Министерство образования Республики Беларусь Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ

И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №5

По теме «Хэш-функция»

Выполнила: студентка гр. 053501 Шурко Т.А.

Проверил: ассистент кафедры информатики Лещенко Е. А.

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc150282930)

[1 Теоретические сведения 4](#_Toc150282931)

[1.1 Алгоритм ГОСТ 34.11 4](#_Toc150282932)

[1.2 Блок-схема алгоритма ГОСТ 34.11 4](#_Toc150282933)

[1.3 Хэш-функция SHA-1 6](#_Toc150282934)

[1.4 Блок-схема хэширования SHA-1 6](#_Toc150282935)

[2 Резульата выполнения задачи 7](#_Toc150282936)

[Заключение 8](#_Toc150282937)

[Приложение А 9](#_Toc150282938)

[Приложение Б 13](#_Toc150282939)

# ВВЕДЕНИЕ

2. Шифрование — обратимое преобразование информации в целях сокрытия от неавторизованных лиц, с предоставлением, в это же время, авторизованным пользователям доступа к ней. Главным образом, шифрование служит задачей соблюдения конфиденциальности передаваемой информации. Важной особенностью любого алгоритма шифрования является использование ключа, который утверждает выбор конкретного преобразования из совокупности возможных для данного алгоритма.
3. Пользователи являются авторизованными, если они обладают определенным аутентичным ключом. Вся сложность и, собственно, задача шифрования состоит в том, как именно реализован этот процесс.
4. В целом, шифрование состоит из двух составляющих — зашифрование и расшифрование.
5. С помощью шифрования обеспечиваются три состояния безопасности информации. Первой является конфиденциальность: шифрование используется для скрытия информации от неавторизованных пользователей при передаче или при хранении. Второе — целостность. Шифрование используется для предотвращения изменения информации при передаче или хранении. Последнее это идентифицируемость. Шифрование используется для аутентификации источника информации и предотвращения отказа отправителя информации от того факта, что данные были отправлены именно им.
6. Для выполнения лабораторной работы 5 необходимо реализовать программное средство контроля целостности сообщений с помощью вычисления хэш-функции SHA-1 и алгоритма ГОСТ 34.11.

# 1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## 1.1 Алгоритм ГОСТ 34.11

ГОСТ 34.11 «Информационная технология. Криптографическая защита информации. Функция хэширования» — [российский](https://ru.wikipedia.org/wiki/%D0%A0%D0%BE%D1%81%D1%81%D0%B8%D1%8F) [криптографический](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F) [стандарт](https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D1%81%D1%83%D0%B4%D0%B0%D1%80%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82) вычисления [хеш-функции](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F), основанный на ГОСТ Р 34.10-94.

Стандарт определяет алгоритм и процедуру вычисления хеш-функции для последовательности символов. Этот стандарт является обязательным для применения в качестве алгоритма хеширования в государственных организациях [РФ](https://ru.wikipedia.org/wiki/%D0%A0%D0%A4) и ряде коммерческих организаций.

Стандарт определяет 2 функции хэширования с длинами хэш-кода 256 и 512 бит.

## 1.2 Блок-схема алгоритма ГОСТ 34.11

В данном подразделе на рисунке 1.1 изображена блок-схема алгоритма 34-11. Процедуру вычисления хэш-функции можно разделить на 3 основных этапа: на первом происходит инициализация величин h, n, sum для вычисления основных шагов алгоритма; на втором этапе идет вычисление хэша для сообщения, размером 512 бит; на финальном этапе вычисление неполного блока и возвращение хэш-функции размером 256 или 512 бит соответственно.

