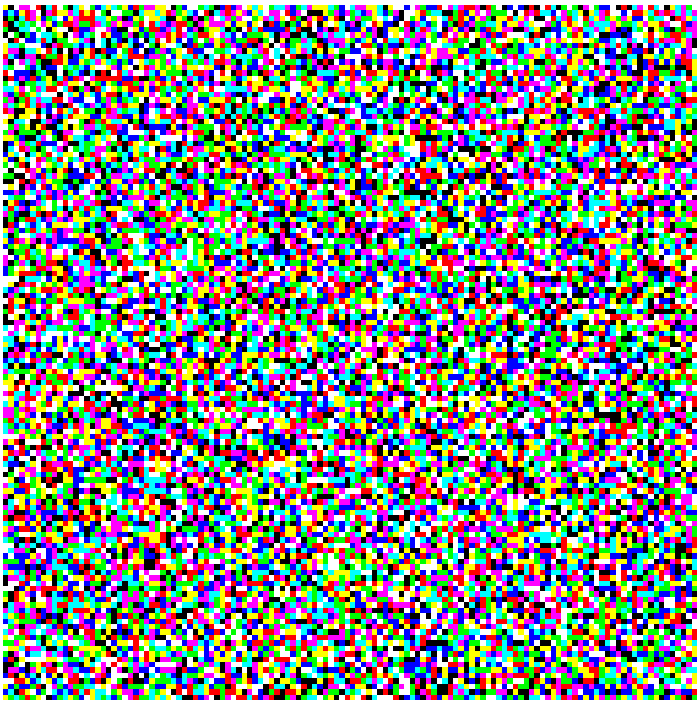
1. Jako dane wejściowe wybrano plik graficzny input3.bmp.
2. Zdefiniowano 64-bitowy klucz: 0x123456789ABCDEF.
3. Wywołano funkcję process_bmp w celu zaszyfrowania pliku input3.bmp i zapisania wyniku jako encrypted.bmp.
4. Ponownie wywołano funkcję process_bmp, tym razem używając encrypted.bmp jako wejścia i tego samego klucza, aby odszyfrować obraz i zapisać wynik jako decrypted.bmp.
5. Porównano wizualnie plik oryginalny (input3.bmp) z plikiem odszyfrowanym (decrypted.bmp).

Wyniki uzyskane po uruchomieniu programu:

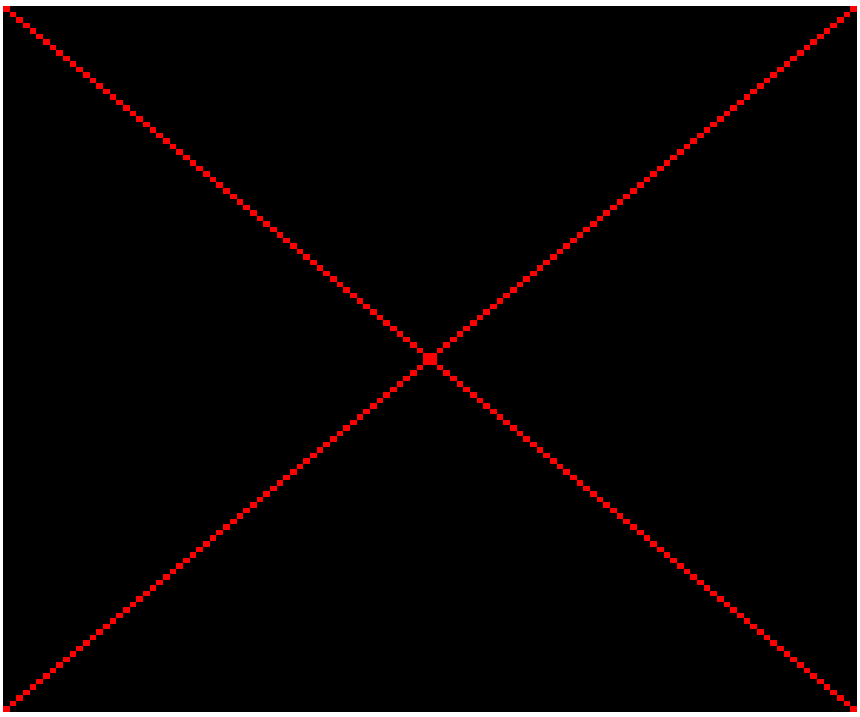
- **Dane wejściowe: input2.bmp:**



- **Obraz zaszyfrowany (encrypted.bmp)**



- **Obraz odszyfrowany (decrypted.bmp):**



5. Omówienie wyników

Przeprowadzone testy, obejmujące zaszyfrowanie obrazu input3.bmp do encrypted.bmp oraz jego późniejsze odszyfrowanie do decrypted.bmp przy użyciu tego samego klucza, potwierdziły podstawową poprawność implementacji. Odszyfrowany obraz (decrypted.bmp) jest wizualnie identyczny z oryginałem, co dowodzi, że proces szyfrowania za pomocą zaimplementowanego algorytmu A5/1 jest odwracalny przy znajomości klucza, a operacja XOR ze strumieniem klucza działa zgodnie z oczekiwaniami.

6. Wnioski

W ramach ćwiczenia pomyślnie zaimplementowano szyfrator strumieniowy A5/1 w języku Python, umożliwiający szyfrowanie i deszyfrowanie plików graficznych w formacie BMP. Implementacja poprawnie odwzorowuje kluczowe mechanizmy A5/1: inicjalizację trzech rejestrów LFSR za pomocą klucza, nieregularne taktowanie sterowane funkcją większościową oraz generowanie strumienia klucza poprzez XORowanie bitów wyjściowych rejestrów. Wykorzystanie operacji XOR do łączenia strumienia klucza z danymi obrazu pozwoliło na zastosowanie tej samej funkcji zarówno do szyfrowania, jak i deszyfrowania. Przeprowadzone testy potwierdziły, że implementacja działa poprawnie, odtwarzając oryginalny obraz po

procesie szyfrowania i deszyfrowania. Ćwiczenie stanowiło praktyczne wprowadzenie do zasad działania szyfratorów strumieniowych opartych na LFSR.