1. Получение бинарного представления хешируемого сообщения

CSTIGERALG =

01000011 01010011 01010100 01001001 01000111 01000101 01010010 01000001 01001100 01000111

Всего 80 бит (10 байт)

1. Добавление бита 10000000 в конец

01000011 01010011 01010100 01001001 01000111 01000101 01010010 01000001 01001100 01000111 10000000

Теперь мы имеем 88 бит (11 байт)

1. Добавление нулевых битов 00000000 до тех пор, пока длина сообщения не будет равна 448 бит (56 байт)

01000011 01010011 01010100 01001001 01000111 01000101 01010010 01000001 01001100 01000111 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

1. Добавление 8 бит, обозначающих длину исходного сообщения. В моём случае длина равна 80 битам (10 байтам), что в бинарном виде будет 00000000 … 01010000. Теперь длина сообщения будет равна 512 битам (64 байтам)

01000011 01010011 01010100 01001001 01000111 01000101 01010010 01000001 01001100 01000111 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01010000

1. Обозначим значения состояний в шестнадцатеричном и двоичном виде

A = 0123456789ABCDEF = 00000001 00100101 00110100 01011000 01100101 01101111 01110101 01111000 01101001 00111010 01111001 00110100 00101000 01100100 01101101 01100100

B = FEDCBA9876543210 = 11111110 11001001 10011000 01111110 11001001 01101001 10010100 00111111 01011001 00111010 00001011 01010101 01111110 11000110 01110011 01111010

C = 0F0E0D0C0B0A0908 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001111 00001110 00001101 00001100 00001011 00001010 00001001 00001000

1. До начала преобразований определим S-блоки, каждый из которых предоставляет 256 пар шестнадцатеричного и бинарного значения.

S1

00: 10101000

…

FF: 10101110

S2

00: 11100010

…

FF: 11100000

S3

00: 10110100

…

FF: 00111101

S4

00: 10110100

…

FF: 00111101

1. Разделяем 512 бит сообщения на 8 блоков по 64 бита.

Блок 1:

01000011 01010011 01010100 01001001 01000111 01000101 01010010 01000001

Блок 2:

01001100 01000111 10000000 00000000 00000000 00000000 00000000 00000000

Блок 3:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Блок 4:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Блок 5:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Блок 6:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

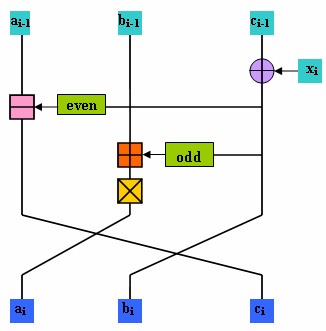
Блок 7:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Блок 8:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 01010000

1. Производим смешивание по следующей схеме



Пример одной итерации

Берём стартовые значения A, B, C и Первый блок из 8 байт (X)

1. C XOR X

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001111 00001110 00001101 00001100 00001011 00001010 00001001 00001000

XOR

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01000011 01010011 01010100 01001001 01000111 01000101 01010010 01000001

=

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01001100 01011101 01011001 01000101 01001100 01001111 01011011 01001001

1. A -= S1[C0] XOR S2[C2] XOR S3[С4] XOR S4[C6]

00000001 00100101 00110100 01011000 01100101 01101111 01110101 01111000 01101001 00111010 01111001 00110100 00101000 01100100 01101101 01100100

-

S1[00000000 00000000] XOR S2[00000000 00000000] XOR

S3[00001111 00001110] XOR S4[00001011 00001010]

=

00000001 00100101 00110100 01011000 01100101 01101111 01110101 01111000 01101001 00111010 01111001 00110100 00101000 01100100 01101101 01100100

-

10101000 10101000 XOR 11100010 11100010 XOR 10011001 10101000 XOR 10001011 11010110

=

00000001 00100101 00110100 01011000 01100101 01101111 01110101 01111000 01101001 00111010 01111001 00110100 00101000 01100100 01101101 01100100

-

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 10001110

=

00000001 00100101 00110100 01011000 01100101 01101111 01110101 01111000 01101001 00111010 01111001 00110100 00101000 01100010 00101010

