Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное образовательное   
учреждение высшего образования

«Новосибирский государственный технический университет»



Кафедра теоретической и прикладной информатики

### Лабораторная работа № 6 по дисциплине «Программные Средства Защиты Информации»

### Асимметричный алгоритм шифрования RSA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Факультет: | ПМИ |  |  |
| Группа: | ПМИМ-01 |  |  |
| Студенты: | Ершов П. К.  Малышкина Е. Д.  Слободчикова А. Э. |  |  |
| Бригада: | 2 |  |  |
| Преподаватель: | Авдеенко Т. В. |  |  |

Новосибирск

2021

1. **Цель работы**

Изучить принцип работы асимметричных алгоритмов шифрования на примере алгоритма RSA. Освоить методику создания комбинированных алгоритмов шифрования, которые совмещают достоинства методов симметричной и асимметричной криптографии.

1. **Задание**
2. Реализовать приложение для шифрования, позволяющее выполнять следующие действия:
   1. Вычислять открытый и закрытый ключи для алгоритма RSA:
   2. числа и генерируются программой или задаются из файла;
   3. числа и должны быть больше, чем 2128;
   4. Шифровать указанным в варианте симметричным алгоритмом открытый текст, а асимметричным – ключ симметричного алгоритма:
      1. ключ для симметричного алгоритма должен генерироваться случайным образом;
      2. в процессе шифрования предусмотреть возможность просмотра и изменения шифруемого текста в шестнадцатеричном и символьном виде;
      3. программа должна уметь работать с текстом произвольной длины.
3. Реализовать приложение для дешифрования:
   1. Приложение расшифровывает зашифрованный ключ с помощью алгоритма RSA, а затем с помощью симметричного алгоритма с ключом расшифровывает зашифрованный текст.
   2. В процессе дешифрования предусмотреть возможность просмотра и изменения зашифрованного текста в шестнадцатеричном и символьном виде.
   3. Программа должна уметь работать с текстом произвольной длины.
4. С помощью реализованных приложений выполнить следующие задания.
   1. Протестировать правильность работы разработанных приложений.
   2. Сделать выводы о проделанной работе.
5. **Исследования**

