Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы защиты информации

ОТЧЁТ

по лабораторной работе

на тему

Симметричная криптография. СТБ 34.101.31-2011

Выполнил

Студент гр. 053502

Юрьев В. А.

Проверил

Ассистент кафедры информатики

Лещенко Е. А.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc146017318)

[2 Теоретические сведения 3](#_Toc146017319)

[3 Блок-схема алгоритма 5](#_Toc146017320)

[4 Демонстрация работы 6](#_Toc146017321)

[5 Вывод 6](#_Toc146017322)

[ПРИЛОЖЕНИЕ А 7](#_Toc146017323)

1. **ПОСТАНОВКА ЗАДАЧИ**

Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма СТБ 34.101.31-2011 в режиме счетчика.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Настоящий стандарт 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.

Криптографические алгоритмы шифрования и контроля целостности делятся на восемь групп:

1) алгоритмы шифрования в режиме простой замены;

2) алгоритмы шифрования в режиме сцепления блоков;

3) алгоритмы шифрования в режиме гаммирования с обратной связью;

4) алгоритмы шифрования в режиме счетчика;

5) алгоритм выработки имитовставки;

6) алгоритмы одновременного шифрования и имитозащиты данных;

7) алгоритмы одновременного шифрования и имитозащиты ключа;

8) алгоритм хэширования.

Первые четыре группы предназначены для обеспечения конфиденциальности сообщений. Каждая группа 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 и сцепления блоков шифруются сообщения, которые содержат хотя бы один блок, а в режимах гаммирования с обратной связью и счетчика – сообщения произвольной длины.

Пятый алгоритм предназначен для контроля целостности сообщений с помощью имитовставок – контрольных слов, которые определяются с использованием ключа. Стороны, располагающие общим ключом, могут организовать контроль целостности при обмене сообщениями путем добавления к ним имитовставок при отправке и проверки имитовставок при получении. Проверка имитовставок дополнительно позволяет стороне-получателю убедиться в том, что сторона-отправитель знает ключ, т. е. позволяет проверить подлинность сообщений.

Шестая и седьмая группы предназначены для обеспечения конфиденциальности и контроля целостности сообщений. Каждая группа включает алгоритмы установки и снятия защиты.

В шестой группе исходное сообщение задается двумя частями: открытой и критической. Алгоритмы защиты предназначены для контроля целостности обеих частей и обеспечения конфиденциальности критической части. При установке защиты вычисляется имитовставка всего сообщения и зашифровывается его критическая часть. При снятии защиты имитовставка проверяется и, если проверка прошла успешно, критическая часть расшифровывается.

В алгоритмах седьмой группы длина защищаемого сообщения должна быть сразу известна, эти алгоритмы рекомендуется применять для защиты ключей. Защищаемый ключ сопровождается открытым заголовком, который содержит открытые атрибуты ключа и одновременно является контрольным значением при проверке целостности. Могут использоваться фиксированные постоянные заголовки, которые служат только для контроля целостности. При установке защиты ключ зашифровывается вместе со своим заголовком и формируется слово, которое является одновременно защищенным ключом и имитовставкой ключа. При снятии защиты выполняется обратное преобразование и расшифрованный заголовок сравнивается с контрольным.

Восьмой алгоритм предназначен для вычисления хэш-значений – контрольных слов, которые определяются без использования ключа. Стороны могут организовать контроль целостности сообщений путем сравнения их хэш-значений с достоверными контрольными хэш-значениями. Изменение сообщения с высокой вероятностью приводит к изменению соответствующего хэш-значения и поэтому хэш-значения могут использоваться вместо самих сообщений, например в системах электронной цифровой подписи.

1. **БЛОК-СХЕМА АЛГОРИТМА**

Схема i-й итерации алгоритма шифрования блока приведена на рисунке 3.1.

