

Akademia Nauk Stosowanych w Nowym Sączu Kryptografia i teoria kodów - projekt	
Nazwisko i imię: <b>Zwolenik Arkadiusz</b>	Ocena projektu:
Data: 17.12.2024	Grupa: <b>P3</b>

## 1. Czym jest kryptografia?

Kryptografia to dziedzina nauki zajmująca się tworzeniem metod zabezpieczania informacji oraz zapewniania jej poufności, integralności i autentyczności. Głównym celem kryptografii jest umożliwienie bezpiecznej komunikacji oraz ochrony danych przed nieautoryzowanym dostępem. Termin pochodzi od greckich słów kryptos (ukryty) i graphein (pisać), co dosłownie oznacza „ukryte pisanie”.

### 1.1 Podstawowe cele kryptografii

Kryptografia dąży do spełnienia kilku kluczowych zasad bezpieczeństwa informacji, takich jak:

- **Poufność** – tylko uprawnione osoby mogą uzyskać dostęp do informacji.
- **Integralność** – dane nie mogą być zmieniane lub modyfikowane przez osoby nieuprawnione.
- **Autentyczność** – możliwość potwierdzenia tożsamości nadawcy wiadomości lub źródła danych.
- **Niezaprzeczalność** – nadawca nie może zaprzeczyć wysłaniu wiadomości, co jest istotne np. w podpisach cyfrowych.

### 1.2 Podstawowe typy kryptografii

- **Kryptografia symetryczna** (jednokluczowa):
  - Wykorzystuje jeden klucz do szyfrowania i deszyfrowania danych, co oznacza, że nadawca i odbiorca muszą posiadać ten sam klucz.

- Jest bardzo wydajna i stosowana m.in. w algorytmach takich jak **DES**, czy **AES**.
- Wadą jest konieczność bezpiecznej wymiany klucza między stronami.
- **Kryptografia asymetryczna** (dwukluczowa):
  - Wykorzystuje dwa różne, ale matematycznie powiązane klucze: **klucz publiczny** (do szyfrowania) i **klucz prywatny** (do deszyfrowania).
  - Klucz publiczny może być udostępniony każdemu, natomiast klucz prywatny jest trzymany w tajemnicy przez właściciela.
  - Używana m.in. w algorytmach **RSA**, **ECC** (krzywych eliptycznych).
  - Jest mniej wydajna od kryptografii symetrycznej, ale nie wymaga bezpiecznej wymiany kluczy.
- **Kryptografia hybrydowa**:
  - Łączy kryptografię symetryczną i asymetryczną, wykorzystując szyfrowanie asymetryczne do bezpiecznej wymiany klucza symetrycznego, a następnie symetryczne do szyfrowania danych.
  - Stosowana np. w protokołach SSL/TLS do bezpiecznych połączeń internetowych.

### 1.3 Podstawowe techniki kryptograficzne

- 1) **Szyfrowanie** – proces przekształcania tekstu jawnego (czytelnego) w zaszyfrowany, czyli tekst nieczytelny dla osób nieupoważnionych.
- 2) **Deszyfrowanie** – proces przywracania zaszyfrowanego tekstu do jego pierwotnej, czytelnej formy.
- 3) **Haszowanie** – przekształcenie danych dowolnej długości w stałej długości skrót (hash), co pozwala sprawdzić integralność danych (algorytmy haszujące to m.in. SHA-256, MD5).
- 4) **Podpis cyfrowy** – technika kryptograficzna, która umożliwia potwierdzenie autentyczności i integralności dokumentu lub wiadomości.

## 1.4 Przykłady zastosowań kryptografii

Kryptografia jest szeroko stosowana w różnych dziedzinach życia codziennego, takich jak:

- **Bankowość internetowa** i transakcje online (zapewnienie bezpieczeństwa przelewów).
- **Komunikacja internetowa** (np. szyfrowane wiadomości w aplikacjach typu WhatsApp czy Signal).
- **Podpisy cyfrowe** stosowane w e-dokumentach i komunikacji e-mail.
- **Bezpieczeństwo Wi-Fi** (protokoły WPA, WPA2, WPA3 do zabezpieczania sieci bezprzewodowych).
- **Autoryzacja i uwierzytelnianie** użytkowników w serwisach internetowych i aplikacjach.

## 2. Cel i zakres pracy

**Celem** pracy jest stworzenie prostej aplikacji webowej w technologii Blazor WebAssembly umożliwiającej użytkownikom przetestowanie różnych algorytmów szyfrowania. Aplikacja będzie dostarczać intuicyjny interfejs, pozwalający na wpisywanie danych, wybór kluczy oraz – w przypadku niektórych metod – możliwość przesłania pliku do zaszyfrowania. Narzędzie ma na celu wprowadzenie użytkowników do zagadnień kryptografii, demonstrując różne podejścia do szyfrowania i deszyfrowania danych.

Aplikacja pozwoli użytkownikowi wybrać spośród dostępnych algorytmów szyfrowania, zaszyfrować i odszyfrować dane przy użyciu wybranego algorytmu oraz zrozumieć podstawowe różnice między poszczególnymi typami szyfrowania.

**Zakres pracy** obejmuje:

- 1) Opracowanie struktury aplikacji w **Blazor WebAssembly**:
  - a) Stworzenie prostego interfejsu użytkownika z **menu**, które pozwala wybrać jeden z dostępnych algorytmów szyfrowania.
  - b) Implementacja formularzy umożliwiających **wprowadzanie danych, podanie klucza szyfrującego** lub **załadowanie pliku** do szyfrowania.
- 2) Implementacja podstawowych algorytmów szyfrowania:
  - a) **Szyfry klasyczne**: np. szyfr polialfabetyczny (Gronsfelda).
  - b) **Szyfrowanie transpozycyjne**: proste algorytmy, które zmieniają pozycję znaków.
  - c) **Szyfrowanie symetryczne**: implementacja podstawowych algorytmów szyfrowania blokowego (DES i AES) oraz strumieniowego.
- 3) Interaktywne **testowanie** algorytmów szyfrowania:
  - a) Każdy moduł umożliwi **wpisanie danych lub wczytanie pliku**, podanie klucza szyfrującego oraz wykonanie operacji szyfrowania i deszyfrowania.
  - b) Wyświetlanie **wyników** szyfrowania w czytelnej formie, z możliwością skopiowania.

- 4) Weryfikacja poprawności działania algorytmów, czyli testowanie algorytmów w celu sprawdzenia poprawności procesu szyfrowania i deszyfrowania, aby wyniki były zgodne z teoretycznymi wzorcami.
- 5) Dokumentacja projektu i wnioski, czyli opis struktury i funkcji aplikacji, wyjaśnienie działania poszczególnych algorytmów, opis interfejsu użytkownika oraz przedstawienie obszarów do dalszego rozwoju aplikacji.

### 3. Aplikacja

Link do github: [https://github.com/Aro3224/Projekt\\_Kryptografia](https://github.com/Aro3224/Projekt_Kryptografia)

#### 3.1 UI

Poniższy zrzut ekranu przedstawia fragment menu głównego aplikacji, umożliwiający wybór spośród różnych typów algorytmów szyfrowania.



