Министерство образования Республики Беларусь Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ

И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По теме «Симметричная криптография. СТБ 34.101.31-2011»

Выполнила: студентка гр. 053501 Шурко Т.А.

Проверил: ассистент кафедры информатики Лещенко Е. А.

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc143718507)

[1 Теоретические сведения 4](#_Toc143718509)

[1.1 Стандарт шифрования СТБ 34.101.31-2011 4](#_Toc143718510)

[1.2 Блок-схема алгоритма 4](#_Toc143718511)

[2 Результат выполнения задачи 10](#_Toc143718513)

[Заключение 11](#_Toc143718515)

[Приложение А. Код программы 12](#_Toc143718517)

# ВВЕДЕНИЕ

2. Шифрование — обратимое преобразование информации в целях сокрытия от неавторизованных лиц, с предоставлением, в это же время, авторизованным пользователям доступа к ней. Главным образом, шифрование служит задачей соблюдения конфиденциальности передаваемой информации. Важной особенностью любого алгоритма шифрования является использование ключа, который утверждает выбор конкретного преобразования из совокупности возможных для данного алгоритма [1].
3. Пользователи являются авторизованными, если они обладают определенным аутентичным ключом. Вся сложность и, собственно, задача шифрования состоит в том, как именно реализован этот процесс.
4. В целом, шифрование состоит из двух составляющих — зашифрование и расшифрование.
5. С помощью шифрования обеспечиваются три состояния безопасности информации. Первой является конфиденциальность: шифрование используется для скрытия информации от неавторизованных пользователей при передаче или при хранении. Второе — целостность. Шифрование используется для предотвращения изменения информации при передаче или хранении. Последнее это идентифицируемость. Шифрование используется для аутентификации источника информации и предотвращения отказа отправителя информации от того факта, что данные были отправлены именно им.
6. Для выполнения лабораторной работы 2 необходимо реализовать программное средство зашифрования и расшифрования текстовых файлов при помощи стандартов шифрования СТБ 34.101.31-2011 в режиме простой замены.

## 1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## 1.1 [Стандарт шифрования СТБ](#_Toc143718510) 34.101.31-2011

Стандарт шифрования СТБ 34.101.31-2011 определяет семейство криптографических алгоритмов, предназначенных для обеспечения конфиденциальности и контроля целостности данных. Обрабатываемыми данными являются двоичные слова (сообщения). Криптографические алгоритмы стандарта построены на основе базовых алгоритмов шифрования блока данных.

Алгоритм шифрования в режиме простой замены предназначен для обеспечения конфиденциальности сообщений, включает алгоритм зашифрования и алгоритм расшифрования. Стороны, располагающие общим ключом, могут организовать конфиденциальный обмен сообщениями путем их зашифрования перед отправкой и расшифрования после получения. В режимах простой замены шифруются сообщения, которые содержат хотя бы один блок.

В алгоритме простой замены используется ключ 𝜃 ∈ {0, 1}256. Ключ должен вырабатываться без возможности предсказания, распространяться с соблюдением мер конфиденциальности и храниться в секрете.

## 1.2 Блок-схема алгоритма

В данном подразделе на рисунке 1 указана блок-схема выполняемого программного продукта, которая подразумевает возможность зашифрования и расшифрования исходного файла в новый файл соответственно.

На рисунках 2 и 3 указаны блок-схемы основного блока зашифрования и расшифрования, который реализует преобразование над полным словом, размер которого составляет 16 байт или 128 бит.

На рисунках 4 и 5 указаны блок-схемы алгоритма, если подаваемое на вход сообщение нельзя разбить на целое число слов размером 128 бит.