1. B += S3[C1] XOR S2[C3] XOR S1[С5] XOR S0[C7]

Аналогичные вычисления

1. B \*= Y – уникальное для каждой итерации нечётное значение
2. Совершаем перемешивание

Ai = Bi-1

Bi = Ci-1

Ci = Ai-1

1. Генерация ключа, который будет оказывать влияние на последующем повторном выполнении шага 8

x0 -= x7 ^ 0xA5A5A5A5A5A5A5A5

x1 ^= x0

x2 += x1

x3 -= x2 ^ ((~x1)<<19)

x4 ^= x3

x5 += x4

x6 -= x5 ^ ((~x4)>>23)

x7 ^= x6

x0 += x7

x1 -= x0 ^ ((~x7)<<19)

x2 ^= x1

x3 += x2

x4 -= x3 ^ ((~x2)>>23)

x5 ^= x4

x6 += x5

x7 -= x6 ^ 0x0123456789ABCDEF

1. Повторяем шаги 8 и 9 и 8 в установленном порядке
2. Изменяем новые значения A, B, C, проводя операции с их исходными версиями

a ^= aa

b -= bb

c += cc

1. Повторяем 7 раз

Code

using System.Text;  
  
namespace Tiger;  
  
public static class TigerAlg  
{  
 static int[] S1 = new int[]  
 {  
 168, 48, 25, 84, 120, 46, 31, 215, 33, 250, 136, 255, 135, 65, 105, 185,  
 208, 161, 12, 98, 145, 194, 38, 78, 227, 177, 118, 66, 219, 18, 6, 95,  
 229, 144, 220, 45, 151, 55, 112, 183, 205, 59, 90, 23, 88, 20, 232, 117,  
 60, 129, 242, 94, 216, 166, 237, 40, 92, 221, 128, 126, 234, 195, 104, 58,  
 252, 8, 184, 114, 52, 15, 165, 96, 30, 228, 26, 80, 142, 238, 32, 157,  
 110, 3, 189, 202, 231, 146, 231, 115, 124, 61, 70, 92, 151, 103, 114, 122,  
 223, 120, 113, 5, 155, 185, 206, 234, 145, 196, 31, 53, 173, 167, 57, 103,  
 187, 201, 43, 146, 149, 47, 12, 231, 19, 125, 42, 97, 162, 36, 233, 116,  
 35, 80, 181, 25, 86, 224, 227, 78, 38, 151, 226, 208, 154, 177, 142, 241,  
 25, 48, 134, 66, 30, 168, 5, 211, 78, 79, 184, 3, 237, 198, 28, 66,  
 147, 206, 169, 211, 53, 59, 49, 93, 173, 207, 4, 58, 175, 224, 170, 63,  
 119, 113, 15, 104, 166, 59, 227, 181, 64, 231, 118, 221, 205, 102, 170, 254,  
 185, 154, 75, 183, 11, 71, 221, 67, 214, 206, 148, 160, 57, 39, 237, 222,  
 108, 164, 92, 204, 244, 115, 137, 48, 124, 179, 172, 200, 91, 82, 73, 89,  
 112, 77, 16, 99, 97, 225, 233, 7, 184, 243, 12, 27, 40, 191, 63, 180,  
 129, 159, 178, 126, 107, 102, 134, 130, 195, 172, 241, 13, 44, 156, 79, 227  
 };  
  
