POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

Implementacja algorytmu 3DES w języku C#

Dominik Sucharski

Poznań, 2020

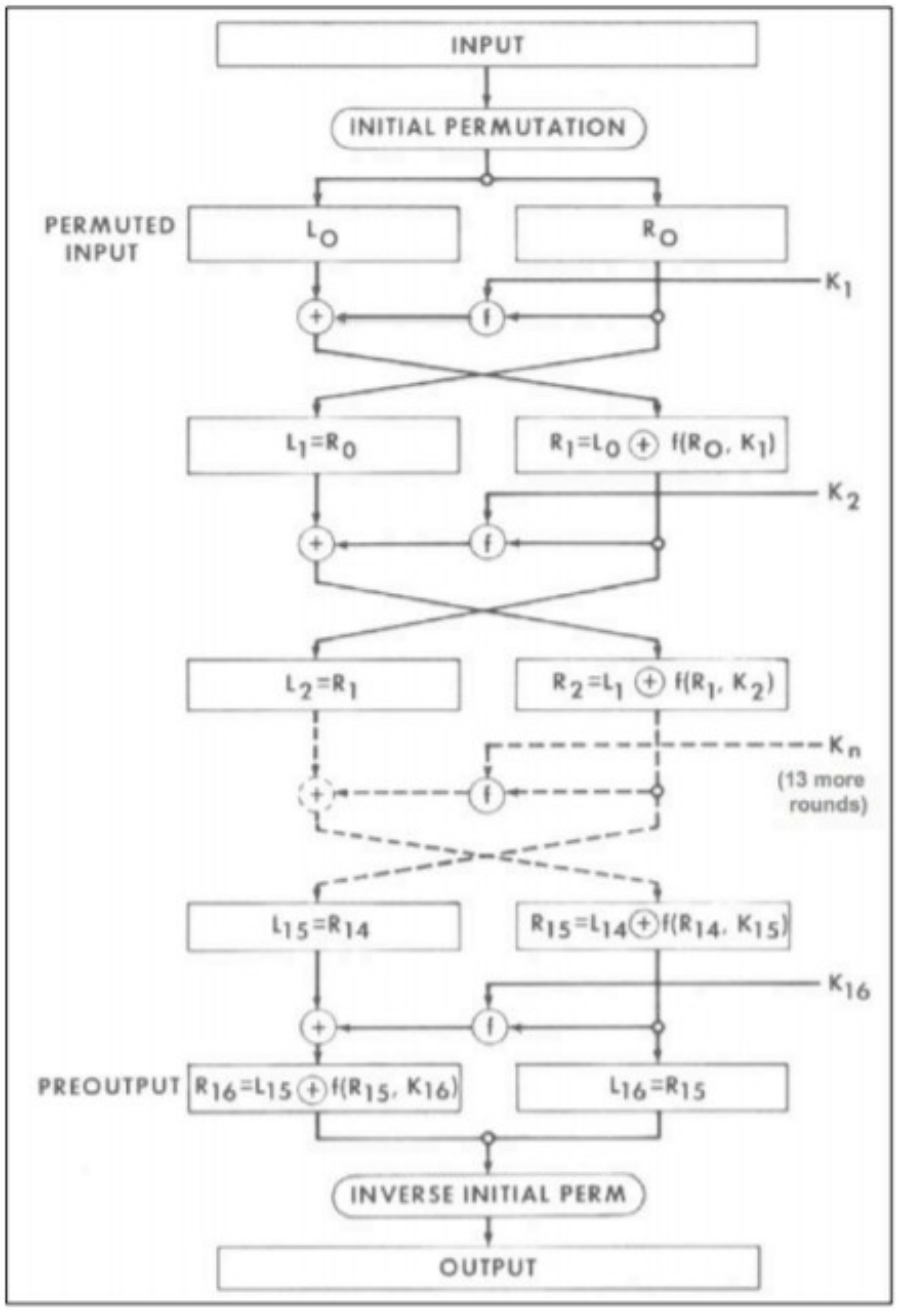
Spis treści

1. Wstęp

Algorytm 3DES opiera się na algorytmie DES. Algorytm DES jest przeznaczony do szyfrowania i deszyfrowania 64 bitowych bloków danych za pomocą 64 bitowego klucza. Szyfrowanie i deszyfrowanie odbywa się za pomocą tego samego klucza. DES oraz 3DES są algorytmami z kluczem symetrycznym.

