|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра теоретической и прикладной информатики | | |
| Лабораторная работа № 1 | | |
| по дисциплине «Информационная безопасность» | | |
| **КРИПТОГРАФИЧЕСКИЕ ХЭШ-ФУНКЦИИ** | | |
|  | | |
|  | Бригада 2 | Хайдаев К.е. |
| Группа ПМИ-82 | Зяблицева У.п. |
| Вариант 2 |  |
|  |  |
|  |  |
|  |  |
| Преподаватели | Авдеенко Т.В. |
|  |  |
| Новосибирск, 2022 | | |

1. **Задание**

I. Реализовать приложение с графическим интерфейсом, позволяющее выполнять следующие действия.

1. Вычислять значение хэш-функции, заданной в варианте:

1) текст сообщения должен считываться из файла;

2) полученное значение хэш-функции должно представляться в

шестнадцатеричном виде и сохраняться в файл;

3) при работе программы должна быть возможность просмотра

и изменения считанного из файла сообщения и вычисленного

значения хэш-функции.

2. Исследовать лавинный эффект на сообщении, состоящем из одного блока:

1) для бита, который будет изменяться, приложение должно

позволять задавать его позицию (номер) в сообщении;

2) приложение должно уметь после каждого раунда (итерации

цикла) вычисления хэш-функции подсчитывать число бит,

изменившихся в значении хэш-функции при изменении одного бита в тексте сообщения;

3) приложение может строить графики зависимости числа бит,

изменившихся в значении хэш-функции, от раунда вычисления хэш-функции, либо графики можно строить в стороннем ПО, но тогда приложение должно сохранять в файл необходимую для построения графиков информацию.

II. С помощью реализованного приложения выполнить следующие

задания.

1. Протестировать правильность работы разработанного приложения.

2. Исследовать лавинный эффект при изменении одного бита в сообщении: для различных позиций изменяемого бита в сообщении

построить графики зависимостей числа бит, изменившихся в значении хэш-функции, от раунда вычисления хэш-функции (всего в отчете должно быть два-три графика).

Вариант: Алгоритм RIPEMD–320.

1. **Теория**

RIPEMD (от англ. RACE Integrity Primitives Evaluation Message Digest) – хэш-функция, разработанная в Лувенском католическом университете Г. Добертином, А. Босселаерсоми, Б. Пренелем. RIPEMD–160 является улучшенной версией RIPEMD, которая в свою очередь использовала принципы MD4 и по производительности сравнима с более популярной SHA–1.

