UNIWERSYTET KOMISJI EDUKACJI NARODOWEJ
W KRAKOWIE

Instytut Bezpieczeństwa i Informatyki

KRYPTOGRAFIA
(ćwiczenia laboratoryjne)

Ćwiczenie numer:
4

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

Czas realizacji zajęć: 135 min
Temat ćwiczenia: Tryby działań algorytmów symetrycznych (ECB, CTR)

1. Wstęp

Celem laboratorium było zapoznanie się z praktycznym zastosowaniem algorytmu IDEA w dwóch różnych trybach działania: ECB (Electronic Codebook) oraz CTR (Counter). Ćwiczenie polegało na zaimplementowaniu w języku Python procedur szyfrowania i deszyfrowania obrazów BMP z wykorzystaniem obu trybów. Istotnym aspektem było również porównanie skutków szyfrowania — zwłaszcza wizualnych — oraz ocena bezpieczeństwa każdego z trybów.

2. Wprowadzenie teoretyczne

- **Tryb ECB (Electronic Codebook)** – to najprostszy tryb działania szyfru blokowego, w którym każdy blok danych jest szyfrowany niezależnie od innych przy użyciu tego samego klucza. Jego główną wadą jest zachowywanie powtarzalnych wzorców w zaszyfrowanym obrazie, co czyni go nieodpowiednim do danych z wysoką redundancją (np. grafiki).

- ∉ **Tryb CTR (Counter Mode)** – działa jak szyfr strumieniowy. Zamiast szyfrować dane bezpośrednio, szyfrowany jest licznik, który rośnie dla każdego bloku. Wynik szyfrowania jest następnie łączony z danymi operacją XOR. Gwarantuje to, że nawet identyczne bloki danych wejściowych dają różne bloki szyfrogramu, co znacznie zwiększa bezpieczeństwo i eliminuje wzorce.

3. Opis implementacji

ECB.py

Zaimplementowano dwa niezależne skrypty: `ECB.py` oraz `CTR.py`.

ECB.py:

- Funkcja `encrypt_bmp()` wczytuje dane BMP, dodaje padding zerami do wielokrotności 8 bajtów, dzieli dane na bloki i szyfruje je algorytmem IDEA. Dane są następnie zapisywane do pliku.

```python
def encrypt_bmp(input_file, output_file, key):
    bmp_header, pixel_data = read_bmp(input_file)
    cipher = IDEA(key)
    padded_data, original_length = pad_data(pixel_data)

    encrypted_data = bytearray()
    for i in range(0, len(padded_data), 8):
        block = int.from_bytes(padded_data[i:i + 8], byteorder='big')
        encrypted_block = cipher.encrypt(block)
        encrypted_data.extend(encrypted_block.to_bytes(8, byteorder='big'))

    write_bmp(output_file, bmp_header, encrypted_data)
    print(f"[+] Encrypted BMP saved as {output_file}")
```

- Funkcja `decrypt_bmp()` wykonuje odwrotną operację, odczytując zaszyfrowany plik i deszyfrując każdy blok. Padding jest usuwany na końcu.

```python

def decrypt_bmp(input_file, output_file, key):
    bmp_header, encrypted_data = read_bmp(input_file)
    cipher = IDEA(key)

    decrypted_data = bytearray()
    for i in range(0, len(encrypted_data), 8):
        block = int.from_bytes(encrypted_data[i:i + 8], byteorder='big')
        decrypted_block = cipher.decrypt(block)
        decrypted_data.extend(decrypted_block.to_bytes(8, byteorder='big'))

    decrypted_data = unpad_data(decrypted_data, len(encrypted_data))
    write_bmp(output_file, bmp_header, decrypted_data)
    print(f"[+] Decrypted BMP saved as {output_file}")