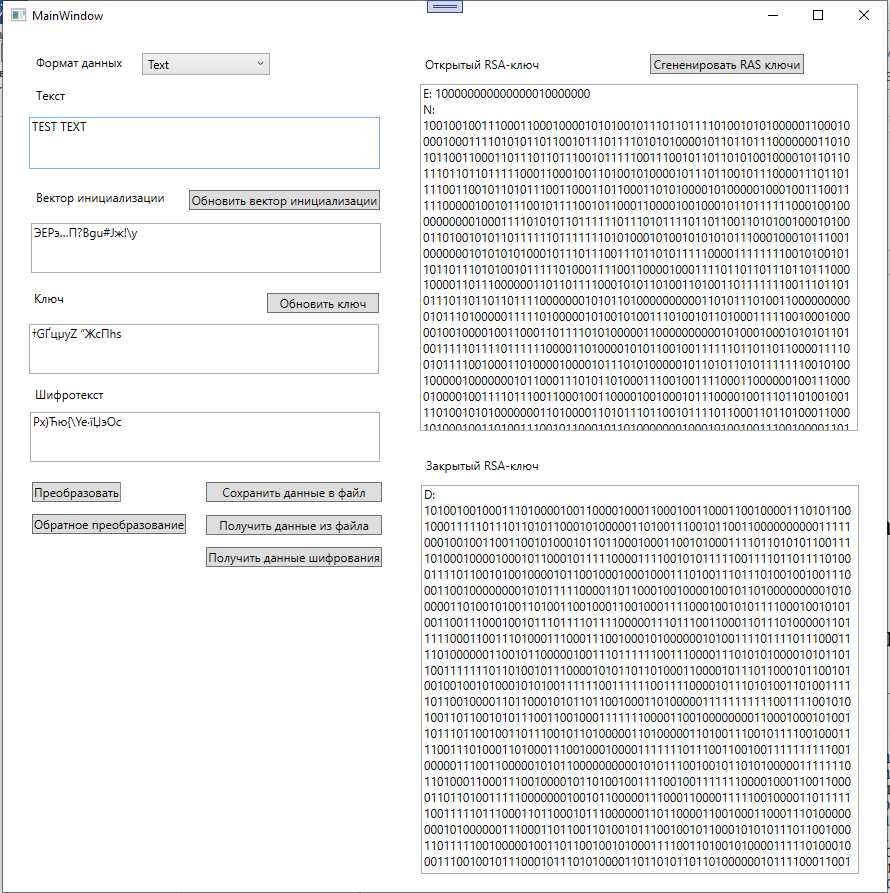


Рисунок 1. Демонстрация шифрования

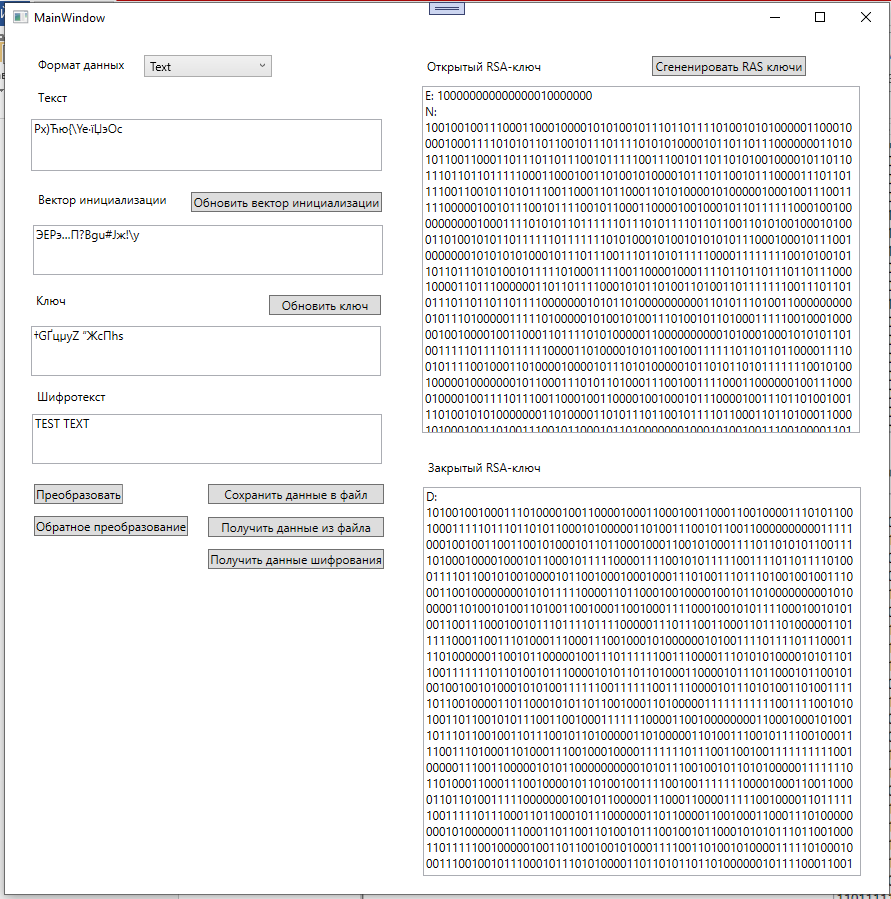


Рисунок 2. Демонстрация дешифрования

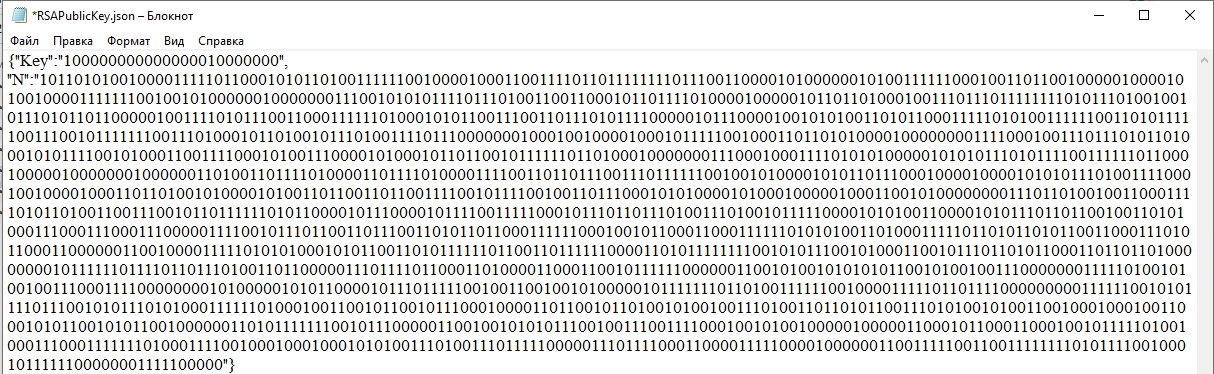


Рисунок 3. Файл с открытым RSA-ключом

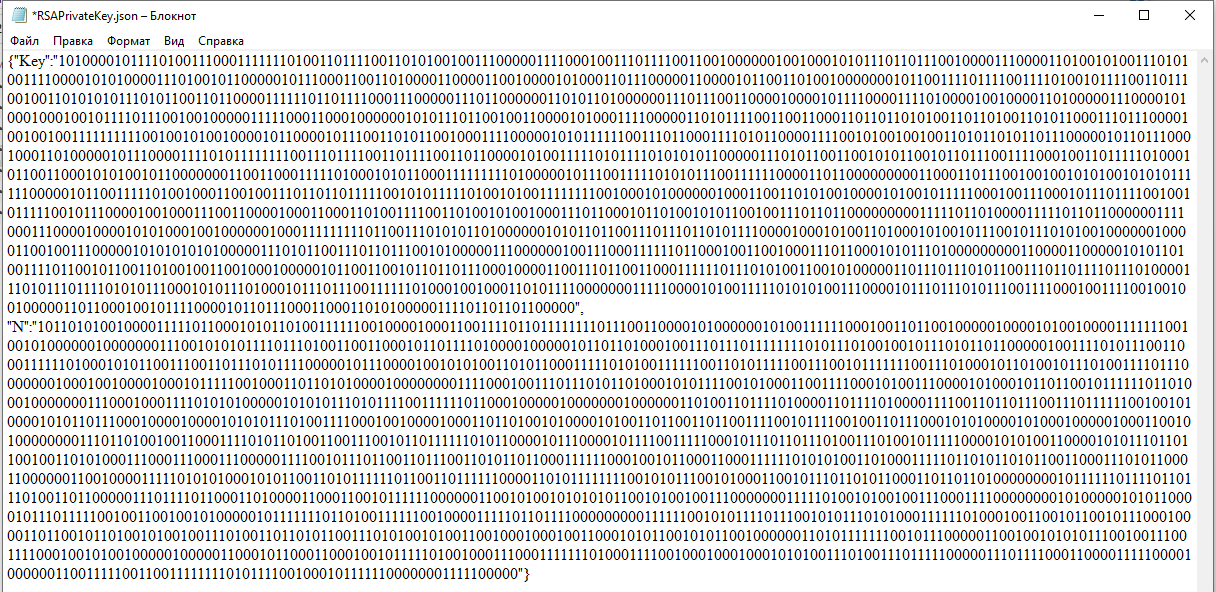


Рисунок 4. Файл с закрытым RSA-ключом

1. **Выводы**

В ходе работы разработано приложение, способное шифровать и дешифровать текст симметричным алгоритмом AES и шифровать и дешифровать ассиметричным алгоритмом RAS ключ открытого алгоритма.

