**ЛАБОРАТОРНА РОБОТА #2**

З дисципліни «Комп'ютерна криптографія»

На тему: «Дослідження блочних алгоритмів симетричних криптографічних перетворень»

Варіант ДСТУ 7624:2014

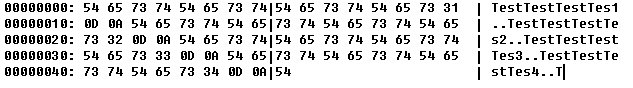
|  |  |
| --- | --- |
| **Виконали**  Студенти групи КВ-61м Щербакова Г.В.  Бондарчук М.Ю.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Прийняв**  Доцент кафедри СПСКС  Тесленко О.К.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

ЗАВДАННЯ

Розробити програму шифрування й розшифрування по стандарту ДСТУ 7624:2014 «Калина» (довжина блоку та ключа – 128 біт).

ФАЙЛИ З ПРИКЛАДАМИ

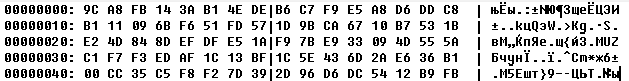
Файл з відкритим текстом



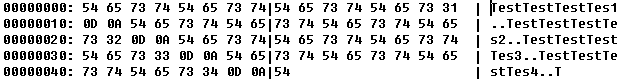
Файл з ключем



Зашифрований файл



Розшифрований файл



ТАБЛИЦЯ ШВИДКОДІЇ АЛГОРИТМУ

Досліди проводились для процесора Intel Pentium B960 2.2 ГГц.

|  |  |  |
| --- | --- | --- |
| Розмір файлу | Час шифрування | Час розшифрування |
| 10 КБ | 00:00:00.0980611 | 00:00:00.1876094 |
| 100 КБ | 00:00:00.8882621 | 00:00:01.9630112 |
| 500 КБ | 00:00:04.5881128 | 00:00:09.7990151 |
| 1 МБ | 00:00:07.8471330 | 00:00:19.5187705 |

ЛІСТИНГ ПРОГРАМИ МОВОЮ C#

Block.cs

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

namespace Kalyna

{

public class Block

{

public List<byte> Data { get; set; } = new List<byte>();

public Block() { }

public Block(Block block)

{

Data = new List<byte>(block.Data);

}

/// <summary>

/// Adds round key

/// </summary>

/// <param name="key"></param>

public void AddRoundKey(Block key)

{

const int n = 8;

for (var i = 0; i < 16; i += n)

{

var dataBi = new BigInteger(Data.Where((d, idx) => i <= idx && idx < i + n).ToArray());

dataBi += new BigInteger(key.Data.Where((d, idx) => i <= idx && idx < i + n).ToArray());

var newData = dataBi.ToByteArray();

for (var j = 0; j < n; j++)

Data[i + j] = j < newData.Length ? newData[j] : (byte)0;

}

}

/// <summary>

/// Substracts round key

/// </summary>

/// <param name="key"></param>

public void SubRoundKey(Block key)

{

const int n = 8;

for (var i = 0; i < 16; i += n)

{

var dataBi = new BigInteger(Data.Where((d, idx) => i <= idx && idx < i + n).ToArray());

dataBi -= new BigInteger(key.Data.Where((d, idx) => i <= idx && idx < i + n).ToArray());

var newData = dataBi.ToByteArray();

for (var j = 0; j < n; j++)

Data[i + j] = j < newData.Length ? newData[j] : (byte)0;

}

}

/// <summary>

/// Cyclic shifts internal state matrix rightwards

/// </summary>

/// <param name="i">Positions number</param>

public void RotateRight(int i)

{

var bi = new BigInteger(Data.ToArray());

bi = (bi >> i % 128) + (bi << (128 - i % 128));

Data = new List<byte>(bi.ToByteArray().Where((t, idx) => idx < 16));

}

/// <summary>

/// Cyclic shifts internal state matrix leftwards

/// </summary>

/// <param name="i">Positions number</param>

public void RotateLeft(int i)

{

var bi = new BigInteger(Data.ToArray());

bi = (bi << i % 128) + (bi >> (128 - i % 128));

Data = new List<byte>(bi.ToByteArray().Where((t, idx) => idx < 16));

}

/// <summary>

/// Shifts internal state matrix leftwards

/// </summary>

/// <param name="i">Positions number</param>

public void ShiftLeft(int i)

{

var bi = new BigInteger(Data.ToArray());

bi <<= i;

Data = new List<byte>(bi.ToByteArray());

}

/// <summary>

/// Changes bytes from S-matrix

/// </summary>

/// <param name="table"></param>

public void SubBytes(byte[][][] table)

{

var trump = 0;

for (var i = Data.Count - 1; 0 <= i; --i)

{

var d = Data[i];

var upper = (d & 0xF0) >> 4;

var lower = d & 0x0F;

Data[i] = table[trump % 4][upper][lower];

trump = ++trump % 4;

}

}

/// <summary>

/// Swap two items of internal state matrix

/// </summary>

/// <param name="i1"></param>

/// <param name="i2"></param>

private void ShiftBytesPair(int i1, int i2)

{

var buf = Data[i1];

Data[i1] = Data[i2];

Data[i2] = buf;

}

/// <summary>

/// Performs internal state matrix rows cyclic right shift (>>> 8)

/// </summary>

public void ShiftRows()

{

for (var i = 11; 8 <= i; --i)

ShiftBytesPair(i, i - 8);

}

/// <summary>

/// Performs internal state matrix rows cyclic left shift (>>> 8)

/// </summary>

public void ShiftRowsRev()

{

for (var i = 11; 8 <= i; --i)

ShiftBytesPair(i - 8, i);

}

/// <summary>

/// Performs Galois multiplication. From Wikipedia

/// </summary>

/// <param name="a"></param>

/// <param name="b"></param>

/// <returns></returns>

private static byte Gmul(int a, int b)

{

byte p = 0; /\* the product of the multiplication \*/

while (b != 0)

{

if ((b & 1) == 1) /\* if b is odd, then add the corresponding a to p (final product = sum of all a's corresponding to odd b's) \*/

p ^= (byte)a; /\* since we're in GF(2^m), addition is an XOR \*/

if ((a & 0x80) == 0x80) /\* GF modulo: if a >= 128, then it will overflow when shifted left, so reduce \*/

a = (a << 1) ^ 0x11d; /\* XOR with the primitive polynomial x^8 + x^4 + x^3 + x + 1 (0b1\_0001\_1011) -- you can change it but it must be irreducible \*/

else

a <<= 1; /\* equivalent to a\*2 \*/

b >>= 1; /\* equivalent to b // 2 \*/

}

return p;