```

CTR.py

- Funkcja `encrypt_bmp_ctr()` działa podobnie, lecz szyfruje licznik (nonce + i), a następnie wynik XORuje z blokiem danych.

```python
def encrypt_bmp_ctr(input_file, output_file, key, nonce):
    bmp_header, pixel_data = read_bmp(input_file)
    cipher = IDEA(key)
    padded_data, original_length = pad_data(pixel_data)

    encrypted_data = bytearray()
    for i in range(0, len(padded_data), 8):
        counter = nonce + (i // 8)
        keystream = cipher.encrypt(counter)
        block = int.from_bytes(padded_data[i:i + 8], byteorder='big')
        encrypted_block = block ^ keystream
        encrypted_data.extend(encrypted_block.to_bytes(8, byteorder='big'))

    write_bmp(output_file, bmp_header, encrypted_data)
    print(f"[+] Encrypted BMP (CTR) saved as {output_file}")
```

- Funkcja `decrypt_bmp_ctr()` wykorzystuje tę samą logikę, ponieważ XOR jest operacją symetryczną – ta sama funkcja może służyć do szyfrowania i deszyfrowania.

```python
def decrypt_bmp_ctr(input_file, output_file, key, nonce):
    bmp_header, encrypted_data = read_bmp(input_file)
    cipher = IDEA(key)

    decrypted_data = bytearray()
    for i in range(0, len(encrypted_data), 8):
        counter = nonce + (i // 8)
        keystream = cipher.encrypt(counter)
        block = int.from_bytes(encrypted_data[i:i + 8], byteorder='big')
        decrypted_block = block ^ keystream
        decrypted_data.extend(decrypted_block.to_bytes(8, byteorder='big'))

    decrypted_data = unpad_data(decrypted_data, len(encrypted_data))
    write_bmp(output_file, bmp_header, decrypted_data)
    print(f"[+] Decrypted BMP (CTR) saved as {output_file}")
```

W obu przypadkach zastosowano wspólne funkcje: `read_bmp()`, `write_bmp()`, `pad_data()`, `unpad_data()` do obsługi plików graficznych i danych binarnych.

```python
def read_bmp(filename):
    with open(filename, 'rb') as f:
        bmp_header = f.read(54)
        pixel_data = f.read()
    return bmp_header, bytearray(pixel_data)


def write_bmp(filename, bmp_header, pixel_data):
    with open(filename, 'wb') as f:
        f.write(bmp_header)
        f.write(pixel_data)


def pad_data(data, block_size=8):
    padding_len = (block_size - len(data) % block_size) % block_size
    return data + bytes([0] * padding_len), len(data)


def unpad_data(data, original_length):
    return data[:original_length]
```