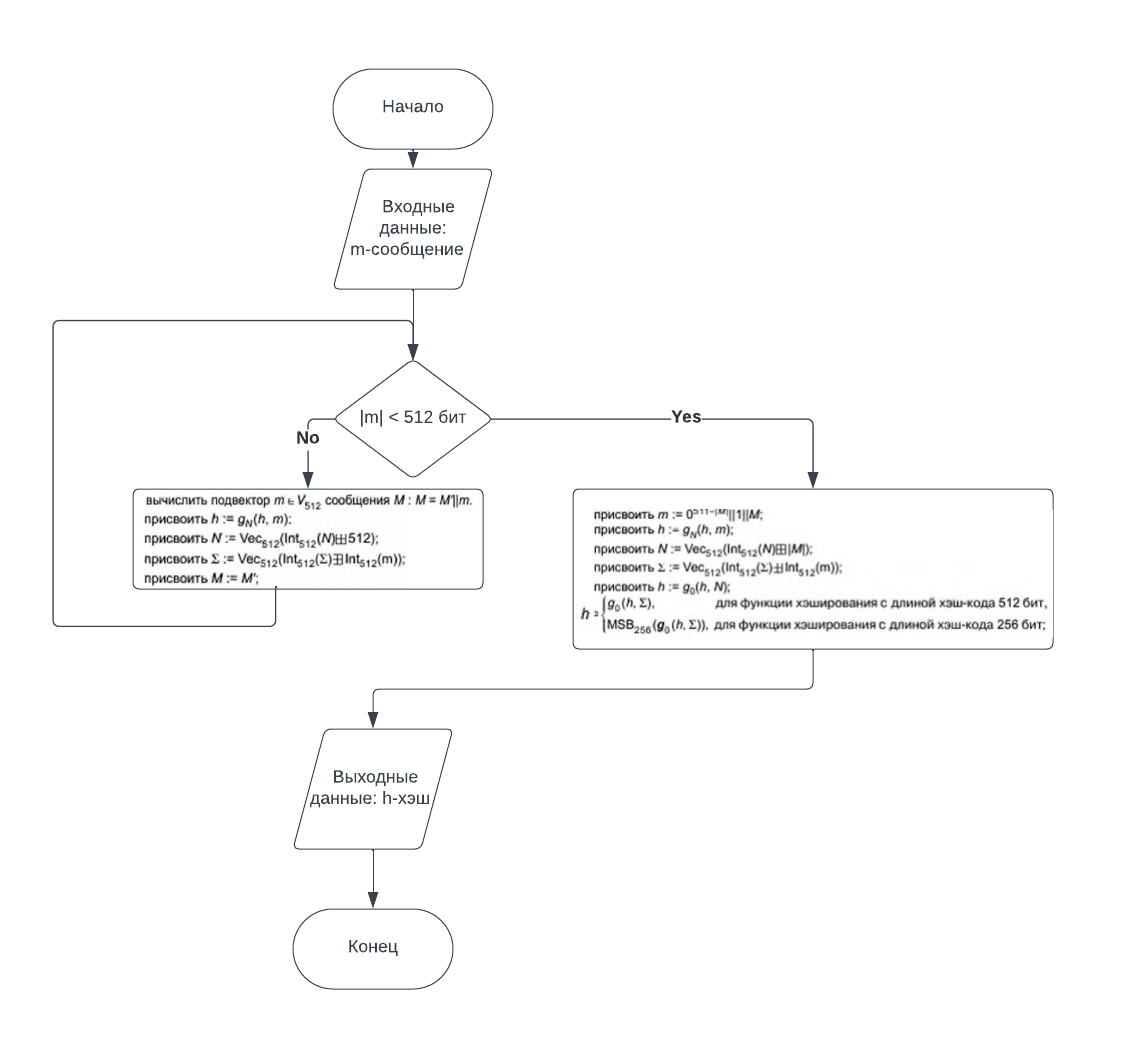


Рисунок 1.1 — Блок-схема алгоритма ГОСТ 34-11

На рисунке 1.2 схема gN-преобразования, которое отображено на основной блок-схеме алгоритма. Преобразование осуществляется с помощью вспомогательной итерационной матрицы констант Сi,j.

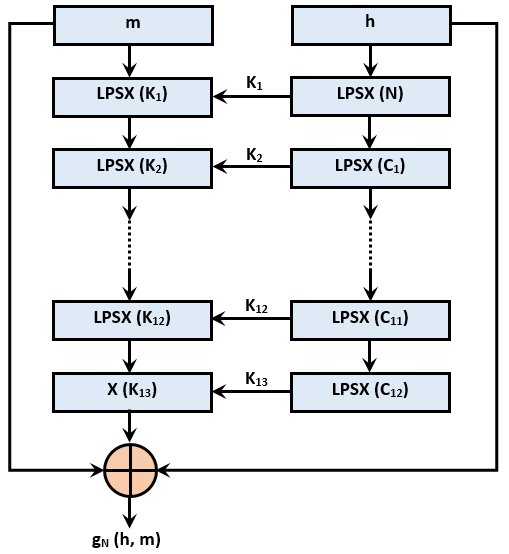


Рисунок 1.2 — Схема gN-преобразования

## 1.3 Хэш-функция SHA-1

Secure Hash Algorithm 1 (SHA-1) — алгоритм [криптографического хеширования](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F).