}

/// <summary>

/// Mixes columns of internal state matrix

/// </summary>

/// <param name="table"></param>

public void MixColumns(byte[][] table)

{

var dataCopy = new List<byte>(Data);

// First column

var trump = Data.Count - 1;

for (var row = 0; row < 8; row++)

{

byte sum = 0;

var hillary = Data.Count - 1;

for (var h = 0; h < 8; h++)

{

sum ^= Gmul(dataCopy[hillary], table[row][h]);

hillary--;

}

Data[trump] = sum;

trump--;

}

// Second column

trump = Data.Count - 1 - 8;

for (var row = 0; row < 8; row++)

{

byte sum = 0;

var hillary = Data.Count - 1 - 8;

for (var h = 0; h < 8; h++)

{

sum ^= Gmul(dataCopy[hillary], table[row][h]);

hillary--;

}

Data[trump] = sum;

trump--;

}

}

/// <summary>

/// Performs exclusive or (XOR) with round key

/// </summary>

/// <param name="key"></param>

public void Xor(Block key)

{

for (var i = 0; i < Data.Count; i++)

Data[i] ^= key.Data[i];

}

}

}

StaticTables.cs

using System.Collections.Generic;

namespace Kalyna

{

public static class StaticTables

{

public static readonly List<byte> V = new List<byte>

{

0, 1, 0, 1, 0, 1, 0, 1,

0, 1, 0, 1, 0, 1, 0, 1

};

public static readonly byte[][][] Π =

{

new [] {

new byte[] { 0xA8, 0x43, 0x5F, 0x06, 0x6B, 0x75, 0x6C, 0x59, 0x71, 0xDF, 0x87, 0x95, 0x17, 0xF0, 0xD8, 0x09 },

new byte[] { 0x6D, 0xF3, 0x1D, 0xCB, 0xC9, 0x4D, 0x2C, 0xAF, 0x79, 0xE0, 0x97, 0xFD, 0x6F, 0x4B, 0x45, 0x39 },

new byte[] { 0x3E, 0xDD, 0xA3, 0x4F, 0xB4, 0xB6, 0x9A, 0x0E, 0x1F, 0xBF, 0x15, 0xE1, 0x49, 0xD2, 0x93, 0xC6 },

new byte[] { 0x92, 0x72, 0x9E, 0x61, 0xD1, 0x63, 0xFA, 0xEE, 0xF4, 0x19, 0xD5, 0xAD, 0x58, 0xA4, 0xBB, 0xA1 },

new byte[] { 0xDC, 0xF2, 0x83, 0x37, 0x42, 0xE4, 0x7A, 0x32, 0x9C, 0xCC, 0xAB, 0x4A, 0x8F, 0x6E, 0x04, 0x27 },

new byte[] { 0x2E, 0xE7, 0xE2, 0x5A, 0x96, 0x16, 0x23, 0x2B, 0xC2, 0x65, 0x66, 0x0F, 0xBC, 0xA9, 0x47, 0x41 },

new byte[] { 0x34, 0x48, 0xFC, 0xB7, 0x6A, 0x88, 0xA5, 0x53, 0x86, 0xF9, 0x5B, 0xDB, 0x38, 0x7B, 0xC3, 0x1E },

new byte[] { 0x22, 0x33, 0x24, 0x28, 0x36, 0xC7, 0xB2, 0x3B, 0x8E, 0x77, 0xBA, 0xF5, 0x14, 0x9F, 0x08, 0x55 },

new byte[] { 0x9B, 0x4C, 0xFE, 0x60, 0x5C, 0xDA, 0x18, 0x46, 0xCD, 0x7D, 0x21, 0xB0, 0x3F, 0x1B, 0x89, 0xFF },

new byte[] { 0xEB, 0x84, 0x69, 0x3A, 0x9D, 0xD7, 0xD3, 0x70, 0x67, 0x40, 0xB5, 0xDE, 0x5D, 0x30, 0x91, 0xB1 },

new byte[] { 0x78, 0x11, 0x01, 0xE5, 0x00, 0x68, 0x98, 0xA0, 0xC5, 0x02, 0xA6, 0x74, 0x2D, 0x0B, 0xA2, 0x76 },

new byte[] { 0xB3, 0xBE, 0xCE, 0xBD, 0xAE, 0xE9, 0x8A, 0x31, 0x1C, 0xEC, 0xF1, 0x99, 0x94, 0xAA, 0xF6, 0x26 },

new byte[] { 0x2F, 0xEF, 0xE8, 0x8C, 0x35, 0x03, 0xD4, 0x7F, 0xFB, 0x05, 0xC1, 0x5E, 0x90, 0x20, 0x3D, 0x82 },

new byte[] { 0xF7, 0xEA, 0x0A, 0x0D, 0x7E, 0xF8, 0x50, 0x1A, 0xC4, 0x07, 0x57, 0xB8, 0x3C, 0x62, 0xE3, 0xC8 },

new byte[] { 0xAC, 0x52, 0x64, 0x10, 0xD0, 0xD9, 0x13, 0x0C, 0x12, 0x29, 0x51, 0xB9, 0xCF, 0xD6, 0x73, 0x8D },

new byte[] { 0x81, 0x54, 0xC0, 0xED, 0x4E, 0x44, 0xA7, 0x2A, 0x85, 0x25, 0xE6, 0xCA, 0x7C, 0x8B, 0x56, 0x80 }

},

new [] {

new byte[] { 0xCE, 0xBB, 0xEB, 0x92, 0xEA, 0xCB, 0x13, 0xC1, 0xE9, 0x3A, 0xD6, 0xB2, 0xD2, 0x90, 0x17, 0xF8 },

new byte[] { 0x42, 0x15, 0x56, 0xB4, 0x65, 0x1C, 0x88, 0x43, 0xC5, 0x5C, 0x36, 0xBA, 0xF5, 0x57, 0x67, 0x8D },

new byte[] { 0x31, 0xF6, 0x64, 0x58, 0x9E, 0xF4, 0x22, 0xAA, 0x75, 0x0F, 0x02, 0xB1, 0xDF, 0x6D, 0x73, 0x4D },

new byte[] { 0x7C, 0x26, 0x2E, 0xF7, 0x08, 0x5D, 0x44, 0x3E, 0x9F, 0x14, 0xC8, 0xAE, 0x54, 0x10, 0xD8, 0xBC },

new byte[] { 0x1A, 0x6B, 0x69, 0xF3, 0xBD, 0x33, 0xAB, 0xFA, 0xD1, 0x9B, 0x68, 0x4E, 0x16, 0x95, 0x91, 0xEE },

new byte[] { 0x4C, 0x63, 0x8E, 0x5B, 0xCC, 0x3C, 0x19, 0xA1, 0x81, 0x49, 0x7B, 0xD9, 0x6F, 0x37, 0x60, 0xCA },