1. **Код программы**

MainWindows.xaml.cs

using Lab1\_Gamming\_Srammbling.CryptoClass;

using Lab1\_Gamming\_Srammbling.Models;

using Lab1\_Gamming\_Srammbling.Utilitiets;

using System;

using System.Numerics;

using System.Windows;

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private string TextFormarFlag = "Text";

private string OldTextFormarFlag = "Text";

private string ResourceFile = "Data.json";

private string TextFile = "Text.json";

private string KeyFile = "Key.json";

private string ChiphrFile = "ChiphrText.json";

private string PUFile = "RSAPublicKey.json";

private string PVFile = "RSAPrivateKey.json";

private string Chiphrmod = "BC";

private byte[] TextArray = null;

private byte[] KeyArray = null;

private byte[] ChiphrArray = null;

private byte[] IVArray = null;

private byte[] SecKeyArray = null;

private RSAKeyClass PU, PV;

private byte[] Sk = null;

private int KeyLenght = 16;

private void Grid\_Loaded(object sender, RoutedEventArgs e)

{

}

private void TextFormat\_DropDownClosed(object sender, EventArgs e)

{

if (TextFormarFlag == "Text")

{

Text.Text = ConverteUtility.UniConvert(Text.Text, TextFormarFlag, TextFormat.Text);

Key.Text = ConverteUtility.UniConvert(Key.Text, TextFormarFlag, TextFormat.Text);

Chiphrtext.Text = ConverteUtility.UniConvert(Chiphrtext.Text, TextFormarFlag, TextFormat.Text);

}

if (TextFormarFlag == "Binary")

if (ConverteUtility.CheckIncorrectFormat(Text.Text, "Bin") && ConverteUtility.CheckIncorrectLength(Text.Text))

{

Text.Text = ConverteUtility.UniConvert(Text.Text, TextFormarFlag, TextFormat.Text);

Key.Text = ConverteUtility.UniConvert(Key.Text, TextFormarFlag, TextFormat.Text);

Chiphrtext.Text = ConverteUtility.UniConvert(Chiphrtext.Text, TextFormarFlag, TextFormat.Text);

}

else

{

MessageBox.Show("Не корректный формат");

TextFormat.SelectedIndex = 1;

}

if (TextFormarFlag == "Hexadecimal")

if (ConverteUtility.CheckIncorrectFormat(Text.Text, "Hex") && ConverteUtility.CheckIncorrectLength(Text.Text))

{

Text.Text = ConverteUtility.UniConvert(Text.Text, TextFormarFlag, TextFormat.Text);

Key.Text = ConverteUtility.UniConvert(Key.Text, TextFormarFlag, TextFormat.Text);

Chiphrtext.Text = ConverteUtility.UniConvert(Chiphrtext.Text, TextFormarFlag, TextFormat.Text);

}

else

{

MessageBox.Show("Не корректный формат");

TextFormat.SelectedIndex = 2;

}

OldTextFormarFlag = TextFormat.Text;

}

private void TextFormat\_DropDownOpened(object sender, EventArgs e)

{

TextFormarFlag = TextFormat.Text;

OldTextFormarFlag = TextFormat.Text;

}

private void CiphButton\_Click(object sender, RoutedEventArgs e)

{

var encryptResults = AESClass.Converter(TextArray, KeyArray, TextFormat.Text, "encrypt", Chiphrmod, IVArray, SecKeyArray);

var tt = RSAClass.GenerateKeyPair();

PU = tt.PublicKey;

PV = tt.PrivateKey;

Sk = RSAClass.Encrypt(KeyArray, PV);

ChiphrArray = encryptResults.chipout;

if (encryptResults.code == 0)

Chiphrtext.Text = encryptResults.output;

if (encryptResults.code == 2)

MessageBox.Show("Не корректная длина текста или ключа");

if (encryptResults.code == 1)

MessageBox.Show("Не корректный формат текста или ключа");

}

private void UpdateKey\_Click(object sender, RoutedEventArgs e)

{

KeyArray = AESClass.RandKey(KeyLenght);

if (TextFormat.Text == "Text")

{

Key.Text = ConverteUtility.ConvertByteArrayToString(AESClass.RandKey(KeyLenght));

}

if (TextFormat.Text == "Binary")

{

Key.Text = ConverteUtility.ConvertByteArraToBinaryStr(AESClass.RandKey(KeyLenght));

}

if (TextFormat.Text == "Hexadecimal")

{

Key.Text = ConverteUtility.ByteArrayToHexString(AESClass.RandKey(KeyLenght));

}

}

private void SaveFile\_Click(object sender, RoutedEventArgs e)

{

var text = new TextModel();

var chiphr = new ChiphrModel();

var key = new KeyModel();

var Pkey = new RSAKeyModel();

var Prkey = new RSAKeyModel();

text.Text = Text.Text;

key.Key = Key.Text;

chiphr.Chiphr = Chiphrtext.Text;

