UNIWERSYTET KOMISJI EDUKACJI NARODOWEJ
W KRAKOWIE

Instytut Bezpieczeństwa i Informatyki

KRYPTOGRAFIA
(ćwiczenia laboratoryjne)

Ćwiczenie numer:
2

Imię i nazwisko: Jakub Radawiec
Numer grupy: L1
Studia stacjonarne

Czas realizacji zajęć: 180 min
Temat ćwiczenia: International Data Encryption Algorithm

1. Wstęp

Celem niniejszego ćwiczenia było zapoznanie się z teorią działania algorytmu szyfrowania blokowego IDEA oraz jego praktyczna implementacja w języku Python. Zadanie obejmowało stworzenie kodu realizującego zarówno proces szyfrowania, jak i deszyfrowania 64-bitowych bloków danych przy użyciu 128-bitowego klucza. Poprawność działania zaimplementowanego algorytmu została zweryfikowana poprzez testy, a cały proces wraz z wynikami udokumentowano w niniejszym sprawozdaniu.

2. Wprowadzenie teoretyczne

Algorytm IDEA (International Data Encryption Algorithm) to symetryczny szyfr blokowy, co oznacza, że ten sam klucz jest używany zarówno do szyfrowania, jak i deszyfrowania. Operuje on na stałych fragmentach danych zwanych blokami, które w przypadku IDEA mają rozmiar 64 bitów. Klucz używany do sterowania procesem szyfrowania jest stosunkowo długi i ma 128 bitów.

Zaprojektowany na początku lat 90. przez Lai i Masseya, IDEA zyskał uznanie dzięki swojej unikalnej strukturze opartej na mieszaniu trzech różnych typów operacji matematycznych, które działają na 16-bitowych częściach bloku danych:

- **XOR bitowy**: Prosta operacja logiczna.
- **Dodawanie modulo 2^16 (65536)**: Standardowe dodawanie, gdzie wynik jest "zawijany" po przekroczeniu 65535.
- **Mnożenie modulo 2^16 + 1 (65537)**: Specyficzne mnożenie, gdzie dodatkowo liczba 0 jest traktowana jako 2^16 (65536) dla celów obliczeń.

Szyfrowanie w IDEA nie odbywa się w jednym kroku. Zamiast tego, 64-bitowy blok danych przechodzi przez 8 identycznych rund obliczeniowych. Każda runda wykorzystuje sześć 16-bitowych fragmentów klucza (podkluczy). Po 8 rundach następuje jeszcze końcowa transformacja wyjściowa, używająca czterech ostatnich podkluczy. W sumie, z oryginalnego 128-bitowego klucza, generowane są 52 podklucze (8 rund * 6 podkluczy/rundę + 4 podklucze końcowe). Proces generowania podkluczy sam w sobie jest częścią algorytmu i zapewnia, że różne części klucza wpływają na różne etapy szyfrowania.

3. Opis implementacji

- **Struktura Klasy IDEA:**

Główna klasa IDEA zawiera wszystkie niezbędne metody do szyfrowania i deszyfrowania. Jej konstruktor __init__ przyjmuje 128-bitowy klucz, inicjalizuje generowanie podkluczy szyfrujących (gen_keys) i deszyfrujących (generate_decryption_keys) oraz przechowuje je w atrybutach _keys i _dkeys.

```
class IDEA:
    def __init__(self, key):
        self._keys = None
        self.gen_keys(key)
        self.generate_decryption_keys()
```

- **Podstawowe Operacje Modulo:**

Zgodnie ze specyfikacją IDEA, zaimplementowano dwie podstawowe operacje: mnożenie modulo 0x10001 (mul_mod) z obsługą wartości 0 jako 0x10000, oraz dodawanie modulo 0x10000 (add_mod).

```
def mul_mod(self, a, b):
    assert 0 <= a <= 0xFFFF
    assert 0 <= b <= 0xFFFF
    if a == 0:
        a = 0x00000
    if b == 0:
        b = 0x00000
    r = (a * b) % 0x10001
    if r == 0x10000:
        r = 0
    return r

def add_mod(self, a, b):
    return (a + b) % 0x10000
```