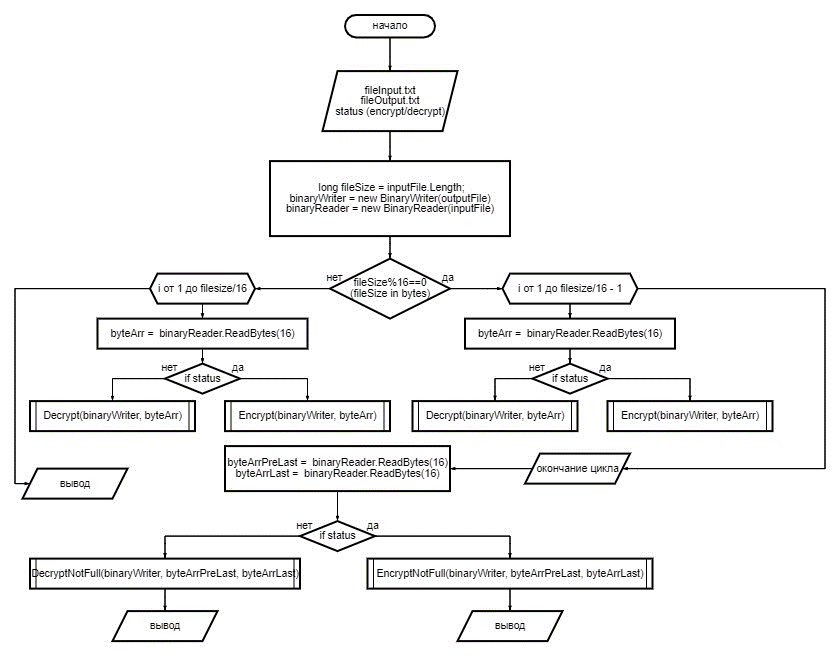


Рисунок 1 — Блок-схема выполняемой программы

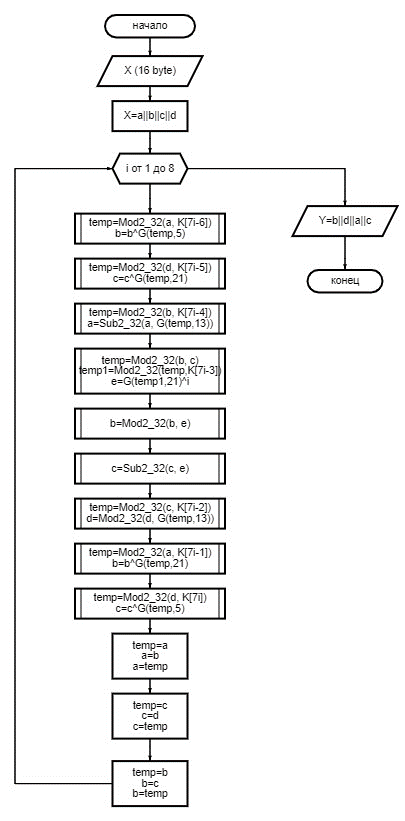


Рисунок 2 — Блок-схема зашифрования одного полного блока

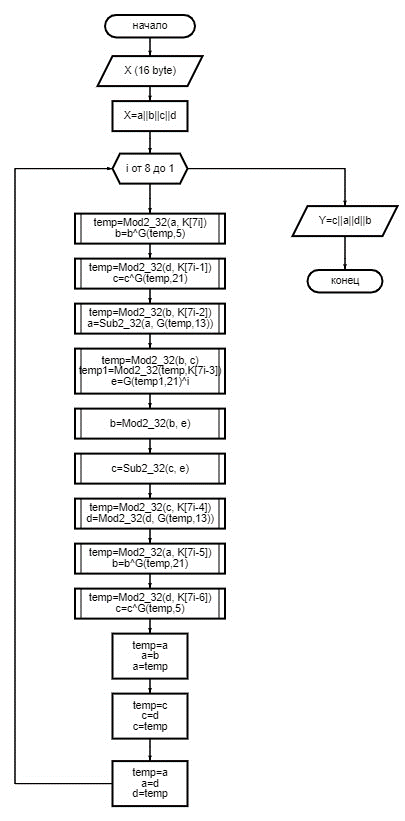


Рисунок 3 — Блок-схема расшифрования одного полного блока

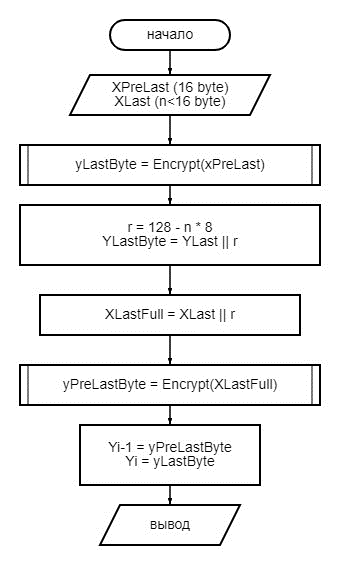


Рисунок 4 — Блок-схема зашифрования в случае неполного блока

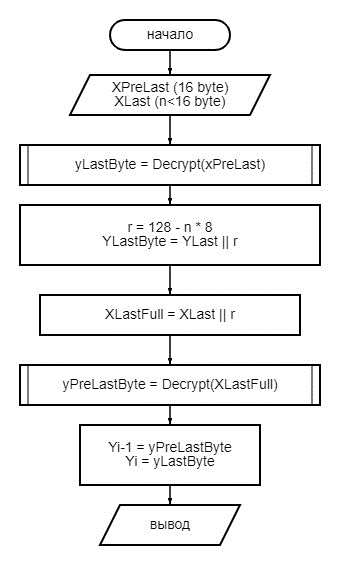


Рисунок 5 — Блок-схема расшифрования в случае неполного блока

# 2 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ЗАДАЧИ

В результате выполнения задачи был реализован алгоритм простой замены на языке программирования С#. Текст для шифрования необходимо поместить в файл и передать его наименование в виде первого параметра, текст после применения алгоритма будет помещен в файл с наименованием, указанным во втором параметре. Текст исполняемой программы в приложении А.

Для запуска программного продукта необходимо подключить класс Crypt, а также указать файл, из которого берется текст и файл, в который будет помещаться результат работы (Рисунок 6). Как параметр можно передать флаг false, который будет указывать на то, что необходимо дешифрование файла. По умолчанию флаг установлен на шифрование.