Также существуют 128,160, 256- и 320-битные версии этого алгоритма,

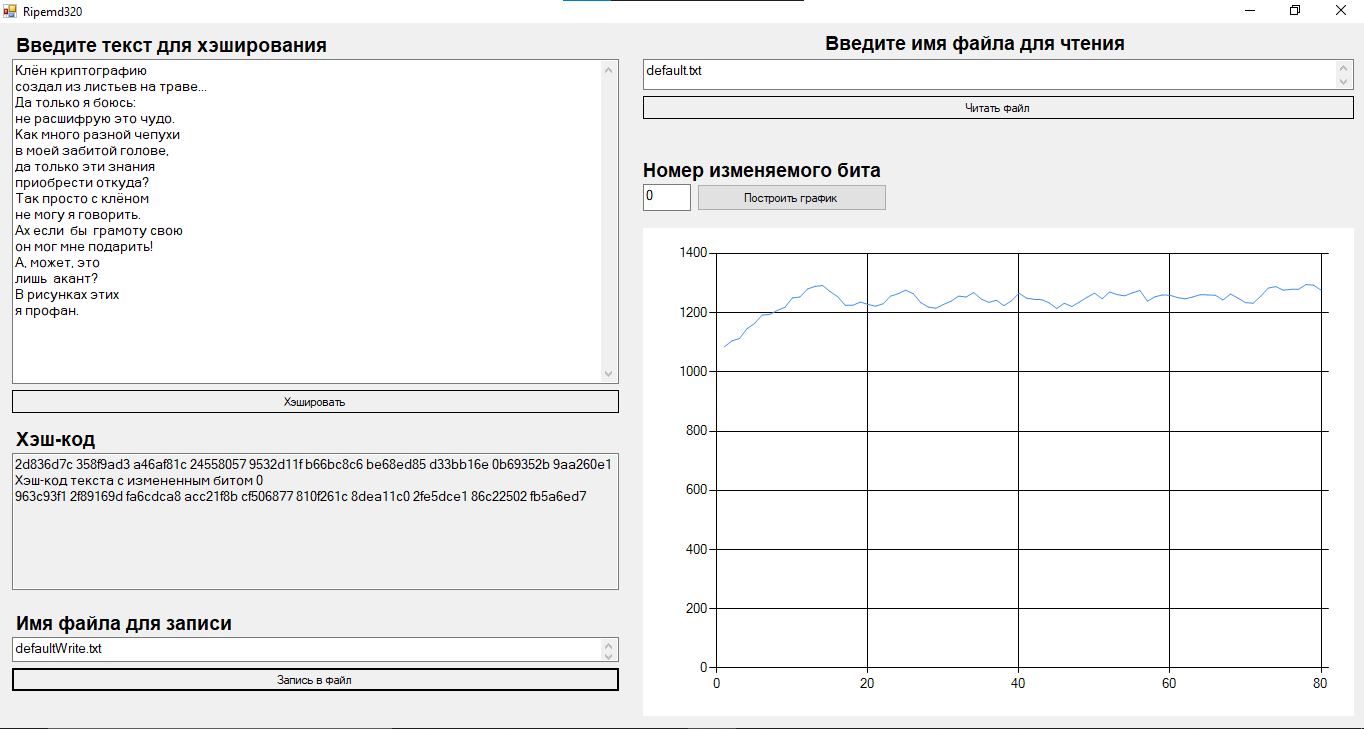
которые называются RIPEMD–128, RIPEMD–256, RIPEMD-160 и RIPEMD–320.128-битная версия представляет собой лишь замену оригинальнойRIPEMD, которая также была 128-битной и в которой были найдены уязвимости. 256- и 320-битные версии отличаются удвоенной длиной дайджеста, что уменьшает вероятность коллизий, но при этом функции не являются более криптостойкими.

RIPEMD разработана в открытом академическом сообществе, в отличие от SHA–1 и SHA–2, которые были созданы NSA. Использование RIPEMD не ограничено какими-либо патентами.

1. **Описание программного средства**

Разработанная программа способна производить шифрование считанного из указанного файла сообщения по алгоритму RIPEMD–320. Также программа способна проводить исследование лавинного эффекта с выводом графика отличающихся битов. Программа учитывает подаваемые ей параметры (имя файла, номер изменяемого бита, их отсутствие, имя несуществующего файла) и адекватно реагировать на них: выдавать соответствующие сообщения.

Интерфейс приложения:



1. **Исследования**

|  |  |  |
| --- | --- | --- |
| N | Сообщение | Хэш-код |
| 1 | Клён криптографию  создал из листьев на траве...  Да только я боюсь:  не расшифрую это чудо.  Как много разной чепухи  в моей забитой голове,  да только эти знания  приобрести откуда?  Так просто с клёном  не могу я говорить.  Ах если бы грамоту свою  он мог мне подарить!  А, может, это  лишь акант?  В рисунках этих  я профан. | 2d836d7c | 358f9ad3 | a46af81c | 24558057 | 9532d11f | b66bc8c6 | be68ed85 | d33bb16e | 0b69352b | 9aa260e1 | |
| 2 | 13222222222222221 4566666666666666664 65876666666666666666666665  655555555555555554  4556 | 8d4de41c | 86c75c66 | c4128b96 | 828cf9fa | 4eaa3216 | 6b396fb9 | 930bae00 | 63405288 | de750762 | 933d6725 | |
| 3 | {}!@#$%^&&\*\*)(dslkfjdsfjv &&&?12930  \*\*\*/\*12134 Ё~`';""><. | 5ad0a9c6 | d9135414 | b544dabc | ce0edc09 | 13fe059d | b2ba2e0a | 3e9695ac | 7a056429 | bd11fa06 | 99218a96 | |
| 4 |  | f3b744a5 | 72daa0e2 | 9d5945b0 | 12260127 | 6433291d | 3be41ff7 | 27055494 | f7f7e9d6 | eb03d126 | 9a419f40 | |

