

Министерство образования Республики Беларусь  
Учреждение образования «Белорусский государственный университет  
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы защиты информации

ОТЧЁТ  
к лабораторной работе №5  
на тему

**ХЕШ-ФУНКЦИИ**

Выполнил: студент гр.253501  
Станишевский А.Д.

Проверил: ассистент кафедры информатики  
Герчик А.В.

Минск 2025

## СОДЕРЖАНИЕ

1 Цель работы .....	3
2 Теоретические сведения .....	4
3 Ход работы.....	5
Заключение .....	6
Приложение А (обязательное) Листинг программного кода .....	7

# 1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является изучение теоретических основ и принципов работы криптографических хеш-функций, в частности алгоритмов ГОСТ 34.11 и SHA-1. В рамках работы требуется разработать программное средство для вычисления хеш-суммы (дайджеста сообщения) для входных данных, а также закрепить навыки работы с криптографическими примитивами.

Результатом выполнения работы должен быть скрипт, который позволяет:

- 1 Вычислить хеш-сумму для текстовых данных с использованием алгоритмов SHA-1 и ГОСТ 34.11.

- 2 Освоить процесс использования криптографических хеш-функций для обеспечения целостности данных.

## 2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Хеш-функции получили широкое распространение в разнообразных алгоритмах быстрого поиска информации.

Однако с появлением криптографии у них появилась вторая, ничуть не меньшая, область применения.

Хеш-функцией (англ. hash – мелко измельчать и перемешивать) называется необратимое преобразование данных, обладающее следующими свойствами:

- на вход алгоритма преобразования может поступать двоичный блок данных произвольной длины;

- на выходе алгоритма получается двоичный блок данных фиксированной длины;

- значения на выходе алгоритма распределяются по равномерному закону по всему диапазону возможных результатов;

- при изменении хотя бы одного бита на входе алгоритма его выход значительно меняется: в идеальном случае инвертируется произвольная половина бит.

В алгоритме ГОСТ 34.11 используются следующие преобразования:

**X-преобразование.** На вход функции X подаются две последовательности длиной 512 бит каждая, выходом функции является XOR этих последовательностей.

**S-преобразование.** Функция S является обычной функцией подстановки. Каждый байт из 512-битной входной последовательности заменяется соответствующим байтом из таблицы подстановок  $\pi$ .

**P-преобразование.** Функция перестановки. Для каждой пары байт из входной последовательности происходит замена одного байта другим.

**L-преобразование.** Представляет собой умножение 64-битного входного вектора на бинарную матрицу A размерами 64x64.

Алгоритм SHA-1 (Secure Hash Algorithm) предложен Институтом Стандартизации США NIST как стандарт хеширования в гражданской криптографии. Этот алгоритм был призван дать еще больший запас прочности к криптоатакам.

SHA-1 реализует хеш-функцию, построенную на идее функции сжатия. Входами функции сжатия являются блок сообщения длиной 512 бит и выход предыдущего блока сообщения.

Выход представляет собой значение всех хеш-блоков до этого момента. Иными словами хеш-блок  $M_i$  равен  $h_i = f(M_i, h_{i-1})$ . Хеш-значением всего сообщения является выход последнего блока.

### 3 ХОД РАБОТЫ

Программное средство реализовано при помощи языка программирования JavaScript. На рисунке 3.1 изображен полученный хеш с использованием алгоритма ГОСТ 34.11.

Введите сообщение

hello143 привет

Алгоритм хеширования

ГОСТ 34.11-512 (512 бит)

ВЫЧИСЛИТЬ ХЕШ

Результат вычисления хеш-функции

fe51f062583636964f101aff25c015157eefbd8dff681cb95521f5b88701f557bb5fd27cf1fdd23a88453c34ddf9e31940efde0cb65faf02900c7214f30a0eae

Рисунок 3.1 – Полученный хеш с использованием алгоритма ГОСТ 34.11

На рисунке 3.2 изображен полученный хеш с использованием алгоритма SHA1.