Для входного сообщения произвольной длины (максимум 264 бита), алгоритм генерирует 160-битное (20 байт) хеш-значение, которое обычно отображается как шестнадцатеричное число длиной в 40 цифр. Используется во многих криптографических приложениях и протоколах. Также рекомендован в качестве основного для государственных учреждений в [США](https://ru.wikipedia.org/wiki/%D0%A1%D0%A8%D0%90).

## 1.4 Блок-схема хэширования SHA-1

На рисунке 1.3 изображена блок-схема алгоритма хэширования SHA-1. Процедуру вычисления хэш-функции можно разделить на 3 основных этапа: на первом происходит инициализация и подготовка исходного сообщения; на втором этапе идет вычисление хэша для сообщения, размером 512 бит; на финальном этапе вычисление хеш-функции, состоящей из вспомогательных переменных h0,h1,h2,h3,h4.

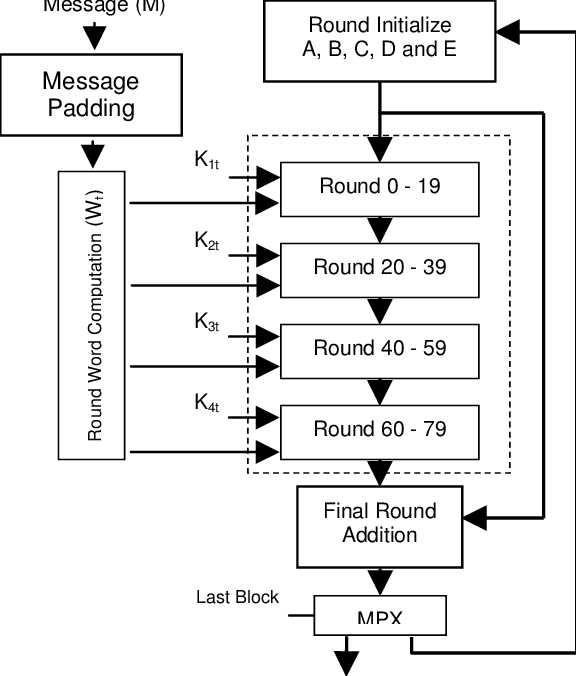


Рисунок 1.3 — Блок-схема SHA-1

# 2 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ЗАДАЧИ

В результате выполнения задачи было реализовано хэширование сообщения из текстового файла с помощью алгоритмов ГОСТ 34.11 и SHA-1 на языке программирования С#. Текст для шифрования необходимо поместить в файл и передать его наименование в виде первого параметра, хэш после применения алгоритма будет помещен в файл с наименованием, указанным во втором параметре. Текст исполняемой программы для хэширования алгоритмом ГОСТ 34.11 в приложении А, для SHA-1 — в приложении Б.

Для запуска программного продукта необходимо подключить классы Hasher3411 и HasherSHA1, а также указать файл, из которого берется текст и файл, в который будет помещаться результат работы (Рисунок 2.1). При запуске хэширования алгоритмом ГОСТ 34-11 можно указать размер хэша, который будет сгенерирован: 256 или 512 бит.