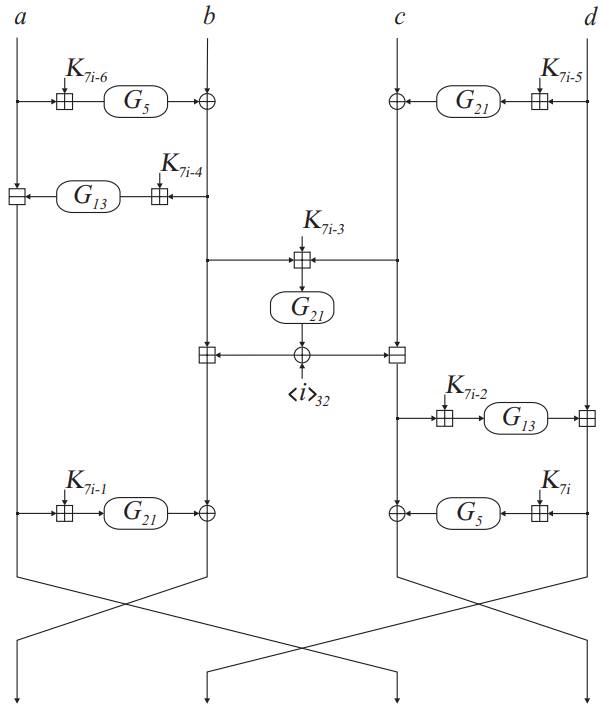


Рисунок 3.1 – Схема i-й итерации шифрования блока

1. **ДЕМОНСТРАЦИЯ РАБОТЫ**

Сообщение, которое необходимо зашифровать берется из файла input.txt. Синхропосылку или ключ можно также считать из файлов или же сгенерировать новые. После шифровки в файл encrypted\_message.txt записывается зашифрованное сообшение, а в файлы key.txt и sync\_message.txt сохраняет ключ и синхропосылку соответственно. При расшифровке входные данные считываются из описанных ранее файлов. Пример работы программы показан на рисунке 4.1.

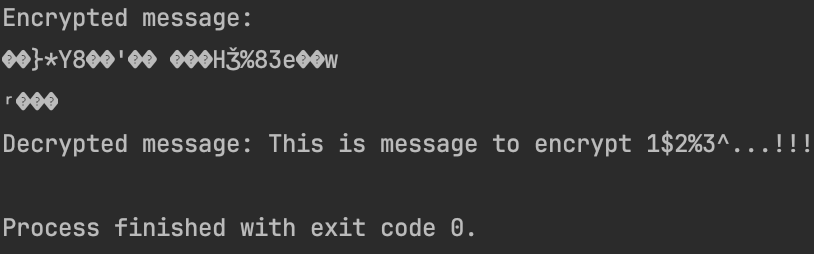


Рисунок 4.1 – Пример работы программы

1. **ВЫВОД**

В ходе выполнения данной лабораторной работы был изучен алгоритм шифрования СТБ 34.101.31-2011. Было разработано программное средство, которое шифрует и расшифровывает текстовые файлы алгоритмом СТБ 34.101.31-2011 в режиме счетчика. Алгоритм СТБ 34.101.31-2011 позволяет шифровать блоки, размеры которых кратны 128 битам. В режиме счетчика можно шифровать любой блок данных, длина которого не более 128 бит.

# **ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Код программы**

namespace lab2;

public class Cryptographer

{

private byte[,] H =

{

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

};

private static List<byte[]> SplitMessageIntoBlocks(byte[] initialBytes)

{

var blocks = new List<byte[]>();

for (var i = 0; i < Math.Ceiling((double)initialBytes.Length / 16); i++)

{

blocks.Add(initialBytes.Skip(16 \* i).Take(16).ToArray());

}

return blocks;

}

private static byte[] XOR(byte[] a, byte[] b)

{

var result = new byte[a.Length];

for (var i = 0; i < a.Length; i++)

{

result[i] = (byte)(a[i] ^ b[i]);

