Автор: Рудченко Д.О.

Група: КН-М922А

Лабораторна робота №1

Гешування

Мета: Дослідити принципи роботи гешування

Завдання: Дослідити існуючі механізми гешування. Реалізувати алгоритм гешування SHA (будь-якої версії). Довести коректність роботи реалізованого алгоритму шляхом порівняння результатів з існуючими реалізаціями.

Хід роботи:

SHA - Група алгоритмів, розроблених NSA Сполучених Штатів. Вони є частиною Федерального стандарту обробки інформації США. Ці алгоритми широко використовуються у кількох криптографічних додатках. Довжина повідомлення варіюється від 160 до 512 біт.

Код програми:

```
using System;
using System.Collections.ObjectModel;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;

namespace LB1
{
	class Program
	{
		static void Main()
	 {
	     string stringToHash = "Rudchenko";
	     byte[] bytesToHash = Encoding.UTF8.GetBytes(stringToHash);
	     Sha256 sha256 = new Sha256();
	     sha256.AddData(bytesToHash, 0, (uint) bytesToHash.Length);
	     byte[] hash = sha256.GetHash().ToArray();
	     string hexStringHash = ByteArrayToHexString(hash);
	     // Console.WriteLine("[{0}]", string.Join("", hash));
	     Console.WriteLine("Input word: " + stringToHash);
	     Console.WriteLine("Encrypted: " + hexStringHash);
	 }

	 public static byte[] HexStringToByteArray(string hex)
	 {
	     if (hex.Length % 2 == 1)
```

```
            {
                throw new Exception("The binary key cannot have an odd number of digits");
            }
            int GetHexVal(char c)
            {
                return c - (c < 58 ? 48 : 55);
            }
            hex = hex.ToUpper();
            byte[] arr = new byte[hex.Length >> 1];
            for (int i = 0; i < hex.Length >> 1; ++i)
            {
                arr[i] = (byte)((GetHexVal(hex[i << 1]) << 4) + (GetHexVal(hex[(i << 1) +
1])));
            }
            return arr;
        }

        public static string ByteArrayToHexString(byte[] ba)
        {
            StringBuilder hex = new StringBuilder(ba.Length * 2);
            foreach (byte b in ba)
            {
                hex.AppendFormat("{0:X2}", b);
            }
            return hex.ToString();
        }
       }
    public class Sha256
    {
        private static readonly UInt32[] K = new UInt32[64] {
            0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5, 0x3956C25B, 0x59F111F1,
0x923F82A4, 0xAB1C5ED5,
            0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3, 0x72BE5D74, 0x80DEB1FE,
0x9BDC06A7, 0xC19BF174,
            0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC, 0x2DE92C6F, 0x4A7484AA,
0x5CB0A9DC, 0x76F988DA,
            0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7, 0xC6E00BF3, 0xD5A79147,
0x06CA6351, 0x14292967,
            0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13, 0x650A7354, 0x766A0ABB,
0x81C2C92E, 0x92722C85,
            0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3, 0xD192E819, 0xD6990624,
0xF40E3585, 0x106AA070,
            0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5, 0x391C0CB3, 0x4ED8AA4A,
0x5B9CCA4F, 0x682E6FF3,
            0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208, 0x90BEFFFA, 0xA4506CEB,
0xBEF9A3F7, 0xC67178F2
        };

        private static UInt32 ROTL(UInt32 x, byte n)
        {
            Debug.Assert(n < 32);
            return (x << n) | (x >> (32 - n));
        }

        private static UInt32 ROTR(UInt32 x, byte n)
        {
            Debug.Assert(n < 32);
            return (x >> n) | (x << (32 - n));
        }
```

```
        private static UInt32 Ch(UInt32 x, UInt32 y, UInt32 z)
        {
            return (x & y) ^ ((~x) & z);
        }

        private static UInt32 Maj(UInt32 x, UInt32 y, UInt32 z)
        {
            return (x & y) ^ (x & z) ^ (y & z);
        }

        private static UInt32 Sigma0(UInt32 x)
        {
            return ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22);
        }

        private static UInt32 Sigma1(UInt32 x)
        {
            return ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25);
        }

        private static UInt32 sigma0(UInt32 x)
        {
            return ROTR(x, 7) ^ ROTR(x, 18) ^ (x >> 3);
        }

        private static UInt32 sigma1(UInt32 x)
        {
            return ROTR(x, 17) ^ ROTR(x, 19) ^ (x >> 10);
        }


        private UInt32[] H = new UInt32[8] {
            0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A, 0x510E527F, 0x9B05688C,
0x1F83D9AB, 0x5BE0CD19
        };

        private byte[] pending_block = new byte[64];
        private uint pending_block_off = 0;
        private UInt32[] uint_buffer = new UInt32[16];