2. DES

2.1. Szyfrowanie



Rysunek 1: Schemat blokowy szyfrowania DES

Pierwszym krokiem jest wygenerowanie 48 bitowych kluczy opisane w podrozdziale 2.3. Klucze są zapisywane w tablicy sub_key, z której później są odczytywane. Kolejnym krokiem jest wykonanie permutacji początkowej danych wejściowych (2.4). Po tej operacji dane są dzielone na dwie części po 32 bity, które przechodzą przez 16 iteracji funkcji **f** opisanej w podrozdziale 2.5. Do funkcji **f** jest przekazywana prawa część danych oraz kolejne klucze wygenerowane w metodzie **Keygen**. Ostatnim etapem jest połączenie części lewej i prawej oraz permutacja końcowa (2.6).

```
/**
 * Szyfrowanie 64 bitowego bloku danych
 */
3 references
public UInt64 Encrypt(UInt64 data, UInt64 key)
{
    // generowanie kluczy
    Keygen(key);
    // permutacja początkowa
    data = InitialPermutation(data);

    // podział danych na dwie części po 32 bity
    UInt32 R = (UInt32)data;  // prawa 32 bitowa część danych
    UInt32 L = (UInt32)(data>>32);  // lewa 32 bitowa część danych

    UInt32 L_prime = 0;
    // 16 rund szyfrowania
    for (byte i = 0; i < 16; i++)
    {
        L_prime = R;  // nowa lewa część danych
        R = L ^ f(R, sub_key[i]);  // nowa prawa część danych
        L = L_prime;
    }
    // łączenie części prawej i lewej
    data = (((UInt64)R) << 32) | (UInt64)L;
    // permutacja końcowa
    return FinalPermutation(data);
}
```

Rysunek 2: Kod metody Encrypt

2.2. Deszyfrowanie

Deszyfrowanie przebiega w taki sam sposób jak szyfrowanie, z wyjątkiem kolejności pobierania kluczy z tablicy sub_key. Klucze te są pobierane od końca.

```
/**
 * Deszyfrowanie 64 bitowego bloku danych
 * Różni się jedynie kolejnością używania kluczy sub_key[15-i]
 */
3 references
public UInt64 Decrypt(UInt64 data, UInt64 key)
{
    // generowanie kluczy
    Keygen(key);
    // permutacja początkowa
    data = InitialPermutation(data);

    // podział danych na dwie części po 32 bity
    UInt32 R = (UInt32)data;  // prawa 32 bitowa część danych
    UInt32 L = (UInt32)(data >> 32);  // lewa 32 bitowa część danych

    UInt32 L_prime = 0;
    // 16 rund szyfrowania
    for (byte i = 0; i < 16; i++)
    {
        L_prime = R;  // nowa lewa część danych
        R = L ^ f(R, sub_key[15-i]);  // nowa prawa część danych
        L = L_prime;
    }
    // łączenie części prawej i lewej
    data = (((UInt64)R) << 32) | (UInt64)L;
    // permutacja końcowa
    return FinalPermutation(data);
}
```

Rysunek 3: Kod metody Decrypt

2.3. Generowanie kluczy

Generowanie kluczy odbywa się za pomocą metody **Keygen**. Metoda wykorzystuje tablice permutacji PC1 i PC2 oraz tablicę ITERATION_SHIFT, która określa przesunięcie każdej 28-bitowej połówki klucza w lewo o jeden lub dwa bity.

```
/**
 * Generowanie bitów klucza.
 */
2 references
private void Keygen(UInt64 key)
{
    // zerowanie tabeli
    Array.Clear(sub_key, 0, sub_key.Length);

    UInt64 permuted_choice_1 = 0;
    for (byte i = 0; i < 56; i++)  // 56 bitów klucza
    {
        permuted_choice_1 <<= 1;
        permuted_choice_1 |= (key >> (64 - PC1[i])) & 0x0000000000000001;  // permutacja PC-1
    }

    UInt32 C = (UInt32)((permuted_choice_1 >> 28) & 0x00000000ffffffff);  // lewa połowa (28 bitów)
    UInt32 D = (UInt32)(permuted_choice_1 & 0x00000000ffffffff);  // prawa połowa (28 bitów)

    // obliczanie 16 kluczy
    for (byte i = 0; i < 16; i++)
    {
        // przesuwanie Ci i Di
        for (byte j = 0; j < ITERATION_SHIFT[i]; j++)
        {
            C = (0x0fffffff & (C << 1)) | (0x00000001 & (C >> 27));
            D = (0x0fffffff & (D << 1)) | (0x00000001 & (D >> 27));
        }

        UInt64 permuted_choice_2 = (((UInt64)C) << 28) | (UInt64)D;
        sub_key[i] = 0; // każdy podklucz składa się z 48 bitów
        for (byte j = 0; j < 48; j++)
        {
            sub_key[i] <<= 1;
            sub_key[i] |= (permuted_choice_2 >> (56 - PC2[j])) & 0x0000000000000001;  // permutacja PC-2
        }
    }
}
```