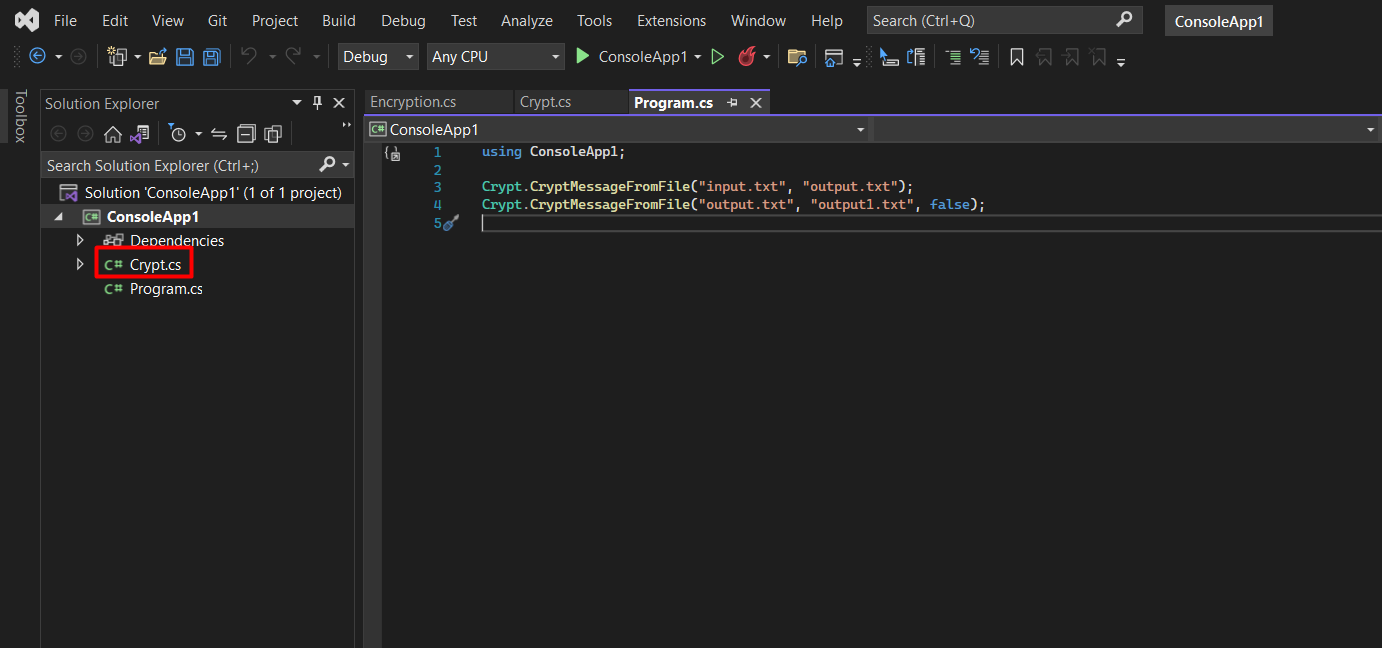


Рисунок 6 — Запускаемая программа

# ЗАКЛЮЧЕНИЕ

При выполнении лабораторной работы был изучен [стандарт шифрования СТБ 34.101.31-2011](#_Toc143718510) в режиме простой замены, рассмотрены и составлены блок-схемы алгоритма, произведена его реализация на языке программирования C#, а также проверена работоспособность программного продукта.

Стандарт СТБ 34.101.31-2011 определяет семейство криптографических алгоритмов шифрования и контроля целостности, которые используются для защиты информации при ее хранении, передаче и обработке. Настоящий стандарт применяется при разработке средств криптографической защиты информации.

Данный стандарт решает проблему разбиения на блоки для зашифрования и расшифрования, так как имеет подразумевает возможность шифрования неполных блоков.

СТБ 34.101.31 является государственным [стандарт](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82)ом [симметричного](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D1%88%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) [шифрования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и контроля целостности Республики Беларусь. Полное название стандарта — СТБ 34.101.31-2007 «Информационные технологии и безопасность. Криптографические алгоритмы шифрования и контроля целостности». Принят в качестве предварительного стандарта в 2007 году. Введён в действие в качестве окончательного стандарта в 2011 году.

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Код программы**

**Crypt.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Data.SqlTypes;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1 {

public class Crypt {

private static readonly byte[][] replacementTable = {

new byte[] { 0xB1,0x94,0xBA,0xC8,0x0A,0x08,0xF5,0x3B,0x36,0x6D,0x00,0x8E,0x58,0x4A,0x5D,0xE4 },

new byte[] { 0x85,0x04,0xFA,0x9D,0x1B,0xB6,0xC7,0xAC,0x25,0x2E,0x72,0xC2,0x02,0xFD,0xCE,0x0D },

new byte[] { 0x5B,0xE3,0xD6,0x12,0x17,0xB9,0x61,0x81,0xFE,0x67,0x86,0xAD,0x71,0x6B,0x89,0x0B },

new byte[] { 0x5C,0xB0,0xC0,0xFF,0x33,0xC3,0x56,0xB8,0x35,0xC4,0x05,0xAE,0xD8,0xE0,0x7F,0x99 },

new byte[] { 0xE1,0x2B,0xDC,0x1A,0xE2,0x82,0x57,0xEC,0x70,0x3F,0xCC,0xF0,0x95,0xEE,0x8D,0xF1 },