Лавинный эффект:

|  |  |  |
| --- | --- | --- |
| N | Бит | График |
| 1 | 0 |  |
| 2 | 61 |  |
| 3 | 200 |  |

1. **Вывод**

В ходе выполненной лабораторной было разработано программное средство, предназначенное для вычисления хэш-функции текстового сообщения алгоритмом RIPEMD–320, а также для исследования лавинного эффекта на примере данного алгоритма.

В ходе исследования лавинного эффекта был, исходя из графиков изменения хэш-функции, можно сделать вывод, что даже изменение одного бита в исходном сообщении приводит к изменению в среднем более половины всех битов хэш-функции уже на первом раунде хэширования.

1. **Код программы**

“Ripemd320.cs”

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

namespace lab1\_ripemd320

{

internal class Ripemd320

{

// оригинальное сообщение

private String originalStr;

private String binaryStr;

//сообщение с выравниванием

private String str;

//хэш коды на каждом раунде. Номер в листе - номер блока, Ключ-номер раунда, Значение-лист H на этом раунде

private List<Dictionary<int, List<long>>> rounds;

//H-лист хэш кодов

private List<String> H;

public List<String> GetH() { return H; }

public String GetOriginalStr() { return originalStr; }

//2^32 для операции mod2^32 - %2^32 остаток от деления

private readonly static long MOD = (long)Math.Pow(2,32);

//внешний лист содержит блоки по 512 бит, внутренний разделяет 512 на слова по 32 бита, т.е. 16 слов

private List<String[]> blocks;

//START

public Ripemd320(String originalStr)

{

this.originalStr = originalStr;

this.binaryStr = stringToBinary(originalStr);

this.str = addBits(binaryStr);

this.str = addBitLength(binaryStr, str);

this.blocks = initializeBlocks(str);

this.H = start();

}

//изменение бита в исходном сообщении

public Ripemd320(String originalStr, int changeBit)

{

this.originalStr = originalStr;

this.binaryStr = stringToBinary(originalStr);

if (binaryStr.Length > changeBit)

{

char[] array = binaryStr.ToCharArray();

if (array[changeBit] == '0')

array[changeBit] = '1';

else

array[changeBit] = '0';

binaryStr = new String(array);

}

else

throw new Exception("Слишком большое число, такого бита в сообщение нету");

this.str = addBits(binaryStr);

this.str = addBitLength(binaryStr, str);

this.blocks = initializeBlocks(str);

this.H = start();

}

// (шаг 0) Сообщение в бинарный формат

private static String stringToBinary(String originalStr)

{

String res = "";

String binary = "";

foreach (var i in originalStr.ToCharArray())

{

//битовый формат

binary = Convert.ToString(i, 2).PadLeft(8, '0');

res += binary;

//res += String.format("%8s", binary).replaceAll(" ", "0");

}

return res;

}

//(шаг 1) Добавляем к сообщению в бинарном формате биты пока они не станут кратны 512, с остатком 448

private static String addBits(String str)

{

String res = str;

res += 1;

while (res.Length % 512 != 448)

{

res += 0;

}

return res;

}

//(шаг 2) добавляем 32 младших бита длины сообщения, а затем 32 старших

private static String addBitLength(String binaryStr, String str)

{

String res = str;

String temp = Convert.ToString(binaryStr.Length,2).PadLeft(64,'0');

char[] array = temp.ToCharArray();

for (int i = 32; i < 64; i++)