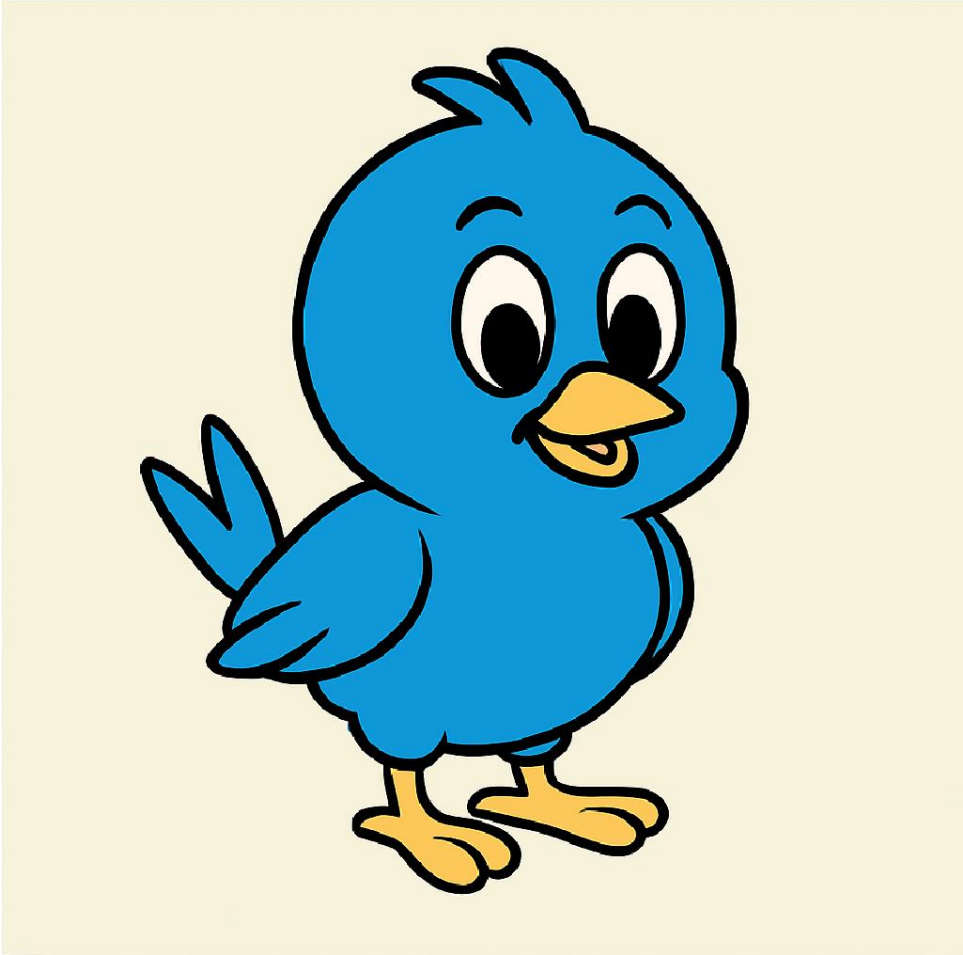
4. Testowanie i wyniki

Do testów wykorzystano obraz `input.bmp` przedstawiający grafikę z wyraźnymi konturami (ptak o niebieskim kolorze na jasnym tle), co pozwala łatwo zaobserwować