new byte[] { 0xC1,0xAB,0x76,0x38,0x9F,0xE6,0x78,0xCA,0xF7,0xC6,0xF8,0x60,0xD5,0xBB,0x9C,0x4F },

new byte[] { 0xF3,0x3C,0x65,0x7B,0x63,0x7C,0x30,0x6A,0xDD,0x4E,0xA7,0x79,0x9E,0xB2,0x3D,0x31 },

new byte[] { 0x3E,0x98,0xB5,0x6E,0x27,0xD3,0xBC,0xCF,0x59,0x1E,0x18,0x1F,0x4C,0x5A,0xB7,0x93 },

new byte[] { 0xE9,0xDE,0xE7,0x2C,0x8F,0x0C,0x0F,0xA6,0x2D,0xDB,0x49,0xF4,0x6F,0x73,0x96,0x47 },

new byte[] { 0x06,0x07,0x53,0x16,0xED,0x24,0x7A,0x37,0x39,0xCB,0xA3,0x83,0x03,0xA9,0x8B,0xF6 },

new byte[] { 0x92,0xBD,0x9B,0x1C,0xE5,0xD1,0x41,0x01,0x54,0x45,0xFB,0xC9,0x5E,0x4D,0x0E,0xF2 },

new byte[] { 0x68,0x20,0x80,0xAA,0x22,0x7D,0x64,0x2F,0x26,0x87,0xF9,0x34,0x90,0x40,0x55,0x11 },

new byte[] { 0xBE,0x32,0x97,0x13,0x43,0xFC,0x9A,0x48,0xA0,0x2A,0x88,0x5F,0x19,0x4B,0x09,0xA1 },

new byte[] { 0x7E,0xCD,0xA4,0xD0,0x15,0x44,0xAF,0x8C,0xA5,0x84,0x50,0xBF,0x66,0xD2,0xE8,0x8A },

new byte[] { 0xA2,0xD7,0x46,0x52,0x42,0xA8,0xDF,0xB3,0x69,0x74,0xC5,0x51,0xEB,0x23,0x29,0x21 },

new byte[] { 0xD4,0xEF,0xD9,0xB4,0x3A,0x62,0x28,0x75,0x91,0x14,0x10,0xEA,0x77,0x6C,0xDA,0x1D }

};

private static readonly byte[][] key = {

new byte[] { 0xB1,0x94,0xBA,0xC8 },

new byte[] { 0x85,0x04,0xFA,0x9D },

new byte[] { 0x5B,0xE3,0xD6,0x12 },

new byte[] { 0x5C,0xB0,0xC0,0xFF },

new byte[] { 0xE1,0x2B,0xDC,0x1A },

new byte[] { 0xC1,0xAB,0x76,0x38 },

new byte[] { 0xF3,0x3C,0x65,0x7B },

new byte[] { 0x3E,0x98,0xB5,0x6E }

};

private static byte[][] KMatrix = new byte[56][];

private static readonly int sizeOfAccumulator = 16;

private static byte[] getArray(byte[] xAccumulator, int index) {

byte[] a = new byte[sizeOfAccumulator / 4];

Array.Copy(xAccumulator, index, a, 0, sizeOfAccumulator / 4);

return a;

}

private static byte HTranformation(byte[] value) {

BitArray bitArr = new BitArray(value);

byte sByteI = 0x00;

int bitIndex = 3;

for (int i = 7; i >= 4; i--) {

if (bitArr[i]) {

sByteI |= (byte)(1 << bitIndex);

}

bitIndex--;

}

byte sByteJ = 0x00;

bitIndex = 3;

for (int i = 3; i >= 0; i--) {

if (bitArr[i]) {

sByteJ |= (byte)(1 << bitIndex);

}

bitIndex--;

}

return replacementTable[sByteI][sByteJ];

