Podstawy Kryptografii – Sprawozdanie z Laboratorium nr 4 (9.04.2025)

Jakub Bilski (155865)

Informatyka (ST), Semestr VI, Rok 2024/2025

Link do repozytorium

1. **Wstęp**

   Algorytm AES operuje na blokach o stałym rozmiarze 128 bitów (16 bajtów), wykorzystując klucz o długości 128 bitów (opcjonalnie 192/265 bitów). Każdy blok jest w kolejnych 10 rundach poddawany operacjom: SubBytes (nieliniowe podstawienie bajtowe), ShiftRows (cykliczne przesunięcie wierszy), MixColumns (mieszanie kolumn w macierzy stanu) i AddRoundKey (operacja XOR ze składową klucza rundy).

   Aby szyfrować dane dłuższe niż pojedyncze blok, wykorzystuje się różne tryby operacji, m.in.:

- ECB (Electronic CodeBook):
  - Każdy 16-bajtowy blok testu jawnego szyfruje się niezależnie tym samym kluczem
  - Niezależne szyfrowanie każdego blokuje pozwala na równoległą implementację algorytmu
  - Identyczne bloki jawne zawsze dają identyczne bloki szyfrogramu, co ujawnia wzorce w danych
- CBC (Cipher Block Chaining):
  - Wykonywana jest operacja XOR na pierwszym bloku plaintextu z losowym wektorem inicjalizacyjnym (IV)
  - Po zaszyfrowaniu pierwszego bloku, kolejny blok jest XOR-owany z poprzednim szyfrogramem itd.
  - Powtarzające się fragmenty tekstu jawnego nie przekładają się na powtarzające się fragmenty szyfrogramu
- CTR (Counter Mode):
  - Generowany jest strumień klucza przez szyfrowania kolejnych wartości licznika (losowy nonce + licznik bloków)
  - Wykonanie operacji XOR między licznikiem a każdym kolejnym blokiem plaintextem
  - Możliwa implementacja równoległa