Rysunek 4: Kod metody Keygen

```
// Tabela permutacji klucza PC-1
private static readonly byte[] PC1 = new byte[56]
{
    57, 49, 41, 33, 25, 17,  9,
     1, 58, 50, 42, 34, 26, 18,
    10,  2, 59, 51, 43, 35, 27,
    19, 11,  3, 60, 52, 44, 36,

    63, 55, 47, 39, 31, 23, 15,
     7, 62, 54, 46, 38, 30, 22,
    14,  6, 61, 53, 45, 37, 29,
    21, 13,  5, 28, 20, 12,  4
};
```

Rysunek 5: Tabela permutacji PC-1

```
// Tabela przesunięcia bitów klucza
private static readonly byte[] ITERATION_SHIFT = new byte[16]
{
    1,  1,  2,  2,  2,  2,  2,  2,  1,  2,  2,  2,  2,  2,  2,  1
};
```

Rysunek 6: Tabela przesunięć bitów klucza

```
// Tabela permutacji klucza PC-2
private static readonly byte[] PC2 = new byte[48]
{
    14, 17, 11, 24,  1,  5,
     3, 28, 15,  6, 21, 10,
    23, 19, 12,  4, 26,  8,
    16,  7, 27, 20, 13,  2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32
};
```

Rysunek 7: Tabela permutacji PC-2

2.4. Permutacja początkowa

Szyfrowany blok jest najpierw poddawany permutacji początkowej (ang. initial permutation).

```
/**
 * Permutacja początkowa.
 * Przyjmuje na wejście liczbę 64 bitową.
 * Zwraca liczbę 64 bitową.
 */
1 reference
private UInt64 InitialPermutation(UInt64 input)
{
    UInt64 result = 0;  // wynik
    for (byte i = 0; i < 64; i++)  // iteracja po wszystkich bitach
    {
        result <<= 1;  // przesunięcie bitowe o 1 w lewo
        // zmiana pozycji bitu zgodnie z tablicą permutacji początkowej
        result |= (input >> (64 - IP[i])) & 0x0000000000000001;
    }
    return result;
}
```

Rysunek 8: Permutacja początkowa - funkcja

Wszystkie bity danych wejściowych są wymieniane zgodnie z tabelą permutacji początkowej IP przedstawionej na rysunku 9. 58 bit danych wejściowych jest pierwszym bitem, 50 drugim i tak dalej aż do bitu 7, który jest ostatnim bitem.

```
// Tabela permutacji początkowej
private static readonly byte[] IP = new byte[64]
{
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17,  9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
};
```