Rysunek 1. Fragment nawigacji pomiędzy różnymi typami szyfrowania

Poniższy zrzut ekranu przedstawia fragment menu głównego aplikacji, umożliwiający wybór spośród różnych typów algorytmów szyfrowania.

```

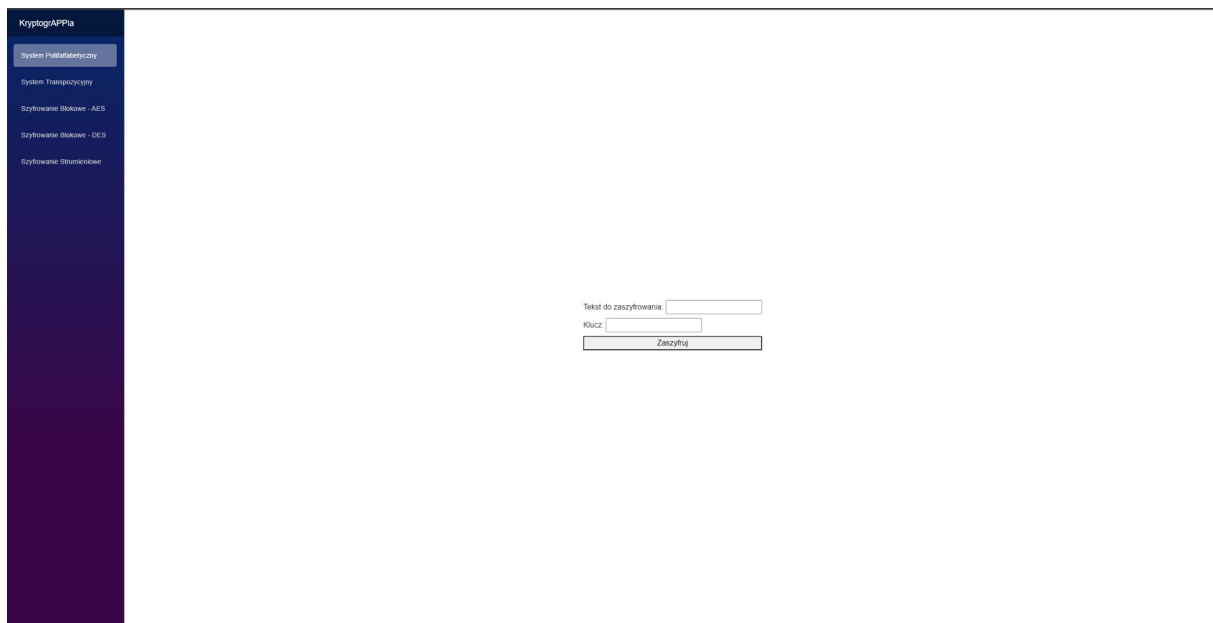
<div class="top-row ps-3 navbar navbar-dark">
  <div class="container-fluid">
    <a class="navbar-brand" href="">KryptogrAPPia</a>
    <button title="Navigation menu" class="navbar-toggler" @onclick="ToggleNavMenu">
      <span class="navbar-toggler-icon"></span>
    </button>
  </div>
</div>

<div class="@NavMenuCssClass nav-scrollable" @onclick="ToggleNavMenu">
  <nav class="flex-column">
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
        <span aria-hidden="true"></span> System Polifalfabetyczny
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="transpozycja">
        <span aria-hidden="true"></span> System Transpozycyjny
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="blokowe_AES">
        <span aria-hidden="true"></span> Szyfrowanie Blokowe - AES
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="blokowe_DES">
        <span aria-hidden="true"></span> Szyfrowanie Blokowe - DES
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="strumieniowe">
        <span aria-hidden="true"></span> Szyfrowanie Strumieniowe
      </NavLink>
    </div>
  </nav>
</div>

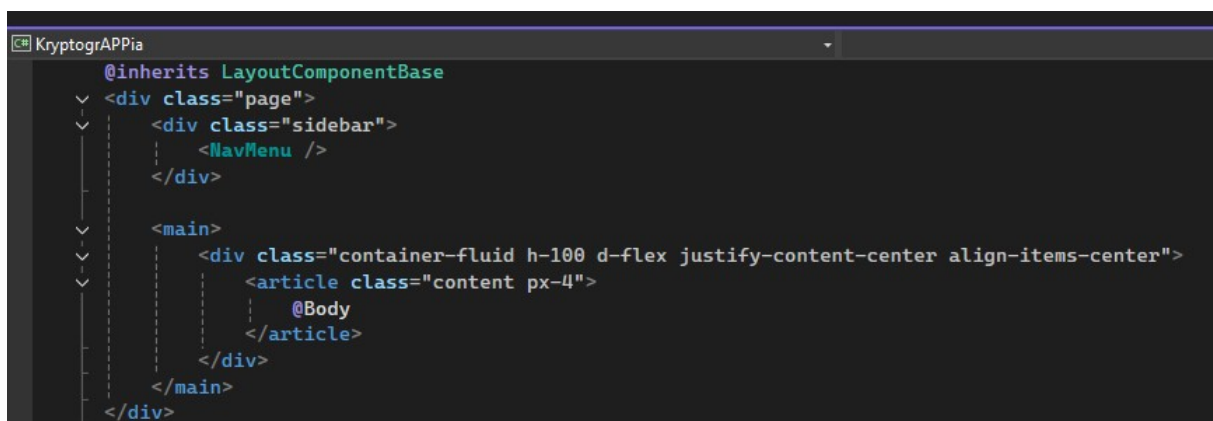
```

Rysunek 2. Zrzut ekranu przedstawiający implementację menu nawigacji

Poniższe zrzuty ekranu przedstawiają implementację głównego layoutu aplikacji, który ułatwia tworzenie funkcjonalności poprzez eliminację potrzeby wielokrotnego powtarzania kodu odpowiedzialnego za ustawianie elementów HTML. Dzięki zastosowanemu układowi, wszelkie zmiany w wyglądzie layoutu można wprowadzać w jednym miejscu, co sprawia, że nie ma potrzeby modyfikowania układu w każdej kategorii szyfrowania z osobna. Takie podejście zapewnia większą spójność i ułatwia zarządzanie interfejsem aplikacji.



Rysunek 3. Strona główna aplikacji



Rysunek 4. Implementacja głównego layoutu aplikacji

## 3.2 Szyfr polialfabetyczny

Szyfr polialfabetyczny to rodzaj algorytmu szyfrowania, w którym do zamiany liter w wiadomości stosuje się różne zestawy znaków (alfabety), w zależności od pozycji liter w tekście. Celem tego podejścia jest zwiększenie bezpieczeństwa szyfrowania poprzez wprowadzenie większej zmienności w przypisaniu liter, co utrudnia złamanie szyfru przy użyciu klasycznych metod, takich jak analiza częstotliwości.

Najbardziej znanym przykładem szyfru polialfabetycznego jest szyfr Vigenère'a, w którym do szyfrowania wykorzystuje się klucz – słowo lub frazę, która okre-



śla, które litery alfabetu będą stosowane do każdej litery w tekście. Klucz jest powtarzany, aby dopasować jego długość do długości wiadomości, a każda litera w wiadomości jest szyfrowana przy użyciu odpowiedniej litery klucza.

W szyfrze Vigenère'a dla każdej litery tekstu jawnego wybierany jest inny alfabet (lub przesunięcie) zgodnie z literą w kluczu. Dzięki temu każda litera tekstu jest szyfrowana innym przesunięciem, co znacząco utrudnia analizę częstotliwościową, która jest skuteczną metodą łamania prostych szyfrów monoalfabetycznych, takich jak szyfr Cezara.

**Szyfr Gronsfelda** to wariant szyfru polialfabetycznego, w którym do szyfrowania tekstu wykorzystuje się liczby zamiast liter. Jest to metoda, która działa podobnie do szyfru Vigenère'a, ale klucz składa się z cyfr, a każda cyfra w kluczu określa liczbę przesunięcia dla odpowiadającej litery w tekście.

Zasada działania szyfru Gronsfelda:

- Klucz składa się z cyfr (np. 3 1 4 1), które określają liczbę przesunięcia dla liter w tekście.
- Każda cyfra w kluczu odpowiada liczbom przesunięć, które są stosowane do liter w tekście jawnym.
- Klucz jest powtarzany w razie potrzeby, aby dopasować go do długości tekstu.
- Aby zaszyfrować tekst, każda litera w wiadomości jest przesuwana o liczbę pozycji w alfabecie równą cyfrze z klucza. Jeśli przesunięcie przekroczy literę Z, zaczyna się od początku alfabetu.

**Przykład:**

Tekst: Ala ma kota

Klucz: 2137

**Szyfrowanie:**

- 1) A (przesunięcie o 2) → C

- 2) L (przesunięcie o 1)  $\rightarrow$  M
- 3) A (przesunięcie o 3)  $\rightarrow$  D
- 4) M (przesunięcie o 7)  $\rightarrow$  T
- 5) A (przesunięcie o 2)  $\rightarrow$  C
- 6) K (przesunięcie o 1)  $\rightarrow$  L
- 7) O (przesunięcie o 3)  $\rightarrow$  R
- 8) T (przesunięcie o 7)  $\rightarrow$  A
- 9) A (przesunięcie o 2)  $\rightarrow$  C

**Wynik:** "Cmd t clrac"

Funkcja GronsfieldCipher implementuje szyfr Gronsfelda, który jest wariantem szyfru Cezara, ale zamiast stałego przesunięcia używa klucza, składającego się z cyfr. Działa na tekście, przesuwając litery o wartość odpowiadającą cyfrom w kluczu. Funkcja przyjmuje trzy argumenty: tekst do zaszyfrowania lub odszyfrowania (text), klucz szyfrujący (cipherKey) i wartość logiczną (encrypt), która określa, czy funkcja ma szyfrować (gdy encrypt jest true) czy odszyfrowywać (gdy encrypt jest false). Funkcja iteruje przez każdy znak w tekście i jeśli jest literą, oblicza przesunięcie na podstawie cyfry w kluczu, cyklicznie przechodząc przez jego cyfry. Jeśli odszyfrowuje, przesunięcie jest obliczane w przeciwnym kierunku, czyli odejmuje wartość klucza od 26. Małe i duże litery są przesuwane w obrębie odpowiedniego alfabetu, a znaki nieliterowe (np. spacje, przecinki) pozostają niezmienione. Na końcu funkcja zwraca zaszyfrowany lub odszyfrowany tekst, w zależności od wartości argumentu encrypt.

```

//Implementacja szyfru Gronsfelda
2 references
private string GronsfeldCipher(string text, string cipherKey, bool encrypt)
{
    var result = new StringBuilder();
    var keyIndex = 0;
    var keyLength = cipherKey.Length;

    foreach (var ch in text)
    {
        //Przesuwanie tylko liter
        if (char.IsLetter(ch))
        {
            //Określenie przesunięcia na podstawie cyfry w kluczu
            int shift = int.Parse(cipherKey[keyIndex % keyLength].ToString());

            //Jeśli odszyfrowujemy, to używamy przesunięcia w przeciwną stronę
            if (!encrypt)
            {
                shift = 26 - shift; //Odejmujemy przesunięcie zamiast dodawać
            }

            //Przesunięcie dla małych liter
            if (char.IsLower(ch))
            {
                result.Append((char) (((ch - 'a') + shift) % 26) + 'a');
            }
            //Przesunięcie dla dużych liter
            else if (char.IsUpper(ch))
            {
                result.Append((char) (((ch - 'A') + shift) % 26) + 'A');
            }

            keyIndex++; //Przechodzimy do kolejnej cyfry w kluczu
        }
        else
        {
            result.Append(ch); //Dodajemy inne znaki bez zmian (np. spacje, przecinki)
        }
    }

    return result.ToString();
}