 static int[] S2 = new int[]  
 {  
 142, 99, 131, 21, 245, 179, 170, 55, 63, 210, 151, 132, 14, 252, 71, 110,  
 38, 178, 8, 98, 173, 206, 93, 137, 124, 160, 244, 19, 218, 176, 253, 82,  
 89, 195, 141, 77, 87, 224, 135, 11, 201, 33, 118, 235, 209, 155, 22, 41,  
 69, 233, 142, 251, 164, 53, 97, 171, 16, 40, 109, 24, 226, 76, 30, 59,  
 191, 214, 190, 83, 92, 13, 175, 159, 87, 234, 202, 169, 121, 48, 141, 243,  
 6, 50, 178, 150, 67, 152, 194, 27, 39, 197, 88, 103, 95, 205, 26, 237,  
 9, 212, 228, 134, 74, 83, 116, 91, 31, 241, 100, 181, 156, 47, 154, 36,  
 62, 221, 198, 119, 158, 203, 27, 212, 68, 222, 4, 25, 208, 237, 99, 194,  
 240, 147, 204, 148, 3, 199, 161, 163, 79, 112, 246, 37, 145, 136, 129, 217,  
 90, 104, 42, 81, 46, 226, 232, 164, 254, 7, 72, 66, 9, 139, 207, 115,  
 35, 96, 138, 230, 125, 162, 85, 211, 130, 122, 216, 223, 177, 66, 213, 74,  
 198, 123, 232, 168, 40, 49, 214, 102, 93, 215, 156, 241, 46, 137, 25, 160,  
 36, 109, 247, 195, 13, 69, 91, 174, 167, 27, 61, 238, 12, 152, 230, 133,  
 53, 87, 190, 150, 206, 101, 165, 128, 22, 145, 240, 33, 112, 191, 154, 94,  
 212, 183, 182, 223, 71, 143, 111, 50, 211, 47, 188, 107, 141, 139, 235, 202,  
 222, 59, 232, 199, 73, 115, 64, 173, 163, 53, 61, 201, 195, 120, 20, 200  
 };  
  
 static int[] S3 = new int[]  
 {  
 72, 35, 81, 237, 158, 225, 46, 108, 156, 130, 231, 120, 217, 149, 199, 24,  
 97, 43, 200, 75, 112, 166, 145, 22, 195, 159, 26, 242, 56, 185, 139, 253,  
 88, 214, 68, 223, 101, 119, 167, 45, 123, 211, 182, 141, 100, 191, 58, 36,  
 77, 64, 218, 240, 151, 15, 72, 249, 35, 161, 180, 57, 109, 11, 234, 239,  
 41, 144, 198, 5, 183, 42, 73, 203, 157, 217, 202, 213, 174, 62, 220, 230,  
 4, 84, 116, 95, 211, 197, 54, 10, 251, 91, 70, 113, 28, 126, 7, 121,  
 133, 8, 244, 127, 147, 90, 33, 29, 204, 155, 132, 18, 12, 6, 32, 110,  
 188, 85, 187, 106, 231, 61, 215, 247, 138, 98, 222, 51, 170, 19, 102, 232,  
 252, 48, 249, 221, 125, 53, 153, 3, 17, 237, 115, 252, 158, 75, 250, 219,  
 233, 186, 207, 76, 128, 218, 205, 139, 236, 222, 68, 118, 70, 91, 100, 252,  
 223, 43, 242, 229, 79, 55, 36, 74, 175, 208, 154, 253, 92, 135, 163, 124,  
 185, 94, 194, 37, 16, 96, 78, 227, 189, 111, 241, 186, 220, 102, 146, 82,  
 12, 97, 59, 209, 88, 236, 135, 144, 139, 250, 234, 162, 240, 248, 45, 172,  
 147, 193, 71, 148, 14, 149, 219, 115, 212, 206, 221, 59, 111, 2, 103, 250,  
 254, 17, 226, 65, 116, 20, 67, 204, 143, 44, 107, 29, 156, 170, 243, 195,  
 215, 173, 65, 81, 231, 12, 67, 225, 205, 74, 94, 163, 204, 11, 218, 249  
 };  
  