zachowanie wzorców w szyfrogramie.

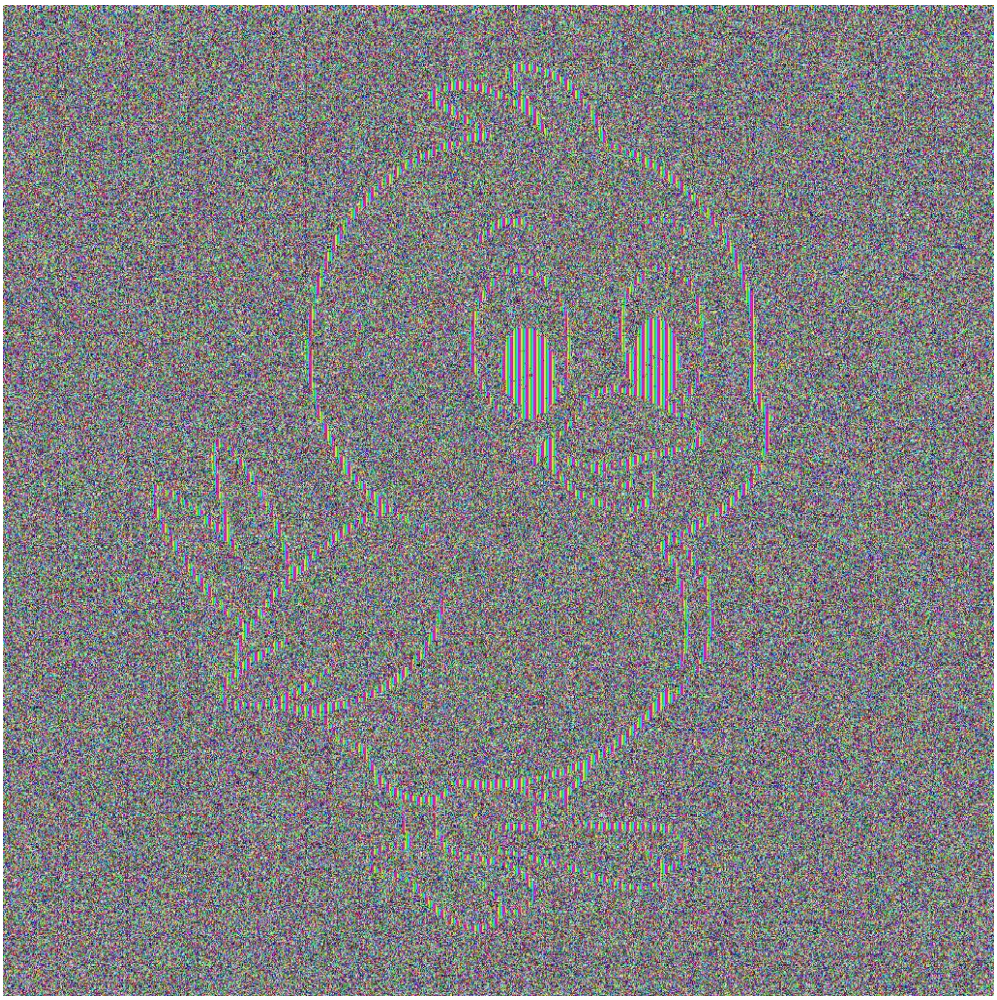
- W każdym uruchomieniu generowano losowy 128-bitowy klucz dla algorytmu IDEA.
- Dla trybu CTR dodatkowo tworzono losowy 64-bitowy nonce.