```

Rysunek 5. Implementacja szyfru Gronsfelda w języku C#

# Szyfr Gronsfelda

Tekst do zaszyfrowania:

Klucz:

Zaszyfrowany tekst: Cmd tc lrac

Odszyfrowany tekst: Ala ma kota

Rysunek 6. Test implementacji

## 3.3 Szyfr transpozycyjny

Szyfr transpozycyjny to rodzaj szyfru, w którym zmienia się kolejność liter w wiadomości, ale same litery pozostają niezmienione. Zamiast zastępować litery innymi znakami (jak w szyfrze podstawieniowym), w szyfrze transpozycyjnym litery w tekście są permutowane (przemieszczane) w określony sposób zgodnie z ustaloną zasadą. Celem jest zamiana oryginalnej wiadomości w jej zaszyfrowaną formę, gdzie struktura tekstu, choć wciąż oparta na tych samych znakach, jest trudniejsza do odczytania bez znajomości klucza, który określa sposób permutacji.

Szyfr transpozycyjny może przyjmować różne formy, takie jak szyfr kolumnowy czy szyfr permutacyjny. W szyfrze kolumnowym, na przykład, tekst jest zapisany w tabeli o określonej liczbie kolumn, a następnie odczytany w innej kolejności, co zmienia rozmieszczenie liter. Ważnym aspektem szyfru transpozycyjnego jest to, że każda litera oryginalnego tekstu pojawia się w zaszyfrowanej wiadomości, ale w zmienionej kolejności, a nie poprzez zmianę samej litery.

Funkcja przedstawiona poniżej na początku tworzy macierz z tekstu na podstawie liczby kolumn. Następnie oblicza liczbę wierszy macierzy na podstawie długości tekstu i liczby kolumn (za pomocą funkcji `Math.Ceiling()`). Potem inicjalizuje tablicę dwuwymiarową (`char[][]`), która reprezentuje macierz. Na końcu wypełnia macierz

literami z tekstu, uzupełniając puste miejsca pustymi znakami ('\0'), jeśli tekst nie wypełnia dokładnie całej macierzy.

```
1 reference
private char[][] GenerateMatrix(string text, int columns)
{
    int rows = (int)Math.Ceiling((double)text.Length / columns);
    var matrix = new char[rows][];
    int charIndex = 0;

    for (int i = 0; i < rows; i++)
    {
        matrix[i] = new char[columns];
        for (int j = 0; j < columns; j++)
        {
            matrix[i][j] = charIndex < text.Length ? text[charIndex++] : '\0';
        }
    }
    return matrix;
}
```

Rysunek 7. Funkcja generująca macierz

Funkcja **TransposeMatrix** wykonuje transpozycję macierzy, czyli przekształca macierz w tekst, stosując różne metody w zależności od tego, czy szyfrujemy, czy odszyfrowujemy. Jeśli szyfrujemy (encrypt = true), funkcja iteruje przez kolumny, a następnie przez wiersze, tworząc ciąg zaszyfrowanego tekstu. Jeśli odszyfrowujemy (encrypt = false), funkcja iteruje przez wiersze, a następnie przez kolumny, tworząc odszyfrowany tekst.

```

2 references
private string TransposeMatrix(char[][] matrix, int columns, bool encrypt)
{
    int rows = matrix.Length;
    string result = "";

    if (encrypt)
    {
        for (int j = 0; j < columns; j++)
        {
            for (int i = 0; i < rows; i++)
            {
                if (matrix[i][j] != '\0')
                {
                    result += matrix[i][j];
                }
            }
        }
    }
    else
    {
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                if (matrix[i][j] != '\0')
                {
                    result += matrix[i][j];
                }
            }
        }
    }

    return result;
}

```

Rysunek 8. Funkcja przekształcająca macierz w tekst

Funkcja **GenerateDecryptionMatrix** tworzy macierz do deszyfrowania tekstu szyfrowanego szyfrem transpozycyjnym. Najpierw oblicza liczbę wierszy w macierzy oraz sposób rozdzielania znaków na kolumny (dzięki różnicy w wysokości kolumn). Następnie dzieli zaszyfrowany tekst na kolumny, umieszczając znaki w odpowiednich miejscach macierzy. Na końcu zależności od tego, czy kolumna jest pełna, czy nie, wypełnia odpowiednie wiersze macierzy.

```

1 reference
private char[][] GenerateDecryptionMatrix(string encryptedText, int columns)
{
    int rows = (int)Math.Ceiling((double)encryptedText.Length / columns);
    var matrix = new char[rows][];
    int charIndex = 0;

    int fullColumns = encryptedText.Length % columns;
    int fullColumnHeight = rows;
    int shortColumnHeight = rows - 1;

    for (int i = 0; i < rows; i++)
    {
        matrix[i] = new char[columns];
    }

    for (int col = 0; col < columns; col++)
    {
        int currentHeight = (col < fullColumns) ? fullColumnHeight : shortColumnHeight;
        for (int row = 0; row < currentHeight; row++)
        {
            if (charIndex < encryptedText.Length)
            {
                matrix[row][col] = encryptedText[charIndex++];
            }
        }
    }

    return matrix;
}

```

Rysunek 9. Funkcja tworząca macierz do deszyfrowania

Tekst do zaszyfrowania:

Liczba kolumn (dla transpozycji):

Macierz szyfrująca:

a	l	a		m
a		k	o	t
a				

Zaszyfrowany tekst: aaal ak omt

Odszyfrowany tekst: ala ma kota

Rysunek 10. Implementacja szyfru transpozycyjnego

### 3.4 Szyfrowanie blokowe – AES

**AES (Advanced Encryption Standard)** to algorytm kryptograficzny, który przekształca dane w sposób uniemożliwiający ich odczytanie przez osoby nieuprawnione. Jest to jeden z najbardziej rozpowszechnionych standardów szyfrowania i został opracowany jako następcą algorytmu DES (Data Encryption Standard). AES jest obecnie uważany za bezpieczny i jest szeroko stosowany w różnych dziedzinach, takich jak bezpieczeństwo danych w sieciach komputerowych, ochrona plików czy szyfrowanie dysków.

W szyfrowaniu blokowym za pomocą algorytmu AES dane są dzielone na bloki o stałej długości (128 bitów, czyli 16 bajtów) oraz każdy blok jest szyfrowany osobno przy użyciu tego samego klucza. AES obsługuje klucze o długości 128, 192 lub 256 bitów, co zapewnia różne poziomy bezpieczeństwa:



- AES-128: klucz ma 16 bajtów
- AES-192: klucz ma 24 bajty
- AES-256: klucz ma 32 bajty

Proces szyfrowania składa się z serii rund, gdzie dane są przekształcane w sposób zwiększający bezpieczeństwo. Liczba rund zależy od długości klucza:

- 10 rund dla AES-128
- 12 rund dla AES-192
- 14 rund dla AES-256

Działanie algorytmu AES:

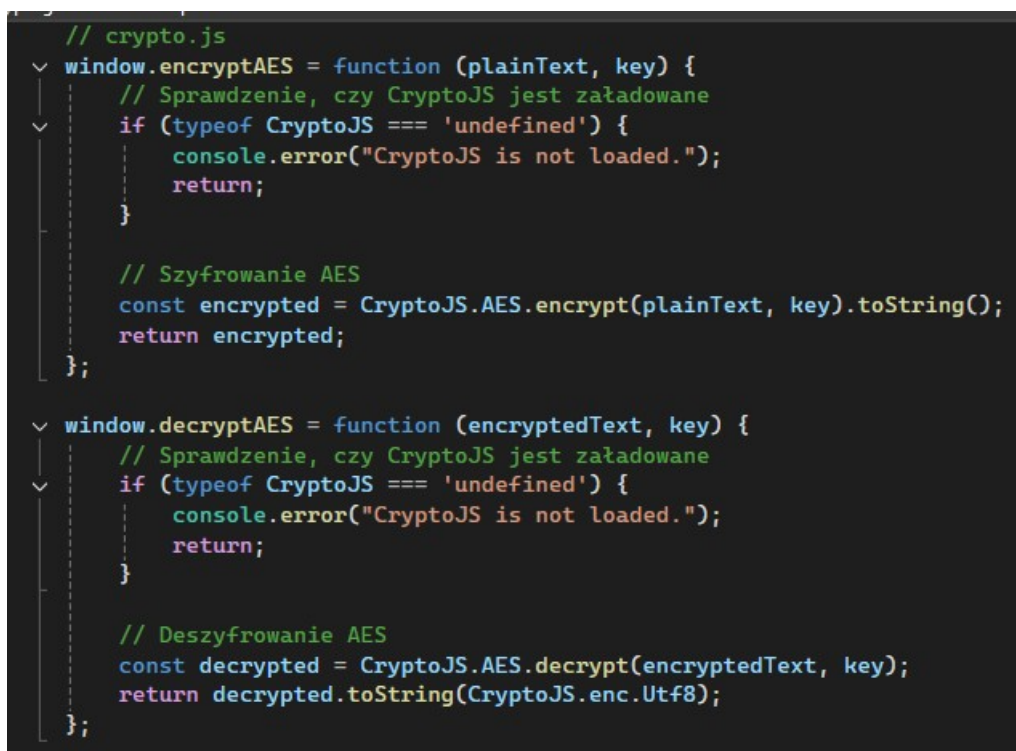
- Długość każdego bloku wynosi 128 bitów. Jeśli dane są krótsze, są odpowiednio uzupełniane (np. przez wypełnienie zerami).
- Użytkownik dostarcza klucz, który jest przekształcany w klucze rundowe przy użyciu tzw. key schedule.
- Dane w bloku przechodzą przez kolejne etapy:
  - Dodawanie klucza (AddRoundKey) – blok danych jest mieszany z kluczem rundowym.
  - Substytucja (SubBytes) – każdy bajt w bloku jest zastępowany innym bajtem na podstawie tabeli (S-Box).
  - Przesunięcia wierszy (ShiftRows) – wiersze bloku są przesuwane o określoną liczbę pozycji.
  - Mieszanie kolumn (MixColumns) – kolumny bloku są przekształcane przy użyciu mnożenia macierzowego.
- Etapy te są powtarzane przez ustaloną liczbę rund, aż do uzyskania zaszyfrowanego bloku.

## Zastosowanie AES:

- Szyfrowanie komunikacji internetowej (np. HTTPS, VPN)
- Szyfrowanie danych na dyskach twardych (np. BitLocker, FileVault)
- Szyfrowanie plików w aplikacjach (np. aplikacje bankowe, komunikatory)
- Ochrona transmisji danych w sieciach bezprzewodowych (np. WPA2)

Podsumowując, AES to nowoczesny, bardzo bezpieczny i uniwersalny algorytm szyfrowania blokowego. Dzięki swojej skuteczności i wydajności jest standardem w ochronie danych na całym świecie.

## Implementacja:



```
// crypto.js
window.encryptAES = function (plainText, key) {
  // Sprawdzenie, czy CryptoJS jest załadowane
  if (typeof CryptoJS === 'undefined') {
    console.error("CryptoJS is not loaded.");
    return;
  }

  // Szyfrowanie AES
  const encrypted = CryptoJS.AES.encrypt(plainText, key).toString();
  return encrypted;
};

window.decryptAES = function (encryptedText, key) {
  // Sprawdzenie, czy CryptoJS jest załadowane
  if (typeof CryptoJS === 'undefined') {
    console.error("CryptoJS is not loaded.");
    return;
  }

  // Deszyfrowanie AES
  const decrypted = CryptoJS.AES.decrypt(encryptedText, key);
  return decrypted.toString(CryptoJS.enc.Utf8);
};
```

Rysunek 11. Użycie biblioteki JS do szyfrowania i deszyfrowania za pomocą AES

```

//Funkcja odpowiedzialna za szyfrowanie tekstu.
1 reference
private async Task EncryptText()
{
    try
    {
        ErrorMessage = string.Empty;

        //Walidacja długości klucza szyfrowania
        if (string.IsNullOrEmpty(EncryptionKey) || EncryptionKey.Length < 8)
        {
            throw new Exception("Klucz szyfrowania musi mieć co najmniej 8 znaków.");
        }

        //Szyfrowanie tekstu
        EncryptedText = await JS.InvokeAsync<string>("encryptAES", InputText, EncryptionKey);
        Console.WriteLine($"Zaszyfrowany tekst: {EncryptedText}");

        isEncrypted = true;
        isDecrypting = false;
    }
    catch (Exception ex)
    {
        ErrorMessage = ex.Message;
    }
}

```

Rysunek 12. Funkcja wywołująca szyfrowanie w AES

```

//Funkcja odpowiedzialna za deszyfrowanie tekstu.
1 reference
private async Task DecryptText()
{
    try
    {
        ErrorMessage = string.Empty;

        //Walidacja długości klucza deszyfrowania
        if (string.IsNullOrEmpty(EncryptionKey) || EncryptionKey.Length < 8)
        {
            throw new Exception("Klucz deszyfrowania musi mieć co najmniej 8 znaków.");
        }

        isDecrypting = true;

        //Odszyfrowanie tekstu
        DecryptedText = await JS.InvokeAsync<string>("decryptAES", EncryptedText, EncryptionKey);
        Console.WriteLine($"Odszyfrowany tekst: {DecryptedText}");

        isEncrypted = false;
        isDecrypting = true;
    }
    catch (Exception ex)
    {
        ErrorMessage = ex.Message;
        isDecrypting = false;
    }
}

```

Rysunek 13. Funkcja wywołująca deszyfrowanie w AES

```

//Funkcja odpowiedzialna za załadowanie pliku tekstowego do zaszyfrowania.
1 reference
private async Task LoadFile(InputFileChangeEventArgs e)
{
    var file = e.File;
    if (file != null)
    {
        var buffer = new byte[file.Size];
        await file.OpenReadStream().ReadAsync(buffer);
        InputText = Encoding.UTF8.GetString(buffer);
    }
}

```

Rysunek 14. Wczytywanie pliku tekstowego

Test:

## Szyfrowanie AES

Tekst do zaszyfrowania:

Klucz szyfrowania:

Nie wybrano pliku

Zaszyfrowany tekst:

Odszyfrowany tekst: ala ma kota

Rysunek 15. Szyfrowanie tekstu w AES

## Szyfrowanie AES

Tekst do zaszyfrowania:

Klucz szyfrowania:

testaes.txt

Zaszyfrowany tekst:

Odszyfrowany tekst: to jest test szyfrowania pliku tekstowego z wykorzystaniem algorytmu aes

Rysunek 16. Szyfrowanie pliku tekstowego w AES

### 3.5 Szyfrowanie blokowe – DES

**DES (Data Encryption Standard)** to klasyczny algorytm szyfrowania blokowego, który został opracowany przez IBM w latach 70-tych XX wieku i przyjęty jako standard przez amerykańską agencję rządową NIST (National Institute of Standards and Technology) w 1977 roku. Szyfrowanie blokowe oznacza, że dane są przetwarzane w blokach o stałej długości, a DES dzieli dane na bloki po 64 bity i szyfruje każdy blok osobno.

DES używa bloków danych o długości 64 bitów (8 bajtów). Cały tekst do zaszyfrowania jest dzielony na bloki 64-bitowe, które są następnie szyfrowane. Klucz używany do szyfrowania w DES ma długość 56 bitów (choć klucz jest zazwyczaj reprezentowany jako 64 bity, ponieważ 8 bitów jest używanych do parzystości). To oznacza, że DES jest stosunkowo wrażliwy na ataki brute force, ponieważ przestrzeń możliwych kluczy jest ograniczona do  $2^{56}$  możliwych kombinacji.

DES jest algorytmem szyfrowania symetrycznego, co oznacza, że ten sam klucz jest używany zarówno do szyfrowania, jak i deszyfrowania danych. Proces szyfrowania składa się z 16 rund, w których stosowane są różne operacje matematyczne, w tym permutacje i operacje XOR, w celu przekształcenia danych w zaszyfrowany tekst.

## Działanie szyfrowania DES:

- Podział na bloki: Dane wejściowe są dzielone na 64-bitowe bloki. Jeśli długość danych nie jest wielokrotnością 64 bitów, algorytm może dodać wypełnienie (padding), aby uzyskać odpowiednią długość.
- Inicjalizacja: Na początku algorytm wykonuje permutację początkową, czyli zmienia pozycje bitów w danych.
- Szyfrowanie (16 rund): Algorytm przeprowadza 16 rund, w których dane są przekształcane poprzez operacje takie jak XOR, permutacje, oraz wykorzystanie 48-bitowych kluczy rundy (wyciąganych z głównego klucza).
- Permutacja końcowa: Po zakończeniu 16 rund, dane przechodzą przez permutację końcową, która tworzy końcowy zaszyfrowany tekst.
- Deszyfrowanie: Proces deszyfrowania jest bardzo podobny do szyfrowania, z tą różnicą, że klucze rundy są używane w odwrotnej kolejności. Proces deszyfrowania przebiega w taki sposób, że każdy blok zaszyfrowany za pomocą DES można odszyfrować tylko przy użyciu tego samego klucza.

Mimo że DES był szeroko stosowany przez wiele lat, jego bezpieczeństwo zostało poważnie naruszone przez rozwój mocy obliczeniowej komputerów. Z racji 56-bitowej długości klucza, DES jest podatny na atak siłowy (brute force), w którym atakujący próbuje wszystkich możliwych kluczy, aby złamać szyfr. W latach 90-tych XX wieku badania i testy wykazały, że DES jest podatny na tego typu atak, dlatego z czasem został zastąpiony przez bardziej bezpieczne algorytmy, takie jak AES (Advanced Encryption Standard).