        private UInt64 bits_processed = 0;

        private bool closed = false;

        private void processBlock(UInt32[] M)
        {
            Debug.Assert(M.Length == 16);

            // 1. Prepare the message schedule (W[t]):
            UInt32[] W = new UInt32[64];
            for (int t = 0; t < 16; ++t)
            {
                W[t] = M[t];
            }

            for (int t = 16; t < 64; ++t)
            {
                W[t] = sigma1(W[t - 2]) + W[t - 7] + sigma0(W[t - 15]) + W[t - 16];
            }
```

```
    // 2. Initialize the eight working variables with the (i-1)-st hash value:
    UInt32 a = H[0],
           b = H[1],
           c = H[2],
           d = H[3],
           e = H[4],
           f = H[5],
           g = H[6],
           h = H[7];

    // 3. For t=0 to 63:
    for (int t = 0; t < 64; ++t)
    {
        UInt32 T1 = h + Sigma1(e) + Ch(e, f, g) + K[t] + W[t];
        UInt32 T2 = Sigma0(a) + Maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;
    }

    // 4. Compute the intermediate hash value H:
    H[0] = a + H[0];
    H[1] = b + H[1];
    H[2] = c + H[2];
    H[3] = d + H[3];
    H[4] = e + H[4];
    H[5] = f + H[5];
    H[6] = g + H[6];
    H[7] = h + H[7];
}

public void AddData(byte[] data, uint offset, uint len)
{
    if (closed)
        throw new InvalidOperationException("Adding data to a closed hasher.");

    if (len == 0)
        return;

    bits_processed += len * 8;

    while (len > 0)
    {
        uint amount_to_copy;

        if (len < 64)
        {
            if (pending_block_off + len > 64)
                amount_to_copy = 64 - pending_block_off;
            else
                amount_to_copy = len;
        }
        else
        {
            amount_to_copy = 64 - pending_block_off;
```

```
                }

                Array.Copy(data, offset, pending_block, pending_block_off, amount_to_copy);
                len -= amount_to_copy;
                offset += amount_to_copy;
                pending_block_off += amount_to_copy;

                if (pending_block_off == 64)
                {
                    toUintArray(pending_block, uint_buffer);
                    processBlock(uint_buffer);
                    pending_block_off = 0;
                }
            }
        }

        public ReadOnlyCollection<byte> GetHash()
        {
            return toByteArray(GetHashUInt32());
        }

        public ReadOnlyCollection<UInt32> GetHashUInt32()
        {
            if (!closed)
            {
                UInt64 size_temp = bits_processed;

                AddData(new byte[1] { 0x80 }, 0, 1);

                uint available_space = 64 - pending_block_off;

                if (available_space < 8)
                    available_space += 64;

                // 0-initialized
                byte[] padding = new byte[available_space];
                // Insert lenght uint64
                for (uint i = 1; i <= 8; ++i)
                {
                    padding[padding.Length - i] = (byte)size_temp;
                    size_temp >>= 8;
                }

                AddData(padding, 0u, (uint)padding.Length);

                Debug.Assert(pending_block_off == 0);

                closed = true;
            }

            return Array.AsReadOnly(H);
        }

        private static void toUintArray(byte[] src, UInt32[] dest)
        {
            for (uint i = 0, j = 0; i < dest.Length; ++i, j += 4)
            {
                dest[i] = ((UInt32)src[j+0] << 24) | ((UInt32)src[j+1] << 16) |
((UInt32)src[j+2] << 8) | ((UInt32)src[j+3]);
            }
```

```
        }

        private static ReadOnlyCollection<byte> toByteArray(ReadOnlyCollection<UInt32> src)
        {
            byte[] dest = new byte[src.Count * 4];
            int pos = 0;

            for (int i = 0; i < src.Count; ++i)
            {
                dest[pos++] = (byte)(src[i] >> 24);
                dest[pos++] = (byte)(src[i] >> 16);
                dest[pos++] = (byte)(src[i] >> 8);
                dest[pos++] = (byte)(src[i]);
            }

            return Array.AsReadOnly(dest);
        }

        public static ReadOnlyCollection<byte> HashFile(Stream fs)
        {
            Sha256 sha = new Sha256();
            byte[] buf = new byte[8196];

            uint bytes_read;
            do
            {
                bytes_read = (uint)fs.Read(buf, 0, buf.Length);
                if (bytes_read == 0)
                    break;

                sha.AddData(buf, 0, bytes_read);
            }
            while (bytes_read == 8196);

            return sha.GetHash();
        }
    }
}
```