new byte[] { 0xE7, 0x2B, 0x48, 0xFD, 0x96, 0x45, 0xFC, 0x41, 0x12, 0x0D, 0x79, 0xE5, 0x89, 0x8C, 0xE3, 0x20 },

new byte[] { 0x30, 0xDC, 0xB7, 0x6C, 0x4A, 0xB5, 0x3F, 0x97, 0xD4, 0x62, 0x2D, 0x06, 0xA4, 0xA5, 0x83, 0x5F },

new byte[] { 0x2A, 0xDA, 0xC9, 0x00, 0x7E, 0xA2, 0x55, 0xBF, 0x11, 0xD5, 0x9C, 0xCF, 0x0E, 0x0A, 0x3D, 0x51 },

new byte[] { 0x7D, 0x93, 0x1B, 0xFE, 0xC4, 0x47, 0x09, 0x86, 0x0B, 0x8F, 0x9D, 0x6A, 0x07, 0xB9, 0xB0, 0x98 },

new byte[] { 0x18, 0x32, 0x71, 0x4B, 0xEF, 0x3B, 0x70, 0xA0, 0xE4, 0x40, 0xFF, 0xC3, 0xA9, 0xE6, 0x78, 0xF9 },

new byte[] { 0x8B, 0x46, 0x80, 0x1E, 0x38, 0xE1, 0xB8, 0xA8, 0xE0, 0x0C, 0x23, 0x76, 0x1D, 0x25, 0x24, 0x05 },

new byte[] { 0xF1, 0x6E, 0x94, 0x28, 0x9A, 0x84, 0xE8, 0xA3, 0x4F, 0x77, 0xD3, 0x85, 0xE2, 0x52, 0xF2, 0x82 },

new byte[] { 0x50, 0x7A, 0x2F, 0x74, 0x53, 0xB3, 0x61, 0xAF, 0x39, 0x35, 0xDE, 0xCD, 0x1F, 0x99, 0xAC, 0xAD },

new byte[] { 0x72, 0x2C, 0xDD, 0xD0, 0x87, 0xBE, 0x5E, 0xA6, 0xEC, 0x04, 0xC6, 0x03, 0x34, 0xFB, 0xDB, 0x59 },

new byte[] { 0xB6, 0xC2, 0x01, 0xF0, 0x5A, 0xED, 0xA7, 0x66, 0x21, 0x7F, 0x8A, 0x27, 0xC7, 0xC0, 0x29, 0xD7 }

},

new [] {

new byte[] { 0x93, 0xD9, 0x9A, 0xB5, 0x98, 0x22, 0x45, 0xFC, 0xBA, 0x6A, 0xDF, 0x02, 0x9F, 0xDC, 0x51, 0x59 },

new byte[] { 0x4A, 0x17, 0x2B, 0xC2, 0x94, 0xF4, 0xBB, 0xA3, 0x62, 0xE4, 0x71, 0xD4, 0xCD, 0x70, 0x16, 0xE1 },

new byte[] { 0x49, 0x3C, 0xC0, 0xD8, 0x5C, 0x9B, 0xAD, 0x85, 0x53, 0xA1, 0x7A, 0xC8, 0x2D, 0xE0, 0xD1, 0x72 },

new byte[] { 0xA6, 0x2C, 0xC4, 0xE3, 0x76, 0x78, 0xB7, 0xB4, 0x09, 0x3B, 0x0E, 0x41, 0x4C, 0xDE, 0xB2, 0x90 },

new byte[] { 0x25, 0xA5, 0xD7, 0x03, 0x11, 0x00, 0xC3, 0x2E, 0x92, 0xEF, 0x4E, 0x12, 0x9D, 0x7D, 0xCB, 0x35 },

new byte[] { 0x10, 0xD5, 0x4F, 0x9E, 0x4D, 0xA9, 0x55, 0xC6, 0xD0, 0x7B, 0x18, 0x97, 0xD3, 0x36, 0xE6, 0x48 },

new byte[] { 0x56, 0x81, 0x8F, 0x77, 0xCC, 0x9C, 0xB9, 0xE2, 0xAC, 0xB8, 0x2F, 0x15, 0xA4, 0x7C, 0xDA, 0x38 },

new byte[] { 0x1E, 0x0B, 0x05, 0xD6, 0x14, 0x6E, 0x6C, 0x7E, 0x66, 0xFD, 0xB1, 0xE5, 0x60, 0xAF, 0x5E, 0x33 },

new byte[] { 0x87, 0xC9, 0xF0, 0x5D, 0x6D, 0x3F, 0x88, 0x8D, 0xC7, 0xF7, 0x1D, 0xE9, 0xEC, 0xED, 0x80, 0x29 },

new byte[] { 0x27, 0xCF, 0x99, 0xA8, 0x50, 0x0F, 0x37, 0x24, 0x28, 0x30, 0x95, 0xD2, 0x3E, 0x5B, 0x40, 0x83 },

new byte[] { 0xB3, 0x69, 0x57, 0x1F, 0x07, 0x1C, 0x8A, 0xBC, 0x20, 0xEB, 0xCE, 0x8E, 0xAB, 0xEE, 0x31, 0xA2 },

new byte[] { 0x73, 0xF9, 0xCA, 0x3A, 0x1A, 0xFB, 0x0D, 0xC1, 0xFE, 0xFA, 0xF2, 0x6F, 0xBD, 0x96, 0xDD, 0x43 },

new byte[] { 0x52, 0xB6, 0x08, 0xF3, 0xAE, 0xBE, 0x19, 0x89, 0x32, 0x26, 0xB0, 0xEA, 0x4B, 0x64, 0x84, 0x82 },

new byte[] { 0x6B, 0xF5, 0x79, 0xBF, 0x01, 0x5F, 0x75, 0x63, 0x1B, 0x23, 0x3D, 0x68, 0x2A, 0x65, 0xE8, 0x91 },

new byte[] { 0xF6, 0xFF, 0x13, 0x58, 0xF1, 0x47, 0x0A, 0x7F, 0xC5, 0xA7, 0xE7, 0x61, 0x5A, 0x06, 0x46, 0x44 },

new byte[] { 0x42, 0x04, 0xA0, 0xDB, 0x39, 0x86, 0x54, 0xAA, 0x8C, 0x34, 0x21, 0x8B, 0xF8, 0x0C, 0x74, 0x67 }