Testowane pliki:

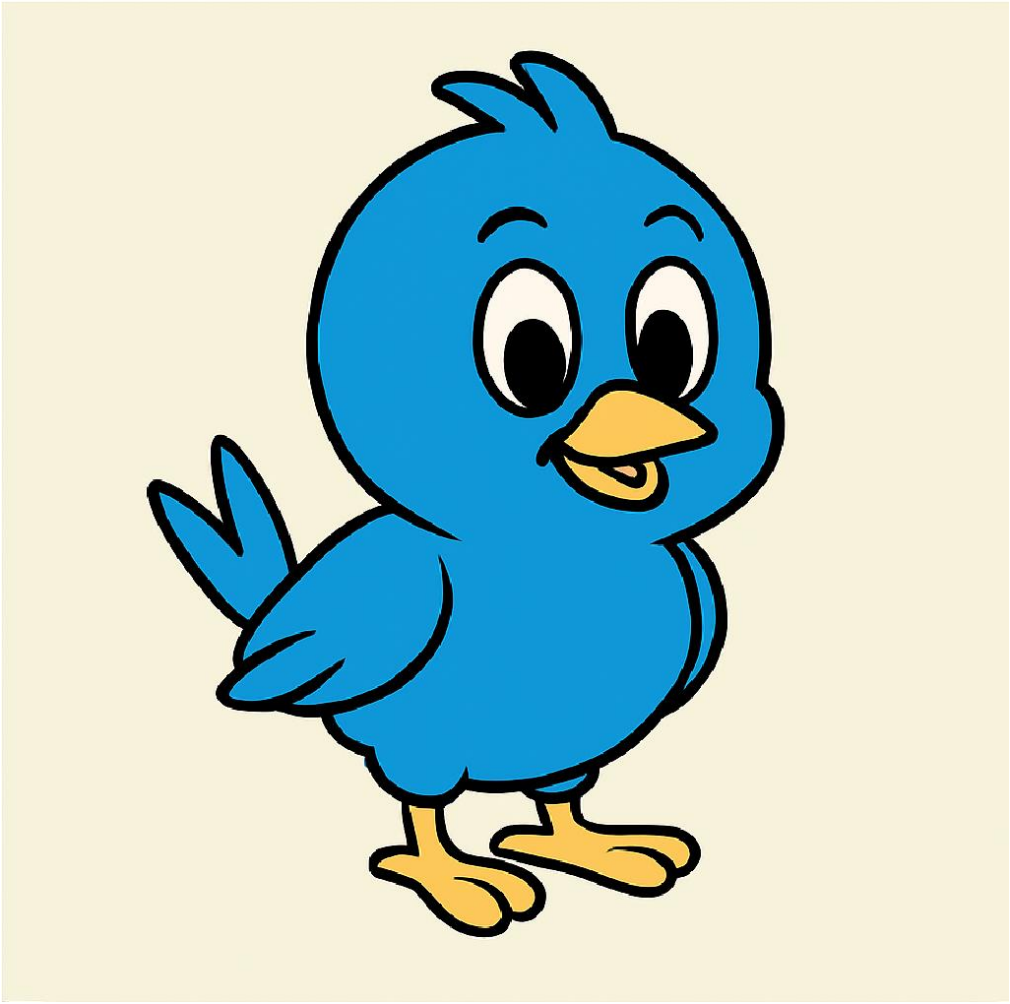
- `input.bmp` – obraz źródłowy



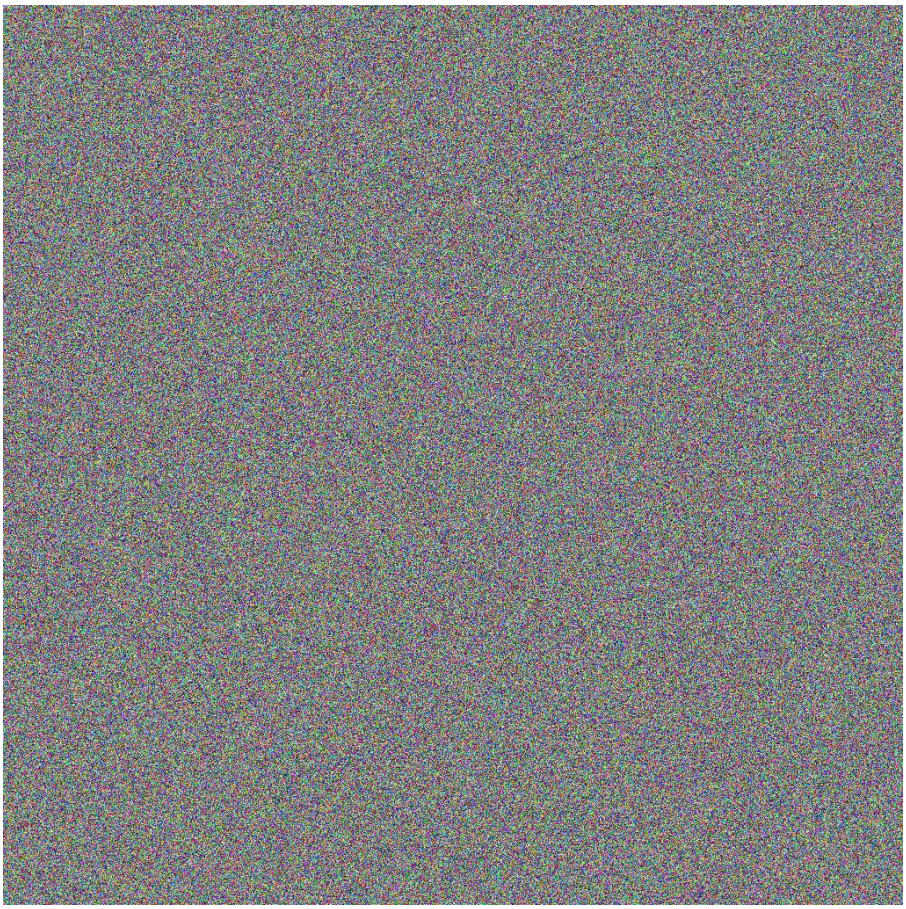
- `encrypted_ecb.bmp` – zaszyfrowany w trybie ECB