Pkey.Key = ConverteUtility.ConvertByteArraToBinaryStr(PU.Key.ToByteArray());

Pkey.N = ConverteUtility.ConvertByteArraToBinaryStr(PU.N.ToByteArray());

Prkey.Key = ConverteUtility.ConvertByteArraToBinaryStr(PV.Key.ToByteArray());

Prkey.N = ConverteUtility.ConvertByteArraToBinaryStr(PV.N.ToByteArray());

FileUtility.JSONSave(PUFile, FileUtility.Serialize(Pkey));

FileUtility.JSONSave(PVFile, FileUtility.Serialize(Prkey));

if (text.Text != "")

FileUtility.JSONSave(TextFile, FileUtility.Serialize(text));

else

MessageBox.Show("Добавте текст");

if (key.Key != "")

FileUtility.JSONSave(KeyFile, FileUtility.Serialize(key));

else

MessageBox.Show("Добавте ключ");

if (chiphr.Chiphr != "")

FileUtility.JSONSave(ChiphrFile, FileUtility.Serialize(chiphr));

else

MessageBox.Show("Добавте шифротекст");

}

private void FileLoad\_Click(object sender, RoutedEventArgs e)

{

Text.Clear();

Key.Clear();

Chiphrtext.Clear();

var text = FileUtility.DeserializeString<TextModel>(FileUtility.JSONSrt(TextFile));

var key = FileUtility.DeserializeString<KeyModel>(FileUtility.JSONSrt(KeyFile));

var PUkey = FileUtility.DeserializeString<RSAKeyModel>(FileUtility.JSONSrt(PUFile));

var PVkey = FileUtility.DeserializeString<RSAKeyModel>(FileUtility.JSONSrt(PVFile));

Text.Text = text.Text;

Key.Text = key.Key;

PU.Key = new BigInteger(ConverteUtility.ConvertBinaryStrToByte(PUkey.Key));

PU.N = new BigInteger(ConverteUtility.ConvertBinaryStrToByte(PUkey.N));

PV.Key = new BigInteger(ConverteUtility.ConvertBinaryStrToByte(PVkey.Key));

PV.N = new BigInteger(ConverteUtility.ConvertBinaryStrToByte(PVkey.N));

}

private void DeciphButton\_Click(object sender, RoutedEventArgs e)

{

Text.Text = Chiphrtext.Text;

TextArray = ChiphrArray;

Chiphrtext.Clear();

var deck = RSAClass.Encrypt(Sk, PU);

var encryptResults = AESClass.Converter(TextArray, deck, TextFormat.Text, "decryt", Chiphrmod, IVArray, SecKeyArray);

if (encryptResults.code == 0)

Chiphrtext.Text = encryptResults.output;

if (encryptResults.code == 2)

MessageBox.Show("Не корректная длина текста или ключа");

if (encryptResults.code == 1)

MessageBox.Show("Не корректный формат текста или ключа");

}

private void LoadChiphFile\_Click(object sender, RoutedEventArgs e)

{

Text.Clear();

Key.Clear();

Chiphrtext.Clear();

var chiphr = FileUtility.DeserializeString<ChiphrModel>(FileUtility.JSONSrt(ChiphrFile));

var key = FileUtility.DeserializeString<KeyModel>(FileUtility.JSONSrt(KeyFile));

Text.Text = chiphr.Chiphr;

Key.Text = key.Key;

}

private void Text\_TextChanged(object sender, System.Windows.Controls.TextChangedEventArgs e)

{

if (Text.Text != "" && TextFormat.Text == OldTextFormarFlag)

{

if (TextFormat.Text == "Text")

TextArray = ConverteUtility.ConvertStringToByteArray(Text.Text);

if (TextFormat.Text == "Binary")

TextArray = ConverteUtility.ConvertBinaryStrToByte(Text.Text);

if (TextFormat.Text == "Hexadecimal")

TextArray = ConverteUtility.HexStringToByteArray(Text.Text);

}

}

private void Key\_TextChanged(object sender, System.Windows.Controls.TextChangedEventArgs e)

{

if (Key.Text != "" && TextFormat.Text == OldTextFormarFlag)

{

if (TextFormat.Text == "Text")

KeyArray = ConverteUtility.ConvertStringToByteArray(Key.Text);

if (TextFormat.Text == "Binary")

KeyArray = ConverteUtility.ConvertBinaryStrToByte(Key.Text);

if (TextFormat.Text == "Hexadecimal")

KeyArray = ConverteUtility.HexStringToByteArray(Key.Text);

}

}

private void UpdateIV\_Click(object sender, RoutedEventArgs e)

{

IVArray = AESClass.RandKey(KeyLenght);

if (TextFormat.Text == "Text")

{

IV.Text = ConverteUtility.ConvertByteArrayToString(IVArray);

}

if (TextFormat.Text == "Binary")

{

IV.Text = ConverteUtility.ConvertByteArraToBinaryStr(IVArray);

}

if (TextFormat.Text == "Hexadecimal")

{

IV.Text = ConverteUtility.ByteArrayToHexString(IVArray);

}

}

}

AESClass.cs

using System;

using Lab1\_Gamming\_Srammbling.Utilitiets;

namespace Lab1\_Gamming\_Srammbling.CryptoClass

{

public class AESClass

{

private static int Nb, Nk, Nr;

private static int[] sbox = { 0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F,