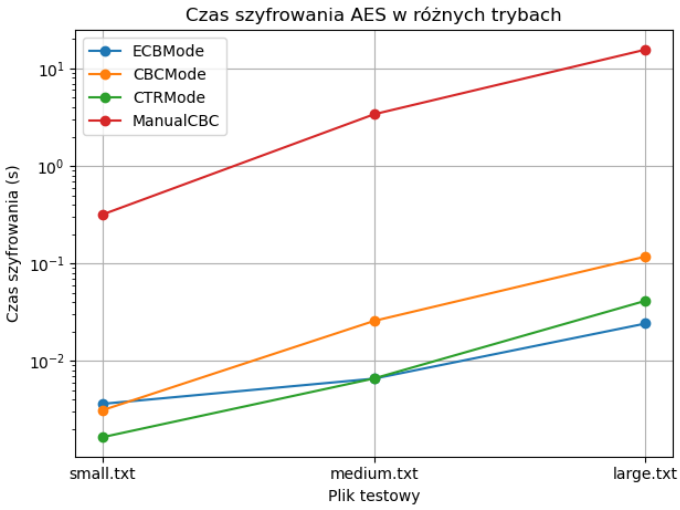
2. **Opis użytego kodu**

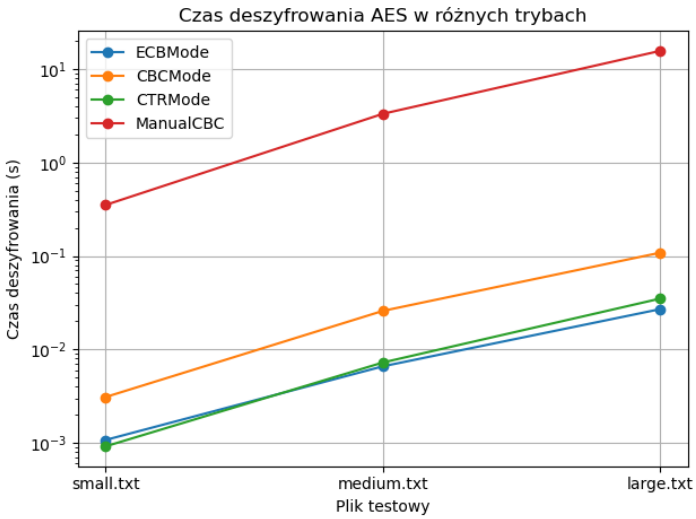
a. **Generacja plików testowych**
   Pliki small.txt (1 MB), medium.txt (10 MB) i large.txt (50 MB) powstały w skrypcie generator.py, który losowo dobiera znaki ASCII i zapisuje łańcuchy o zadanym rozmiarze.
b. **Definicje trybów szyfrowania**
   W pliku ciphers.py zaimplementowano klasy:
   - ECBMode, CBCMode, CTRMode – korzystają bezpośrednio z Crypto.Cipher.AES
   - ManualCBC – ręczna implementacja CBC poprzez:
     - Dzielenie na bloki po 16 bajtów
     - XOR z poprzednim blokiem/IV
     - Szyfrowanie w trybie ECB
     - Padding PKCS#7
c. **Pomiar czasu i analiza propagacji błędu**
   W utils.py funkcja measure_time mierzy czas wykonania funkcji, a analyze_error_propagation:
   - Szyfruje oryginalne dane
   - Szyfruje dane ze zmienionym jednym bitem, porównuje szyfrogramy
   - Deszyfruje szyfrogram ze zmienionym jednym bitem
   - Porównuje uzyskany tekst jawny
d. **Główny plik main.py**
   - Generowanie plików testowych
   - Dla każdego pliku i trybu mierzy czasy i propagację błędu
   - Zapisuje wyniki do plików .csv
   - Wywołuje plot_results.py do rysowania wykresów