 static int[] S4 = new int[]  
 {  
 57, 247, 162, 201, 39, 194, 165, 150, 19, 243, 64, 232, 66, 25, 14, 170,  
 221, 48, 192, 154, 216, 69, 217, 180, 28, 99, 105, 90, 8, 59, 144, 139,  
 111, 17, 227, 153, 15, 214, 63, 137, 155, 31, 100, 238, 80, 128, 98, 250,  
 96, 56, 51, 199, 239, 140, 244, 10, 219, 167, 235, 215, 22, 95, 174, 84,  
 197, 77, 205, 87, 164, 176, 34, 103, 84, 110, 133, 131, 136, 85, 71, 228,  
 7, 72, 145, 230, 182, 146, 117, 43, 52, 148, 149, 251, 109, 174, 140, 99,  
 188, 88, 192, 20, 227, 41, 216, 75, 73, 119, 86, 190, 247, 2, 29, 44,  
 112, 83, 184, 249, 178, 161, 183, 35, 167, 60, 218, 106, 229, 220, 114, 147,  
 137, 142, 250, 211, 204, 171, 130, 181, 50, 78, 6, 175, 91, 94, 237, 160,  
 71, 49, 58, 40, 30, 120, 9, 55, 25, 4, 93, 212, 65, 1, 118, 193,  
 45, 68, 243, 239, 169, 246, 52, 248, 115, 47, 166, 95, 249, 37, 210, 125,  
 243, 66, 109, 57, 20, 116, 162, 208, 172, 152, 26, 82, 187, 101, 201, 3,  
 36, 187, 207, 163, 12, 171, 54, 123, 214, 122, 81, 238, 93, 121, 211, 123,  
 23, 18, 143, 176, 65, 34, 24, 189, 222, 49, 67, 184, 104, 108, 124, 104,  
 74, 247, 138, 132, 155, 223, 177, 125, 29, 233, 38, 141, 5, 240, 121, 89,  
 212, 190, 38, 241, 146, 250, 227, 114, 93, 23, 241, 102, 203, 108, 177, 181  
 };  
   
 public static string GetHash(string message)  
 {  
 byte[] binaryMessage = Encoding.UTF8.GetBytes(message);  
  
 byte[] fullMessage = new byte[64];  
 int length = binaryMessage.Length;  
  
 for (int i = 0; i < length; i++)  
 {  
 fullMessage[i] = binaryMessage[i];  
 }  
  
 fullMessage[length] = 0x80;  
  
 for (int i = length; i < 64; i++)  
 {  
 fullMessage[i] = 0;  
 }  
  
 for (int i = 62 - (length \* 8 / 256); i < 62; i++)  
 {  
 fullMessage[i] = 255;  
 }  
  
 for (int i = 62 - (length \* 8 / 256); i < 62; i++)  
 {  
 fullMessage[i] = 255;  
 }  
  
 fullMessage[63] = (byte)((length \* 8) % 256);  
  
 List<List<byte>> blocks = new List<List<byte>>();  
  
 for (int i = 0; i < 8; i++)  
 {  
 blocks.Add(new List<byte>());  
 for (int j = 0; j < 8; j++)  
 {  
 blocks[i].Add(fullMessage[j + (8 \* i)]);  
 }  
 }  
  
 byte[] a = BitConverter.GetBytes(0x0123456789ABCDEF);  
 byte[] b = BitConverter.GetBytes(0xFEDCBA9876543210);  
 byte[] c = BitConverter.GetBytes(0xF096A5B4C3B2E187);  
  
 Tuple<byte[], byte[], byte[]> tpl = new Tuple<byte[], byte[], byte[]>(a, b, c);  
 Tuple<byte[], byte[], byte[]> tplPrevious = new Tuple<byte[], byte[], byte[]>(a, b, c);  
  
  
 for (int i = 0; i < 8; i++)  
 {  
 tplPrevious = new Tuple<byte[], byte[], byte[]>(tpl.Item1, tpl.Item2, tpl.Item3);  
  
 tpl = Pass(tpl, blocks, 5);  
 Key(blocks);  
 tpl = Pass(tpl, blocks, 7);  
 Key(blocks);  
 tpl = Pass(tpl, blocks, 9);  
 Key(blocks);  
  
 for (int j = 0; j < 8; j++)  
 {  
 tpl.Item1[j] ^= a[j];  
 }  
  
 for (int j = 0; j < 8; j++)  
 {  
 tpl.Item2[j] -= b[j];  
 }  
  
 for (int j = 0; j < 8; j++)  
 {  
 tpl.Item3[j] += c[j];  
 }  
 }  
  
 string result = "";  
  
 foreach (var block in blocks)  
 {  
 foreach (var bt in block)  
 {  
 result += $"{bt:X}";  
 }  
 }  
  
 return result;  
 }  
  
 private static void Key(List<List<byte>> blocks)  
 {  
 var x0 = blocks[0];  
 var x1 = blocks[1];  
 var x2 = blocks[2];  
 var x3 = blocks[3];  
 var x4 = blocks[4];  
 var x5 = blocks[5];  
 var x6 = blocks[6];  
 var x7 = blocks[7];  
   