},

new [] {

new byte[] {0x68, 0x8D, 0xCA, 0x4D, 0x73, 0x4B, 0x4E, 0x2A, 0xD4, 0x52, 0x26, 0xB3, 0x54, 0x1E, 0x19, 0x1F },

new byte[] {0x22, 0x03, 0x46, 0x3D, 0x2D, 0x4A, 0x53, 0x83, 0x13, 0x8A, 0xB7, 0xD5, 0x25, 0x79, 0xF5, 0xBD },

new byte[] {0x58, 0x2F, 0x0D, 0x02, 0xED, 0x51, 0x9E, 0x11, 0xF2, 0x3E, 0x55, 0x5E, 0xD1, 0x16, 0x3C, 0x66 },

new byte[] {0x70, 0x5D, 0xF3, 0x45, 0x40, 0xCC, 0xE8, 0x94, 0x56, 0x08, 0xCE, 0x1A, 0x3A, 0xD2, 0xE1, 0xDF },

new byte[] {0xB5, 0x38, 0x6E, 0x0E, 0xE5, 0xF4, 0xF9, 0x86, 0xE9, 0x4F, 0xD6, 0x85, 0x23, 0xCF, 0x32, 0x99 },

new byte[] {0x31, 0x14, 0xAE, 0xEE, 0xC8, 0x48, 0xD3, 0x30, 0xA1, 0x92, 0x41, 0xB1, 0x18, 0xC4, 0x2C, 0x71 },

new byte[] {0x72, 0x44, 0x15, 0xFD, 0x37, 0xBE, 0x5F, 0xAA, 0x9B, 0x88, 0xD8, 0xAB, 0x89, 0x9C, 0xFA, 0x60 },

new byte[] {0xEA, 0xBC, 0x62, 0x0C, 0x24, 0xA6, 0xA8, 0xEC, 0x67, 0x20, 0xDB, 0x7C, 0x28, 0xDD, 0xAC, 0x5B },

new byte[] {0x34, 0x7E, 0x10, 0xF1, 0x7B, 0x8F, 0x63, 0xA0, 0x05, 0x9A, 0x43, 0x77, 0x21, 0xBF, 0x27, 0x09 },

new byte[] {0xC3, 0x9F, 0xB6, 0xD7, 0x29, 0xC2, 0xEB, 0xC0, 0xA4, 0x8B, 0x8C, 0x1D, 0xFB, 0xFF, 0xC1, 0xB2 },

new byte[] {0x97, 0x2E, 0xF8, 0x65, 0xF6, 0x75, 0x07, 0x04, 0x49, 0x33, 0xE4, 0xD9, 0xB9, 0xD0, 0x42, 0xC7 },

new byte[] {0x6C, 0x90, 0x00, 0x8E, 0x6F, 0x50, 0x01, 0xC5, 0xDA, 0x47, 0x3F, 0xCD, 0x69, 0xA2, 0xE2, 0x7A },

new byte[] {0xA7, 0xC6, 0x93, 0x0F, 0x0A, 0x06, 0xE6, 0x2B, 0x96, 0xA3, 0x1C, 0xAF, 0x6A, 0x12, 0x84, 0x39 },

new byte[] {0xE7, 0xB0, 0x82, 0xF7, 0xFE, 0x9D, 0x87, 0x5C, 0x81, 0x35, 0xDE, 0xB4, 0xA5, 0xFC, 0x80, 0xEF },

new byte[] {0xCB, 0xBB, 0x6B, 0x76, 0xBA, 0x5A, 0x7D, 0x78, 0x0B, 0x95, 0xE3, 0xAD, 0x74, 0x98, 0x3B, 0x36 },

new byte[] {0x64, 0x6D, 0xDC, 0xF0, 0x59, 0xA9, 0x4C, 0x17, 0x7F, 0x91, 0xB8, 0xC9, 0x57, 0x1B, 0xE0, 0x61 }

}

};

public static readonly byte[][][] ΠRev =

{

new[]

{

new byte[] {0xA4, 0xA2, 0xA9, 0xC5, 0x4E, 0xC9, 0x03, 0xD9, 0x7E, 0x0F, 0xD2, 0xAD, 0xE7, 0xD3, 0x27, 0x5B},

new byte[] {0xE3, 0xA1, 0xE8, 0xE6, 0x7C, 0x2A, 0x55, 0x0C, 0x86, 0x39, 0xD7, 0x8D, 0xB8, 0x12, 0x6F, 0x28},

new byte[] {0xCD, 0x8A, 0x70, 0x56, 0x72, 0xF9, 0xBF, 0x4F, 0x73, 0xE9, 0xF7, 0x57, 0x16, 0xAC, 0x50, 0xC0},

new byte[] {0x9D, 0xB7, 0x47, 0x71, 0x60, 0xC4, 0x74, 0x43, 0x6C, 0x1F, 0x93, 0x77, 0xDC, 0xCE, 0x20, 0x8C},

new byte[] {0x99, 0x5F, 0x44, 0x01, 0xF5, 0x1E, 0x87, 0x5E, 0x61, 0x2C, 0x4B, 0x1D, 0x81, 0x15, 0xF4, 0x23},

new byte[] {0xD6, 0xEA, 0xE1, 0x67, 0xF1, 0x7F, 0xFE, 0xDA, 0x3C, 0x07, 0x53, 0x6A, 0x84, 0x9C, 0xCB, 0x02},

new byte[] {0x83, 0x33, 0xDD, 0x35, 0xE2, 0x59, 0x5A, 0x98, 0xA5, 0x92, 0x64, 0x04, 0x06, 0x10, 0x4D, 0x1C},

new byte[] {0x97, 0x08, 0x31, 0xEE, 0xAB, 0x05, 0xAF, 0x79, 0xA0, 0x18, 0x46, 0x6D, 0xFC, 0x89, 0xD4, 0xC7},

new byte[] {0xFF, 0xF0, 0xCF, 0x42, 0x91, 0xF8, 0x68, 0x0A, 0x65, 0x8E, 0xB6, 0xFD, 0xC3, 0xEF, 0x78, 0x4C},

new byte[] {0xCC, 0x9E, 0x30, 0x2E, 0xBC, 0x0B, 0x54, 0x1A, 0xA6, 0xBB, 0x26, 0x80, 0x48, 0x94, 0x32, 0x7D},

new byte[] {0xA7, 0x3F, 0xAE, 0x22, 0x3D, 0x66, 0xAA, 0xF6, 0x00, 0x5D, 0xBD, 0x4A, 0xE0, 0x3B, 0xB4, 0x17},

new byte[] {0x8B, 0x9F, 0x76, 0xB0, 0x24, 0x9A, 0x25, 0x63, 0xDB, 0xEB, 0x7A, 0x3E, 0x5C, 0xB3, 0xB1, 0x29},

new byte[] {0xF2, 0xCA, 0x58, 0x6E, 0xD8, 0xA8, 0x2F, 0x75, 0xDF, 0x14, 0xFB, 0x13, 0x49, 0x88, 0xB2, 0xEC},

new byte[] {0xE4, 0x34, 0x2D, 0x96, 0xC6, 0x3A, 0xED, 0x95, 0x0E, 0xE5, 0x85, 0x6B, 0x40, 0x21, 0x9B, 0x09},

new byte[] {0x19, 0x2B, 0x52, 0xDE, 0x45, 0xA3, 0xFA, 0x51, 0xC2, 0xB5, 0xD1, 0x90, 0xB9, 0xF3, 0x37, 0xC1},

new byte[] {0x0D, 0xBA, 0x41, 0x11, 0x38, 0x7B, 0xBE, 0xD0, 0xD5, 0x69, 0x36, 0xC8, 0x62, 0x1B, 0x82, 0x8F}