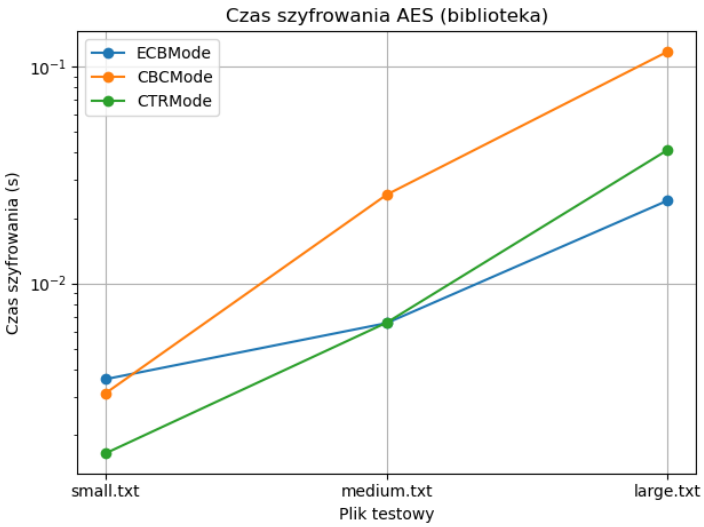
3. Wyniki pomiarów i analiza

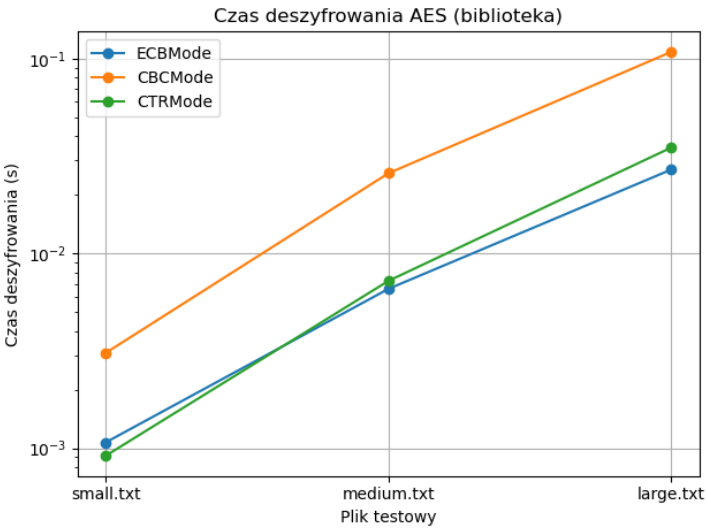
a. Czasy szyfrowania i deszyfrowania

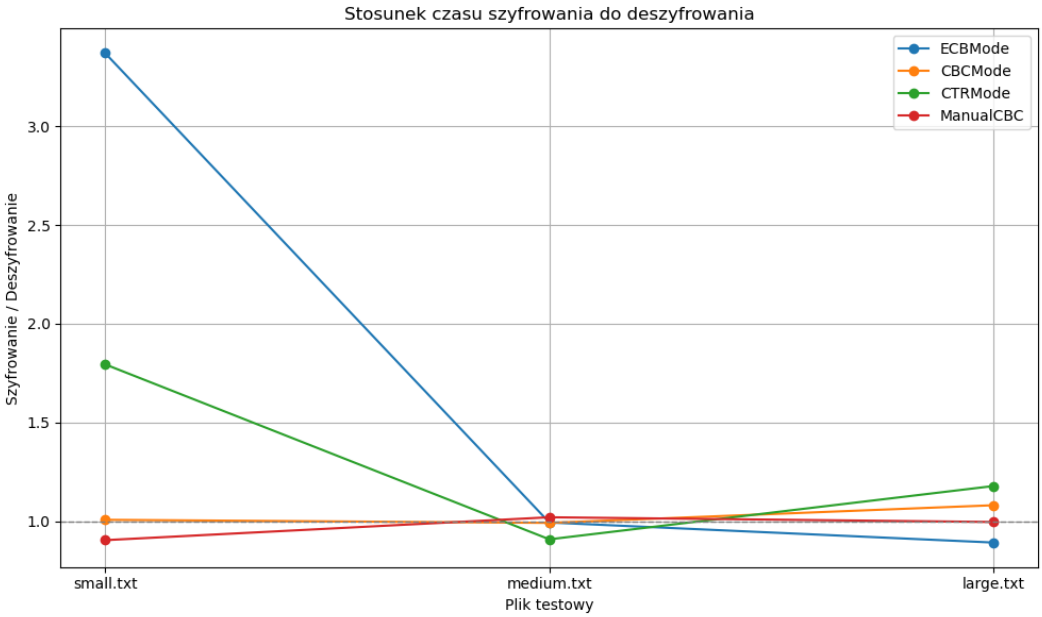
File	Mode	Encryption Time (s)	Decryption Time (s)
small.txt	ECBMode	0.003613	0.001072
small.txt	CBCMode	0.003113	0.003088
small.txt	CTRMode	0.001644	0.000916
small.txt	ManualCBC	0.317642	0.350915
medium.txt	ECBMode	0.006555	0.006591
medium.txt	CBCMode	0.025658	0.025848
medium.txt	CTRMode	0.006588	0.007244
medium.txt	ManualCBC	3.390828	3.321024
large.txt	ECBMode	0.024114	0.027001
large.txt	CBCMode	0.117399	0.108514
large.txt	CTRMode	0.041294	0.035021
large.txt	ManualCBC	15.634471	15.679946