0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76, 0xCA, 0x82,

0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C,

0xA4, 0x72, 0xC0, 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,

0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15, 0x04, 0xC7, 0x23,

0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27,

0xB2, 0x75, 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52,

0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84, 0x53, 0xD1, 0x00, 0xED,

0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58,

0xCF, 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9,

0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8, 0x51, 0xA3, 0x40, 0x8F, 0x92,

0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,

0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E,

0x3D, 0x64, 0x5D, 0x19, 0x73, 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A,

0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB, 0xE0,

0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62,

0x91, 0x95, 0xE4, 0x79, 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E,

0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08, 0xBA, 0x78,

0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B,

0xBD, 0x8B, 0x8A, 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,

0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E, 0xE1, 0xF8, 0x98,

0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55,

0x28, 0xDF, 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41,

0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16 };

private static int[] inv\_sbox = { 0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5,

0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB, 0x7C, 0xE3,

0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4,

0xDE, 0xE9, 0xCB, 0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D,

0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E, 0x08, 0x2E, 0xA1,

0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B,

0xD1, 0x25, 0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4,

0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92, 0x6C, 0x70, 0x48, 0x50,

0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D,

0x84, 0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4,

0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06, 0xD0, 0x2C, 0x1E, 0x8F, 0xCA,

0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,

0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF,

0xCE, 0xF0, 0xB4, 0xE6, 0x73, 0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD,

0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E, 0x47,

0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E,

0xAA, 0x18, 0xBE, 0x1B, 0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79,

0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4, 0x1F, 0xDD,

0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27,

0x80, 0xEC, 0x5F, 0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D,

0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF, 0xA0, 0xE0, 0x3B,

0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53,

0x99, 0x61, 0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1,

0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D };

private static int[] Rcon = { 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,

0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,

0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,

0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,

0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,

0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,

0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,

0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,

0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,

0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,

0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,

0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,

0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,

0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,

0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,

0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb };

public static byte[] RandKey(int N)

{

Random rnd = new Random((int)DateTime.Now.Ticks);

var bt = new Byte[N];

rnd.NextBytes(bt);

return bt;

}

private static byte[] xor\_func(byte[] a, byte[] b)

{

byte[] outp = new byte[a.Length];

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

{

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

}

return outp;

}

private static byte[,] generateSubkeys(byte[] key)

{

byte[,] tmp = new byte[Nb \* (Nr + 1), 4];

int i = 0;

while (i < Nk)

{

tmp[i, 0] = key[i \* 4];

tmp[i, 1] = key[i \* 4 + 1];

tmp[i, 2] = key[i \* 4 + 2];

tmp[i, 3] = key[i \* 4 + 3];

i++;

}

i = Nk;

while (i < Nb \* (Nr + 1))

{

byte[] temp = new byte[4];

for (int k = 0; k < 4; k++)

temp[k] = tmp[i - 1, k];

if (i % Nk == 0)

{

temp = SubWord(rotateWord(temp));

temp[0] = (byte)(temp[0] ^ (Rcon[i / Nk] & 0xff));

}

else if (Nk > 6 && i % Nk == 4)

{

temp = SubWord(temp);

}

byte[] tmp2 = new byte[4] { tmp[i - Nk, 0], tmp[i - Nk, 1], tmp[i - Nk, 2], tmp[i - Nk, 3] };

byte[] result = new byte[4];

result = xor\_func(tmp2, temp);

tmp[i, 0] = result[0];

tmp[i, 1] = result[1];

tmp[i, 2] = result[2];

tmp[i, 3] = result[3];

i++;

}

return tmp;

}

private static byte[] SubWord(byte[] inp)

{

byte[] tmp = new byte[inp.Length];

for (int i = 0; i < tmp.Length; i++)

tmp[i] = (byte)(sbox[inp[i] & 0x000000ff] & 0xff);

return tmp;

}

private static byte[] rotateWord(byte[] input)

{

byte[] tmp = new byte[input.Length];

tmp[0] = input[1];

tmp[1] = input[2];

tmp[2] = input[3];

tmp[3] = input[0];

return tmp;

}

private static byte[,] AddRoundKey(byte[,] state, byte[,] w, int round)

{

byte[,] tmp = new byte[4, 4];

for (int c = 0; c < Nb; c++)

{

for (int l = 0; l < 4; l++)

tmp[l, c] = (byte)(state[l, c] ^ w[round \* Nb + c, l]);

}

return tmp;

}

private static byte[,] SubBytes(byte[,] state)

{

byte[,] tmp = new byte[4, 4];

for (int row = 0; row < 4; row++)

for (int col = 0; col < Nb; col++)

tmp[row, col] = (byte)(sbox[(state[row, col] & 0x000000ff)] & 0xff);

return tmp;

}

private static byte[,] InvSubBytes(byte[,] state)

{

for (int row = 0; row < 4; row++)

for (int col = 0; col < Nb; col++)

state[row, col] = (byte)(inv\_sbox[(state[row, col] & 0x000000ff)] & 0xff);

return state;

}

private static byte[,] ShiftRows(byte[,] state)

{

byte[] t = new byte[4];

for (int r = 1; r < 4; r++)

{

for (int c = 0; c < Nb; c++)

t[c] = state[r, (c + r) % Nb];

for (int c = 0; c < Nb; c++)

state[r, c] = t[c];

}

return state;

}

private static byte[,] InvShiftRows(byte[,] state)

{

byte[] t = new byte[4];

for (int r = 1; r < 4; r++)

{

for (int c = 0; c < Nb; c++)

t[(c + r) % Nb] = state[r, c];

for (int c = 0; c < Nb; c++)

state[r, c] = t[c];