 byte[] constant1 = BitConverter.GetBytes(0xA5A5A5A5A5A5A5A5);  
 byte[] constant2 = BitConverter.GetBytes(0x0123456789ABCDEF);  
   
 int length = Math.Max(x0.Count, x7.Count);  
 for (int i = x0.Count; i < length; i++) x0.Add(0);  
 for (int i = x7.Count; i < length; i++) x7.Add(0);  
   
 for (int i = 0; i < length; i++)  
 {  
 x0[i] = (byte)(x0[i] - (x7[i] ^ constant1[i % 8]));  
 x1[i] ^= x0[i];  
 x2[i] = (byte)(x2[i] + x1[i]);  
 x3[i] = (byte)(x3[i] - (x2[i] ^ (~x1[i] << 19)));  
 x4[i] ^= x3[i];  
 x5[i] = (byte)(x5[i] + x4[i]);  
 x6[i] = (byte)(x6[i] - (x5[i] ^ (~x4[i] >> 23)));  
 x7[i] ^= x6[i];  
 x0[i] = (byte)(x0[i] + x7[i]);  
 x1[i] = (byte)(x1[i] - (x0[i] ^ (~x7[i] << 19)));  
 x2[i] ^= x1[i];  
 x3[i] = (byte)(x3[i] + x2[i]);  
 x4[i] = (byte)(x4[i] - (x3[i] ^ (~x2[i] >> 23)));  
 x5[i] ^= x4[i];  
 x6[i] = (byte)(x6[i] + x5[i]);  
 x7[i] = (byte)(x7[i] - (x6[i] ^ constant2[i % 8]));  
 }  
 }  
  
 private static Tuple<byte[], byte[], byte[]> Pass(Tuple<byte[], byte[], byte[]> tpl, List<List<byte>> blocks,  
 int mul)  
 {  
 tpl = Round(tpl.Item1, tpl.Item2, tpl.Item3, blocks[0], mul);  
 tpl = Round(tpl.Item2, tpl.Item3, tpl.Item1, blocks[1], mul);  
 tpl = Round(tpl.Item3, tpl.Item1, tpl.Item2, blocks[2], mul);  
 tpl = Round(tpl.Item1, tpl.Item2, tpl.Item3, blocks[3], mul);  
 tpl = Round(tpl.Item2, tpl.Item3, tpl.Item1, blocks[4], mul);  
 tpl = Round(tpl.Item3, tpl.Item1, tpl.Item2, blocks[5], mul);  
 tpl = Round(tpl.Item1, tpl.Item2, tpl.Item3, blocks[6], mul);  
 tpl = Round(tpl.Item2, tpl.Item3, tpl.Item1, blocks[7], mul);  
  
 return tpl;  
 }  
  
 private static Tuple<byte[], byte[], byte[]> Round(byte[] a, byte[] b, byte[] c, List<byte> x, int mul)  
 {  
 for (int i = 0; i < 8; i++)  
 {  
 c[i] ^= x[i];  
 }  
  
 var tempA = BitConverter.GetBytes(BitConverter.ToInt32(a) - S1[c[0]] ^ S2[c[2]] ^ S3[c[4]] ^ S4[c[6]]);  
  
 for (int i = 0; i < 8; i++)  
 {  
 a[i] = 0;  
 }  
  
 Array.Copy(tempA, 0, a, 8 - tempA.Length, tempA.Length);  
  
 var tempB = BitConverter.GetBytes(BitConverter.ToInt32(b) + S4[c[1]] ^ S3[c[3]] ^ S2[c[5]] ^ S1[c[7]]);  
  
 for (int i = 0; i < 8; i++)  
 {  
 a[i] = 0;  
 }  
  
 Array.Copy(tempB, 0, b, 8 - tempB.Length, tempB.Length);  
  
 var tempBB = BitConverter.GetBytes(BitConverter.ToInt32(b) \* mul);  
  
 for (int i = 0; i < 8; i++)  
 {  
 a[i] = 0;  
 }  
  
 Array.Copy(tempBB, 0, b, 8 - tempBB.Length, tempBB.Length);  
  
 return new Tuple<byte[], byte[], byte[]>(a, b, c);  
 }  
}