Podsumowując, szyfrowanie blokowe DES jest klasycznym algorytmem, który służył jako standard w zakresie szyfrowania danych przez kilka dziesięcioleci. Mimo swojej historycznej roli, dziś jest uznawany za niebezpieczny ze względu na krótki klucz i wrażliwość na ataki brute force. Współczesne zastosowania kryptografii wykorzystują bardziej zaawansowane i bezpieczniejsze algorytmy, takie jak AES.

Implementacja:

```

window.encryptDES = function (plainText, key) {
    // Sprawdzenie, czy CryptoJS jest załadowane
    if (typeof CryptoJS === 'undefined') {
        console.error("CryptoJS is not loaded.");
        return;
    }

    // Szyfrowanie DES
    const encrypted = CryptoJS.DES.encrypt(plainText, key).toString();
    return encrypted;
};

window.decryptDES = function (encryptedText, key) {
    // Sprawdzenie, czy CryptoJS jest załadowane
    if (typeof CryptoJS === 'undefined') {
        console.error("CryptoJS is not loaded.");
        return;
    }

    // Deszyfrowanie DES
    const decrypted = CryptoJS.DES.decrypt(encryptedText, key);
    return decrypted.toString(CryptoJS.enc.Utf8);
};

```

Rysunek 17. Użycie biblioteki JS do szyfrowania/deszyfrowania za pomocą DES

```

// Funkcja odpowiedzialna za szyfrowanie tekstu.
1 reference
private async Task EncryptText()
{
    try
    {
        ErrorMessage = string.Empty;

        // Walidacja długości klucza szyfrowania
        if (string.IsNullOrEmpty(EncryptionKey) || EncryptionKey.Length < 8)
        {
            throw new Exception("Klucz szyfrowania musi mieć co najmniej 8 znaków.");
        }

        // Szyfrowanie tekstu za pomocą DES
        EncryptedText = await JS.InvokeAsync<string>("encryptDES", InputText, EncryptionKey);
        Console.WriteLine($"Zaszyfrowany tekst: {EncryptedText}");

        isEncrypted = true;
        isDecrypting = false;
    }
    catch (Exception ex)
    {
        ErrorMessage = ex.Message;
    }
}

```

Rysunek 18. Algorytm używający funkcji JS do szyfrowania DES'em



```

// Funkcja odpowiedzialna za deszyfrowanie tekstu.
1 reference
private async Task DecryptText()
{
    try
    {
        ErrorMessage = string.Empty;

        // Walidacja długości klucza deszyfrowania
        if (string.IsNullOrEmpty(EncryptionKey) || EncryptionKey.Length < 8)
        {
            throw new Exception("Klucz deszyfrowania musi mieć co najmniej 8 znaków.");
        }

        isDecrypting = true;

        // Odszyfrowanie tekstu za pomocą DES
        DecryptedText = await JS.InvokeAsync<string>("decryptDES", EncryptedText, EncryptionKey);
        Console.WriteLine($"Odszyfrowany tekst: {DecryptedText}");

        isEncrypted = false;
        isDecrypting = false;
    }
    catch (Exception ex)
    {
        ErrorMessage = ex.Message;
        isDecrypting = false;
    }
}

```

Rysunek 19. Algorytm używający funkcji JS do deszyfrowania DES'em

Test:

## Szyfrowanie DES

Tekst do zaszyfrowania:

Klucz szyfrowania:

Wybierz plik

Zaszyfrowany tekst:

Odszyfrowany tekst: ala ma kota

Rysunek 20. Test szyfrowania DES'em



### 3.6 Szyfrowanie strumieniowe – AES

Szyfrowanie strumieniowe z użyciem AES to metoda szyfrowania danych, w której blok algorytmu AES jest używany w trybie strumieniowym, aby zaszyfrować dane w sposób ciągły, byte po byte, zamiast na całych blokach jak w tradycyjnych trybach blokowych.

Szyfrowanie strumieniowe jest metodą, w której dane są szyfrowane strumieniowo, tj. po jednym bicie lub bajcie, w przeciwieństwie do szyfrowania blokowego, które przetwarza dane w większych jednostkach (blokach). Szyfrowanie strumieniowe pozwala na szyfrowanie danych o zmiennej długości i może być bardziej elastyczne w niektórych scenariuszach.

Chociaż AES jest algorytmem szyfrowania blokowego, może być używany w trybach strumieniowych, które umożliwiają jego zastosowanie w szyfrowaniu strumieniowym. Jednym z najpopularniejszych trybów wykorzystywanych do tego celu jest CTR (Counter Mode), który zamienia AES w szyfr strumieniowy. W tym trybie AES działa jako generator strumienia klucza, który jest następnie XOR-owany z danymi do zaszyfrowania.

Działanie szyfrowania strumieniowego z użyciem AES:

- 1) Inicjalizacja (IV i klucz): Szyfrowanie strumieniowe przy użyciu AES wymaga klucza oraz wektora inicjalizacyjnego (IV), który jest używany w trybie CTR do generowania strumienia klucza.
- 2) Generowanie strumienia klucza: W trybie CTR, AES działa na liczniku (counter), który jest modyfikowany dla każdego bloku danych. Wektory inicjalizacyjne i licznik są szyfrowane przez algorytm AES, tworząc strumień klucza.
- 3) XOR z danymi: Wygenerowany strumień klucza (ze szyfrowania licznika) jest następnie XOR-owany z danymi, które mają zostać zaszyfrowane. XORowanie oznacza, że każda jednostka danych (bajt) zostaje połączona z odpowiednim bajtem strumienia klucza w taki sposób, że wynik jest szyfrogramem.
- 4) Deszyfrowanie: Proces deszyfrowania w trybie strumieniowym jest identyczny, ponieważ XOR z tym samym strumieniem klucza spowoduje przywrócenie oryginalnych danych.

Szyfrowanie strumieniowe z użyciem AES jest stosunkowo szybkie, elastyczne i nadaje się do dynamicznego szyfrowania danych. Dzięki użyciu trybu CTR, AES staje się szyfrem strumieniowym, co daje możliwość szyfrowania pojedynczych bajtów danych w sposób szybki i bez opóźnień, co jest przydatne w przypadku transmisji danych w czasie rzeczywistym, takich jak przesyłanie plików, streaming czy komunikacja w aplikacjach na żywo.

#### Implementacja:

Funkcja `encryptStream` przyjmuje trzy argumenty: tekst do zaszyfrowania (`plainText`), klucz szyfrowania (`key`) oraz wektor inicjalizacyjny (`iv`). Najpierw konwertuje klucz i wektor inicjalizacyjny na odpowiedni format (`CryptoJS.enc.Utf8.parse`), ponieważ `CryptoJS` wymaga, aby klucz i IV były w formacie `WordArray`. Następnie używa algorytmu AES do szyfrowania tekstu w trybie CTR, gdzie każda część tekstu jest szyfrowana poprzez dodanie strumienia generowanego przez licznik (który jest modyfikowany w trakcie operacji) do bloków danych. W trybie CTR nie stosuje się paddingu, dlatego parametr `padding: CryptoJS.pad.NoPadding` jest użyty. Zaszyfrowany tekst jest następnie konwertowany na ciąg znaków (w formacie `Base64`), który jest zwracany do aplikacji Blazor.

```
//Funkcja szyfrująca tekst w trybie CTR za pomocą AES
window.encryptStream = function (plainText, key, iv) {
    //Sprawdzenie, czy CryptoJS jest załadowane
    if (typeof CryptoJS === 'undefined') {
        console.error("CryptoJS is not loaded.");
        return;
    }

    //Konwertowanie klucza i wektora inicjalizacyjnego do formatu WordArray
    var keyHex = CryptoJS.enc.Utf8.parse(key);
    var ivHex = CryptoJS.enc.Utf8.parse(iv);

    //Szyfrowanie za pomocą AES w trybie CTR
    const encrypted = CryptoJS.AES.encrypt(plainText, keyHex, {
        iv: ivHex,
        mode: CryptoJS.mode.CTR,
        padding: CryptoJS.pad.NoPadding
    });

    return encrypted.toString(); //Zwracamy zaszyfrowany tekst w formacie Base64
};
```

Rysunek 21. Algorytm szyfrowania strumieniowego z użyciem AES'a i wektora inicjalizacyjnego