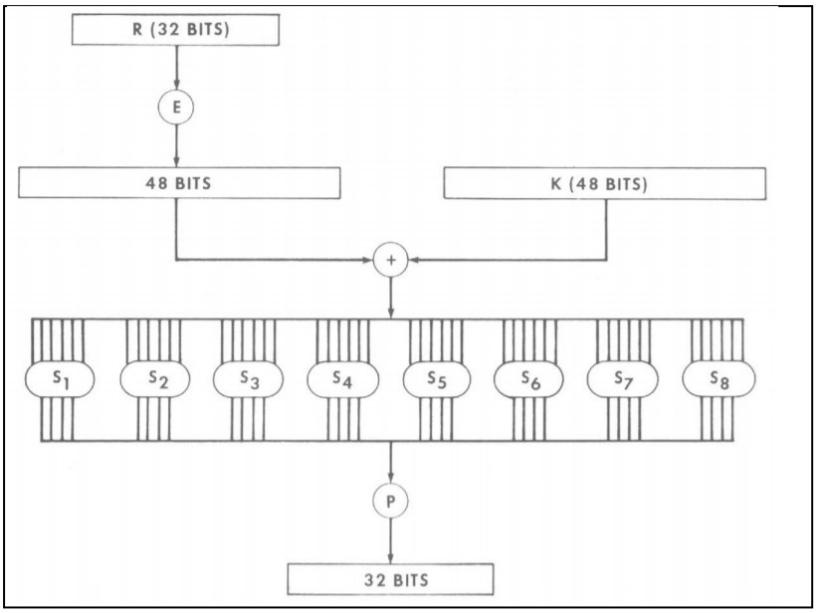
Rysunek 9: Tabela permutacji początkowej

Następnie dane wejściowe są dzielone na dwie części po 32 bity.

2.5. Funkcja f

```
/**
 * Funkcja f.
 * Przyjmuje 32 bity danych wejściowych i
 * 48 bitowy podklucz
 * Zwraca liczbę 32 bitową.
 */
2 references
private UInt32 f(UInt32 R, UInt64 K)
{
    UInt64 expanded_input = 0;
    // rozszerzenie z 32 bitów do 48 bitów
    for (byte i = 0; i < 48; i++)
    {
        expanded_input <<= 1;
        expanded_input |= (UInt64)((R >> (32 - EXPANSION[i])) & 0x00000001);
    }

    // XOR z podkluczem
    expanded_input ^= K;

    // S1(B1)S2(B2)S3(B3)S4(B4)S5(B5)S6(B6)S7(B7)S8(B8)
    UInt32 output_s = 0;
    // 8 s-bloków
    for (byte i = 0; i < 8; i++)
    {
        // wybranie pierwszego i ostatniego bitu z każdej 6 bitowej części
        byte row = (byte)((expanded_input & (UInt64)(0x0000840000000000 >> 6 * i)) >> (42 - 6 * i));
        row = (byte)((row >> 4) | (row & 0x01));

        // wybranie 4 środkowych bitów
        byte column = (byte)((expanded_input & (UInt64)(0x0000780000000000 >> 6 * i)) >> (43 - 6 * i));

        output_s <<= 4;  // przezsunięcie o 4 pozycje
        output_s |= (UInt32)(SBOX[i, 16 * row + column] & 0x0f);  // dodanie 4 bitów wyjściowych
    }

    UInt32 output_p = 0;
    // permutacja w P-blokach
    for (byte i = 0; i < 32; i++)
    {
        output_p <<= 1;
        output_p |= (output_s >> (32 - PBOX[i])) & 0x00000001;
    }

    return output_p;
}
```

Rysunek 10: Kod metody f



Rysunek 11: Obliczanie funkcji f(R,K)

Pierwszym etapem jest rozszerzenie danych wejściowych za pomocą tabeli permutacji rozszerzającej.

```
// Tabela permutacji rozszerzającej (E BIT-SELECTION TABLE)
private static readonly byte[] EXPANSION = new byte[48]
{
    32,  1,  2,  3,  4,  5,
     4,  5,  6,  7,  8,  9,
     8,  9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32,  1
};
```

Rysunek 12: Tabela permutacji rozszerzającej

Następnie rozszerzone dane są łączone z 48 bitowym kluczem odpowiednim dla danej iteracji za pomocą operacji XOR (suma modulo 2).