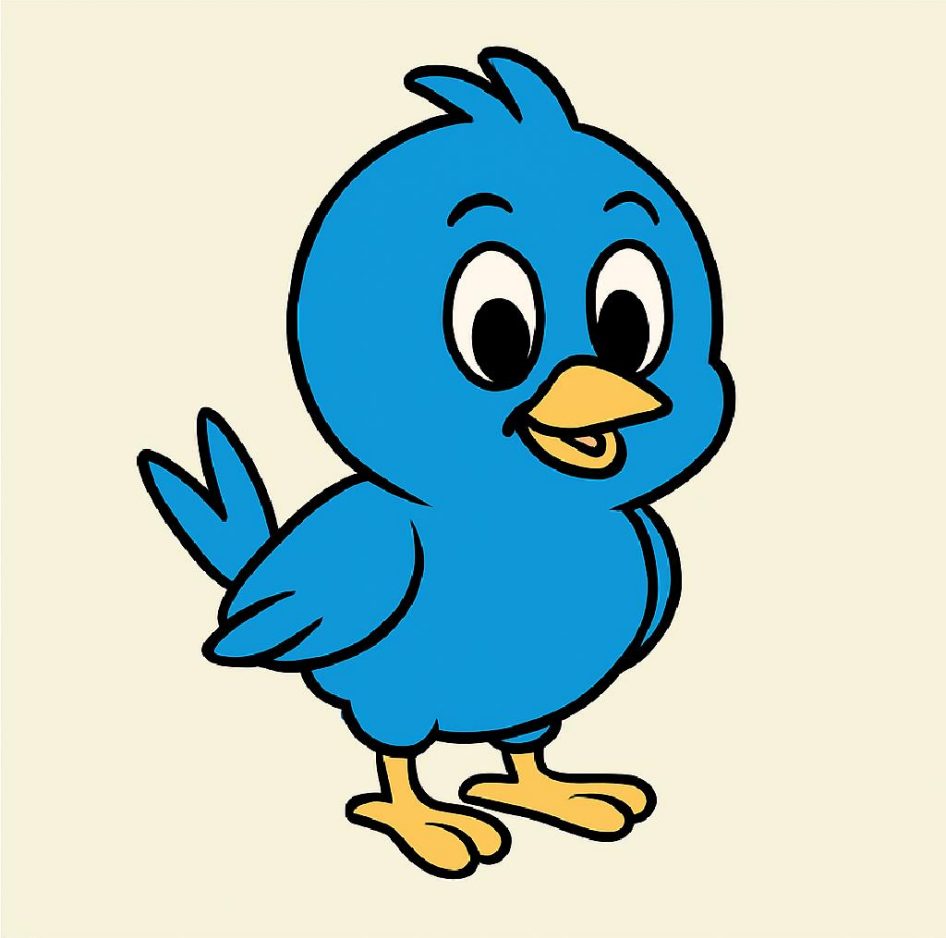
- `decrypted_ecb.bmp` – odszyfrowany w trybie ECB



- `encrypted_ctr.bmp` – zaszyfrowany w trybie CTR



- `decrypted_ctr.bmp` – odszyfrowany w trybie CTR



1. Porównanie wyników

ECB:

- Obraz zaszyfrowany w trybie ECB zachował charakterystyczne wzory i kontury oryginału.
- Powtarzalne fragmenty danych (np. tło lub kształty) są widoczne w szyfrogramie, co stanowi istotne zagrożenie dla poufności danych.

CTR:

- Szyfrogram uzyskany w trybie CTR przypominał losowy szum.
- Nie dało się odróżnić żadnych wzorców — nawet przy powtarzających się blokach wejściowych.
- Odszyfrowany obraz był identyczny z oryginałem, co potwierdza poprawność działania.

5. Wnioski

Laboratorium pokazało praktyczne różnice między dwoma trybami działania szyfru blokowego. Tryb ECB, mimo prostoty implementacji, nie zapewnia odpowiedniego poziomu bezpieczeństwa dla danych o powtarzalnej strukturze, takich jak obrazy BMP. Tryb CTR natomiast eliminuje te słabości, oferując znacznie lepsze zabezpieczenie przy porównywalnej złożoności obliczeniowej.

Zarówno implementacja, jak i testy zakończyły się sukcesem. Zrealizowano wszystkie założenia ćwiczenia, a wyniki jednoznacznie wskazują na przewagę trybu CTR nad ECB w kontekście szyfrowania danych graficznych.