Funkcja `decryptStream` działa w sposób odwrotny. Przyjmuje trzy argumenty: zaszyfrowany tekst (`encryptedText`), klucz szyfrowania (`key`) oraz wektor inicjalizacyjny (`iv`). Tak jak w przypadku szyfrowania, klucz i IV są konwertowane na format `WordArray`. Następnie AES w trybie CTR jest używany do odszyfrowania danych. Strumień generowany przez licznik (który był użyty w czasie szyfrowania) jest odwrócony, a zaszyfrowane dane są odzyskiwane. Odszyfrowany tekst jest konwertowany na format UTF-8 i zwracany do aplikacji Blazor.

```
// Funkcja deszyfrująca tekst w trybie CTR za pomocą AES
window.decryptStream = function (encryptedText, key, iv) {
    // Sprawdzenie, czy CryptoJS jest załadowane
    if (typeof CryptoJS === 'undefined') {
        console.error("CryptoJS is not loaded.");
        return;
    }

    //Konwertowanie klucza i wektora inicjalizacyjnego do formatu WordArray
    var keyHex = CryptoJS.enc.Utf8.parse(key);
    var ivHex = CryptoJS.enc.Utf8.parse(iv);

    //Deszyfrowanie za pomocą AES w trybie CTR
    const decrypted = CryptoJS.AES.decrypt(encryptedText, keyHex, {
        iv: ivHex,
        mode: CryptoJS.mode.CTR,
        padding: CryptoJS.pad.NoPadding
    });

    //Zwracamy odszyfrowany tekst
    return decrypted.toString(CryptoJS.enc.Utf8);
};
```

Rysunek 22. Algorytm deszyfrowania strumieniowego z użyciem AES'a i wektora inicjalizacyjnego

### 3.7 Szyfrowanie strumieniowe – DES

Funkcja `encryptDESStream` w JavaScript służy do szyfrowania tekstu za pomocą algorytmu DES w trybie strumieniowym, używając klucza szyfrowania oraz wektora inicjalizacyjnego (IV). Najpierw konwertuje klucz szyfrowania i wektor inicjalizacyjny na odpowiedni format, czyli na tzw. obiekty hex w CryptoJS, ponieważ DES wymaga, aby te wartości były reprezentowane w postaci szeregów bajtów. Następnie, używając funkcji `CryptoJS.DES.encrypt()`, tekst jest szyfrowany przy zastosowaniu trybu CBC (Cipher Block Chaining), który jest trybem blokowym z wykorzystaniem wektora IV. Funkcja ta zwraca zaszyfrowany tekst w formacie Base64.

```

window.encryptDESStream = function (plainText, key, iv) {
    // Sprawdzenie, czy CryptoJS jest załadowane
    if (typeof CryptoJS === 'undefined') {
        console.error("CryptoJS is not loaded.");
        return;
    }

    //Konwersja klucza i wektora do odpowiednich formatów
    const keyHex = CryptoJS.enc.Utf8.parse(key);
    const ivHex = CryptoJS.enc.Utf8.parse(iv);

    // Szyfrowanie w trybie strumieniowym (ECB z IV)
    const encrypted = CryptoJS.DES.encrypt(plainText, keyHex, {
        iv: ivHex,
        mode: CryptoJS.mode.CBC,
        padding: CryptoJS.pad.Pkcs7
    });

    return encrypted.toString();
};

```

Rysunek 23. Szyfrowanie strumieniowe z wykorzystaniem algorytmu DES

Funkcja `decryptDESStream` działa odwrotnie. Odbiera zaszyfrowany tekst, klucz szyfrowania oraz wektor inicjalizacyjny. Tak jak w przypadku szyfrowania, klucz i wektor są najpierw konwertowane na format hex. Następnie, przy pomocy `CryptoJS.DES.decrypt()`, przeprowadza deszyfrowanie tekstu, używając tego samego algorytmu DES w trybie CBC z tymi samymi wartościami klucza i IV. Odszyfrowany wynik jest następnie konwertowany na tekst w formacie UTF-8, co umożliwia odczytanie oryginalnego tekstu.

```

window.decryptDESStream = function (encryptedText, key, iv) {
    //Sprawdzenie, czy CryptoJS jest załadowane
    if (typeof CryptoJS === 'undefined') {
        console.error("CryptoJS is not loaded.");
        return;
    }

    //Konwersja klucza i wektora do odpowiednich formatów
    const keyHex = CryptoJS.enc.Utf8.parse(key);
    const ivHex = CryptoJS.enc.Utf8.parse(iv);

    //Deszyfrowanie w trybie strumieniowym (ECB z IV)
    const decrypted = CryptoJS.DES.decrypt(encryptedText, keyHex, {
        iv: ivHex,
        mode: CryptoJS.mode.CBC,
        padding: CryptoJS.pad.Pkcs7
    });

    return decrypted.toString(CryptoJS.enc.Utf8);
};

```

Rysunek 24. Deszyfrowanie strumieniowe z wykorzystaniem algorytmu DES

Test

## Szyfrowanie Strumieniowe DES

Tekst do zaszyfrowania:

Klucz szyfrowania:

Wektor inicjalizacyjny (IV):

Nie wybrano pliku

Zaszyfrowany tekst:

Odszyfrowany tekst: Hello, World!

Rysunek 25. Test szyfrowania strumieniowego za pomocą DES'a z wektorem inicjalizacyjnym

### 3.8 Symulacja protokołu Diffiego-Hellmana

Protokół Diffiego-Hellmana (Diffie-Hellman Key Exchange) to jeden z pierwszych protokołów umożliwiających bezpieczną wymianę kluczy pomiędzy dwoma stronami. Został opracowany w 1976 roku przez Whitfielda Diffiego i Marlina Hellmana. Jego celem jest umożliwienie stronom wspólnej generacji klucza do szyfrowania za pomocą niezabezpieczonego kanału komunikacyjnego, co jest fundamentalnym krokiem w kontekście zapewnienia tajności w kryptografii.

Działanie protokołu:

Zakłada się, że obie strony znają pewne publiczne wartości: wielką liczbę pierwszą ( $p$ ) oraz generator grupy ( $g$ ). Generator grupy  $g$  to liczba, która może wytworzyć wszystkie liczby w grupie, kiedy podniesiona do potęgi dowolnej liczby mniejszej niż  $p$ .

1) Ustalenie klucza publicznego:

- a) Każda ze stron generuje swoją prywatną liczbę kluczową (np.  $a$  dla Alice i  $b$  dla Boba), która jest przypadkową liczbą w zakresie od 1 do  $p-2$ .
- b) Alice oblicza swój klucz publiczny  $A$  jako  $A = g^a \bmod p$ .
- c) Bob oblicza swój klucz publiczny  $B$  jako  $B = g^b \bmod p$ .

2) Wymiana kluczy publicznych:

- a) Alice wysyła wartość  $A$  do Boba, a Bob wysyła wartość  $B$  do Alice. Te wymiany mogą odbywać się przez niebezpieczny, niechroniony kanał.

3) Krok 3: Wspólna generacja klucza:

- a) Obie strony otrzymują klucze publiczne drugiej strony. Alice oblicza wspólny klucz jako  $K = B^a \bmod p$ , a Bob oblicza  $K = A^b \bmod p$ .

Implementacja:

Funkcja `GenerateKeys` na początku weryfikuje, czy podana liczba  $p$  jest liczbą pierwszą. Jeśli nie jest, ustawia odpowiedni komunikat błędu. W przeciwnym wypadku generuje ona losowe klucze prywatne. Dla strony  $A$  i  $B$  generowany jest losowy



klucz prywatny (PrivateKeyA i PrivateKeyB) w zakresie od 1 do  $p-1$ . Na końcu obliczane zostają klucze publiczne używając wzoru  $g^{\text{privateKey}} \bmod p$ .

```
private void GenerateKeys()
{
    try
    {
        ErrorMessage = string.Empty;

        if (!IsPrime(PrimeInput))
        {
            ErrorMessage = "Podana liczba nie jest liczbą pierwszą.";
            return;
        }

        var p = int.Parse(PrimeInput);
        var g = int.Parse(BaseInput);

        PrivateKeyA = new Random().Next(1, p - 1).ToString();
        PublicKeyA = CalculatePublicKey(g, int.Parse(PrivateKeyA), p);

        PrivateKeyB = new Random().Next(1, p - 1).ToString();
        PublicKeyB = CalculatePublicKey(g, int.Parse(PrivateKeyB), p);
    }
    catch (Exception ex)
    {
        ErrorMessage = $"Błąd: {ex.Message}";
    }
}
```

Rysunek 26. Funkcja obliczająca klucze publiczne

Funkcja ExchangeKeys sprawdza, czy dla obu stron (PublicKeyA i PublicKeyB) są już wygenerowane. Jeśli nie, ustawia odpowiedni komunikat błędu. Dla strony A, używając wzoru  $\text{publicKeyB}^{\text{privateKeyA}} \bmod p$ , oblicza wspólny sekret (SharedSecretA) (dla strony B robi to samo ze wzorem  $\text{publicKeyA}^{\text{privateKeyB}} \bmod p$ ). Wynikowe wspólne sekrety dla obu stron są przechowywane w zmiennych SharedSecretA i SharedSecretB.

```

private void ExchangeKeys()
{
    try
    {
        ErrorMessage = string.Empty;

        if (string.IsNullOrEmpty(PublicKeyA) || string.IsNullOrEmpty(PublicKeyB))
        {
            ErrorMessage = "Najpierw wygeneruj klucze publiczne obu stron.";
            return;
        }

        var p = int.Parse(PrimeInput);

        SharedSecretA = CalculateSharedSecret(int.Parse(PublicKeyB), int.Parse(PrivateKeyA), p);
        SharedSecretB = CalculateSharedSecret(int.Parse(PublicKeyA), int.Parse(PrivateKeyB), p);
    }
    catch (Exception ex)
    {
        ErrorMessage = $"Błąd: {ex.Message}";
    }
}

```

Rysunek 27. Funkcja obliczająca wspólny sekret

### 3.9 Szyfrowanie RSA

Szyfrowanie RSA to jeden z najważniejszych algorytmów kryptografii asymetrycznej, opracowany w 1977 roku przez Ronalda Rivesta, Adiego Shamira i Leonarda Adlemana, od których nazwisk pochodzi jego nazwa. RSA umożliwia szyfrowanie, deszyfrowanie oraz podpisy cyfrowe, wykorzystując parę kluczy: publiczny do szyfrowania i weryfikacji podpisów oraz prywatny do deszyfrowania i tworzenia podpisów. Głównym założeniem RSA jest oparcie bezpieczeństwa na trudności faktoryzacji dużych liczb na czynniki pierwsze.