}

return state;

}

private static byte[,] MixColumns(byte[,] s)

{

int[] sp = new int[4];

byte b02 = (byte)0x02, b03 = (byte)0x03;

for (int c = 0; c < 4; c++)

{

sp[0] = FFMul(b02, s[0, c]) ^ FFMul(b03, s[1, c]) ^ s[2, c] ^ s[3, c];

sp[1] = s[0, c] ^ FFMul(b02, s[1, c]) ^ FFMul(b03, s[2, c]) ^ s[3, c];

sp[2] = s[0, c] ^ s[1, c] ^ FFMul(b02, s[2, c]) ^ FFMul(b03, s[3, c]);

sp[3] = FFMul(b03, s[0, c]) ^ s[1, c] ^ s[2, c] ^ FFMul(b02, s[3, c]);

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

s[i, c] = (byte)(sp[i]);

}

return s;

}

private static byte[,] InvMixColumns(byte[,] s)

{

int[] sp = new int[4];

byte b02 = (byte)0x0e, b03 = (byte)0x0b, b04 = (byte)0x0d, b05 = (byte)0x09;

for (int c = 0; c < 4; c++)

{

sp[0] = FFMul(b02, s[0, c]) ^ FFMul(b03, s[1, c]) ^ FFMul(b04, s[2, c]) ^ FFMul(b05, s[3, c]);

sp[1] = FFMul(b05, s[0, c]) ^ FFMul(b02, s[1, c]) ^ FFMul(b03, s[2, c]) ^ FFMul(b04, s[3, c]);

sp[2] = FFMul(b04, s[0, c]) ^ FFMul(b05, s[1, c]) ^ FFMul(b02, s[2, c]) ^ FFMul(b03, s[3, c]);

sp[3] = FFMul(b03, s[0, c]) ^ FFMul(b04, s[1, c]) ^ FFMul(b05, s[2, c]) ^ FFMul(b02, s[3, c]);

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

s[i, c] = (byte)(sp[i]);

}

return s;

}

public static byte FFMul(byte a, byte b)

{

byte aa = a, bb = b, r = 0, t;

while (aa != 0)

{

if ((aa & 1) != 0)

r = (byte)(r ^ bb);

t = (byte)(bb & 0x80);

bb = (byte)(bb << 1);

if (t != 0)

bb = (byte)(bb ^ 0x1b);

aa = (byte)((aa & 0xff) >> 1);

}

return r;

}

public static byte[] encryptBloc(byte[] inp, byte[,] w\_k)

{

byte[] tmp = new byte[inp.Length];

byte[,] state = new byte[4, Nb];

for (int i = 0; i < inp.Length; i++)

state[i / 4, i % 4] = inp[i % 4 \* 4 + i / 4];

state = AddRoundKey(state, w\_k, 0);

for (int round = 1; round < Nr; round++)

{

state = SubBytes(state);

state = ShiftRows(state);

state = MixColumns(state);

state = AddRoundKey(state, w\_k, round);

}

state = SubBytes(state);

state = ShiftRows(state);

state = AddRoundKey(state, w\_k, Nr);

for (int i = 0; i < tmp.Length; i++)

tmp[i % 4 \* 4 + i / 4] = state[i / 4, i % 4];

return tmp;

}

public static byte[] decryptBloc(byte[] inp, byte[,] w\_k)

{

byte[] tmp = new byte[inp.Length];

byte[,] state = new byte[4, Nb];

for (int i = 0; i < inp.Length; i++)

state[i / 4, i % 4] = inp[i % 4 \* 4 + i / 4];

state = AddRoundKey(state, w\_k, Nr);

for (int round = Nr - 1; round >= 1; round--)

{

state = InvSubBytes(state);

state = InvShiftRows(state);

state = AddRoundKey(state, w\_k, round);

state = InvMixColumns(state);

}

state = InvSubBytes(state);

state = InvShiftRows(state);

state = AddRoundKey(state, w\_k, 0);

for (int i = 0; i < tmp.Length; i++)

tmp[i % 4 \* 4 + i / 4] = state[i / 4, i % 4];

return tmp;

}

public static byte[] encryptBC(byte[] inp, byte[] key, byte[] iv = null)

{

Nb = 4;

Nk = key.Length / 4;

Nr = Nk + 6;

int lenght = 0;

var padding = new byte[1];

int i;

lenght = 16 - inp.Length % 16;

padding = new byte[lenght];

padding[0] = (byte)0x00;

for (i = 1; i < lenght; i++)

padding[i] = 0;

var tmp = new byte[inp.Length + lenght];

var bloc = new byte[16];

var f\_array = new byte[16];

Array.Copy(iv, 0, f\_array, 0, iv.Length);

var w\_k = generateSubkeys(key);

int count = 0;

for (i = 0; i < tmp.Length; i++)

{

if (i > 0 && i % 16 == 0)

{

var xorbloc = xor\_func(bloc, f\_array);

bloc = encryptBloc(xorbloc, w\_k);

Array.Copy(bloc, 0, tmp, i - 16, bloc.Length);

f\_array = xor\_func(bloc, f\_array);

}

if (i < inp.Length)

bloc[i % 16] = inp[i];

else

{

bloc[i % 16] = padding[count % 16];

count++;

}

}

if (bloc.Length == 16)

{

var xorbloc = xor\_func(bloc, f\_array);

bloc = encryptBloc(xorbloc, w\_k);

