## 5.1 Vigenere full code

## 5.1.1 Vigenere Class:

public class VigenereCipher

{

private string key;

private char[,] table;

public VigenereCipher(string key)

{

if (key != null && key.Length > 2)

this.key = key.ToUpper();

else

this.key = ("VigenereCipherTestKey").ToUpper();

removeDuplicatesFromKey();

updateTable();

}

public string encrypt(string toEncrypt)

{

padKeyIfTooShort(toEncrypt);

key = key.ToUpper();

char[] keyChar = key.ToCharArray();

char[] messageChar = toEncrypt.ToUpper().ToCharArray();

char[] encryptedText = new char[keyChar.Length];

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

{

int keyInt = mapCharToInt(keyChar[a]);

int messageInt = mapCharToInt(messageChar[a]);

encryptedText[a] = table[keyInt, messageInt];

}

StringBuilder sb = new StringBuilder();

sb.Append(encryptedText);

return sb.ToString();

}

private int mapCharToInt(char letter)

{

return letter - 65;

}

private char mapIntToChar(int number)

{

return (char)(number + 65);

}

public string decrypt(string toDecrypt)

{

padKeyIfTooShort(toDecrypt);

char[] keyChar = key.ToUpper().ToCharArray();

char[] messageChar = toDecrypt.ToUpper().ToCharArray();

char[] decryptedText = new char[keyChar.Length];

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

{

int keyInt = mapCharToInt(keyChar[a]);

for(int b = 0; b < 26; b++)

{

if(table[keyInt, b] == messageChar[a])

decryptedText[a] = mapIntToChar(b);

}

}

StringBuilder sb = new StringBuilder();

sb.Append(decryptedText);

return sb.ToString();

}

private void padKeyIfTooShort(string message)

{

//Message: HelloWorld

//Key: Lemon

//Padded: LemonLemon

if(key.Length < message.Length)

{

//We need to pad the key

int amountToAdd = message.Length - key.Length;

for(int a = 0; a < amountToAdd; a++)

{

key += key.ToCharArray().ElementAt(a).ToString();

}

}

}

private void removeDuplicatesFromKey()

{

char[] listOfUniqueCharacters = new char[key.Length];

int count = 0;

for(int a = key.Length; a > 0; a--)

{

if (listOfUniqueCharacters.Contains(key.ToCharArray()[a - 1]))

//Already in the list, remove the duplicate

key.Remove(a - 1, 1);

else

{

listOfUniqueCharacters[count] = key.ToCharArray()[a - 1];

count++;

}

}

StringBuilder sb = new StringBuilder();

Console.WriteLine(sb.Append(listOfUniqueCharacters).ToString());

sb.Clear();

Array.Resize(ref listOfUniqueCharacters, count);

Console.WriteLine(sb.Append(listOfUniqueCharacters).ToString());

sb.Clear();

Array.Reverse(listOfUniqueCharacters);

Console.WriteLine(sb.Append(listOfUniqueCharacters).ToString());

sb.Clear();

sb.Append(listOfUniqueCharacters);

key = sb.ToString();

}

private void updateTable()

{

table = new char[26, 26];

for(int a = 0; a < 26; a++)

{

for(int b = 0; b < 26; b++)

{

if (a + b > 25)

table[a, b] = (char)((a + b) + 65 - 26);

else

table[a, b] = (char)((a + b) + 65);

}

}

}

}

## 5.1.2 Vigenere GUI code:

public partial class Vigenere : Form

{

private VigenereCipher vc;

public Vigenere()

{

InitializeComponent();

vc = new VigenereCipher("aaa");

}

private void btnEncrypt\_Click(object sender, EventArgs e)

{

//

if (tbKey.Text == "")

tbKey.Text = "aaa";

else

vc = new VigenereCipher(tbKey.Text);

tbCipher.Text = vc.encrypt(tbPlain.Text);

}

private void btnDecrypt\_Click(object sender, EventArgs e)

{

//

if (tbCipher.Text == "" || tbKey.Text == "")

MessageBox.Show("Enter the cipher text and the key.");

else

{

vc = new VigenereCipher(tbKey.Text);

tbDecrypted.Text = vc.decrypt(tbCipher.Text);

}

}

private void btnClear\_Click(object sender, EventArgs e)

{

tbCipher.Text = "";

tbDecrypted.Text = "";

tbKey.Text = "";

tbPlain.Text = "";

vc = new VigenereCipher("aaa");

}

private void tbPlain\_KeyPress(object sender, KeyPressEventArgs e)

{

if (!System.Text.RegularExpressions.Regex.IsMatch(e.KeyChar.ToString(), @"^[a-zA-Z]+$"))

e.Handled = true;

}

private void tbKey\_KeyPress(object sender, KeyPressEventArgs e)