{

res += array[i];

}

for (int i = 0; i < 32; i++)

{

res += array[i];

}

//res+=temp;

return res;

}

//(шаг 2) разделение сообщение на блоки по 512 битов - "(?<=\\G.{512})" прикольная регулярка, надо почитать

private static List<String[]> initializeBlocks(String str)

{

List<String[]> result = new List<String[]>();

str += "1";

List<String> external = Regex.Split(str,"(?<=\\G.{512})").ToList();

foreach (var item in external)

{

//result.add(item.split("(?<=\\G.{32})"));

//записываем слова в обратном порядке например вместе (a b c d) будет (d c b a)

String[] res = new String[16];

String[] tempArray1 = Regex.Split(item,"(?<=\\G.{32})");

for (int i = 0; i < tempArray1.Length-1; i++)

{

String temp = "";

String[] tempArray2 = Regex.Split(tempArray1[i],"(?<=\\G.{8})");

for (int j = tempArray2.Length - 1; j >= 0; j--)

{

temp += tempArray2[j];

}

res[i] = temp;

}

result.Add(res);

}

return result;

}

//(шаг3.1) Нелинейная побитовая функция

private static String nonLinearBitFunc(int j, String X, String Y, String Z)

{

String result = "";

long res = 0;

long x = Convert.ToInt64(X, 16);

long y = Convert.ToInt64(Y, 16);

long z = Convert.ToInt64(Z, 16);

if (j >= 0 || j <= 15)

{

res = x ^ y ^ z;

}

else if (j >= 16 || j <= 31)

{

res = (x & y) | (~x & z);

}

else if (j >= 32 || j <= 47)

{

res = (x | (~y)) ^ z;

}

else if (j >= 48 || j <= 63)

{

res = (x & z) | (y & (~z));

}

else if (j >= 64 || j <= 79)

{

res = x ^ (y | (~z));

}

result = Convert.ToString(res,16);

return result;

}

private static long nonLinearBitFunc(int j, long x, long y, long z)

{

long res = 0;

if (j >= 0 || j <= 15)

{

res = x ^ y ^ z;

}

else if (j >= 16 || j <= 31)

{

res = (x & y) | (~x & z);

}

else if (j >= 32 || j <= 47)

{

res = (x | (~y)) ^ z;

}

else if (j >= 48 || j <= 63)

{

res = (x & z) | (y & (~z));

}

else if (j >= 64 || j <= 79)

{

res = x ^ (y | (~z));

}

return res;

}

//(шаг 3.2) Добавление 16-ричных констант K1

private static String addHexConstK1(int j)

{

String res = "";

if (j >= 0 || j <= 15)

{

res = HEX\_K1[0];

}

else if (j >= 16 || j <= 31)

{

res = HEX\_K1[1];

}

else if (j >= 32 || j <= 47)

{

res = HEX\_K1[2];

}

else if (j >= 48 || j <= 63)

{

res = HEX\_K1[3];

}

else if (j >= 64 || j <= 79)

{

res = HEX\_K1[4];

}

return res;

}

//(шаг 3.2) Добавление 16-ричных констант K2

private static String addHexConstK2(int j)

{

String res = "";

if (j >= 0 || j <= 15)

{

res = HEX\_K2[0];

}

else if (j >= 16 || j <= 31)

{

res = HEX\_K2[1];

}

else if (j >= 32 || j <= 47)

{

res = HEX\_K2[2];

}

else if (j >= 48 || j <= 63)

{

res = HEX\_K2[3];

}

else if (j >= 64 || j <= 79)

{

res = HEX\_K2[4];

}

return res;

}

//16-ричные константы для шага 3.2

public readonly static String[] HEX\_K1 = { "00000000", "5a827999", "6ed9eba1", "8f1bbcdc", "a953fd4e" };

public readonly static String[] HEX\_K2 = { "50a28be6", "5c4dd124", "6d703ef3", "7a6d76e9", "00000000" };