48 bitowy wynik operacji XOR jest dzielony na 6 bitowe bloki. Z i-tego bloku pierwszy i ostatni bit odpowiada za wybór wiersza z i-tego SBOX-a, a 4 środkowe odpowiadają za wybór kolumny. Odczytana wartość z tablicy SBOX stanowi 4 bity wyjściowe. Po przejściu po wszystkich 6 bitowych blokach, dokonywana jest permutacja P-bloku.

```
// S-bloki [8*16*4]
private static readonly byte[,] SBOX = new byte[,]
{
    {  // S1
        14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7,
         0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
         4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
        15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13
    },
    {  // S2
        15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10,
         3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5,
         0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15,
        13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9
    },
    {  // S3
        10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8,
        13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1,
        13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7,
         1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12
    },
    {  // S4
         7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15,
        13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9,
        10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4,
         3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14
    },
    {  // S5
         2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9,
        14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6,
         4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14,
        11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3
    },
    {  // S6
        12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11,
        10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8,
         9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6,
         4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13
    },
    {  // S7
         4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1,
        13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6,
         1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2,
         6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12
    },
    {  // S8
        13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7,
         1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2,
         7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8,
         2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11
    }
};
```

```
// Tabela permutacji w P-Bloku
private static readonly byte[] PBOX = new byte[32]
{
    16,  7, 20, 21,
    29, 12, 28, 17,
     1, 15, 23, 26,
     5, 18, 31, 10,
     2,  8, 24, 14,
    32, 27,  3,  9,
    19, 13, 30,  6,
    22, 11,  4, 25
};
```

Rysunek 13: Tabela permutacji w P-bloku

2.6. Permutacja końcowa

Permutacja końcowa jest odwrotnością permutacji początkowej opisanej w podrozdziale 2.4.

```
/**
* Permutacja końcowa.
 * Przyjmuje na wejście liczbę 64 bitową.
 * Zwraca liczbę 64 bitową.
 */
1 reference
private UInt64 FinalPermutation(UInt64 input)
{
    UInt64 result = 0;
    for (byte i = 0; i < 64; i++)
    {
        result <<= 1;
        result |= (input >> (64 - FP[i])) & 0x0000000000000001;
    }
    return result;
}
```

Rysunek 14: Permutacja końcowa - funkcja

```
// Tabela permutacji końcowej
private static readonly byte[] FP = new byte[64]
{
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41,  9, 49, 17, 57, 25
};
```

Rysunek 15: Tabela permutacji końcowej

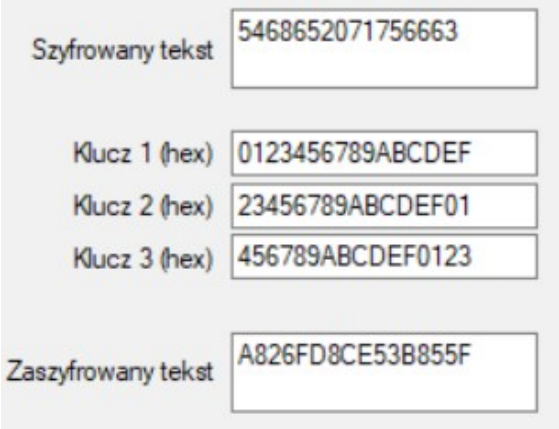
3. 3DES

3DES został zaimplementowany w klasie **tdes**. Metoda **EncryptBlock** pozwala zaszyfrować 64 bitowy blok danych, natomiast metoda **DecryptBlock** odpowiada za deszyfrowanie 64 bitowego bloku danych. Każdy z trzech kluczy przekazywanych jako parametry funkcji musi mieć 64 bity długości.

```
private des _des;
2 references
public tdes()
{
    _des = new des();
}

1 reference
public UInt64 EncryptBlock(UInt64 block, UInt64 key1, UInt64 key2, UInt64 key3)
{
    // input, key1
    UInt64 result = _des.Encrypt(block, key1);
    // key 2
    result = _des.Decrypt(result, key2);
    // key 3
    result = _des.Encrypt(result, key3);
    return result;
}

1 reference
public UInt64 DecryptBlock(UInt64 block, UInt64 key1, UInt64 key2, UInt64 key3)
{
    // input, key3
    UInt64 result = _des.Decrypt(block, key3);
    // key 2
    result = _des.Encrypt(result, key2);
    // key 1
    result = _des.Decrypt(result, key1);
    return result;
}
```

4. Porównanie z referencją

Porównanie z załącznikiem B znajdującym się w pliku dostępnym pod adresem https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-67r1.pdf pozwoliło potwierdzić prawidłowe działanie zaimplementowanego szyfru 3DES w trybie ECB. Tryb ten charakteryzuje się niezależnym szyfrowaniem kolejnych bloków danych.