Działanie algorytmu RSA opiera się na trzech głównych krokach:

- 1) Generowanie kluczy:
  - a) Wybiera się dwie duże liczby pierwsze  $p$  i  $q$ , które są podstawą obliczeń.
  - b) Oblicza się  $n = p \cdot q$ , gdzie  $n$  staje się częścią klucza publicznego i prywatnego.
  - c) Oblicza się funkcję Eulera  $\phi(n) = (p-1) * (q-1)$ , która służy do ustalenia właściwości matematycznych klucza.



- d) Wybiera się liczbę  $e$  (publiczny wykładnik), która musi być względnie pierwsza względem  $\phi(n)$  (tj. ich największy wspólny dzielnik wynosi 1).
- e) Oblicza się prywatny wykładnik  $d$  jako odwrotność modularną  $e$  względem  $\phi(n)$ , tak że  $e \cdot d \equiv 1 \pmod{\phi(n)}$ .

## 2) Szyfrowanie:

- a) Dane (reprezentowane jako liczba  $M$ ) są szyfrowane przy użyciu klucza publicznego  $(e, n)$  za pomocą wzoru  $C = M^e \pmod{n}$ .
- b) Wynik  $C$  to szyfrogram, który można przesłać odbiorcy.

## 3) Deszyfrowanie:

- a) Odbiorca, używając klucza prywatnego  $(d, n)$  odszyfrowuje szyfrogram za pomocą wzoru  $M = C^d \pmod{n}$ .
- b) Wynik  $M$  to oryginalna wiadomość.

Podsumowując, RSA znajduje szerokie zastosowanie w systemach bezpieczeństwa, takich jak HTTPS (SSL/TLS), podpisy cyfrowe i kryptografia klucza publicznego w ogólności.

## Implementacja:

Funkcja **generateRSAKeys** generuje parę kluczy RSA (publiczny i prywatny) o długości 2048 bitów i zapisuje je w formacie PEM. Klucze RSA są wykorzystywane do szyfrowania i deszyfrowania danych. Funkcja zaczyna od sprawdzenia, czy biblioteka Forge jest załadowana, sprawdzając, czy zmienna `forge` jest zdefiniowana. Jeśli nie, wypisuje błąd w konsoli. Funkcja `forge.pki.rsa.generateKeyPair` generuje parę kluczy RSA. Długość klucza wynosi 2048 bitów, a wykorzystywana jest publiczna wartość  $e$ , która wynosi `0x10001` (standardowa wartość dla RSA). Po wygenerowaniu kluczy, funkcja wykorzystuje metodę `forge.pki.publicKeyToPem` i `forge.pki.privateKeyToPem` do konwersji kluczy do formatu PEM, który jest szeroko stosowany w kryptografii.

```

window.generateRSAKeys = function () {
  if (typeof forge === 'undefined') {
    console.error("Forge library is not loaded.");
    return;
  }

  const keypair = forge.pki.rsa.generateKeyPair({ bits: 2048, e: 0x10001 });
  return {
    PublicKey: forge.pki.publicKeyToPem(keypair.publicKey),
    PrivateKey: forge.pki.privateKeyToPem(keypair.privateKey),
  };
};

```

Rysunek 28. Funkcja w JS generująca klucze RSA

Funkcja **encryptRSA** jest odpowiedzialna za szyfrowanie wiadomości przy użyciu klucza publicznego RSA. Jak poprzednio, funkcja sprawdza, czy biblioteka Forge jest dostępna. Jeśli nie, wyświetla błąd. Za pomocą `forge.pki.publicKeyFromPem` funkcja ładuje klucz publiczny z formatu PEM, który został przekazany do funkcji. Funkcja używa metody `encrypt` z obiektu klucza publicznego, aby zaszyfrować wiadomość. Używa algorytmu RSA w trybie RSA-OAEP, który jest bezpiecznym trybem szyfrowania opartym na RSA. Po zaszyfrowaniu wiadomości, wynik jest kodowany do formatu base64 za pomocą `forge.util.encode64`. Base64 jest powszechnie stosowane do reprezentowania danych binarnych w formie tekstowej, co umożliwia ich bezpieczne przesyłanie lub przechowywanie.

```

window.encryptRSA = function (message, publicKeyPem) {
  if (typeof forge === 'undefined') {
    console.error("Forge library is not loaded.");
    return;
  }

  const publicKey = forge.pki.publicKeyFromPem(publicKeyPem);
  const encrypted = publicKey.encrypt(message, 'RSA-OAEP');
  return forge.util.encode64(encrypted);
};

```

Rysunek 29. Funkcja szyfrująca tekst za pomocą RSA

Ostatnia funkcja **decryptRSA** deszyfruje wiadomość za pomocą klucza prywatnego RSA. Za pomocą `forge.pki.privateKeyFromPem` funkcja ładuje klucz prywat-

ny z formatu PEM, który został przekazany do funkcji. Funkcja najpierw dekoduje zaszyfrowaną wiadomość z formatu base64 za pomocą `forge.util.decode64`. Używając metody `decrypt` z obiektu klucza prywatnego, funkcja odszyfrowuje wiadomość. Używany jest również tryb RSA-OAEP, który jest bezpiecznym trybem deszyfrowania RSA. Na końcu funkcja zwraca odszyfrowaną wiadomość w postaci tekstowej.

```
window.decryptRSA = function (encryptedMessage, privateKeyPem) {  
  if (typeof forge === 'undefined') {  
    console.error("Forge library is not loaded.");  
    return;  
  }  
  
  const privateKey = forge.pki.privateKeyFromPem(privateKeyPem);  
  const encryptedBytes = forge.util.decode64(encryptedMessage);  
  return privateKey.decrypt(encryptedBytes, 'RSA-OAEP');  
};
```

Rysunek 30. Funkcja deszyfrująca tekst za pomocą RSA

Test:

RSA Encryption/Decryption

Klucz publiczny:

-----BEGIN PUBLIC KEY-----  
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCA

Klucz prywatny:

-----BEGIN RSA PRIVATE KEY-----  
MIIEpAIBAAKCAQEA mCC8RGK+HSn5PJA/YtTJzH5x

Wiadomość do zaszyfrowania:

Szyfrogram:

TAK3Z4HnuXh1XVShCvHyZKss/AADSIItbY8RU9zxdgU  
KbJYLwEoCnIARHAI4wj35qThqZm52UFzVnAipfGdNFS

Odszyfrowana wiadomość:

hello world

### 3.10 Podpis cyfrowy

Podpis cyfrowy to mechanizm kryptograficzny umożliwiający weryfikację autentyczności i integralności danych oraz zapewnienie, że pochodzi od określonego nadawcy. Jego działanie opiera się na kryptografii asymetrycznej (klucze publiczne i prywatne) i funkcjach skrótu (hash).

Działanie podpisu cyfrowego:

1) Tworzenie podpisu cyfrowego:

- a) Hashowanie danych - nadawca tworzy skrót wiadomości (hash) za pomocą funkcji skrótu, np. SHA-256. Funkcja hash zapewnia, że wynik jest unikalnym skrótem danych wejściowych,
- b) Szyfrowanie skrótu kluczem prywatnym - skrót (hash) jest szyfrowany za pomocą klucza prywatnego nadawcy, tworząc podpis cyfrowy. Klucz prywatny jest znany tylko nadawcy.

2) Wysłłka:

- a) Nadawca przesyła odbiorcy oryginalną wiadomość oraz podpis cyfrowy.

3) Weryfikacja podpisu cyfrowego:

- a) Hashowanie wiadomości - odbiorca oblicza skrót (hash) z otrzymanej wiadomości za pomocą tej samej funkcji skrótu, której użył nadawca,
- b) Deszyfrowanie podpisu - podpis cyfrowy jest deszyfrowany za pomocą klucza publicznego nadawcy. Klucz publiczny jest powszechnie dostępny.
- c) Porównanie hashy - jeśli hash zdeszyfrowany z podpisu cyfrowego i hash obliczony z wiadomości są identyczne:
  - 1) Wiadomość jest autentyczna i pochodzi od nadawcy (klucz prywatny pasuje do klucza publicznego),
  - 2) Wiadomość nie została zmieniona podczas transmisji.

d) Jeśli hashe różnią się:

1) Wiadomość została zmodyfikowana lub podpis jest nieautentyczny.