}

private static BitArray CyclicShift(BitArray bitArr, int num) {

for (int k = 0; k < num; k++) {

if (bitArr.Count > 1) {

var tmp = bitArr[bitArr.Count - 1];

for (var i = bitArr.Count - 1; i != 0; --i) bitArr[i] = bitArr[i - 1];

bitArr[0] = tmp;

}

}

return bitArr;

}

private static byte[] ConvertBitsArrayIntoByteArray(BitArray bitsArr) {

int byteArraySize = bitsArr.Count / 8;

byte[] byteArr = new byte[byteArraySize];

for (int l = 0; l < byteArraySize; l++) {

byte sByte = 0x00;

int bitIndex = 0;

for (int m = 0; m < 8; m++) {

if (bitsArr[l \* 8 + m]) {

sByte |= (byte)(1 << bitIndex);

}

bitIndex++;

}

byteArr[l] = sByte;

}

return byteArr;

}

private static BitArray GFunction(byte[] array, int index) {

byte[] arrayResult = new byte[4];

for (int i = 0; i < 4; i++) {

arrayResult[i] = HTranformation(new byte[] { array[i] });

}

BitArray bitArr = new BitArray(arrayResult);

BitArray result = CyclicShift(bitArr, index);

return result;

}

private static void fillKMatrix() {

for (int i = 0; i < 56; i++) {

KMatrix[i] = key[i%8];

}

}

private static byte[] Mod2\_32(byte[] a, byte[] b) {

byte[] res;

UInt32 A = BitConverter.ToUInt32(a, 0);

UInt32 B = BitConverter.ToUInt32(b, 0);

A += B;

res = BitConverter.GetBytes(A);

return res;

}

private static byte[] Difference(byte[] a, byte[] b) {

byte[] res;

UInt32 A = BitConverter.ToUInt32(a, 0);

UInt32 B = BitConverter.ToUInt32(b, 0);

A -= B;

res = BitConverter.GetBytes(A);

return res;

}

public static byte[] EncryptFullBlock(BinaryWriter binaryWriter, byte[] xAccumulator, bool isSave=true) {

byte[] a = getArray(xAccumulator, 0);

byte[] b = getArray(xAccumulator, sizeOfAccumulator / 4);

byte[] c = getArray(xAccumulator, (sizeOfAccumulator / 4) \* 2);

byte[] d = getArray(xAccumulator, (sizeOfAccumulator / 4) \* 3);

byte[] kTemp = new byte[4];

int index = 0;

for (int l = 1; l <= 8; l++) {

//step 1

index = 7 \* l - 6;

kTemp = KMatrix[index - 1];

byte[] modResult = Mod2\_32(kTemp, a);

BitArray bitArrResult = GFunction(modResult, 5);

BitArray bitArray = new BitArray(b);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

b = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 2

index = 7 \* l - 5;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, d);

bitArrResult = GFunction(modResult, 21);

bitArray = new BitArray(c);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

c = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 3

index = 7 \* l - 4;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, b);

bitArrResult = GFunction(modResult, 13);

byte[] resultArr = ConvertBitsArrayIntoByteArray(bitArrResult);

a = Difference(a, resultArr);

//step 4

index = 7 \* l - 3;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(b, c);

modResult = Mod2\_32(modResult, kTemp);

bitArrResult = GFunction(modResult, 21);

bitArray = new BitArray(l);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

BitArray e = bitArrResult;

//step 5

byte[] eByteArr = ConvertBitsArrayIntoByteArray(e);

b = Mod2\_32(b, eByteArr);

//step 6

c = Difference(c, eByteArr);

//step 7

index = 7 \* l - 2;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, c);

bitArrResult = GFunction(modResult, 13);

d = Mod2\_32(d, ConvertBitsArrayIntoByteArray(bitArrResult));