Array.Copy(bloc, 0, tmp, i - 16, bloc.Length);

f\_array = xor\_func(bloc, f\_array);

}

return tmp;

}

public static byte[] decryptBC(byte[] inp, byte[] key, byte[] iv = null)

{

int i;

var tmp = new byte[inp.Length];

var bloc = new byte[16];

Nb = 4;

Nk = key.Length / 4;

Nr = Nk + 6;

var w\_k = generateSubkeys(key);

var f\_array = new byte[16];

Array.Copy(iv, 0, f\_array, 0, iv.Length);

for (i = 0; i < inp.Length; i++)

{

if (i > 0 && i % 16 == 0)

{

var bloc\_d = decryptBloc(bloc, w\_k);

var xorbloc = xor\_func(bloc\_d, f\_array);

Array.Copy(xorbloc, 0, tmp, i - 16, bloc.Length);

f\_array = xor\_func(bloc, f\_array);

}

if (i < inp.Length)

bloc[i % 16] = inp[i];

}

bloc = decryptBloc(bloc, w\_k);

var x = xor\_func(bloc, f\_array);

Array.Copy(x, 0, tmp, i - 16, bloc.Length);

return tmp;

}

public static byte[] encryptDevisPrice(byte[] inp, byte[] key, byte[] iv = null, byte[] key2 = null)

{

Nb = 4;

Nk = key.Length / 4;

Nr = Nk + 6;

int lenght = 0;

byte[] padding = new byte[1];

int i;

lenght = 16 - inp.Length % 16;

padding = new byte[lenght];

padding[0] = (byte)0x00;

for (i = 1; i < lenght; i++)

padding[i] = 0;

byte[] tmp = new byte[inp.Length + lenght];

byte[] bloc = new byte[16];

byte[] f\_array = new byte[16];

Array.Copy(iv, 0, f\_array, 0, iv.Length);

var w\_k = generateSubkeys(key);

var w\_k\_sec = generateSubkeys(key2);

int count = 0;

for (i = 0; i < tmp.Length; i++)

{

if (i > 0 && i % 16 == 0)

{

var bloc\_first = encryptBloc(f\_array, w\_k);

var xorbloc = xor\_func(bloc\_first, bloc);

var bloc\_second = encryptBloc(xorbloc, w\_k\_sec);

Array.Copy(bloc\_second, 0, tmp, i - 16, bloc.Length);

Array.Copy(bloc\_second, 0, f\_array, 0, bloc.Length);

}

if (i < inp.Length)

bloc[i % 16] = inp[i];

else

{

bloc[i % 16] = padding[count % 16];

count++;

}

}

if (bloc.Length == 16)

{

var bloc\_first = encryptBloc(f\_array, w\_k);

var xorbloc = xor\_func(bloc\_first, bloc);

var bloc\_second = encryptBloc(xorbloc, w\_k\_sec);

Array.Copy(bloc\_second, 0, tmp, i - 16, bloc.Length);

Array.Copy(bloc\_second, 0, f\_array, 0, bloc.Length);

}

return tmp;

}

public static byte[] decryptDevisPrice(byte[] inp, byte[] key, byte[] iv = null, byte[] key2 = null)

{

int i;

byte[] tmp = new byte[inp.Length];

byte[] bloc = new byte[16];

Nb = 4;

Nk = key.Length / 4;

Nr = Nk + 6;

var w\_k = generateSubkeys(key);

var w\_k\_sec = generateSubkeys(key2);

byte[] f\_array = new byte[16];

Array.Copy(iv, 0, f\_array, 0, iv.Length);

for (i = 0; i < inp.Length; i++)

{

if (i > 0 && i % 16 == 0)

{

var bloc\_second = decryptBloc(bloc, w\_k\_sec);

var bloc\_first = encryptBloc(f\_array, w\_k);

var xorbloc = xor\_func(bloc\_second, bloc\_first);

Array.Copy(xorbloc, 0, tmp, i - 16, bloc.Length);

Array.Copy(bloc, 0, f\_array, 0, bloc.Length);

}

if (i < inp.Length)

bloc[i % 16] = inp[i];

}

var bloc\_s = decryptBloc(bloc, w\_k\_sec);

var bloc\_f = encryptBloc(f\_array, w\_k);

var x = xor\_func(bloc\_f, bloc\_s);

Array.Copy(x, 0, tmp, i - 16, bloc.Length);

return tmp;

}

private static byte[] deletePadding(byte[] input)

{

int count = 0;

int i = input.Length - 1;

while (input[i] == 0)

{

count++;

i--;

}

byte[] tmp = new byte[input.Length - count - 1];

Array.Copy(input, 0, tmp, 0, tmp.Length);

return tmp;

}

public static (string output, int code, byte[] chipout) Converter(byte[] Text, byte[] Key, string flag = "Text",

string type = "encrypt", string mod = "BC", byte[] iv = null, byte[] skey = null) //Универсальный преобразователь

{

string chiphrtext = "";

int code = 0;

byte[] tt = null;

if (flag == "Text")

{

if (type == "encrypt")

{

if (mod == "BC")

tt = encryptBC(Text, Key, iv);

else

tt = encryptDevisPrice(Text, Key, iv, skey);

}

else

{

if (mod == "BC")

tt = decryptBC(Text, Key, iv);

else

tt = decryptDevisPrice(Text, Key, iv, skey);

}

chiphrtext = ConverteUtility.ConvertByteArrayToString(tt);

}

if (flag == "Binary")

{

if (type == "encrypt")