{

if (!System.Text.RegularExpressions.Regex.IsMatch(e.KeyChar.ToString(), @"^[a-zA-Z]+$"))

e.Handled = true;

}

private void tbCipher\_KeyPress(object sender, KeyPressEventArgs e)

{

if (!System.Text.RegularExpressions.Regex.IsMatch(e.KeyChar.ToString(), @"^[a-zA-Z]+$"))

e.Handled = true;

}

}

}

## 5.2 Vernam Cipher Code

## 5.2.1 Vernam Class:

public static class VernamClass

{

private static string plainText { get; set; }

private static string cipherText { get; set; }

private static int[] key { get; set; }

private static Random keyGen { get; set; } = new Random();

//encryption of plaintext using keyGen

public static CipherClass encrypt(string plaintext)

{

cipherText = "";

//key = generateKey(plaintext.Length);

key = generateKey(plaintext.Length);

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

{

int character = plaintext[i];

cipherText += Convert.ToString(Convert.ToChar((character + (key[i]) % 26)));

}

//rewrite van voor tot agter :(

CipherClass cipher = new CipherClass(cipherText, key);

return cipher;

}

//Encrypt a file

public static CipherClass encrypt(string fileName, bool isFile)

{

DateTime start = DateTime.Now;

byte[] plainText = File.ReadAllBytes(fileName);

byte[] key = generateKey(plainText.Length, true);

byte[] cipher = new byte[plainText.Length];

cipher = exclusiveOR(plainText, key);

CipherClass cipherObj = new CipherClass(cipher, key);

ByteArrayToFile(fileName + ".vernam", cipher);

DateTime end = DateTime.Now;

Console.WriteLine(end - start);

Console.WriteLine("Encryption complete");

return cipherObj;

}

//decryption of text using Cipher datatype

public static string decrypt(CipherClass cipher)

{

if(cipher.cipher.GetType() == typeof(string))

{

cipherText = (string)cipher.cipher;

key = cipher.key;

plainText = "";

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

{

int character = cipherText[i];

plainText += Convert.ToString(Convert.ToChar(character - (key[i] % 26)));

}

return plainText;

}

return "Error";

}

public static bool decrypt(string fileName, byte[] key)

{

byte[] cipherText = File.ReadAllBytes(fileName);

byte[] plainText = new byte[cipherText.Length];

plainText = exclusiveOR(cipherText, key);

fileName = fileName.Replace(".vernam", "");

string fileExtension = fileName.Substring(fileName.LastIndexOf('.'));

ByteArrayToFile(fileName + ".decrypted" + fileExtension, plainText);

Console.WriteLine("Decryption Done");

return true;

}

//generates valid keys based on the length of the plaintext

private static int[] generateKey(int length)

{

key = new int[length];

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

{

key[i] = keyGen.Next(0, 26);

}

return key;

}

public static byte[] generateKey(int length, bool isFile)

{

byte[] key = new byte[length];

keyGen.NextBytes(key);

return key;

}

public static bool ByteArrayToFile(string fileName, byte[] byteArray)

{

try

{

using (var fs = new FileStream(fileName, FileMode.Create, FileAccess.Write))

{

fs.Write(byteArray, 0, byteArray.Length);

return true;

}

}

catch (Exception ex)

{

Console.WriteLine("Exception caught in process: {0}", ex);

return false;

}

}

public static byte[] exclusiveOR(byte[] data, byte[] key)

{

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

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

result[i] = (byte)(data[i] ^ key[i]);

return result;

}

}

}

## 5.2.2 Vernam GUI code:

public partial class Vernam : Form

{

CipherClass encryptedCipher;

string fileName;

public Vernam()

{

InitializeComponent();

}

private void btnEncrypt\_Click(object sender, EventArgs e)

{

encryptedCipher = VernamClass.encrypt(tbxPlainText.Text);

tbxCipherText.Text = (string)encryptedCipher.cipher;

string output = "";

foreach (int key in encryptedCipher.key)

{

output += key.ToString() + " ";

}

tbxKey.Text = output;

}

private void btnDecrypt\_Click(object sender, EventArgs e)

{

tbxDecryptedText.Text = VernamClass.decrypt(encryptedCipher);

}

private void btnClear\_Click(object sender, EventArgs e)

{

encryptedCipher = null;

tbxCipherText.Clear();

tbxDecryptedText.Clear();

tbxKey.Clear();

tbxPlainText.Clear();

}

private void btnSelectFile\_Click(object sender, EventArgs e)

{

OpenFileDialog ofd = new OpenFileDialog();

ofd.ShowDialog();

try

{

fileName = ofd.FileName;

tbxFileName.Text = fileName;

}

catch (Exception)

{

}

}

private void btnEncryptFile\_Click(object sender, EventArgs e)

{

try

{

pgrStatus.Value = 0;

pgrStatus.Step = 33;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Starting Encryption";