//step 8

index = 7 \* l - 1;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, a);

bitArrResult = GFunction(modResult, 21);

bitArray = new BitArray(b);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

b = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 9

index = 7 \* l;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, d);

bitArrResult = GFunction(modResult, 5);

bitArray = new BitArray(c);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

c = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 10-12

byte[] temp = a;

a = b;

b = temp;

temp = c;

c = d;

d = temp;

temp = b;

b = c;

c = temp;

}

byte[] resultY = new byte[a.Length \* 4];

Array.Copy(b, 0, resultY, 0, a.Length);

Array.Copy(d, 0, resultY, a.Length, a.Length);

Array.Copy(a, 0, resultY, a.Length \* 2, a.Length);

Array.Copy(c, 0, resultY, a.Length \* 3, a.Length);

if (isSave) {

binaryWriter.Write(b);

binaryWriter.Write(d);

binaryWriter.Write(a);

binaryWriter.Write(c);

}

return resultY;

}

public static byte[] DecryptFullBlock(BinaryWriter binaryWriter, byte[] xAccumulator, bool isSave=true) {

byte[] a = getArray(xAccumulator, 0);

byte[] b = getArray(xAccumulator, sizeOfAccumulator / 4);

byte[] c = getArray(xAccumulator, (sizeOfAccumulator / 4) \* 2);

byte[] d = getArray(xAccumulator, (sizeOfAccumulator / 4) \* 3);

byte[] kTemp = new byte[4];

int index = 0;

for (int l = 8; l >= 1; l--) {

//step 1

index = 7 \* l;

kTemp = KMatrix[index - 1];

byte[] modResult = Mod2\_32(kTemp, a);

BitArray bitArrResult = GFunction(modResult, 5);

BitArray bitArray = new BitArray(b);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

b = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 2

index = 7 \* l - 1;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, d);

bitArrResult = GFunction(modResult, 21);

bitArray = new BitArray(c);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

c = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 3

index = 7 \* l - 2;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, b);

bitArrResult = GFunction(modResult, 13);

byte[] resultArr = ConvertBitsArrayIntoByteArray(bitArrResult);

a = Difference(a, resultArr);

//step 4

index = 7 \* l - 3;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(b, c);

modResult = Mod2\_32(modResult, kTemp);

bitArrResult = GFunction(modResult, 21);

bitArray = new BitArray(l);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

BitArray e = bitArrResult;

//step 5

byte[] eByteArr = ConvertBitsArrayIntoByteArray(e);

b = Mod2\_32(b, eByteArr);

//step 6

c = Difference(c, eByteArr);

//step 7

index = 7 \* l - 4;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, c);

bitArrResult = GFunction(modResult, 13);

d = Mod2\_32(d, ConvertBitsArrayIntoByteArray(bitArrResult));

//step 8

index = 7 \* l - 5;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, a);

bitArrResult = GFunction(modResult, 21);

bitArray = new BitArray(b);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

b = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 9

index = 7 \* l - 6;

kTemp = KMatrix[index - 1];

modResult = Mod2\_32(kTemp, d);

bitArrResult = GFunction(modResult, 5);

bitArray = new BitArray(c);

for (int j = 0; j < bitArray.Length; j++) {

bitArrResult[j] = bitArrResult[j] ^ bitArray[j];

}

c = ConvertBitsArrayIntoByteArray(bitArrResult);

//step 10-12

byte[] temp = a;

a = b;

b = temp;

temp = c;

c = d;

d = temp;

temp = a;

a = d;

d = temp;

}

byte[] resultY = new byte[a.Length \* 4];

Array.Copy(c, 0, resultY, 0, a.Length);

Array.Copy(a, 0, resultY, a.Length, a.Length);

Array.Copy(d, 0, resultY, a.Length\*2, a.Length);

Array.Copy(b, 0, resultY, a.Length\*3, a.Length);

if (isSave) {

binaryWriter.Write(c);

binaryWriter.Write(a);

binaryWriter.Write(d);

binaryWriter.Write(b);

}

return resultY;