{

if (mod == "BC")

tt = encryptBC(Text, Key, iv);

else

tt = encryptDevisPrice(Text, Key, iv, skey);

}

else

{

if (mod == "BC")

tt = decryptBC(Text, Key, iv);

else

tt = decryptDevisPrice(Text, Key, iv, skey);

}

chiphrtext = ConverteUtility.ConvertByteArraToBinaryStr(tt);

}

if (flag == "Hexadecimal")

{

if (type == "encrypt")

{

if (mod == "BC")

tt = encryptBC(Text, Key, iv);

else

tt = encryptDevisPrice(Text, Key, iv, skey);

}

else

{

if (mod == "BC")

tt = decryptBC(Text, Key, iv);

else

tt = decryptDevisPrice(Text, Key, iv, skey);

}

chiphrtext = ConverteUtility.ByteArrayToHexString(tt);

}

return (chiphrtext, code, tt);

}

public static int ChangedBits(byte[] origin, byte[] novel)

{

var origin\_str = ConverteUtility.ConvertByteArraToBinaryStr(origin);

var novel\_str = ConverteUtility.ConvertByteArraToBinaryStr(novel);

var changedBits = 0;

for (int i = 0; i < origin\_str.Length; i++)

{

if (origin\_str[i] != novel\_str[i])

changedBits++;

}

return changedBits;

}

}

}

RSAClass.cs

using System.Linq;

using System.Numerics;

using System.Security.Cryptography;

namespace Lab1\_Gamming\_Srammbling.CryptoClass

{

public static class RSAClass

{

public static BigInteger RandomBigIntInRange(BigInteger min, BigInteger max)

{

RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

if (min > max)

{

BigInteger temp = min;

min = max;

max = temp;

}

BigInteger offset = -min;

min = 0;

max += offset;

BigInteger value = RandomBigIntFromZero(rng, max) - offset;

return value;

}

public static BigInteger RandomBigIntFromZero(RandomNumberGenerator rng, BigInteger max)

{

BigInteger value;

byte[] bytes = max.ToByteArray();

byte ZeroBitsMask = 0b00000000;

byte MostSignificantByte = bytes[bytes.Length - 1];

for (int i = 7; i >= 0; i--)

{

if ((MostSignificantByte & (0b1 << i)) != 0)

{

int ZeroBits = 7 - i;

ZeroBitsMask = (byte)(0b11111111 >> ZeroBits);

break;

}

}

do

{

rng.GetBytes(bytes);

bytes[bytes.Length - 1] &= ZeroBitsMask;

value = new BigInteger(bytes);

}

while (value > max);

return value;

}

public static bool MillerRabinTest(BigInteger N, BigInteger D)

{

BigInteger a = RandomBigIntInRange(2, N - 2);

BigInteger x = BigInteger.ModPow(a, D, N);

if (x == 1 || x == N - 1)

return true;

else

return false;

}

public static bool IsPrime(BigInteger N)

{

if (N < 2)

return false;

if (N == 2 || N == 3)

return true;

if (N % 2 == 0)

return false;

BigInteger D = N - 1;

while (D % 2 == 0)

D /= 2;

for (int k = 0; k < 64; k++)

{

if (!MillerRabinTest(N, D))

return false;

}

return true;

}

public static BigInteger GetFirstPrime(BigInteger N)

{

int Limit = 10000000;

while (Limit-- > 0)

{

if (IsPrime(N))

return N;

N++;

}

return 2;

}

public static BigInteger GetLargeRandomPrime()

{

byte[] max = Enumerable.Repeat((byte)0xFF, 128).ToArray();

max[max.Length - 1] &= 0x7F;

BigInteger Bmax = new BigInteger(max);

BigInteger N = RandomBigIntInRange(Bmax / 8, Bmax);

if (IsPrime(N))

return N;

else

return GetFirstPrime(N);

}

public static BigInteger GCD(BigInteger a, BigInteger b)

{

while (a != 0 && b != 0)

{

if (a > b)

a %= b;

else

b %= a;

}

return a == 0 ? b : a;

}

public static BigInteger ModInverse(BigInteger a, BigInteger n)

{

BigInteger i = n, v = 0, d = 1;

while (a > 0)

{

BigInteger t = i / a, x = a;

a = i % x;

i = x;

x = d;

d = v - t \* x;

v = x;

}

v %= n;

if (v < 0) v = (v + n) % n;

return v;

}

public static (RSAKeyClass PublicKey, RSAKeyClass PrivateKey) GenerateKeyPair()

{

BigInteger P = GetLargeRandomPrime();

BigInteger Q = GetLargeRandomPrime();

BigInteger N = P \* Q;

BigInteger Phi = (P - 1) \* (Q - 1);

BigInteger e;

e = 65537;

while (GCD(e, Phi) != 1)

{

e = GetFirstPrime(e);

}

BigInteger d = ModInverse(e, Phi);

var PublicKey = new RSAKeyClass(e, N);

var PrivateKey = new RSAKeyClass(d, N);

return (PublicKey, PrivateKey);

}

public static byte[] Encrypt(byte[] M, RSAKeyClass EncryptionKey) => BigInteger.ModPow(new BigInteger(M), EncryptionKey.Key, EncryptionKey.N).ToByteArray();

}

}

RSAKeyClass.cs

using System.Numerics;

namespace Lab1\_Gamming\_Srammbling.CryptoClass

{

public class RSAKeyClass

{

public BigInteger Key;

public BigInteger N;

public RSAKeyClass(BigInteger Key, BigInteger N)

{

this.Key = Key;

this.N = N;

}

}

}