Rysunek 16: Okno programu podczas testu

5. Przyjmowanie plików na wejście

Szyfrowanie i deszyfrowanie odbywa się w 64 bitowych blokach danych. Dane wejściowe muszą mieć rozmiar, który jest wielokrotnością wielkości bloku. Odczytany plik wejściowy jest dzielony na bloki, które są szyfrowane lub deszyfrowane. Przetworzone bloki są zapisywane w pliku wynikowym. Podstęp operacji jest wyświetlany na pasku statusu i jest aktualizowany co 16 kB, ze względu na znaczny spadek szybkości szyfrowania podczas częstych aktualizacji okna programu.

```
/**
 * Szyfrowanie pliku
 */
1 reference
public void EncryptFile(string inputFileName, string outputFileName, UInt64 key1, UInt64 key2, UInt64 key3,
    ToolStripStatusLabel statusLabel, StatusStrip statusStrip)
{
    // sprawdzenie czy plik istnieje
    if (!File.Exists(inputFileName))
    {
        statusLabel.Text = "Plik nie istnieje";
        return;
    }
    FileStream fileStreamRead = new FileStream(inputFileName, FileMode.Open);

    // sprawdzenie rozmiaru pliku, wymagana wieloktorność 8 bajtów (64 bity)
    if (fileStreamRead.Length % 8 != 0)
    {
        statusLabel.Text = "Rozmiar pliku musi być wielokrotnością 8 bajtów";
        return;
    }

    FileStream fileStreamWrite = new FileStream(outputFileName, FileMode.Create, FileAccess.Write);

    int block_count = (int)(fileStreamRead.Length / 8);
    UInt64 block = 0;
    for (int i = 0; i < fileStreamRead.Length; i+=8)
    {
        // tworzenie bloku
        block = (UInt64)fileStreamRead.ReadByte();  // pierwszy bajt
        for (int j = 1; j < 8; j++)
        {
            UInt64 blockByte = (UInt64)fileStreamRead.ReadByte();
            blockByte <<= (j * 8);
            block |= blockByte;
        }

        // szyfrowanie bloku
        UInt64 result = EncryptBlock(block, key1, key2, key3);

        // zapisywanie bloku w pliku
        byte resultByte = (byte)(result & (UInt64)0xff);  // pierwszy bajt
        fileStreamWrite.WriteByte(resultByte);
        for (int j = 1; j < 8; j++)
        {
            resultByte = (byte)((result >> (8 * j)) & 0x00000000000000ff);
            fileStreamWrite.WriteByte(resultByte);
        }
        if ((i % 16000) == 0)
        {
            statusLabel.Text = "Zaszyfrowano " + ((i + 8) / 8).ToString() + " / " + block_count.ToString() + " bloków";
            statusStrip.Refresh();
            Application.DoEvents();
        }

    }
    statusLabel.Text = "Zaszyfrowano " + block_count.ToString() + " / " + block_count.ToString() + " bloków";
    fileStreamWrite.Close();
    fileStreamRead.Close();
}
```