}

public static void EncryptNotFullBlock(BinaryWriter binaryWriter, byte[] xPreLastAccumulator, byte[] xLastAccumulator, int r) {

byte[] yLastByteArr = EncryptFullBlock(binaryWriter, xPreLastAccumulator, false);

byte[] yLast = new byte[(128 - r) / 8];

Array.Copy(yLastByteArr, yLast, yLast.Length);

byte[] rByteArr = new byte[r / 8];

Array.Copy(yLastByteArr, yLast.Length, rByteArr,0, yLastByteArr.Length - yLast.Length);

byte[] result = new byte[yLastByteArr.Length];

Array.Copy(xLastAccumulator, 0, result, 0, xLastAccumulator.Length);

Array.Copy(rByteArr, 0, result, xLastAccumulator.Length, rByteArr.Length);

byte[] yPreLast = EncryptFullBlock(binaryWriter, result, false);

binaryWriter.Write(yPreLast);

binaryWriter.Write(yLast);

}

public static void DecryptNotFullBlock(BinaryWriter binaryWriter, byte[] xPreLastAccumulator, byte[] xLastAccumulator, int r) {

byte[] yLastByteArr = DecryptFullBlock(binaryWriter, xPreLastAccumulator, false);

byte[] yLast = new byte[(128 - r) / 8];

Array.Copy(yLastByteArr, yLast, yLast.Length);

byte[] rByteArr = new byte[r / 8];

Array.Copy(yLastByteArr, yLast.Length, rByteArr, 0, yLastByteArr.Length - yLast.Length);

byte[] result = new byte[yLastByteArr.Length];

Array.Copy(xLastAccumulator, 0, result, 0, xLastAccumulator.Length);

Array.Copy(rByteArr, 0, result, xLastAccumulator.Length, rByteArr.Length);

byte[] yPreLast = DecryptFullBlock(binaryWriter, result, false);

binaryWriter.Write(yPreLast);

binaryWriter.Write(yLast);

}

public static void СryptMessageFromFile(string fileInputName, string fileOutputName, bool status = true) {

fillKMatrix();

using (FileStream outputFile = File.OpenWrite(fileOutputName)) {

using (FileStream inputFile = File.OpenRead(fileInputName)) {

long fileSize = inputFile.Length; // размер файла в байтах

Console.WriteLine(fileSize);

using (BinaryReader binaryReader = new BinaryReader(inputFile)) {

using (BinaryWriter binaryWriter = new BinaryWriter(outputFile)) {

if (fileSize % 8 == 0) {

for (int p = 0; p < fileSize / 16; p++) {

// заполнение накопителей

byte[] xAccumulator = binaryReader.ReadBytes(sizeOfAccumulator);

if (status) {

EncryptFullBlock(binaryWriter, xAccumulator);

} else {

DecryptFullBlock(binaryWriter, xAccumulator);

}

}

} else {

for (int p = 0; p < fileSize / 16 - 1; p++) {

// заполнение накопителей

byte[] xAccumulator = binaryReader.ReadBytes(sizeOfAccumulator);

if (status) {

EncryptFullBlock(binaryWriter, xAccumulator);

} else {

DecryptFullBlock(binaryWriter, xAccumulator);

}

}

byte[] xPreLast = binaryReader.ReadBytes(sizeOfAccumulator);

byte[] xLast = binaryReader.ReadBytes(sizeOfAccumulator);

int r = 128 - xLast.Length \* 8;

if (status) {

EncryptNotFullBlock(binaryWriter, xPreLast, xLast, r);

} else {

DecryptNotFullBlock(binaryWriter, xPreLast, xLast, r);

}

}

}

}

}

}

return;

}

}

}

**Program.cs**

using ConsoleApp1;

Crypt.СryptMessageFromFile("input.txt", "output.txt");

Crypt.СryptMessageFromFile("output.txt", "output1.txt", false);