- **Operacje Odwrotne:**

Do generowania kluczy deszyfrujących oraz do samej deszyfracji niezbędne są operacje odwrotne: odwrotność addytywna modulo 0x10000 (add_inv) oraz odwrotność multiplikatywna modulo 0x10001 (mul_inv), obliczana za pomocą Rozszerzonego Algorytmu Euklidesa.

```python
def add_inv(self, key):
    return (0x10000 - key) % 0x10000

def mul_inv(self, key):
    if key == 0:
        return 0
    t0, t1 = 0, 1
    r0, r1 = 0x10001, key
    while r1 != 0:
        q = r0 // r1
        r0, r1 = r1, r0 - q * r1
        t0, t1 = t1, t0 - q * t1
    return t0 % 0x10001
```

- **Generowanie Podkluczy Szyfrujących:**

Metoda gen_keys pobiera 128-bitowy klucz główny i generuje 52 16-bitowe podklucze szyfrujące. Wykorzystuje ekstrakcję 16-bitowych fragmentów oraz cykliczne przesunięcie klucza głównego o 25 bitów po każdych 8 wygenerowanych podkluczach. Wynik jest przechowywany w _keys.

```python
def generate_decryption_keys(self):
    dkeys = []
    for i in range(9):
        k = self._keys[i]
        if i == 0 or i == 8:
            k1 = self.mul_inv(k[0])
            k2 = self.add_inv(k[1 if i == 0 else 2])
            k3 = self.add_inv(k[2 if i == 0 else 1])
            k4 = self.mul_inv(k[3])
        else:
            k1 = self.mul_inv(k[0])
            k2 = self.add_inv(k[2])
            k3 = self.add_inv(k[1])
            k4 = self.mul_inv(k[3])
        k5 = k[4]
        k6 = k[5]
        dkeys.insert(0, (k1, k2, k3, k4, k5, k6))
    self._dkeys = dkeys
```

- **Funkcja Rundy:**

Metoda round realizuje logikę pojedynczej rundy obliczeniowej algorytmu IDEA. Przyjmuje cztery 16-bitowe części bloku danych oraz 6 podkluczy rundowych, wykonuje sekwencję operacji XOR, dodawania modulo i mnożenia modulo, zwracając cztery przetworzone 16-bitowe części bloku.

```
def round(self, p1, p2, p3, p4, keys):
    k1, k2, k3, k4, k5, k6 = keys
    p1 = self.mul_mod(p1, k1)
    p2 = self.add_mod(p2, k2)
    p3 = self.add_mod(p3, k3)
    p4 = self.mul_mod(p4, k4)
    x = p1 ^ p3
    t0 = self.mul_mod(x, k5)
    x = self.add_mod(p2 ^ p4, t0)
    t1 = self.mul_mod(x, k6)
    t2 = self.add_mod(t0, t1)
    return p1 ^ t1, t2 ^ p2, t2 ^ p3, p4 ^ t1
```

- **Szyfrowanie:**

Metoda encrypt odpowiada za cały proces szyfrowania 64-bitowego bloku. Dzieli blok wejściowy na 4 części, przepuszcza je przez 8 rund (wywołując metodę round z odpowiednimi podkluczami z _keys), wykonuje końcową transformację wyjściową (pół-rundę) i łączy wynikowe części w 64-bitowy szyfrogram.

```
def encrypt(self, plain):
    p1 = (plain >> 48) & 0xFFFF
    p2 = (plain >> 32) & 0xFFFF
    p3 = (plain >> 16) & 0xFFFF
    p4 = plain & 0xFFFF
    for i in range(8):
        p1, p2, p3, p4 = self.round(p1, p2, p3, p4, self._keys[i])
    k1, k2, k3, k4, _, _ = self._keys[8]
    y1 = self.mul_mod(p1, k1)
    y2 = self.add_mod(p3, k2)
    y3 = self.add_mod(p2, k3)
    y4 = self.mul_mod(p4, k4)
    return (y1 << 48) | (y2 << 32) | (y3 << 16) | y4
```