//то же что выше, только в 10 форме

public readonly static long[] DEC\_K1 = {

Convert.ToInt64(HEX\_K1[0], 16), Convert.ToInt64(HEX\_K1[1], 16), Convert.ToInt64(HEX\_K1[2], 16),

Convert.ToInt64(HEX\_K1[3], 16), Convert.ToInt64(HEX\_K1[4], 16)};

public readonly static long[] DEC\_K2 = {

Convert.ToInt64(HEX\_K2[0], 16), Convert.ToInt64(HEX\_K2[1], 16), Convert.ToInt64(HEX\_K2[2], 16),

Convert.ToInt64(HEX\_K2[3], 16), Convert.ToInt64(HEX\_K2[4], 16)};

private static long addDecConstK1(int j)

{

long res = 0;

if (j >= 0 || j <= 15)

{

res = Ripemd320.DEC\_K1[0];

}

else if (j >= 16 || j <= 31)

{

res = DEC\_K1[1];

}

else if (j >= 32 || j <= 47)

{

res = DEC\_K1[2];

}

else if (j >= 48 || j <= 63)

{

res = DEC\_K1[3];

}

else if (j >= 64 || j <= 79)

{

res = DEC\_K1[4];

}

return res;

}

private static long addDecConstK2(int j)

{

long res = 0;

if (j >= 0 || j <= 15)

{

res = DEC\_K2[0];

}

else if (j >= 16 || j <= 31)

{

res = DEC\_K2[1];

}

else if (j >= 32 || j <= 47)

{

res = DEC\_K2[2];

}

else if (j >= 48 || j <= 63)

{

res = DEC\_K2[3];

}

else if (j >= 64 || j <= 79)

{

res = DEC\_K2[4];

}

return res;

}

//(шаг3.3) Номера выбираемых сообщений из 32-битных слов

public readonly static int[] NUM\_OF\_WORDS\_32BIT\_R1 = {

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,

7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8,

3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12,

1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2,

4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13};

public readonly static int[] NUM\_OF\_WORDS\_32BIT\_R2 = {

5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12,

6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2,

15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13,

8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14,

12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11,

};

//(шаг 3.4) количество бит для сдвига

public readonly static int[] NUM\_OF\_BITS\_TO\_SHIFT\_S1 = {

11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8,

7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12,

11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5,

11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12,

9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6

};

public readonly static int[] NUM\_OF\_BITS\_TO\_SHIFT\_S2 = {

8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6,

9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11,

9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5,

15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8,

8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11

};

//(шаг 3.5) начальные значения хэшей от h0 до h9

public static readonly String[] INITIAL\_HASH\_VALUE\_16\_H = {

"67452301", "efcdab89", "98badcfe",

"10325476", "c3d2e1f0", "76543210",

"fedcba98", "89abcdef", "01234567",

"3c2d1e0f"};

//тоже самое только в 10 форме

public static readonly long[] INITIAL\_HASH\_VALUE\_H = {

Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[0], 16), Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[1], 16), Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[2], 16),

Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[3], 16), Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[4], 16), Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[5], 16),

Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[6], 16), Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[7], 16), Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[8], 16),

Convert.ToInt64(INITIAL\_HASH\_VALUE\_16\_H[9], 16)};

//(шаг 4)

private List<String> start()

{

List<String> result = new List<String>();

rounds = new List<Dictionary<int, List<long>>>();

//H = H0->H9

//задаем начальные значения

long[] H = INITIAL\_HASH\_VALUE\_H.ToArray();

long temp = 0; // временная переменная, просто удобнее вычислять

//Проходим по всем блокам из 512 бит

for (int i = 0; i < blocks.Count(); i++)

{

rounds.Add(new Dictionary<int, List<long>>());

//задаем A B C D E значения равные массиву H полученному на прошлой итерации

long A1 = H[0];

long B1 = H[1];

long C1 = H[2];

long D1 = H[3];

long E1 = H[4];

long A2 = H[5];

long B2 = H[6];

long C2 = H[7];

long D2 = H[8];

long E2 = H[9];