Implementacja:

Funkcja `generateRSAKeys` generuje klucz publiczny i prywatny za pomocą biblioteki `Forge`. Klucz publiczny służy do weryfikacji podpisów, natomiast klucz prywatny jest używany do ich tworzenia.

Funkcja `signMessage` hashuje wiadomość algorytmem `SHA-256`, po czym hash jest szyfrowany kluczem prywatnym, tworząc podpis cyfrowy.

Funkcja `verifyMessage` ponownie oblicza hash wiadomości, następnie podpis jest de-szyfrowany za pomocą klucza publicznego i porównywany z obliczonym hashem.

```

window.generateRSAKeys = function () {
    if (typeof forge === 'undefined') {
        console.error("Forge library is not loaded.");
        return;
    }

    const keypair = forge.pki.rsa.generateKeyPair({ bits: 2048, e: 0x10001 });
    return {
        PublicKey: forge.pki.publicKeyToPem(keypair.publicKey),
        PrivateKey: forge.pki.privateKeyToPem(keypair.privateKey),
    };
};

window.signMessage = function (message, privateKeyPem) {
    if (typeof forge === 'undefined') {
        console.error("Forge library is not loaded.");
        return;
    }

    const privateKey = forge.pki.privateKeyFromPem(privateKeyPem);
    const md = forge.md.sha256.create();
    md.update(message, "utf8");
    const signature = privateKey.sign(md);
    return forge.util.encode64(signature);
};

window.verifyMessage = function (message, signature, publicKeyPem) {
    if (typeof forge === 'undefined') {
        console.error("Forge library is not loaded.");
        return false;
    }

    const publicKey = forge.pki.publicKeyFromPem(publicKeyPem);
    const md = forge.md.sha256.create();
    md.update(message, "utf8");
    const decodedSignature = forge.util.decode64(signature);
    return publicKey.verify(md.digest().bytes(), decodedSignature);
};

```

Rysunek 32. Implementacja podpisu cyfrowego w JavaScript z użyciem biblioteki Forge

Test:

**Podpis Cyfrowy**

Klucz publiczny:

```
-----BEGIN PUBLIC KEY-----  
MIIBIjANBgkqhkiG9w0BAQEFAAOCA
```

Klucz prywatny:

```
-----BEGIN RSA PRIVATE KEY-----  
MIIEpAIBAAKCAQEAI1K5OrLLkIaNN
```

Wiadomość:

Podpis cyfrowy:

```
OjW1POMkrey1qd6FEebkrOrWu4OU  
/apS1rbph8THt2VpxyqaSBopZhaW76
```

Podpis jest prawidłowy.

Rysunek 33. Prawidłowa weryfikacja podpisu

Na rysunku poniżej została zmieniona sama wiadomość. Po kliknięciu w przycisk „zweryfikuj podpis” pojawia się powiadomienie, które pokazuje, że podpis nie został zweryfikowany poprawnie.

## Podpis Cyfrowy

Generuj klucze RSA

Klucz publiczny:  
-----BEGIN PUBLIC KEY-----  
MIIBIjANBgkqhkiG9w0BAQEFAAOCA

Klucz prywatny:  
-----BEGIN RSA PRIVATE KEY-----  
MIIEpAIBAAKCAQEAqO6ZnNu3qcO

Wiadomość:

Podpisz wiadomość

Podpis cyfrowy:  
Uyz89JcaeR5f/dHWqp4Ra1QpXVqDI  
Zwl94XG8t5UIA3kSZJ5C/1kBP8ly21c

Zweryfikuj podpis

Podpis jest nieprawidłowy.

Rysunek 34. Nieprawidłowa weryfikacja podpisu

### 3.11 Kod HMAC

HMAC (Hash-based Message Authentication Code) to mechanizm kryptograficzny służący do zapewnienia integralności i uwierzytelnienia wiadomości. HMAC wykorzystuje funkcję skrótu (np. SHA-256, SHA-1) w połączeniu z tajnym kluczem, aby wygenerować kod uwierzytelniający. Kod ten jest przesyłany razem z wiadomością i pozwala odbiorcy zweryfikować, czy wiadomość pochodzi od autoryzowanego nadawcy i czy nie została zmodyfikowana podczas transmisji.

Działanie HMAC:



1) Dane wejściowe:

- a) Wiadomość,
- b) Tajny klucz (znany tylko nadawcy i odbiorcy),
- c) Funkcja skrótu (np. SHA-256, SHA-512).

2) Tworzenie kodu HMAC:

- a) Najpierw klucz jest dopasowywany do odpowiedniego rozmiaru wymaganej funkcji skrótu (jeśli jest zbyt długi, zostaje "przecięty", a jeśli zbyt krótki, jest uzupełniany zerami).
- b) Klucz jest przekształcany przez dwa operatory logiczne:
  - 1) Opadek zewnętrzny (outer padding): Klucz jest XOR-owany z zestawem bitów zwanych opad,
  - 2) Opadek wewnętrzny (inner padding): Klucz jest XOR-owany z zestawem bitów zwanych ipad.
- c) Wiadomość jest łączona z przekształconym kluczem:
  - 1) Najpierw ipad + wiadomość są hashowane,
  - 2) Następnie wynik z powyższego kroku jest łączony z opad i ponownie hashowany.

3) Wynik HMAC:

- a) Wynik końcowy to skrót wiadomości zabezpieczony kluczem, który nazywamy kodem uwierzytelniającym (HMAC).

Implementacja:

Rysunek 30 opisuje funkcję w języku JavaScript tworzącą klucz HMAC. Na początku klucz i wiadomość są konwertowane na Uint8Array za pomocą TextEncoder. Następnie klucz jest importowany jako obiekt kryptograficzny za pomocą Web Crypto API. Potem algorytm HMAC generuje kod na podstawie klucza i wiadomości. Na końcu wynik jest kodowany w Base64 i zwracany do Blazora.

```

window.generateHMAC = async function (message, key, algorithm) {
  try {
    if (!window.crypto || !window.crypto.subtle) {
      throw new Error("Web Crypto API nie jest obsługiwane w tej przeglądarce.");
    }

    const encoder = new TextEncoder();
    const keyData = encoder.encode(key);
    const messageData = encoder.encode(message);

    const cryptoKey = await window.crypto.subtle.importKey(
      "raw",
      keyData,
      { name: "HMAC", hash: { name: algorithm } },
      false,
      ["sign"]
    );

    const signature = await window.crypto.subtle.sign(
      "HMAC",
      cryptoKey,
      messageData
    );

    return btoa(String.fromCharCode(...new Uint8Array(signature)));
  } catch (error) {
    console.error("Błąd generowania HMAC:", error);
    throw error;
  }
};

```

Rysunek 35. Implementacja kodu HMAC w JavaScript

Test:

### Generowanie HMAC

Wiadomość:

Klucz:

Algorytm (np. SHA-256):

Wynik HMAC:

EKdK5EFzrsX/SBRoC5YNEDHu7LWjRz  
 KDFDITnek1L/I=

Rysunek 36. Test kodu HMAC

## 4. Podsumowanie i wnioski

Projekt koncentrował się na głębszym zrozumieniu i praktycznej implementacji różnych algorytmów kryptograficznych oraz technik zabezpieczeń, które są kluczowe dla zapewnienia bezpieczeństwa danych w aplikacjach internetowych. Każdy algorytm, od szyfru Gronsfelda po RSA, HMAC, Diffie-Hellman, oraz szyfrowanie blokowe i strumieniowe, miał na celu zabezpieczenie danych podczas ich przesyłania i przechowywania. Dzięki projektowi uczestnicy mieli możliwość zgłębić teorię oraz praktyczną stronę kryptografii, co umożliwiło im lepsze zrozumienie mechanizmów, które chronią integralność, poufność oraz autentyczność danych.

Projekt pozwolił na zrozumienie, jak różne algorytmy kryptograficzne pełnią różne funkcje. Szyfry takie jak Gronsfelda, transpozycyjne, oraz RSA różnią się zastosowaniami i poziomem bezpieczeństwa, co pokazało różnorodność i elastyczność kryptografii.

Tworzenie tych algorytmów w realnych aplikacjach, takich jak Blazor, pozwala na zrozumienie ich działania w praktyce. Na przykład, algorytm Diffie-Hellmana pozwala na bezpieczną wymianę kluczy, co jest kluczowe dla zapewnienia bezpieczeństwa komunikacji w sieci. Symulacja protokołu Diffie-Hellmana pokazała, jak łatwo można wymienić klucze, a jednocześnie zabezpieczyć komunikację.

Projekt pozwolił uczestnikom zrozumieć nie tylko jak działają poszczególne algorytmy kryptograficzne, ale także jak je odpowiednio zastosować w praktyce. Bezpieczne zarządzanie kluczami oraz wybór odpowiednich algorytmów zależy od specyficznych wymagań systemu i kontekstu użycia. Ważne jest, aby zrozumieć ograniczenia każdego algorytmu i stosować odpowiednie techniki zabezpieczeń w zależności od specyfiki aplikacji. Symulacje protokołu Diffie-Hellmana i RSA pokazały, jak algorytmy te mogą być używane do ochrony komunikacji i danych w sieci.