}

return result;

}

public byte[] CounterMode(byte[] messageBytes, byte[] key, byte[] syncMessage)

{

if (messageBytes.Length == 0)

{

throw new ArgumentException("Message length should be more then 0");

}

if (key.Length != 32)

{

throw new ArgumentException("Key should be 32 bytes (256 bits)");

}

if (syncMessage.Length != 16)

{

throw new ArgumentException("SyncMessage should be 16 bytes (128 bits)");

}

var s = F(syncMessage, key);

var X = SplitMessageIntoBlocks(messageBytes);

var Y = new List<byte[]>();

foreach (var block in X)

{

s = new byte[12].Concat(BitConverter.GetBytes(ModuloAddition(BitConverter.ToUInt32(s), 1))).ToArray();

var result = XOR(block, L(F(s, key), block.Length));

Y.Add(result);

}

return Y.SelectMany(y => y).ToArray();

}

private static byte[] L(byte[] word, int count)

{

return word.Select((b, i) => i < count ? b : (byte)0).ToArray();

}

private uint SubstitutionH(uint word)

{

var wordBytes = BitConverter.GetBytes(word);

for (var i = 0; i < 4; i++)

{

var u = (uint)wordBytes[i];

var x = (int)(u >> 4);

var y = (int)((u << 28) >> 28);

wordBytes[i] = H[x, y];

}

return BitConverter.ToUInt32(wordBytes);

}

private uint G(uint word, int r)

{

word = SubstitutionH(word);

return (word << (r % 32) & uint.MaxValue - 1) ^ (word >> (32 - r) % 32);

}

private byte[] F(byte[] block, byte[] key)

{

if (block.Length != 16)

{

throw new ArgumentException("Block size should be 16 bytes (128 bits)");

}

var a = BitConverter.ToUInt32(block[..4], 0);

var b = BitConverter.ToUInt32(block[4..8], 0);

var c = BitConverter.ToUInt32(block[8..12], 0);

var d = BitConverter.ToUInt32(block[12..16], 0);

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

{

b ^= G(ModuloAddition(a, GetKeyPart(key, 7 \* (i + 1) - 6)), 5);

c ^= G(ModuloAddition(d, GetKeyPart(key, 7 \* (i + 1) - 5)), 21);

a = ModuloSubtraction(a, G(ModuloAddition(b, GetKeyPart(key, 7 \* (i + 1) - 4)), 13));

var e = G(ModuloAddition(ModuloAddition(b, c), GetKeyPart(key, 7 \* (i + 1) - 3)), 21) ^ (uint)(i + 1);

b = ModuloAddition(b, e);

c = ModuloSubtraction(c, e);

d = ModuloAddition(d, G(ModuloAddition(c, GetKeyPart(key, 7 \* (i + 1) - 2)), 13));

b ^= G(ModuloAddition(a, GetKeyPart(key, 7 \* (i + 1) - 1)), 21);

c ^= G(ModuloAddition(d, GetKeyPart(key, 7 \* (i + 1))), 5);

(a, b) = (b, a);

(c, d) = (d, c);

(b, c) = (c, b);

}

var aBytes = BitConverter.GetBytes(a);

var bBytes = BitConverter.GetBytes(b);

var cBytes = BitConverter.GetBytes(c);

var dBytes = BitConverter.GetBytes(d);

var result = aBytes.Concat(bBytes).Concat(cBytes).Concat(dBytes).ToArray();

return result;

}

private static uint GetKeyPart(byte[] key, int part)

{

var keyPart = key.Skip((part % 8 - 1) \* 4).Take(4).ToArray();

return BitConverter.ToUInt32(keyPart, 0);

}

private static uint ModuloAddition(uint num1, uint num2)

{

return (num1 + num2) % uint.MaxValue;

}

private static uint ModuloSubtraction(uint num1, uint num2)

{

return (num1 - num2) % uint.MaxValue;

}

}