//обрабатываем блок по алгоритму (a+b)=(a+b)%2^32=(a+b)%MOD

for (int j = 0; j < 80; j++)

{

long b=Convert.ToInt64(blocks[i][NUM\_OF\_WORDS\_32BIT\_R1[j]], 2);

temp = (cyclicShiftLeft((((A1 + nonLinearBitFunc(j, B1, C1, D1)) % MOD + Convert.ToInt64(blocks[i][NUM\_OF\_WORDS\_32BIT\_R1[j]], 2)) % MOD + addDecConstK1(j)) % MOD, NUM\_OF\_BITS\_TO\_SHIFT\_S1[j]) + E1) % MOD;

A1 = E1;

E1 = D1;

D1 = cyclicShiftLeft(C1, 10);

C1 = B1;

B1 = temp;

temp = (cyclicShiftLeft((((A2 + nonLinearBitFunc(79 - j, B2, C2, D2)) % MOD + Convert.ToInt64(blocks[i][NUM\_OF\_WORDS\_32BIT\_R2[j]], 2)) % MOD + addDecConstK2(j)) % MOD, NUM\_OF\_BITS\_TO\_SHIFT\_S2[j]) + E2) % MOD;

A2 = E2;

E2 = D2;

D2 = cyclicShiftLeft(C2, 10);

C2 = B2;

B2 = temp;

if (j == 15)

{

temp = B1;

B1 = B2;

B2 = temp;

}

if (j == 31)

{

temp = D1;

D1 = D2;

D2 = temp;

}

if (j == 47)

{

temp = A1;

A1 = A2;

A2 = temp;

}

if (j == 63)

{

temp = C1;

C1 = C2;

C2 = temp;

}

if (j == 79)

{

temp = E1;

E1 = E2;

E2 = temp;

}

rounds[i].Add(j , new List<long>(){ A1, B1, C1, D1, E1, A2, B2, C2, D2, E2} );

}

H[0] = (H[0] + A1) % MOD;

H[1] = (H[1] + B1) % MOD;

H[2] = (H[2] + C1) % MOD;

H[3] = (H[3] + D1) % MOD;

H[4] = (H[4] + E1) % MOD;

H[5] = (H[5] + A2) % MOD;

H[6] = (H[6] + B2) % MOD;

H[7] = (H[7] + C2) % MOD;

H[8] = (H[8] + D2) % MOD;

H[9] = (H[9] + E2) % MOD;

}

foreach (var h in H)

{

result.Add(Convert.ToString(h,16).PadLeft(8,'0'));

}

return result;

}

//циклический сдвиг влево на i позиций

private static long cyclicShiftLeft(long num, int i)

{

String temp = Convert.ToString(num,2);

char[] array = temp.ToCharArray();

temp = "";

if (num != 0)

{

for (int j = i; j < array.Length; j++)

{

temp += array[j];

}

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

{

temp += array[j];

}

}

else

temp = "0";

return Convert.ToInt64(temp, 2);

}

public int[] getChangedBits(Ripemd320 other)

{

int[] result = new int[80];

if (this.rounds.Count != other.rounds.Count)

return result;

for (int i = 0; i < rounds.Count; i++)

{

for (int j = 0; j < rounds[i].Count; j++)

{

for (int k = 0; k < rounds[i][j].Count; k++)

{

char[] array1 = Convert.ToString(rounds[i][j][k],2).PadLeft(32,'0').ToCharArray();

char[] array2 = Convert.ToString(other.rounds[i][j][k], 2).PadLeft(32, '0').ToCharArray();

for (int l = 0; l < array1.Length; l++)

{

if (array1[l] != array2[l])

result[j]++;

}

}

}

}

return result;

}

public override string ToString()

{

StringBuilder sb = new StringBuilder(120);

foreach (var item in H)

{

sb.Append(item);

sb.Append(" | ");

}

return sb.ToString();

}

}

}