},

new[]

{

new byte[] {0x83, 0xF2, 0x2A, 0xEB, 0xE9, 0xBF, 0x7B, 0x9C, 0x34, 0x96, 0x8D, 0x98, 0xB9, 0x69, 0x8C, 0x29},

new byte[] {0x3D, 0x88, 0x68, 0x06, 0x39, 0x11, 0x4C, 0x0E, 0xA0, 0x56, 0x40, 0x92, 0x15, 0xBC, 0xB3, 0xDC},

new byte[] {0x6F, 0xF8, 0x26, 0xBA, 0xBE, 0xBD, 0x31, 0xFB, 0xC3, 0xFE, 0x80, 0x61, 0xE1, 0x7A, 0x32, 0xD2},

new byte[] {0x70, 0x20, 0xA1, 0x45, 0xEC, 0xD9, 0x1A, 0x5D, 0xB4, 0xD8, 0x09, 0xA5, 0x55, 0x8E, 0x37, 0x76},

new byte[] {0xA9, 0x67, 0x10, 0x17, 0x36, 0x65, 0xB1, 0x95, 0x62, 0x59, 0x74, 0xA3, 0x50, 0x2F, 0x4B, 0xC8},

new byte[] {0xD0, 0x8F, 0xCD, 0xD4, 0x3C, 0x86, 0x12, 0x1D, 0x23, 0xEF, 0xF4, 0x53, 0x19, 0x35, 0xE6, 0x7F},

new byte[] {0x5E, 0xD6, 0x79, 0x51, 0x22, 0x14, 0xF7, 0x1E, 0x4A, 0x42, 0x9B, 0x41, 0x73, 0x2D, 0xC1, 0x5C},

new byte[] {0xA6, 0xA2, 0xE0, 0x2E, 0xD3, 0x28, 0xBB, 0xC9, 0xAE, 0x6A, 0xD1, 0x5A, 0x30, 0x90, 0x84, 0xF9},

new byte[] {0xB2, 0x58, 0xCF, 0x7E, 0xC5, 0xCB, 0x97, 0xE4, 0x16, 0x6C, 0xFA, 0xB0, 0x6D, 0x1F, 0x52, 0x99},

new byte[] {0x0D, 0x4E, 0x03, 0x91, 0xC2, 0x4D, 0x64, 0x77, 0x9F, 0xDD, 0xC4, 0x49, 0x8A, 0x9A, 0x24, 0x38},

new byte[] {0xA7, 0x57, 0x85, 0xC7, 0x7C, 0x7D, 0xE7, 0xF6, 0xB7, 0xAC, 0x27, 0x46, 0xDE, 0xDF, 0x3B, 0xD7},

new byte[] {0x9E, 0x2B, 0x0B, 0xD5, 0x13, 0x75, 0xF0, 0x72, 0xB6, 0x9D, 0x1B, 0x01, 0x3F, 0x44, 0xE5, 0x87},

new byte[] {0xFD, 0x07, 0xF1, 0xAB, 0x94, 0x18, 0xEA, 0xFC, 0x3A, 0x82, 0x5F, 0x05, 0x54, 0xDB, 0x00, 0x8B},

new byte[] {0xE3, 0x48, 0x0C, 0xCA, 0x78, 0x89, 0x0A, 0xFF, 0x3E, 0x5B, 0x81, 0xEE, 0x71, 0xE2, 0xDA, 0x2C},

new byte[] {0xB8, 0xB5, 0xCC, 0x6E, 0xA8, 0x6B, 0xAD, 0x60, 0xC6, 0x08, 0x04, 0x02, 0xE8, 0xF5, 0x4F, 0xA4},

new byte[] {0xF3, 0xC0, 0xCE, 0x43, 0x25, 0x1C, 0x21, 0x33, 0x0F, 0xAF, 0x47, 0xED, 0x66, 0x63, 0x93, 0xAA}

},

new[]

{

new byte[] {0x45, 0xD4, 0x0B, 0x43, 0xF1, 0x72, 0xED, 0xA4, 0xC2, 0x38, 0xE6, 0x71, 0xFD, 0xB6, 0x3A, 0x95},

new byte[] {0x50, 0x44, 0x4B, 0xE2, 0x74, 0x6B, 0x1E, 0x11, 0x5A, 0xC6, 0xB4, 0xD8, 0xA5, 0x8A, 0x70, 0xA3},

new byte[] {0xA8, 0xFA, 0x05, 0xD9, 0x97, 0x40, 0xC9, 0x90, 0x98, 0x8F, 0xDC, 0x12, 0x31, 0x2C, 0x47, 0x6A},

new byte[] {0x99, 0xAE, 0xC8, 0x7F, 0xF9, 0x4F, 0x5D, 0x96, 0x6F, 0xF4, 0xB3, 0x39, 0x21, 0xDA, 0x9C, 0x85},

new byte[] {0x9E, 0x3B, 0xF0, 0xBF, 0xEF, 0x06, 0xEE, 0xE5, 0x5F, 0x20, 0x10, 0xCC, 0x3C, 0x54, 0x4A, 0x52},

new byte[] {0x94, 0x0E, 0xC0, 0x28, 0xF6, 0x56, 0x60, 0xA2, 0xE3, 0x0F, 0xEC, 0x9D, 0x24, 0x83, 0x7E, 0xD5},

new byte[] {0x7C, 0xEB, 0x18, 0xD7, 0xCD, 0xDD, 0x78, 0xFF, 0xDB, 0xA1, 0x09, 0xD0, 0x76, 0x84, 0x75, 0xBB},

new byte[] {0x1D, 0x1A, 0x2F, 0xB0, 0xFE, 0xD6, 0x34, 0x63, 0x35, 0xD2, 0x2A, 0x59, 0x6D, 0x4D, 0x77, 0xE7},

new byte[] {0x8E, 0x61, 0xCF, 0x9F, 0xCE, 0x27, 0xF5, 0x80, 0x86, 0xC7, 0xA6, 0xFB, 0xF8, 0x87, 0xAB, 0x62},