- **Deszyfrowanie:**

Metoda decrypt realizuje proces odwrotny do szyfrowania. Strukturalnie jest bardzo podobna do encrypt, ale wykorzystuje podklucze deszyfrujące (_dkeys) w odpowiedniej kolejności, aby odtworzyć oryginalny 64-bitowy blok tekstu jawnego z szyfrogramu.

```python
def decrypt(self, encrypted):
    p1 = (encrypted >> 48) & 0xFFFF
    p2 = (encrypted >> 32) & 0xFFFF
    p3 = (encrypted >> 16) & 0xFFFF
    p4 = encrypted & 0xFFFF
    for i in range(8):
        p1, p2, p3, p4 = self.round(p1, p2, p3, p4, self._dkeys[i])
    k1, k2, k3, k4, _, _ = self._dkeys[8]
    y1 = self.mul_mod(p1, k1)
    y2 = self.add_mod(p3, k2)
    y3 = self.add_mod(p2, k3)
    y4 = self.mul_mod(p4, k4)
    return (y1 << 48) | (y2 << 32) | (y3 << 16) | y4
```

4. Testowanie i wyniki

Do przetestowania implementacji wykorzystano funkcję main, która wykonuje następujące kroki:

1. Definiuje 128-bitowy klucz główny składający się z samych zer.
2. Definiuje 64-bitowy blok tekstu jawnego składający się z samych zer.
3. Tworzy instancję klasy IDEA z zadanym kluczem.
4. Szyfruje blok tekstu jawnego.
5. Deszyfruje uzyskany szyfrogram.
6. Wyświetla klucz, tekst jawny, szyfrogram oraz odszyfrowany tekst w postaci binarnej.
7. Sprawdza, czy odszyfrowany tekst jest identyczny z oryginalnym tekstem jawnym i wyświetla odpowiedni komunikat.

Wyniki uzyskane po uruchomieniu programu:

```
C:\Users\Michał\Desktop\crypto\2>python IDEA.py
Klucz:  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Tekst jawny:    0000000000000000000000000000000000000000000000000000000000000000
Zaszyfrowany:   0000000000000000000000000000000000000000000000000000000000000000
Odszyfrowany:   0000000000000000000000000000000000000000000000000000000000000000
Deszyfrowanie poprawne – odzyskano oryginalny blok.
```

5. Omówienie wyników

Zgodnie z oczekiwaniami, zaimplementowany algorytm IDEA poprawnie przetworzył dane wejściowe. Proces szyfrowania dla klucza zerowego i tekstu jawnego zerowego wygenerował określony szyfrogram. Następnie, proces deszyfrowania, wykorzystując wygenerowane klucze deszyfrujące, był w stanie dokładnie odwrócić operację szyfrowania. Odszyfrowany blok danych

okazał się identyczny z oryginalnym tekstem jawnym, co potwierdza poprawność implementacji zarówno algorytmu szyfrowania, deszyfrowania, jak i procedur generowania kluczy.

6. Wnioski

W ramach ćwiczenia pomyślnie zaimplementowano algorytm szyfrowania blokowego IDEA w języku Python. Zaimplementowano wszystkie kluczowe komponenty algorytmu: operacje modulo (dodawanie i mnożenie), generowanie podkluczy szyfrujących i deszyfrujących, funkcję rundy oraz procesy szyfrowania i deszyfrowania. Przeprowadzone testy potwierdziły, że implementacja działa poprawnie i jest zgodna ze specyfikacją algorytmu IDEA, umożliwiając skuteczne szyfrowanie i odwracalne deszyfrowanie danych.