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Creating Ciphertext";

encryptedCipher = VernamClass.encrypt(fileName, true);

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Saving .key File";

ByteArrayToFile(fileName + ".key", encryptedCipher.byteKey);

pgrStatus.Value = 100;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Done";

}

catch (OutOfMemoryException)

{

MessageBox.Show("Out of Memory Exception occurred. Please select a smaller file.");

}

catch (Exception)

{

MessageBox.Show("Something went wrong.");

}

}

public static bool ByteArrayToFile(string fileName, byte[] byteArray)

{

try

{

using (var fs = new FileStream(fileName, FileMode.Create, FileAccess.Write))

{

fs.Write(byteArray, 0, byteArray.Length);

return true;

}

}

catch (Exception ex)

{

Console.WriteLine("Exception caught in process: {0}", ex);

return false;

}

}

private void btnDecryptFile\_Click(object sender, EventArgs e)

{

try

{

pgrStatus.Value = 0;

pgrStatus.Step = 33;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Starting Decryption";

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Creating Plaintext";

VernamClass.decrypt(fileName, File.ReadAllBytes(fileName.Replace(".vernam",".key")));

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Saving .decrypted File";

pgrStatus.Value = 100;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Done";

}

catch (IOException)

{

MessageBox.Show("Please ensure that the .key file is in the same directory as the encrypted file.");

}

catch (Exception)

{

MessageBox.Show("Something went wrong");