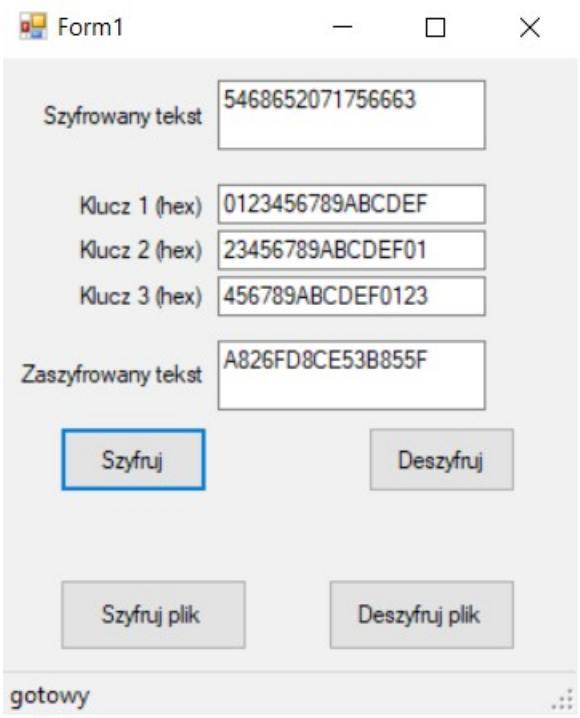
Rysunek 17: Kod metody szyfrującej pliki

```
/**
 * Deszyfrowanie pliku
 */
1 reference
public void DecryptFile(string inputFileName, string outputFileName, UInt64 key1, UInt64 key2, UInt64 key3,
    System.Windows.Forms.ToolStripStatusLabel statusLabel, StatusStrip statusStrip)
{
    // sprawdzenie czy plik istnieje
    if (!File.Exists(inputFileName))
    {
        statusLabel.Text = "Plik nie istnieje";
        return;
    }
    FileStream fileStreamRead = new FileStream(inputFileName, FileMode.Open);

    // sprawdzenie rozmiaru pliku, wymagana wieloktorność 8 bajtów (64 bity)
    if (fileStreamRead.Length % 8 != 0)
    {
        statusLabel.Text = "Rozmiar pliku musi być wielokrotnością 8 bajtów";
        return;
    }

    FileStream fileStreamWrite = new FileStream(outputFileName, FileMode.Create, FileAccess.Write);

    int block_count = (int)(fileStreamRead.Length / 8);
    UInt64 block = 0;
    for (int i = 0; i < fileStreamRead.Length; i += 8)
    {
        // tworzenie bloku
        block = (UInt64)fileStreamRead.ReadByte();  // pierwszy bajt
        for (int j = 1; j < 8; j++)
        {
            UInt64 blockByte = (UInt64)fileStreamRead.ReadByte();
            blockByte <<= (j * 8);
            block |= blockByte;
        }

        // deszyfrowanie bloku
        UInt64 result = DecryptBlock(block, key1, key2, key3);

        // zapisywanie bloku w pliku
        byte resultByte = (byte)(result & (UInt64)0xff);
        fileStreamWrite.WriteByte(resultByte);
        for (int j = 1; j < 8; j++)
        {
            resultByte = (byte)((result >> (8 * j)) & 0x00000000000000ff);
            fileStreamWrite.WriteByte(resultByte);
        }
        if ((i % 16000) == 0)
        {
            statusLabel.Text = "Deszyfrowano " + ((i + 8) / 8).ToString() + " / " + block_count.ToString() + " bloków";
            statusStrip.Refresh();
            Application.DoEvents();
        }
    }
    statusLabel.Text = "Deszyfrowano " + block_count.ToString() + " / " + block_count.ToString() + " bloków";
    fileStreamWrite.Close();
    fileStreamRead.Close();
}
```

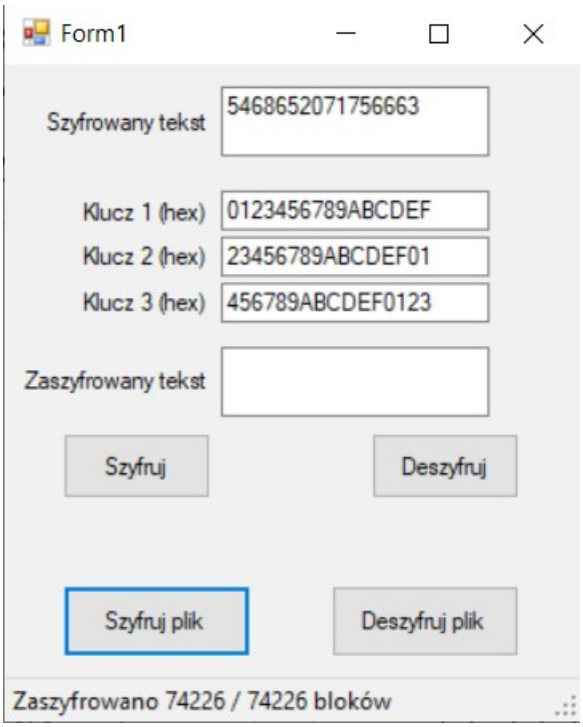
Rysunek 18: Kod metody deszyfrującej pliki

6. Podsumowanie i wnioski

Implementacja algorytmu 3DES działa prawidłowo. Pozwala szyfrować pliki o rozmiarze 128MB, a także dowolne inne dane w postaci 64 bitowych bloków. Dodatkowe testy przeprowadzone na plikach graficznych także przebiegły pomyślnie.



Rysunek 20: Okno programu – szyfrowanie liczby w formacie hex



Rysunek 19: Okno programu - po szyfrowaniu pliku