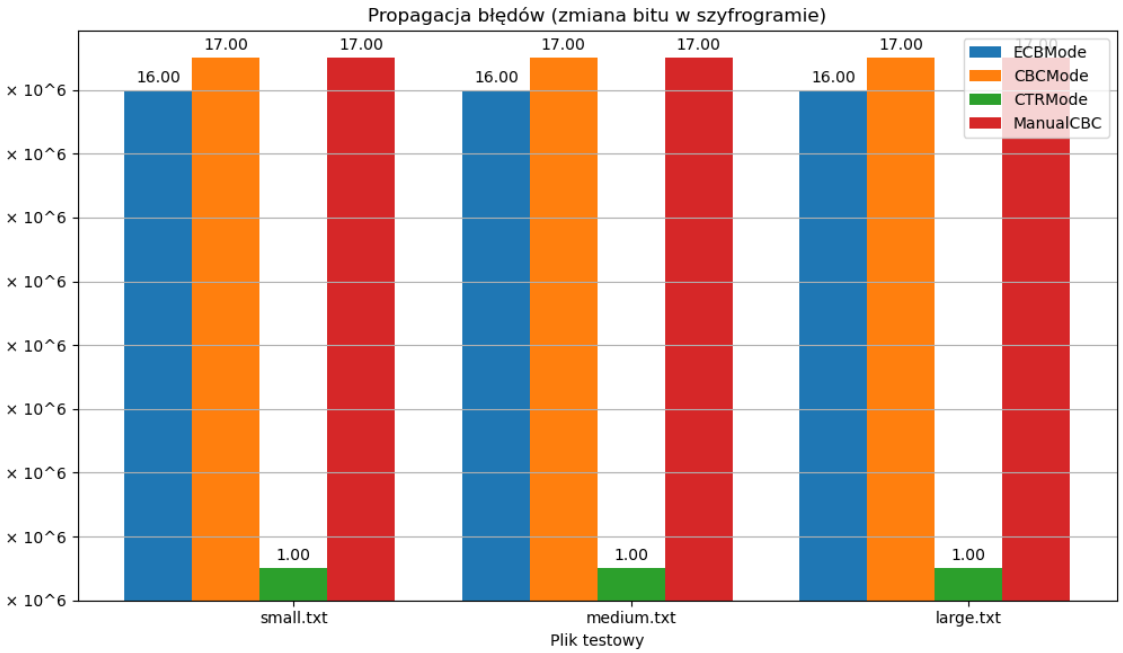


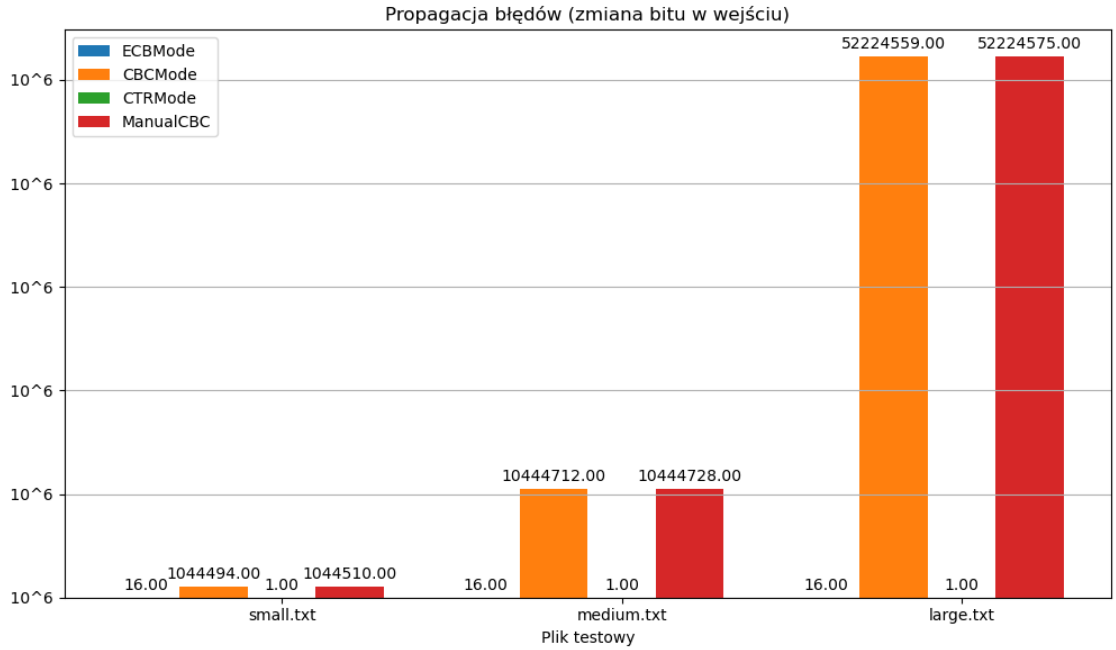
- **CTR vs ECB przy 1 MB**
  - Dla małego pliku cały zestaw danych mieści się w pamięci podręcznej procesora, więc generowanie strumienia klucza (AES+XOR) w trybie CTR i czyste szyfrowanie bloków w ECB odbywa się opóźnieniem. CTR wypada lepiej, bo XOR jest szybką operacją, a całość trzyma się w cache.
- **Przełom przy 10 MB**
  - Gdy dane mogą zacząć wykraczać poza pamięć cache, ważny jest także koszt przesyłu przez RAM. W ECB wywoływany jest tylko silnik AES, a w CTR dodatkowo wykorzystuje strumień klucza i XOR-uje blok – dodatkowe odczyty/zapisy mają wpływ

- **Dominacja ECB przy 50 MB**
    - Na największych plikach wpływ pamięci RAM jest jeszcze większy – CTR musi dwukrotnie czytać/zapisać każdy blok (generowanie strumienia klucza + XOR), a ECB tylko jednorazowo przechodzi po buforze szyfrując go.
- **CBC**
    - Szyfrowanie nie może odbyć się równolegle – dla każdego kolejnego bloku czeka się na wyniki szyfrowania bloku poprzedniego.
    - Bibliotek pycrypto ma zaimplementowany rdzeń AES i iterację po blokach bezpośrednio w języku C, który jest wielokrotnie szybszy niż Python
    - Ręcznie zaimplementowany CBC jest o wiele wolniejszy, ponieważ jest zaimplementowany w Pythonie: dla każdego 16-bajtowego bloku zachodzą operacje: np.frombuffer -> np.bitwise_xor -> tobytes() -> AES.encrypt
    - Ręczna implementacja z wykorzystaniem biblioteki NumPy (operacje wykonywane są w języku C) pozwoliła zmniejszyć różnice między dwiema wersjami – gdyby zdecydować się na implementację całkowicie za pomocą standardowej biblioteki Python, ManualCBC byłby jeszcze wolniejszy