}

}

}

## 5.3 Columnar Transposition Code

## 5.3.1 Columnar Transposition class:

## public static string encrypt(string plaintext, string key)

## {

## //make all letters uppercase and remove spaces

## plaintext = plaintext.ToUpper();

## plaintext = plaintext.Replace(" ", "");

## //Sort key in alphabetical order

## char[] brokenKey = key.ToArray();

## Array.Sort(brokenKey);

## string sortedKey = new string(brokenKey);

## 

## //Calculate the number of rows needed in the matrix

## int rows = (int)Math.Ceiling((double)plaintext.Length / key.Length);

## //create the matrix

## string[,] plainTextMatrix = new string[rows, key.Length];

## int ptIndex = 0;

## //Fill matrix with data and padding

## for(int row = 0; row<rows; row++)

## {

## for(int col = 0; col<key.Length; col++)

## {

## if (ptIndex < plaintext.Length)

## {

## plainTextMatrix[row,col] = plaintext.Substring(ptIndex++, 1);

## }else

## {

## plainTextMatrix[row, col] = "|";

## }

## }

## }

## //Map sorted key indices to original key indices

## int[] alphaToKeyMap = new int[key.Length];

## char[] tempKey = key.ToCharArray();

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

## {

## int index = -1;

## for(int charEl = 0; charEl<alphaToKeyMap.Length; charEl++)

## {

## if (sortedKey[i] == tempKey[charEl])

## {

## index = charEl;

## tempKey[charEl] = '#';

## break;

## }

## }

## alphaToKeyMap[i] = index;

## }

## string cipherText = "";

## //Fill cipherText with columns selected according to previous mapping

## for (int col = 0; col < key.Length; col++)

## {

## for (int row = 0; row < rows; row++)

## {

## cipherText += plainTextMatrix[row, alphaToKeyMap[col]];

## }

## }

## return cipherText;

## }

## public static string decrypt(string ciphertext, string key)

## {

## //Sort key in alphabetical order

## char[] brokenKey = key.ToArray();

## Array.Sort(brokenKey);

## string sortedKey = new string(brokenKey);

## //get amount of rows the plain text matrix has

## int rows = (int) Math.Ceiling((double)ciphertext.Length / key.Length);

## string[,] cipherTextMatrix = new string[rows, key.Length];

## //reconstruct the key-sorted plaintext matrix

## int index = 0;

## //Fill matrix with encrypted data

## for (int col = 0; col<key.Length; col++)

## {

## for(int row = 0; row<rows; row++)

## {

## cipherTextMatrix[row, col] = Convert.ToString(ciphertext[index++]);

## }

## }

## //Map sorted key indices to original key indices

## int[] alphaToKeyMap = new int[key.Length];

## char[] tempKey = sortedKey.ToCharArray();

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

## {

## int index3 = -1;

## for (int charEl = 0; charEl < alphaToKeyMap.Length; charEl++)

## {

## if (key[i] == tempKey[charEl])

## {

## index3 = charEl;

## tempKey[charEl] = '#';

## break;

## }

## }

## alphaToKeyMap[i] = index3;

## }

## //fill output variable with decrypted data

## string plainText = "";

## for(int row = 0; row<rows; row++)

## {

## for(int col=0; col<key.Length; col++)

## {

## plainText += cipherTextMatrix[row, alphaToKeyMap[col]];

## }

## }

## //Remove padding added by encryption

## return plainText.Trim(new char[] { '|'});

## }

## public static bool encryptFile(string fileName, string key)

## {

## //Read bytes from plaintext file

## byte[] plainText = File.ReadAllBytes(fileName);

## 

## //Sort key in alphabetical order

## char[] brokenKey = key.ToArray();

## Array.Sort(brokenKey);

## string sortedKey = new string(brokenKey);

## //calculate number of rows each column in the matrix will have

## int rows = (int)Math.Ceiling((double)plainText.Length / key.Length);

## //create matrix object

## byte[,] plainTextMatrix = new byte[rows, key.Length];

## int ptIndex = 0;

## byte paddingStartIndex = 0;

## bool paddingHasStarted = false;

## for (int row = 0; row < rows; row++)

## {

## for (int col = 0; col < key.Length; col++)

## {

## if (ptIndex < plainText.Length)

## {

## //fill matrix with bytes until there are not any bytes left

## plainTextMatrix[row, col] = plainText[ptIndex++];

## }

## else

## {

## //after all bytes are read start adding padding to the remainder of the last row

## //Save the index of this padding to be added to cipher file

## if (!paddingHasStarted)

## {

## paddingHasStarted = true;

## paddingStartIndex = Convert.ToByte(col);

## }

## plainTextMatrix[row, col] = 0;

## }

## }

## }

## 

## //create an array that maps the original key to the sorted key. This will be used to

## //select matrix columns in the correct order for encryption

## int[] alphaToKeyMap = new int[key.Length];

## char[] tempKey = key.ToCharArray();

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

## {

## int index = -1;

## for (int charEl = 0; charEl < alphaToKeyMap.Length; charEl++)

## {

## if (sortedKey[i] == tempKey[charEl])

## {

## index = charEl;

## tempKey[charEl] = '#';

## break;

## }

## }

## alphaToKeyMap[i] = index;

## }

## //create output byte array and fill byte array with columns in the order corresponding to

## //the mapping made in the previous block. Thus the first column to be selected is the

## //index of the first character (in the sorted key) in the original key

## byte[] cipherText = new byte[plainTextMatrix.LongLength + 1];

## int index2 = 0;

## for (int col = 0; col < key.Length; col++)

## {

## for (int row = 0; row < rows; row++)

## {

## cipherText[index2++] += plainTextMatrix[row, alphaToKeyMap[col]];

## }

## }

## //Save padding index to the byte array

## cipherText[cipherText.Length - 1] = paddingStartIndex;

## //Write the byte array to disk

## ByteArrayToFile(fileName + ".transposition", cipherText);

## return true;

## }

## public static bool decryptFile(string fileName, string key)

## {

## //Read bytes from plaintext file

## byte[] cipherText = File.ReadAllBytes(fileName);

## byte paddingStartIndex = cipherText[cipherText.Length - 1];

## 

## 

## //Sort key in alphabetical order

## char[] brokenKey = key.ToArray();

## Array.Sort(brokenKey);

## string sortedKey = new string(brokenKey);

## //calculate number of rows each column in the matrix will have

## int rows = (int)Math.Floor((double)(cipherText.Length) / key.Length);

## //create matrix object

## byte[,] cipherTextMatrix = new byte[rows, key.Length];

## //reconstruct the key-sorted ciphertext matrix

## int index = 0;

## for (int col = 0; col < key.Length; col++)

## {

## for (int row = 0; row < rows; row++)

## {

## //Fill matrix with file data

## cipherTextMatrix[row, col] = cipherText[index++];

## }

## }

## //create an array that maps the sorted key to the original key. This will be used to

## //select matrix columns in the correct order for encryption

## int[] alphaToKeyMap = new int[key.Length];

## char[] tempKey = sortedKey.ToCharArray();

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

## {

## int index3 = -1;

## for (int charEl = 0; charEl < alphaToKeyMap.Length; charEl++)

## {

## if (key[i] == tempKey[charEl])

## {

## index3 = charEl;

## tempKey[charEl] = '#';

## break;

## }

## }

## alphaToKeyMap[i] = index3;

## }

## //create output byte array and fill byte array with columns in the order corresponding to

## //the mapping made in the previous block. Thus the first column to be selected is the

## //index of the first character (in the original key) in the sorted key

## byte[] plainText = new byte[cipherTextMatrix.LongLength];

## int index2 = 0;

## for (int row = 0; row < rows; row++)

## {

## for (int col = 0; col < key.Length; col++)

## {

## plainText[index2++] = cipherTextMatrix[row, alphaToKeyMap[col]];

## }

## }

## //remove paddings bytes

## Array.Resize(ref plainText, plainText.Length-(key.Length - paddingStartIndex));

## fileName = fileName.Replace(".transposition", "");

## string fileExtension = fileName.Substring(fileName.LastIndexOf('.'));

## //Write the byte array to disk

## ByteArrayToFile(fileName+".decrypted"+fileExtension, plainText);

## return true;

## }

## //Method for writing byte array to file using filestream

## public static bool ByteArrayToFile(string fileName, byte[] byteArray)

## {

## try

## {

## using (var fs = new FileStream(fileName, FileMode.Create, FileAccess.Write))

## {

## fs.Write(byteArray, 0, byteArray.Length);

## return true;

## }

## }

## catch (Exception ex)

## {

## Console.WriteLine("Exception caught in process: {0}", ex);

## return false;

## }

## }

## }

}

## 5.3.2 Transposition GUI

public partial class Transposition : Form

{

string encryptedText;