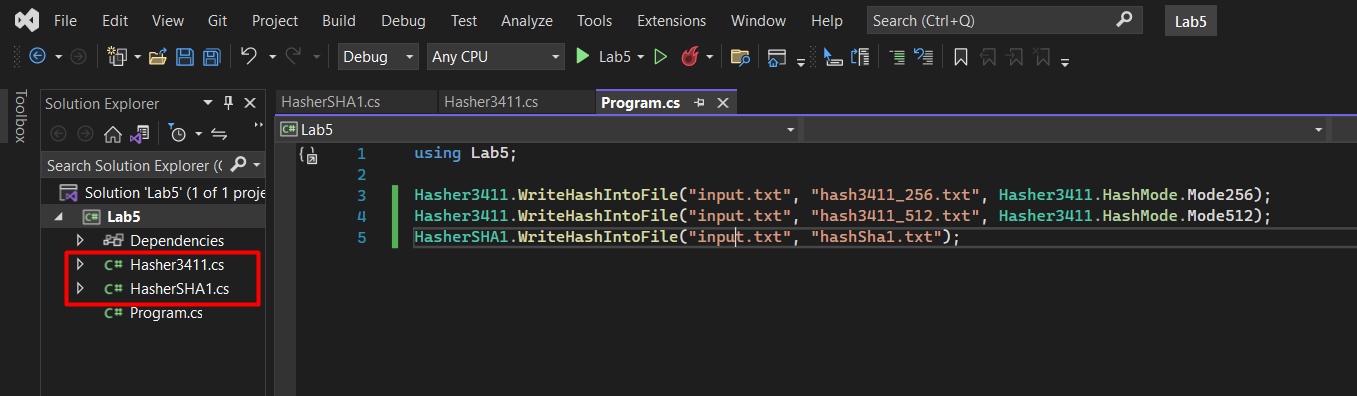


Рисунок 2.1 — Запускаемая программа

# ЗАКЛЮЧЕНИЕ

При выполнении лабораторной работы были изучены и реализованы алгоритмы хэширования ГОСТ 34-11 и SHA-1, рассмотрены и составлены блок-схемы алгоритмов, произведена их реализация на языке программирования C#, а также проверена работоспособность программного продукта.

Алгоритм ГОСТ 34-11 определяет процедуру вычисления хэш-функции для любой последовательности двоичных символов, которые применяются в криптографических методах обработки и защиты информации, в том числе для реализации процедур обеспечения целостности, аутентичности, электронной цифровой подписи при передаче обработке и хранении информации в автоматизированных системах. Данная хэш-функция используется при реализации систем электронной цифровой подписи на базе ассиметричного криптографического алгоритма ГОСТ 34.10.

Хеш-функции SHA-1 используются в [системах контроля версий](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8), системах электронной подписи, а также для построения кодов [аутентификации](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F).

SHA-1 является наиболее распространенным из всего семейства [SHA](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%8B_%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B3%D0%BE_%D1%85%D1%8D%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F&action=edit&redlink=1) и применяется в различных широко распространенных криптографических приложениях и алгоритмах.

Теоретическая криптостойкость алгоритма ГОСТ 34-11 намного превышает SHA-1, от использования которого постепенно отказываются мировые лидеры, вроде Google и Yandex.

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Код программы**

**Hasher3411.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

using System.Security.Cryptography;

using System.Text;

using System.Threading.Tasks;

namespace Lab5 {

public class Hasher3411 {

// 256

private static readonly byte[] piSubstitution = new byte[] {..};

// 64

private static readonly byte[] tauSubstitution = new byte[] {…};

//64\*16

private static readonly ulong[] matrixA = new ulong[] {…};

//12\*64

private static readonly byte[][] iterationConstants = new byte[12][] {…};

public enum HashMode {

Mode256 = 256,

Mode512 = 512

}

private static byte[] Xor64(byte[] byteArr1, byte[] byteArr2) {

if (byteArr1.Length != 64 || byteArr2.Length != 64) {

throw new Exception("Wrong input parameters. Expecting byte array, length 64");

}

byte[] result = new byte[64];

for (int i = 0; i < 64; i++) {

result[i] = (byte)(byteArr1[i] ^ byteArr2[i]);

}

return result;

}

private static byte[] TranslationS64(byte[] bytes) {

if (bytes.Length != 64) {

throw new Exception("Wrong input parameters. Expecting byte array, length 64");

}

byte[] result = new byte[64];

for (int i = 0; i < 64; i++) {

result[i] = piSubstitution[bytes[i]];

}

return result;

}

private static byte[] TranslationP64(byte[] bytes) {

if (bytes.Length != 64) {

throw new Exception("Wrong input parameters. Expecting byte array, length 64");

}

byte[] result = new byte[64];

for (int i = 0; i < 64; i++) {

result[i] = bytes[tauSubstitution[i]];

}

return result;

}

private static byte[] TranslationL64(byte[] bytes) {

if (bytes.Length != 64) {

throw new Exception("Wrong input parameters. Expecting byte array, length 64");

}

byte[] result = new byte[64];

byte[] tempArray = new byte[8];

for (int i = 0; i < 8; i++) {

Array.Copy(bytes, 8 \* i, tempArray, 0, 8);

UInt64 uIntBytes = BitConverter.ToUInt64(tempArray);

ulong accum = 0;

for (int j = 0; j < 64; j++) {

accum ^= ((uIntBytes >> j) & 1) \* matrixA[63 - j];