new byte[] {0x3F, 0xDF, 0x48, 0x00, 0x14, 0x9A, 0xBD, 0x5B, 0x04, 0x92, 0x02, 0x25, 0x65, 0x4C, 0x53, 0x0C},

new byte[] {0xF2, 0x29, 0xAF, 0x17, 0x6C, 0x41, 0x30, 0xE9, 0x93, 0x55, 0xF7, 0xAC, 0x68, 0x26, 0xC4, 0x7D},

new byte[] {0xCA, 0x7A, 0x3E, 0xA0, 0x37, 0x03, 0xC1, 0x36, 0x69, 0x66, 0x08, 0x16, 0xA7, 0xBC, 0xC5, 0xD3},

new byte[] {0x22, 0xB7, 0x13, 0x46, 0x32, 0xE8, 0x57, 0x88, 0x2B, 0x81, 0xB2, 0x4E, 0x64, 0x1C, 0xAA, 0x91},

new byte[] {0x58, 0x2E, 0x9B, 0x5C, 0x1B, 0x51, 0x73, 0x42, 0x23, 0x01, 0x6E, 0xF3, 0x0D, 0xBE, 0x3D, 0x0A},

new byte[] {0x2D, 0x1F, 0x67, 0x33, 0x19, 0x7B, 0x5E, 0xEA, 0xDE, 0x8B, 0xCB, 0xA9, 0x8C, 0x8D, 0xAD, 0x49},

new byte[] {0x82, 0xE4, 0xBA, 0xC3, 0x15, 0xD1, 0xE0, 0x89, 0xFC, 0xB1, 0xB9, 0xB5, 0x07, 0x79, 0xB8, 0xE1}

},

new[]

{

new byte[] {0xB2, 0xB6, 0x23, 0x11, 0xA7, 0x88, 0xC5, 0xA6, 0x39, 0x8F, 0xC4, 0xE8, 0x73, 0x22, 0x43, 0xC3},

new byte[] {0x82, 0x27, 0xCD, 0x18, 0x51, 0x62, 0x2D, 0xF7, 0x5C, 0x0E, 0x3B, 0xFD, 0xCA, 0x9B, 0x0D, 0x0F},

new byte[] {0x79, 0x8C, 0x10, 0x4C, 0x74, 0x1C, 0x0A, 0x8E, 0x7C, 0x94, 0x07, 0xC7, 0x5E, 0x14, 0xA1, 0x21},

new byte[] {0x57, 0x50, 0x4E, 0xA9, 0x80, 0xD9, 0xEF, 0x64, 0x41, 0xCF, 0x3C, 0xEE, 0x2E, 0x13, 0x29, 0xBA},

new byte[] {0x34, 0x5A, 0xAE, 0x8A, 0x61, 0x33, 0x12, 0xB9, 0x55, 0xA8, 0x15, 0x05, 0xF6, 0x03, 0x06, 0x49},

new byte[] {0xB5, 0x25, 0x09, 0x16, 0x0C, 0x2A, 0x38, 0xFC, 0x20, 0xF4, 0xE5, 0x7F, 0xD7, 0x31, 0x2B, 0x66},

new byte[] {0x6F, 0xFF, 0x72, 0x86, 0xF0, 0xA3, 0x2F, 0x78, 0x00, 0xBC, 0xCC, 0xE2, 0xB0, 0xF1, 0x42, 0xB4},

new byte[] {0x30, 0x5F, 0x60, 0x04, 0xEC, 0xA5, 0xE3, 0x8B, 0xE7, 0x1D, 0xBF, 0x84, 0x7B, 0xE6, 0x81, 0xF8},

new byte[] {0xDE, 0xD8, 0xD2, 0x17, 0xCE, 0x4B, 0x47, 0xD6, 0x69, 0x6C, 0x19, 0x99, 0x9A, 0x01, 0xB3, 0x85},

new byte[] {0xB1, 0xF9, 0x59, 0xC2, 0x37, 0xE9, 0xC8, 0xA0, 0xED, 0x4F, 0x89, 0x68, 0x6D, 0xD5, 0x26, 0x91},

new byte[] {0x87, 0x58, 0xBD, 0xC9, 0x98, 0xDC, 0x75, 0xC0, 0x76, 0xF5, 0x67, 0x6B, 0x7E, 0xEB, 0x52, 0xCB},

new byte[] {0xD1, 0x5B, 0x9F, 0x0B, 0xDB, 0x40, 0x92, 0x1A, 0xFA, 0xAC, 0xE4, 0xE1, 0x71, 0x1F, 0x65, 0x8D},

new byte[] {0x97, 0x9E, 0x95, 0x90, 0x5D, 0xB7, 0xC1, 0xAF, 0x54, 0xFB, 0x02, 0xE0, 0x35, 0xBB, 0x3A, 0x4D},

new byte[] {0xAD, 0x2C, 0x3D, 0x56, 0x08, 0x1B, 0x4A, 0x93, 0x6A, 0xAB, 0xB8, 0x7A, 0xF2, 0x7D, 0xDA, 0x3F},

new byte[] {0xFE, 0x3E, 0xBE, 0xEA, 0xAA, 0x44, 0xC6, 0xD0, 0x36, 0x48, 0x70, 0x96, 0x77, 0x24, 0x53, 0xDF},

new byte[] {0xF3, 0x83, 0x28, 0x32, 0x45, 0x1E, 0xA4, 0xD3, 0xA2, 0x46, 0x6E, 0x9C, 0xDD, 0x63, 0xD4, 0x9D}

}

};

public static readonly byte[][] Mds = {

new byte[]{ 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04 },

new byte[]{ 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07 },

new byte[]{ 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06 },

new byte[]{ 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08 },

new byte[]{ 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01 },

new byte[]{ 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05 },

new byte[]{ 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01 },

new byte[]{ 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01 }

};

//public static readonly byte[][] MdsRev = {

// new byte[]{ 0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA },

// new byte[]{ 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA, 0xAD },

// new byte[]{ 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA, 0xAD, 0x95 },

// new byte[]{ 0xA8, 0x2F, 0x49, 0xD7, 0xCA, 0xAD, 0x95, 0x76 },

// new byte[]{ 0x2F, 0x49, 0xD7, 0xCA, 0xAD, 0x95, 0x76, 0xA8 },

// new byte[]{ 0x49, 0xD7, 0xCA, 0xAD, 0x95, 0x76, 0xA8, 0x2F },

// new byte[]{ 0xD7, 0xCA, 0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49 },