string key;

string fileName;

public Transposition()

{

InitializeComponent();

}

private void btnEncrypt\_Click(object sender, EventArgs e)

{

key = tbxKey.Text;

encryptedText = TranspositionClass.encrypt(tbxPlainText.Text, key);

tbxCipherText.Text = encryptedText;

}

private void btnDecrypt\_Click(object sender, EventArgs e)

{

tbxDecryptedText.Text = TranspositionClass.decrypt(encryptedText, key);

}

private void btnSelectFile\_Click(object sender, EventArgs e)

{

OpenFileDialog ofd = new OpenFileDialog();

ofd.ShowDialog();

try

{

fileName = ofd.FileName;

tbxFileName.Text = fileName;

}

catch (Exception)

{

}

}

private void btnEncryptFile\_Click(object sender, EventArgs e)

{

try

{

pgrStatus.Value = 0;

pgrStatus.Step = 33;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Starting Encryption";

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Creating Ciphertext";

TranspositionClass.encryptFile(fileName, tbxKey.Text);

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Saving .key File";

pgrStatus.Value = 100;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Done";

}

catch (OutOfMemoryException)

{

MessageBox.Show("Out of Memory Exception occurred. Please select a smaller file.");

}

catch (Exception)

{

MessageBox.Show("Something went wrong.");

}

}

private void btnDecryptFile\_Click(object sender, EventArgs e)

{

try

{

pgrStatus.Value = 0;

pgrStatus.Step = 33;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Starting Encryption";

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Creating Ciphertext";

TranspositionClass.decryptFile(fileName, tbxKey.Text);

pgrStatus.PerformStep();

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Saving .key File";

pgrStatus.Value = 100;

lblStatus.Text = pgrStatus.Value.ToString() + "%";

lblStatusAction.Text = "Done";

}

catch (OutOfMemoryException)

{

MessageBox.Show("Out of Memory Exception occurred. Please select a smaller file.");

}

catch (Exception)

{

MessageBox.Show("Something went wrong.");

}

}

}

}

## RSA Class

class RSAImplementation

{

//Public component

private BigInteger e;

//Private component

private BigInteger d;

//Prime numbers

private BigInteger p, q;

//Multiplication of p, q -> Used as modulus for pri/pub keys -> Length is key length

private BigInteger n;

//Carmichael's totient function

private BigInteger ctf;

// Tmep varialbe, unneeded here tbh

private byte[] randomNumbers;

private RSACryptoServiceProvider rsa;

public RSAImplementation()

{

rsa = new RSACryptoServiceProvider();

generateKeys();

}

public int getKeyLength()

{

return n.ToString().Length;

}

// Multiplicative Inverse function

private BigInteger mul\_inv(BigInteger a, BigInteger b)

{

BigInteger b0 = b, t, q;

BigInteger x0 = BigInteger.Zero, x1 = BigInteger.One;

if (BigInteger.Compare(b, BigInteger.One) == 0)

return BigInteger.One;

while (BigInteger.Compare(a, BigInteger.One) == 1)

{

q = BigInteger.Divide(a, b);

t = b;

b = BigInteger.Remainder(a, b);

a = t;

t = x0;

x0 = BigInteger.Subtract(x1, BigInteger.Multiply(q, x0));

x1 = t;

}

if (BigInteger.Compare(x1, BigInteger.Zero) == -1)

x1 = BigInteger.Add(x1, b0);

return x1;

}

// Least Common multiple

private BigInteger LCM(BigInteger a, BigInteger b)

{

return BigInteger.Divide(BigInteger.Multiply(a, b), BigInteger.GreatestCommonDivisor(a, b));

}

private bool IsProbablePrime(BigInteger source, int certainty)

{

if (BigInteger.Compare(source, BigInteger.Add(BigInteger.One, BigInteger.One)) == 0 || BigInteger.Compare(source, BigInteger.Add(BigInteger.Add(BigInteger.One, BigInteger.One), BigInteger.One)) == 0)

return true;

if (BigInteger.Compare(source, BigInteger.Add(BigInteger.One, BigInteger.One)) == -1 || BigInteger.Remainder(source, BigInteger.Add(BigInteger.One, BigInteger.One)) == 0)

return false;