}

result[8 \* i] = (byte)accum;

result[8 \* i + 1] = (byte)(accum >> 8);

result[8 \* i + 2] = (byte)(accum >> 16);

result[8 \* i + 3] = (byte)(accum >> 24);

result[8 \* i + 4] = (byte)(accum >> 32);

result[8 \* i + 5] = (byte)(accum >> 40);

result[8 \* i + 6] = (byte)(accum >> 48);

result[8 \* i + 7] = (byte)(accum >> 56);

}

return result;

}

private static byte[] TranslationE64(byte[] k, byte[] m) {

byte[] result = new byte[64];

m.CopyTo(result, 0);

for (int i = 0; i < 12; i++) {

result = Xor64(k, result);

result = TranslationS64(result);

result = TranslationP64(result);

result = TranslationL64(result);

k = Xor64(k, iterationConstants[i]);

k = TranslationS64(k);

k = TranslationP64(k);

k = TranslationL64(k);

}

result = Xor64(k, result);

return result;

}

private static byte[] TranslationG64(byte[] n, byte[] h, byte[] m) {

byte[] result = new byte[64];

result = Xor64(h, n);

result = TranslationS64(result);

result = TranslationP64(result);

result = TranslationL64(result);

result = TranslationE64(result, m);

result = Xor64(result, h);

result = Xor64(result, m);

return result;

}

public static byte[] GetHash(byte[] data, HashMode mode=HashMode.Mode256) {

// Get Block Parameters

int size = data.Length;

int fullBlocks = size / 64;

int partialBlock = size % 64;

// Initialize arrays

byte[] h = new byte[64];

byte[] n = new byte[64];

byte[] sum = new byte[64];

byte[] message = new byte[64];

for (int i = 0; i < 64; i++) {

h[i] = 0;

n[i] = 0;

sum[i] = 0;

}

for (int i = 1; i <= fullBlocks; i++) {

Array.Copy(data, size - 64 \* i, message, 0, 64);

h = TranslationG64(n, h, message);

byte[] tempArrayN = (new BigInteger(n, isUnsigned: true) + 512).ToByteArray();

Array.Copy(tempArrayN, n, tempArrayN.Length);

byte[] tempArraySum = (new BigInteger(sum, isUnsigned: true) + new BigInteger(message, isUnsigned: true)).ToByteArray();

Array.Copy(tempArraySum, sum, tempArraySum.Length);

}

if (partialBlock != 0) {

byte[] newMessage = new byte[64];

byte[] tempArray = new byte[partialBlock];

Array.Copy(data, 0, tempArray, 0, partialBlock);

int index = 0;

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

if (i < 64 - partialBlock - 2) {

newMessage[i] = 0;

} else if (i == 64 - partialBlock - 1) {

newMessage[i] = 1;

index = 0;

} else {

newMessage[i] = tempArray[index];

index++;

}

}

h = TranslationG64(n, h, newMessage);

byte[] tempArrayN = (new BigInteger(n, isUnsigned: true) + partialBlock\*8).ToByteArray();

Array.Copy(tempArrayN, n, tempArrayN.Length);

byte[] tempArraySum = (new BigInteger(sum, isUnsigned: true) + new BigInteger(newMessage, isUnsigned: true)).ToByteArray();

Array.Copy(tempArraySum, sum, tempArraySum.Length);

}

h = TranslationG64(new byte[64], h, n);

h = TranslationG64(new byte[64], h, sum);

if (mode == HashMode.Mode256) {

if (h.Length < 32) {

byte[] expanded = new byte[32];

Array.Copy(h, expanded, h.Length);

h = expanded;

}

if (h.Length > 32) {

h = h[..32];

}

}

return h.ToArray();

}

public static void WriteHashIntoFile(string inputFileName, string ouputFileName, HashMode mode=HashMode.Mode256) {

if (!File.Exists(inputFileName)) {

File.WriteAllText(inputFileName, string.Empty);

}

byte[] message = File.ReadAllBytes(inputFileName);

byte[] hash = GetHash(message, mode);

string hashedString3411 = BitConverter.ToString(hash).Replace("-", string.Empty).ToLowerInvariant();

File.WriteAllText(ouputFileName, hashedString3411);

}

}

}

**Program.cs**

using Lab5;

Hasher3411.WriteHashIntoFile("input.txt", "hash3411\_256.txt", Hasher3411.HashMode.Mode256);

Hasher3411.WriteHashIntoFile("input.txt", "hash3411\_512.txt", Hasher3411.HashMode.Mode512);