// new byte[]{ 0xCA, 0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7 }

//};

public static readonly byte[][] MdsRev = {

new byte[]{ 0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA },

new byte[]{ 0xCA, 0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7 },

new byte[]{ 0xD7, 0xCA, 0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49 },

new byte[]{ 0x49, 0xD7, 0xCA, 0xAD, 0x95, 0x76, 0xA8, 0x2F },

new byte[]{ 0x2F, 0x49, 0xD7, 0xCA, 0xAD, 0x95, 0x76, 0xA8 },

new byte[]{ 0xA8, 0x2F, 0x49, 0xD7, 0xCA, 0xAD, 0x95, 0x76 },

new byte[]{ 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA, 0xAD, 0x95 },

new byte[]{ 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA, 0xAD }

};

}

}

FileEncoderDecoder.cs

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

namespace Kalyna

{

public class FileEncoderDecoder

{

public string PlainTextFileName { private get; set; }

public string EncryptedTextFileName { private get; set; }

public string DecryptedTextFileName { private get; set; }

public string KeyFileName { private get; set; }

private Random Random { get; } = new Random();

private int BlocksNumber { get; set; }

private static string GetFullFilePath(string fileName)

{

DirectoryInfo directoryInfo = Directory.GetParent(Directory.GetCurrentDirectory()).Parent;

return directoryInfo == null ? string.Empty : Path.Combine(directoryInfo.FullName, fileName);

}

private static void AddByteToBlock(ref byte[] block, byte data)

{

var newArray = new byte[block.Length + 1];

block.CopyTo(newArray, 1);

newArray[0] = data;

block = newArray;

}

private Block GetKey()

{

//var key = new Block

//{

// Data = new List<byte>

// {

// 15, 14, 13, 12, 11, 10, 9, 8,

// 7, 6, 5, 4, 3, 2, 1, 0

// }

//};

var keyFilePath = GetFullFilePath(KeyFileName);

if (!File.Exists(keyFilePath)) return null;

using (var reader = new BinaryReader(File.Open(keyFilePath, FileMode.Open)))

{

var key = reader.ReadBytes(16);

return key.Length != 16 ? null : new Block { Data = new List<byte>(key) };

}

}

public void Encode()

{

var plainFilePath = GetFullFilePath(PlainTextFileName);

var encryptedFilePath = GetFullFilePath(EncryptedTextFileName);

if (!File.Exists(plainFilePath)) return;

var algorithm = new Algorithm();

var key = GetKey();

algorithm.GenerateRoundsKeys(key);

var areAddedRandomBytes = false;

using (var reader = new BinaryReader(File.Open(plainFilePath, FileMode.Open)))

using (var writer = new BinaryWriter(File.Open(encryptedFilePath, FileMode.Create)))

{

var watch = System.Diagnostics.Stopwatch.StartNew();

var fileSize = 0;

var block = reader.ReadBytes(16);

var previousBlock = block;

while (block.Length != 0)

{

fileSize += block.Length;

BlocksNumber++;

if (block.Length < 16)

{

areAddedRandomBytes = true;

var numberOfAddedBytes = (byte)(16 - block.Length);

var len = block.Length;

for (var i = 0; i < 16 - len - 1; i++)

AddByteToBlock(ref block, (byte)Random.Next(255));

AddByteToBlock(ref block, numberOfAddedBytes);

fileSize += numberOfAddedBytes;

}

var cipherText = algorithm.Encrypt(new Block

{

Data = new List<byte>(block)

}, key);

writer.Write(cipherText.Data.ToArray());

previousBlock = block;

block = reader.ReadBytes(16);

}

if (!areAddedRandomBytes && 1 <= previousBlock[0] && previousBlock[0] <= 16)

{

for (var i = 0; i < 15; i++)

AddByteToBlock(ref block, (byte)Random.Next(255));

AddByteToBlock(ref block, 16);

var cipherText = algorithm.Encrypt(new Block

{

Data = new List<byte>(block)

}, key);

writer.Write(cipherText.Data.ToArray());

}

watch.Stop();

Console.WriteLine($"Encryption completed\nTime: {watch.Elapsed}\nFile size: {fileSize} Bytes" +

$"\nSpeed: {(int)(fileSize / watch.Elapsed.TotalSeconds)} Bytes per second");

}

}

public void Decode()

{

var encryptedFilePath = GetFullFilePath(EncryptedTextFileName);

var decryptedFilePath = GetFullFilePath(DecryptedTextFileName);

if (!File.Exists(encryptedFilePath)) return;

var algorithm = new Algorithm();

var key = GetKey();

algorithm.GenerateRoundsKeys(key);

using (var reader = new BinaryReader(File.Open(encryptedFilePath, FileMode.Open)))

using (var writer = new BinaryWriter(File.Open(decryptedFilePath, FileMode.Create)))

{

var watch = System.Diagnostics.Stopwatch.StartNew();

var fileSize = 0;

var block = reader.ReadBytes(16);

var nextBlock = reader.ReadBytes(16);

while (block.Length != 0)

{

fileSize += block.Length;

var plainText = algorithm.Decrypt(new Block

{

Data = new List<byte>(block)

}, key);

if (nextBlock.Length == 0 && plainText.Data[0] <= 16)

{

var numberOfAddedBytes = plainText.Data[0];

if (numberOfAddedBytes != 16)

writer.Write(plainText.Data.Where((d, idx) => numberOfAddedBytes <= idx).ToArray());

}

else

writer.Write(plainText.Data.ToArray());

block = nextBlock;

nextBlock = reader.ReadBytes(16);

}

watch.Stop();

Console.WriteLine($"\nDecryption completed\nTime: {watch.Elapsed}\nFile size: {fileSize} Bytes" +

$"\nSpeed: {(int)(fileSize / watch.Elapsed.TotalSeconds)} Bytes per second");

}

}

}

}

Algorithm.cs

using System;

using System.Collections.Generic;

using System.Numerics;

namespace Kalyna

{

public class Algorithm

{

public bool UseLog { get; set; } = false;

private List<Block> RoundsKeys { get; } = new List<Block>();

private static void Log(string message, Block block)

{

Console.WriteLine($"{message,-30} {new BigInteger(block.Data.ToArray()).ToString("X32")}");

}

private Block GenerateKt(Block key)

{

var kt = new Block

{

Data = new List<byte>

{

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5

}

};

kt.AddRoundKey(key);

if (UseLog)

Log("state[0].add\_rkey:", kt);

kt.SubBytes(StaticTables.Π);

if (UseLog)

Log("state[0].s\_box:", kt);

kt.ShiftRows();

if (UseLog)

Log("state[0].s\_row:", kt);

kt.MixColumns(StaticTables.Mds);

if (UseLog)

Log("state[0].m\_col:", kt);

kt.Xor(key);

if (UseLog)

Log("state[0].xor\_rkey:", kt);

kt.SubBytes(StaticTables.Π);

if (UseLog)