BigInteger d = BigInteger.Subtract(source, BigInteger.One);

int s = 0;

while (BigInteger.Compare(BigInteger.Remainder(d, BigInteger.Add(BigInteger.One, BigInteger.One)), BigInteger.Zero) == 0)

{

d = BigInteger.Divide(d, BigInteger.Add(BigInteger.One, BigInteger.One));

s += 1;

}

RandomNumberGenerator rng = RandomNumberGenerator.Create();

byte[] bytes = new byte[source.ToByteArray().Length];

BigInteger a;

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

{

do

{

rng.GetBytes(bytes);

a = new BigInteger(bytes);

}

while (BigInteger.Compare(a, BigInteger.Add(BigInteger.One, BigInteger.One)) == -1 || BigInteger.Compare(a, BigInteger.Subtract(source, BigInteger.Add(BigInteger.One, BigInteger.One))) >= 0);

BigInteger x = BigInteger.ModPow(a, d, source);

if (BigInteger.Compare(x, BigInteger.One) == 0 || BigInteger.Compare(x, BigInteger.Subtract(source, BigInteger.One)) == 0)

continue;

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

{

x = BigInteger.ModPow(x, BigInteger.Add(BigInteger.One, BigInteger.One), source);

if (BigInteger.Compare(x, BigInteger.One) == 0)

return false;

if (BigInteger.Compare(x, BigInteger.Subtract(source, BigInteger.One)) == 0)

break;

}

if (BigInteger.Compare(x, BigInteger.Subtract(source, BigInteger.One)) != 0)

return false;

}

return true;

}

// https://en.wikipedia.org/wiki/RSA\_(cryptosystem)#Key\_generation

public void generateKeys()

{

Console.WriteLine("\nStep 1.1");

RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

do

{

randomNumbers = new byte[32];

rng.GetBytes(randomNumbers);

p = new BigInteger(randomNumbers);

p = BigInteger.Abs(p);

} while (!IsProbablePrime(p, 10));

Console.WriteLine("p: " + p.ToString());

Console.WriteLine("\nStep 1.2");

do

{

randomNumbers = new byte[32];

rng.GetBytes(randomNumbers);

q = new BigInteger(randomNumbers);

q = BigInteger.Abs(q);

} while (!IsProbablePrime(q, 10));

Console.WriteLine("q: " + q.ToString());

rng.Dispose();

randomNumbers = new byte[1];

Console.WriteLine("\nStep 2");

n = BigInteger.Multiply(p, q);

Console.WriteLine("n: " + n.ToString());

Console.WriteLine("\nStep 3");

ctf = LCM(BigInteger.Add(p, BigInteger.MinusOne), BigInteger.Add(q, BigInteger.MinusOne));

Console.WriteLine("ctf: " + ctf.ToString());

// Possible mistake here, review with internet

Console.WriteLine("\nStep 4");

rng = new RNGCryptoServiceProvider();

do

{

byte[] tempBytes = ctf.ToByteArray();

rng.GetBytes(tempBytes);

e = new BigInteger(tempBytes);

e = BigInteger.Abs(e);

} while (!(BigInteger.Compare(BigInteger.GreatestCommonDivisor(e, ctf), BigInteger.One) == 0));

Console.WriteLine("e: " + e.ToString());

rng.Dispose();

Console.WriteLine("\nStep 5");

d = mul\_inv(e, ctf);

Console.WriteLine("d: " + d.ToString());

}

public BigInteger getPubKey()

{

return e;

}

public BigInteger getPrivKey()

{

return d;

}

public BigInteger encryptText(string toEncrypt)

{

BigInteger m = new BigInteger(Encoding.UTF8.GetBytes(toEncrypt));

return BigInteger.ModPow(m, e, n);

}

public string decryptText(BigInteger toDecrypt)

{

BigInteger c = toDecrypt;

return Encoding.UTF8.GetString(BigInteger.ModPow(c, d, n).ToByteArray());

}

public SymmetricAlgorithm encryptFile(string inName, string outName)

{

//Create the file streams to handle the input and output files.

FileStream fin = new FileStream(inName, FileMode.Open, FileAccess.Read);

FileStream fout = new FileStream(outName, FileMode.OpenOrCreate, FileAccess.Write);

fout.SetLength(0);

//Create variables to help with read and write.

byte[] bin = new byte[100]; //This is intermediate storage for the encryption.

long rdlen = 0; //This is the total number of bytes written.

long totlen = fin.Length; //This is the total length of the input file.

int len; //This is the number of bytes to be written at a time.