# ПРИЛОЖЕНИЕ Б

**(обязательное)**

**Код программы**

**HasherSHA1.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using static Lab5.Hasher3411;

namespace Lab5 {

public class HasherSHA1 {

private static uint h0 = 0x67452301;

private static uint h1 = 0xEFCDAB89;

private static uint h2 = 0x98BADCFE;

private static uint h3 = 0x10325476;

private static uint h4 = 0xC3D2E1F0;

private static uint LeftRotate(uint x, int c)

=> (x << c) | (x >> (32 - c));

private static uint GetUintFromBytes(byte[] bytes) {

byte[] value = new byte[8];

Array.Reverse(bytes);

Array.Copy(bytes, value, bytes.Length);

uint result = (uint)BitConverter.ToUInt64(value, 0);

return result;

}

public static byte[] GetHash(byte[] bytes) {

// message length always is a multiple of 8 bits

var processedInputBuilder = new List<byte>(bytes) { 0x80 };

while (processedInputBuilder.Count % 64 != 56)

processedInputBuilder.Add(0x0);

processedInputBuilder.AddRange(BitConverter.GetBytes((long)bytes.Length \* 8).Reverse()); // bit converter returns big-endian

byte[] processedInput = processedInputBuilder.ToArray();

int blockAmount = processedInput.Length / 64;

for (int i = 0; i < blockAmount; i++) {

byte[] temp = new byte[64];

Array.Copy(processedInput, i\*64, temp, 0, 64);

byte[][] w = new byte[80][];

for (int j = 0; j < 16; j++) {

w[j] = new byte[4];

Array.Copy(temp, 4\*j, w[j], 0, 4);

}

for (int j = 16; j < 79; j++) {

w[j] = new byte[4];

w[j] = Xor(w[j], w[j - 3]);

w[j] = Xor(w[j], w[j - 8]);

w[j] = Xor(w[j], w[j - 14]);

w[j] = Xor(w[j], w[j - 16]);

// Left Rotate 1 (differs from sha-0)

byte tempByte = w[j][0];

for (int p = 0; p < 3; p++) {

w[j][p] = w[j][p + 1];

}

w[j][3] = tempByte;

}

uint a = h0, b=h1, c=h2, d=h3, e=h4, f, k;

for (int j = 0; j < 79; j++) {

if (i >= 0 && i <= 19) {

f = (b & c) | (~b & d);

k = 0x5A827999;

} else if (i >= 20 && i <= 39) {

f = b ^ c ^ d;

k = 0x6ED9EBA1;

} else if (i >= 40 && i <= 59) {

f = (b & c) | (b & d) | (c & d);

k = 0x8F1BBCDC;

} else {

f = b ^ c ^ d;

k = 0xCA62C1D6;

}

uint tempUint = LeftRotate(a, 5) + f + e + k + GetUintFromBytes(w[j]);

e = d;

d = c;

c = LeftRotate(b, 30);

b = a;

a = tempUint;

}

h0 = h0 + a;

h1 = h1 + b;

h2 = h2 + c;

h3 = h3 + d;

h4 = h4 + e;

}

byte[] hh = new byte[20];

Array.Copy(BitConverter.GetBytes(h0), 0, hh, 0, 4);

Array.Copy(BitConverter.GetBytes(h1), 0, hh, 4, 4);

Array.Copy(BitConverter.GetBytes(h2), 0, hh, 8, 4);

Array.Copy(BitConverter.GetBytes(h3), 0, hh, 12, 4);

Array.Copy(BitConverter.GetBytes(h4), 0, hh, 16, 4);

return hh;

}

private static byte[] Xor(byte[] byteArr1, byte[] byteArr2) {

if (byteArr1.Length != 4 || byteArr2.Length != 4) {

throw new Exception("Wrong input parameters. Expecting byte array, length 64");

}

byte[] result = new byte[4];

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

result[i] = (byte)(byteArr1[i] ^ byteArr2[i]);

}

return result;

}

public static void WriteHashIntoFile(string inputFileName, string ouputFileName) {

if (!File.Exists(inputFileName)) {

File.WriteAllText(inputFileName, string.Empty);

}

byte[] message = File.ReadAllBytes(inputFileName);

byte[] hash = GetHash(message);

string hashedString = BitConverter.ToString(hash).Replace("-", string.Empty).ToLowerInvariant();

File.WriteAllText(ouputFileName, hashedString);

}

}

}

**Program.cs**

using Lab5;

HasherSHA1.WriteHashIntoFile("input.txt", "hashSha1.txt");