b. Propagacja błędu

File	Mode	Input Diff	Cipher Diff
small.txt	ECBMode	16	16
small.txt	CBCMode	1044494	17
small.txt	CTRMode	1	1
small.txt	ManualCBC	1044510	17
medium.txt	ECBMode	16	16
medium.txt	CBCMode	10444712	17
medium.txt	CTRMode	1	1
medium.txt	ManualCBC	10444728	17
large.txt	ECBMode	16	16
large.txt	CBCMode	52224559	17
large.txt	CTRMode	1	1
large.txt	ManualCBC	52224575	17





- **ECB – błąd w wejściu**
  - Zmiana jednego bitu w tekście jawnym wpływa wyłącznie na ten blok AES, więc w szyfrogramie różnicujemy dokładnie 16 bajtów
- **ECB – błąd w szyfrogramie**
  - Analogicznie, uszkodzenie bitu w szyfrogramie przekłada się po odszyfrowaniu na zmianę tylko w jednej macierzy stanu, pozostałe bloki są odszyfrowywane poprawnie

- **CTR – izolacja strumienia**
  - Tryb CTR generuje strumień klucza bajt po bajcie i XOR-uje z danymi: każdy błąd bitowy w wejściu lub szyfrogramie propaguje się do dokładnie 1 bajtu wyścia, bo operacje są niezależne blokowo i strumieniowo
- **CBC - błąd w wejściu**
  - W CBC każdy blok plaintextu jest XOR-owany z poprzednim szyfrogramem. Zmiana bitu w pierwszym bloku powoduje, że każdy kolejny blok szyfrogramu „dziedziczy" błąd w XOR -> praktycznie wszystkie bajty szyfrogramu różnią się
- **CBC – błąd w szyfrogramie**
  - Uszkodzony blok szyfrogramu psuje cały odpowiadający mu blok po odszyfrowaniu (16 bajtów) oraz dodatkowo 1 bajt kolejnego bloku (z powodu XOR z uszkodzonym n-tym blokiem szyfrogramu)

4. **Wnioski i ocena**
   a. **Wydajność**
      i. CTR oferuje najwyższą przepustowość przy małych i średnich danych, dzięki lekkiej operacji XOR i pełnej równoległości
      ii. ECB charakteryzuje się minimalnym narzutem i liniowym skalowaniem; dla najwyższych plików (50 MB) przewyższa CTR, bo unika dwukrotnego przechodzenia po buforze
      iii. CBC (biblioteczne) dodaje koszt XOR-u per blok i sekwencyjności (ok. 2-4 razy względem ECB), co czyni go wolniejszym przy wszystkich rozmiarach testowanych plików
      iv. ManualCBC jest ok. 100 razy wolniejsze od CBC bibliotecznego, głównie z powodu kosztownych konwersji Python <-> NumPy i pętli w czystym Pythonie
   b. **Odporność na błędy**
      i. ECB i CTR izolują błędy do pojedynczego bloku lub bajtu, co sprzyja odporności na przypadkowe uszkodzenia transmisji

ii. CBC świadomie rozlewa błąd – zmiana jednego bitu w wejściu uszkadza niemal cały szyfrogram, co w zastosowaniach sieciowych może wymagać retransmisji całej wiadomości

c. **Bezpieczeństwo**
   i. ECB jest niezalecany do danych z jakimikolwiek wzorcami (np. obrazy, powtarzające się struktury)
   ii. CBC zapewnia ukrycie powtarzalnych bloków (ale wymaga bezpiecznego IV i zarządzania paddingiem)
   iii. CTR daje podobny poziom ukrycia, ale wymaga unikalnego nonce dla każdej sesji; powtórzenie nonce łamie bezpieczeństwo

d. **Rekomendacje**
   i. Tryb CTR – w systemach wymagających maksymalnej wydajności i równoległości (np. szyfrowania dużych plików, aplikacje wielowątkowe)
   ii. Tryb CBC – tam, gdzie ważna jest odporność na odkrywanie struktur danych i mamy pewność co do poprawnego zarządzania IV
   iii. Tryb ECB – wyłącznie do małych, jednorazowych, losowych bloków danych lub tam, gdzie pewne wzorce nie są istotne