SymmetricAlgorithm rijn = SymmetricAlgorithm.Create(); //Creates the default implementation, which is RijndaelManaged.

rijn.GenerateIV();

rijn.GenerateKey();

CryptoStream encStream = new CryptoStream(fout, rijn.CreateEncryptor(), CryptoStreamMode.Write);

Console.WriteLine("Encrypting...");

//Read from the input file, then encrypt and write to the output file.

while (rdlen < totlen)

{

len = fin.Read(bin, 0, 100);

encStream.Write(bin, 0, len);

rdlen = rdlen + len;

Console.WriteLine("{0} bytes processed", rdlen);

}

encStream.Close();

fout.Close();

fin.Close();

return rijn;

}

public void decryptFile(string inName, string outName, SymmetricAlgorithm rijn)

{

FileStream fin = new FileStream(inName, FileMode.Open, FileAccess.Read);

FileStream fout = new FileStream(outName, FileMode.OpenOrCreate, FileAccess.Write);

fout.SetLength(0);

byte[] bin = new byte[100];

long rdlen = 0;

long totlen = fin.Length;

int len;

CryptoStream decStream = new CryptoStream(fout, rijn.CreateDecryptor(), CryptoStreamMode.Write);

Console.WriteLine("Decrypting...");

while (rdlen < totlen)

{

len = fin.Read(bin, 0, 100);

decStream.Write(bin, 0, len);

rdlen = rdlen + len;

Console.WriteLine("{0} bytes processed", rdlen);

}

decStream.Close();

fout.Close();

fin.Close();

}

}

}

**RSA GUI Code**

class RSAImplementation

{

//Public component

private BigInteger e;

//Private component

private BigInteger d;

//Prime numbers

private BigInteger p, q;

//Multiplication of p, q -> Used as modulus for pri/pub keys -> Length is key length

private BigInteger n;

//Carmichael's totient function

private BigInteger ctf;

// Tmep varialbe, unneeded here tbh

private byte[] randomNumbers;

private RSACryptoServiceProvider rsa;

public RSAImplementation()

{

rsa = new RSACryptoServiceProvider();

generateKeys();

}

public int getKeyLength()

{

return n.ToString().Length;

}

// Multiplicative Inverse function

private BigInteger mul\_inv(BigInteger a, BigInteger b)

{

BigInteger b0 = b, t, q;

BigInteger x0 = BigInteger.Zero, x1 = BigInteger.One;

if (BigInteger.Compare(b, BigInteger.One) == 0)

return BigInteger.One;

while (BigInteger.Compare(a, BigInteger.One) == 1)

{

q = BigInteger.Divide(a, b);

t = b;

b = BigInteger.Remainder(a, b);

a = t;

t = x0;

x0 = BigInteger.Subtract(x1, BigInteger.Multiply(q, x0));

x1 = t;

}

if (BigInteger.Compare(x1, BigInteger.Zero) == -1)

x1 = BigInteger.Add(x1, b0);

return x1;

}

// Least Common multiple

private BigInteger LCM(BigInteger a, BigInteger b)

{

return BigInteger.Divide(BigInteger.Multiply(a, b), BigInteger.GreatestCommonDivisor(a, b));

}

private bool IsProbablePrime(BigInteger source, int certainty)

{

if (BigInteger.Compare(source, BigInteger.Add(BigInteger.One, BigInteger.One)) == 0 || BigInteger.Compare(source, BigInteger.Add(BigInteger.Add(BigInteger.One, BigInteger.One), BigInteger.One)) == 0)

return true;

if (BigInteger.Compare(source, BigInteger.Add(BigInteger.One, BigInteger.One)) == -1 || BigInteger.Remainder(source, BigInteger.Add(BigInteger.One, BigInteger.One)) == 0)

return false;

BigInteger d = BigInteger.Subtract(source, BigInteger.One);

int s = 0;

while (BigInteger.Compare(BigInteger.Remainder(d, BigInteger.Add(BigInteger.One, BigInteger.One)), BigInteger.Zero) == 0)

{

d = BigInteger.Divide(d, BigInteger.Add(BigInteger.One, BigInteger.One));

s += 1;

}

RandomNumberGenerator rng = RandomNumberGenerator.Create();

byte[] bytes = new byte[source.ToByteArray().Length];

BigInteger a;

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

{

do

{

rng.GetBytes(bytes);

a = new BigInteger(bytes);

}

while (BigInteger.Compare(a, BigInteger.Add(BigInteger.One, BigInteger.One)) == -1 || BigInteger.Compare(a, BigInteger.Subtract(source, BigInteger.Add(BigInteger.One, BigInteger.One))) >= 0);

BigInteger x = BigInteger.ModPow(a, d, source);

if (BigInteger.Compare(x, BigInteger.One) == 0 || BigInteger.Compare(x, BigInteger.Subtract(source, BigInteger.One)) == 0)

continue;

