Tryby pracy szyfrów blokowych

1. Przeanalizuj dostępne tryby pracy szyfrów blokowych w wybranym środowisku programowania i zmierz czasy szyfrowania i deszyfrowania dla 3 różnej wielkości plików we wszystkich 5 podstawowych trybach ECB, CBC, OFB, CFB, i CTR. Zinterpretuj otrzymane wyniki.

Dla bible.txt(wielokrotnie skopiowane txt biblii) o rozmiarze 135MB:

Tryb pracy	Czas szyfrowania	Czas deszyfrowania	Całkowity czas
ECB	0.106965	0.108965	0.215930
CBC	0.402868	0.393875	0.796742
CTR	0.112961	0.113963	0.226924

Tryb ECB jest zazwyczaj najszybszy, ponieważ nie wymaga operacji łańcuchowania ani generowania wektorów inicjalizujących, tryby CBC i CTR są nieco wolniejsze ze względu na dodatkowe operacje.

W trybie ECB, oraz trybie CTR zaletą jest to, że równoległe można szyfrować jak I deszyfrować dane. Dzięki temu obie te opcje bardzo szybko są w stanie przetwarzać dane. Jak widzimy na załączonej tabeli różnica czasu pomiędzy ECB a CTR jest bardzo mała, wręcz marginalna. Dużo wolniejszy zdaje się być tryb CBC.

Tryb CBC równolegle pozwala tylko na deszyfrowanie. Dla mimo wszystko tak małego pliku różnica ta(pomiędzy szyfrowaniem a deszyfrowaniem) jest bardzo mała, na pewno dużo mniejsza niż różnica w czasie całkowitym pomiędzy przykładowo CTR a CBC.
Czas deszyfrowania mimo wszystko jest wolniejszy niż w dwóch innych trybach, a to wynika z potrzeby odczytania poprzednich bloków szyfrogramu, oraz wykonania dodatkowego XORa po odszyfrowaniu każdego bloku.

2. Przeanalizuj propagację błędów w wyżej wymienionych trybach pracy. Czy błąd w szyfrogramie będzie skutkował niemożnością odczytania po deszyfrowaniu całej wiadomości, fragmentu, ..? Zinterpretuj wyniki obserwacji.

ECB: Każdy blok szyfrowany jest niezależnie, co pozwala na równoległe przetwarzanie, ale ujawnia wzorce danych. Błąd w szyfrogramie psuje tylko jeden blok tekstu jawnego.

CBC: Każdy blok jest XOR-owany z poprzednim szyfrogramem, zapewniając lepsze bezpieczeństwo, ale wymaga sekwencyjnego szyfrowania. Błąd uszkadza dwa bloki tekstu.

CTR: Działa jak strumieniowy szyfr, umożliwiając pełną równoległość i szybkość. Błąd wpływa tylko na pojedynczy bit w danym bloku, bez propagacji.

Mode	Effect of bit errors in C_i
ECB	Random bit errors in P_i
CBC	Random bit errors in P_i Specific bit errors in P_{i+1}
CFB	Specific bit errors in P_i Random bit errors in P_{i+1}, …, until synchronization is restored
OFB	Specific bit errors in P_i
CTR	Specific bit errors in P_i

Figure 1: Screenshot z Wikipedii "Summary of Effect of Bit Errors on Decryption"

3. Zaimplementuj tryb CBC (korzystając z dostępnego w wybranym środowisku programowania trybu ECB).

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
import os


class CBCCipher:
    def __init__(self, key, iv=None):
        self.key = key
        self.block_size = AES.block_size

        if iv is None:
            self.iv = get_random_bytes(self.block_size)
        else:
            if len(iv) != self.block_size:
                raise ValueError(f"IV musi mieć długość {self.block_size} bajtów")
            self.iv = iv

        # Inicjalizacja podstawowego szyfru ECB
        self.ecb_cipher = AES.new(self.key, AES.MODE_ECB)

    def encrypt(self, plaintext):
        # Szyfrowanie w trybie CBC

        plaintext = pad(plaintext, self.block_size)
        ciphertext = bytearray()
        previous_block = self.iv

        for i in range(0, len(plaintext), self.block_size):
            block = plaintext[i:i + self.block_size]

            # XOR z poprzednim blokiem szyfrogramu (lub IV dla pierwszego bloku)
            xored_block = bytes(a ^ b for a, b in zip(block, previous_block))

            # Szyfrowanie bloku w trybie ECB
            encrypted_block = self.ecb_cipher.encrypt(xored_block)
            ciphertext.extend(encrypted_block)

            # Aktualizacja poprzedniego bloku dla następnej iteracji
```

```
            previous_block = encrypted_block

        # Zwracamy IV razem z zaszyfrowanym tekstem (standardowa praktyka)
        return self.iv + ciphertext

    def decrypt(self, ciphertext):
        # Deszyfrowanie w trybie CBC

        if len(ciphertext) < 2 * self.block_size:
            raise ValueError("Ciphertext jest zbyt krótki")

        # Wydzielenie IV i właściwego szyfrogramu
        iv = ciphertext[:self.block_size]
        ciphertext = ciphertext[self.block_size:]

        plaintext = bytearray()
        previous_block = iv

        for i in range(0, len(ciphertext), self.block_size):
            block = ciphertext[i:i + self.block_size]

            # Odszyfrowanie bloku w trybie ECB
            decrypted_block = self.ecb_cipher.decrypt(block)

            # XOR z poprzednim blokiem szyfrogramu (lub IV dla pierwszego bloku)
            xored_block = bytes(a ^ b for a, b in zip(decrypted_block, previous_block))
            plaintext.extend(xored_block)

            # Aktualizacja poprzedniego bloku dla następnej iteracji
            previous_block = block

        # Usunięcie paddingu
        return unpad(plaintext, self.block_size)


if __name__ == "__main__":
    # Generowanie klucza i testowego IV
    key = get_random_bytes(16)  # AES-128
    iv = get_random_bytes(16)

    # Inicjalizacja naszego szyfru CBC
    cbc = CBCCipher(key, iv)

    # Testowy tekst do zaszyfrowania
    plaintext = b"To jest przykladowy tekst do zaszyfrowania w trybie CBC z uzyciem ECB"

    # Szyfrowanie
    ciphertext = cbc.encrypt(plaintext)
    print(f"Zaszyfrowany tekst (IV + ciphertext): {ciphertext.hex()}")

    # Deszyfrowanie
    decrypted = cbc.decrypt(ciphertext)
    print(f"Odszyfrowany tekst: {decrypted.decode('utf-8')}")

    # Weryfikacja
    assert decrypted == plaintext, "Deszyfrowanie nie powiodlo sie!"
    print("Weryfikacja pomyslna - tekst odszyfrowany poprawnie")
```