Log("state[0].s\_box:", kt);

kt.ShiftRows();

if (UseLog)

Log("state[0].s\_row:", kt);

kt.MixColumns(StaticTables.Mds);

if (UseLog)

Log("state[0].m\_col:", kt);

kt.AddRoundKey(key);

if (UseLog)

Log("state[0].add\_rkey:", kt);

kt.SubBytes(StaticTables.Π);

if (UseLog)

Log("state[0].s\_box:", kt);

kt.ShiftRows();

if (UseLog)

Log("state[0].s\_row:", kt);

kt.MixColumns(StaticTables.Mds);

if (UseLog)

Log("state[0].m\_col:", kt);

return kt;

}

public List<Block> GenerateRoundsKeys(Block key)

{

if (UseLog)

Log("Key", key);

for (var i = 0; i <= 10; i++)

RoundsKeys.Add(new Block());

// Even keys

//var kt = new Block

//{

// Data = new List<byte>

// {

// 0x7D, 0xD8, 0xE2, 0x38, 0x2F, 0xBC, 0x5C, 0xD0,

// 0xA1, 0x5B, 0x77, 0x3B, 0x65, 0x1F, 0x2F, 0x86

// }

//};

var kt = GenerateKt(key);

if (UseLog)

Log("KT", kt);

for (var i = 0; i <= 10; i += 2)

{

if (UseLog)

Console.WriteLine();

var roundKey = RoundsKeys[i];

roundKey.Data = new List<byte>(StaticTables.V);

roundKey.ShiftLeft(i / 2);

if (UseLog)

Log($"state[{i}].ShiftLeft (tmv):", roundKey);

var keyCopy = new Block(key);

keyCopy.RotateRight(32 \* i);

if (UseLog)

Log($"state[{i}].Rotate (id):", keyCopy);

roundKey.AddRoundKey(kt);

if (UseLog)

Log($"state[{i}].add\_rkey (tmv):", roundKey);

var copy = new Block(roundKey);

roundKey.AddRoundKey(keyCopy);

if (UseLog)

Log($"state[{i}].add\_rkey (kt\_round):", roundKey);

roundKey.SubBytes(StaticTables.Π);

if (UseLog)

Log($"state[{i}].s\_box:", roundKey);

roundKey.ShiftRows();

if (UseLog)

Log($"state[{i}].s\_row:", roundKey);

roundKey.MixColumns(StaticTables.Mds);

if (UseLog)

Log($"state[{i}].m\_col:", roundKey);

roundKey.Xor(copy);

if (UseLog)

Log($"state[{i}].xor\_rkey (kt\_round):", roundKey);

roundKey.SubBytes(StaticTables.Π);

if (UseLog)

Log($"state[{i}].s\_box:", roundKey);

roundKey.ShiftRows();

if (UseLog)

Log($"state[{i}].s\_row:", roundKey);

roundKey.MixColumns(StaticTables.Mds);

if (UseLog)

Log($"state[{i}].m\_col:", roundKey);

roundKey.AddRoundKey(copy);

if (UseLog)

{

Log($"state[{i}].add\_rkey (tmv):", copy);

Log($"state[{i}].add\_rkey (kt\_round):", roundKey);

}

RoundsKeys[i] = roundKey;

}

// Odd keys

for (var i = 1; i <= 10; i += 2)

{

RoundsKeys[i].Data = RoundsKeys[i - 1].Data;

if (i == 7)

{

}

RoundsKeys[i].RotateLeft(56);

}

return RoundsKeys;

}

public Block Encrypt(Block plainText, Block key)

{

if (RoundsKeys.Count == 0)

GenerateRoundsKeys(key);

var cipherText = new Block(plainText);

cipherText.AddRoundKey(RoundsKeys[0]);

for (var i = 1; i <= 9; i++)

{

if (UseLog)

Console.WriteLine();

cipherText.SubBytes(StaticTables.Π);

if (UseLog)

Log($"round[{i}].s\_box:", cipherText);

cipherText.ShiftRows();

if (UseLog)

Log($"round[{i}].s\_row:", cipherText);

cipherText.MixColumns(StaticTables.Mds);

if (UseLog)

Log($"round[{i}].m\_col:", cipherText);

cipherText.Xor(RoundsKeys[i]);

if (UseLog)

Log($"round[{i}].xor\_rkey:", cipherText);

}

if (UseLog)

Console.WriteLine();

cipherText.SubBytes(StaticTables.Π);

if (UseLog)

Log("round[10].s\_box:", cipherText);

cipherText.ShiftRows();

if (UseLog)

Log("round[10].s\_row:", cipherText);

cipherText.MixColumns(StaticTables.Mds);

if (UseLog)

Log("round[10].m\_col:", cipherText);

cipherText.AddRoundKey(RoundsKeys[10]);

if (UseLog)

Log("round[10].add\_rkey:", cipherText);

return cipherText;

}

public Block Decrypt(Block cipherText, Block key)

{

if (RoundsKeys.Count == 0)

GenerateRoundsKeys(key);

var plainText = new Block(cipherText);

plainText.SubRoundKey(RoundsKeys[10]);

if (UseLog)

Log("round[10].sub\_rkey:", plainText);

plainText.MixColumns(StaticTables.MdsRev);

if (UseLog)

Log("round[10].m\_col:", plainText);

plainText.ShiftRowsRev();

if (UseLog)

Log("round[10].s\_row:", plainText);

plainText.SubBytes(StaticTables.ΠRev);

if (UseLog)

Log("round[10].s\_box:", plainText);

for (var i = 9; 1 <= i; --i)

{

if (UseLog)

Console.WriteLine();

plainText.Xor(RoundsKeys[i]);

if (UseLog)

Log($"round[{i}].xor\_rkey:", plainText);

plainText.MixColumns(StaticTables.MdsRev);

if (UseLog)

Log($"round[{i}].m\_col:", plainText);

plainText.ShiftRowsRev();

if (UseLog)

Log($"round[{i}].s\_row:", plainText);

plainText.SubBytes(StaticTables.ΠRev);

if (UseLog)

Log($"round[{i}].s\_box:", plainText);

}

plainText.SubRoundKey(RoundsKeys[0]);

if (UseLog)

Log("round[0].sub\_rkey:", plainText);

return plainText;

}

}

}

Program.cs

namespace Kalyna

{

internal static class Program

{

private static void Main()

{

var f = new FileEncoderDecoder

{

PlainTextFileName = "Files\\Plain.txt",

EncryptedTextFileName = "Files\\Encrypted.txt",

DecryptedTextFileName = "Files\\Decrypted.txt",

KeyFileName = "Files\\Key.txt"

};

f.Encode();

f.Decode();

}

}

}