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

{

x = BigInteger.ModPow(x, BigInteger.Add(BigInteger.One, BigInteger.One), source);

if (BigInteger.Compare(x, BigInteger.One) == 0)

return false;

if (BigInteger.Compare(x, BigInteger.Subtract(source, BigInteger.One)) == 0)

break;

}

if (BigInteger.Compare(x, BigInteger.Subtract(source, BigInteger.One)) != 0)

return false;

}

return true;

}

// https://en.wikipedia.org/wiki/RSA\_(cryptosystem)#Key\_generation

public void generateKeys()

{

Console.WriteLine("\nStep 1.1");

RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

do

{

randomNumbers = new byte[32];

rng.GetBytes(randomNumbers);

p = new BigInteger(randomNumbers);

p = BigInteger.Abs(p);

} while (!IsProbablePrime(p, 10));

Console.WriteLine("p: " + p.ToString());

Console.WriteLine("\nStep 1.2");

do

{

randomNumbers = new byte[32];

rng.GetBytes(randomNumbers);

q = new BigInteger(randomNumbers);

q = BigInteger.Abs(q);

} while (!IsProbablePrime(q, 10));

Console.WriteLine("q: " + q.ToString());

rng.Dispose();

randomNumbers = new byte[1];

Console.WriteLine("\nStep 2");

n = BigInteger.Multiply(p, q);

Console.WriteLine("n: " + n.ToString());

Console.WriteLine("\nStep 3");

ctf = LCM(BigInteger.Add(p, BigInteger.MinusOne), BigInteger.Add(q, BigInteger.MinusOne));

Console.WriteLine("ctf: " + ctf.ToString());

// Possible mistake here, review with internet

Console.WriteLine("\nStep 4");

rng = new RNGCryptoServiceProvider();

do

{

byte[] tempBytes = ctf.ToByteArray();

rng.GetBytes(tempBytes);

e = new BigInteger(tempBytes);

e = BigInteger.Abs(e);

} while (!(BigInteger.Compare(BigInteger.GreatestCommonDivisor(e, ctf), BigInteger.One) == 0));

Console.WriteLine("e: " + e.ToString());

rng.Dispose();

Console.WriteLine("\nStep 5");

d = mul\_inv(e, ctf);

Console.WriteLine("d: " + d.ToString());

}

public BigInteger getPubKey()

{

return e;

}

public BigInteger getPrivKey()

{

return d;

}

public BigInteger encryptText(string toEncrypt)

{

BigInteger m = new BigInteger(Encoding.UTF8.GetBytes(toEncrypt));

return BigInteger.ModPow(m, e, n);

}

public string decryptText(BigInteger toDecrypt)

{

BigInteger c = toDecrypt;

return Encoding.UTF8.GetString(BigInteger.ModPow(c, d, n).ToByteArray());

}

public SymmetricAlgorithm encryptFile(string inName, string outName)

{

//Create the file streams to handle the input and output files.

FileStream fin = new FileStream(inName, FileMode.Open, FileAccess.Read);

FileStream fout = new FileStream(outName, FileMode.OpenOrCreate, FileAccess.Write);

fout.SetLength(0);

//Create variables to help with read and write.

byte[] bin = new byte[100]; //This is intermediate storage for the encryption.

long rdlen = 0; //This is the total number of bytes written.

long totlen = fin.Length; //This is the total length of the input file.

int len; //This is the number of bytes to be written at a time.

SymmetricAlgorithm rijn = SymmetricAlgorithm.Create(); //Creates the default implementation, which is RijndaelManaged.

rijn.GenerateIV();

rijn.GenerateKey();

CryptoStream encStream = new CryptoStream(fout, rijn.CreateEncryptor(), CryptoStreamMode.Write);

Console.WriteLine("Encrypting...");

//Read from the input file, then encrypt and write to the output file.

while (rdlen < totlen)

{

len = fin.Read(bin, 0, 100);

encStream.Write(bin, 0, len);

rdlen = rdlen + len;

Console.WriteLine("{0} bytes processed", rdlen);

}

encStream.Close();

fout.Close();

fin.Close();

return rijn;

}

public void decryptFile(string inName, string outName, SymmetricAlgorithm rijn)

{

FileStream fin = new FileStream(inName, FileMode.Open, FileAccess.Read);

FileStream fout = new FileStream(outName, FileMode.OpenOrCreate, FileAccess.Write);

fout.SetLength(0);

byte[] bin = new byte[100];

long rdlen = 0;

long totlen = fin.Length;

int len;

CryptoStream decStream = new CryptoStream(fout, rijn.CreateDecryptor(), CryptoStreamMode.Write);

Console.WriteLine("Decrypting...");

while (rdlen < totlen)

{

len = fin.Read(bin, 0, 100);

decStream.Write(bin, 0, len);

rdlen = rdlen + len;

Console.WriteLine("{0} bytes processed", rdlen);

}

decStream.Close();

fout.Close();

fin.Close();

}

}

}