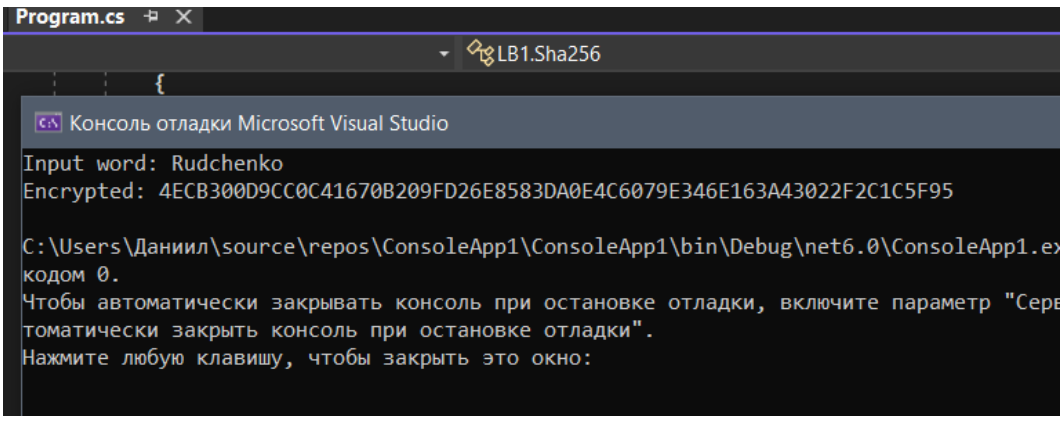

Рис. 1 – Результат роботи програми

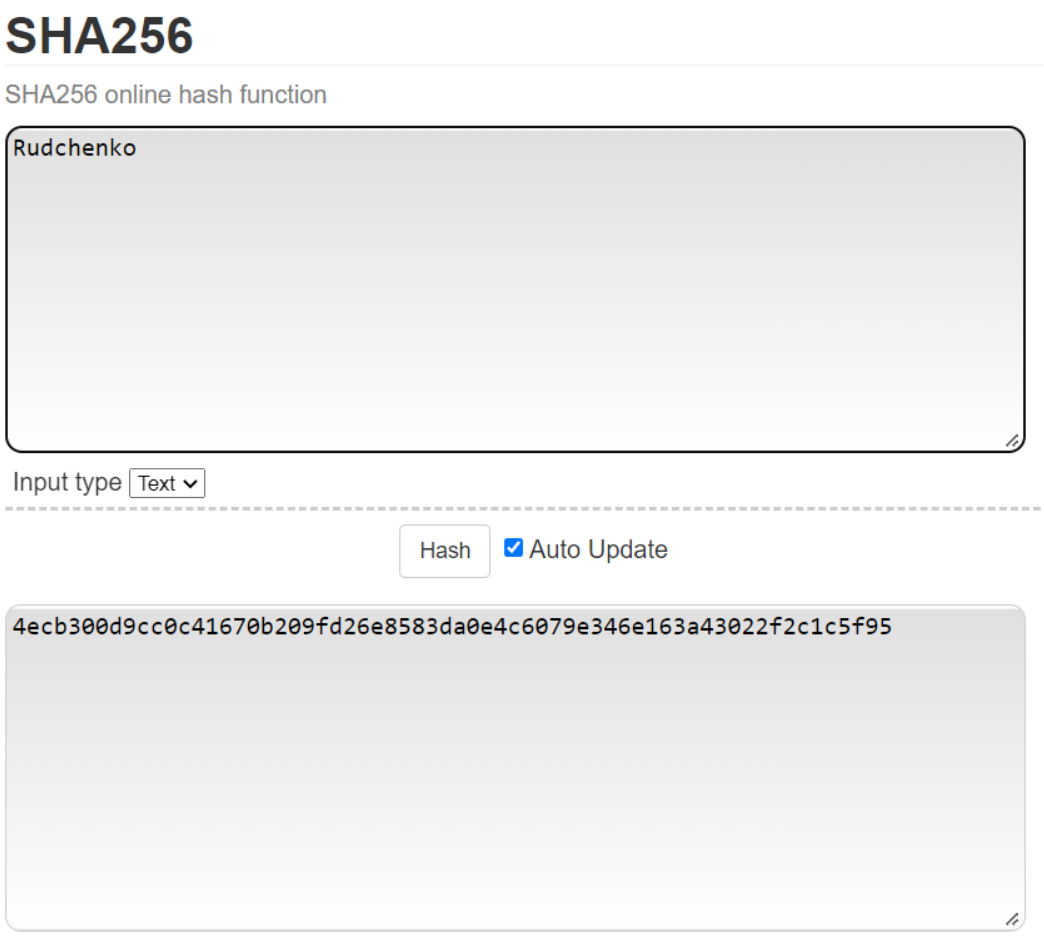

Рис. 2 – Перевірка результатів (успішно)

Висновок: в результаті виконання лабораторноі роботи було досліджено принципи роботи гешування.