Введите сообщение

hello143 привет

Алгоритм хеширования

SHA-1 (160 бит)

ВЫЧИСЛИТЬ ХЕШ

Результат вычисления хеш-функции

5a4bdb33b207339ce02e518ad580021bfd1990af

Рисунок 3.2 – Полученный хеш с использованием алгоритма SHA1

Таким образом, программа успешно справляется с поставленными задачами.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы были изучены принципы криптографического хеширования, включая теоретические основы алгоритмов ГОСТ 34.11 и SHA-1. Были рассмотрены ключевые свойства хеш-функций: необратимость, фиксированная длина выхода и лавинный эффект.

Разработанное программное средство позволяет вычислять дайджесты сообщений для текстовых файлов, демонстрируя ключевые свойства хеш-функций на практике.

Полученные практические навыки работы с криптографическими хеш-функциями позволяют успешно применять их для обеспечения целостности и аутентификации данных.

Таким образом, цель лабораторной работы достигнута, и приобретенные знания и навыки могут быть использованы для решения задач обеспечения информационной безопасности.

# ПРИЛОЖЕНИЕ А

## (обязательное)

### Листинг программного кода

```
const C = Array.from({ length: 12 }, (_, i) => {
  const c = new Uint8Array(64);
  for (let j = 0; j < 64; j++) {
    c[j] = (i * 64 + j) * 0x01010101 & 0xFF;
  }
  return c;
});

function bytesToHex(bytes) {
  return Array.from(bytes).map(b => b.toString(16).padStart(2, '0')).join('');
}

function stringToBytes(str) {
  const encoder = new TextEncoder();
  return encoder.encode(str);
}

function xor(a, b) {
  const result = new Uint8Array(64);
  for (let i = 0; i < 64; i++) {
    result[i] = a[i] ^ b[i];
  }
  return result;
}

function S(data) {
  const result = new Uint8Array(64);
  for (let i = 0; i < 64; i++) {
    result[i] = PI[data[i]];
  }
  return result;
}

function P(data) {
  const result = new Uint8Array(64);
  const tau = [
    0, 8, 16, 24, 32, 40, 48, 56,
    1, 9, 17, 25, 33, 41, 49, 57,
    2, 10, 18, 26, 34, 42, 50, 58,
    3, 11, 19, 27, 35, 43, 51, 59,
    4, 12, 20, 28, 36, 44, 52, 60,
    5, 13, 21, 29, 37, 45, 53, 61,
    6, 14, 22, 30, 38, 46, 54, 62,
    7, 15, 23, 31, 39, 47, 55, 63
  ];
  for (let i = 0; i < 64; i++) {
    result[i] = data[tau[i]];
  }
  return result;
}

function L(data) {
  const result = new Uint8Array(64);
  const A = [
    0x8e20faa72ba0b470n, 0x47107ddd9b505a38n, 0xad08b0e0c3282d1cn,
    0xd8045870ef14980en,
```

```

        0x6c022c38f90a4c07n,      0x3601161cf205268dn,      0x1b8e0b0e798c13c8n,
0x83478b07b2468764n
    ];

    for (let i = 0; i < 8; i++) {
        let r = 0n;
        for (let j = 0; j < 8; j++) {
            r = (r << 8n) | BigInt(data[i * 8 + j]);
        }

        let t = 0n;
        for (let j = 0; j < 64; j++) {
            if ((r >> BigInt(j)) & 1n) {
                t ^= A[j >> 3] >> BigInt(j & 7);
            }
        }

        for (let j = 0; j < 8; j++) {
            result[i * 8 + j] = Number((t >> BigInt(56 - j * 8)) & 0xFFn);
        }
    }

    return result;
}

function KeySchedule(K, i) {
    K = xor(K, C[i]);
    K = S(K);
    K = P(K);
    K = L(K);
    return K;
}

function E(K, m) {
    let state = xor(K, m);

    for (let i = 0; i < 12; i++) {
        state = S(state);
        state = P(state);
        state = L(state);
        K = KeySchedule(K, i);
        state = xor(state, K);
    }

    return state;
}

function g(N, m, h) {
    let K = xor(h, N);
    K = S(K);
    K = P(K);
    K = L(K);

    const t = E(K, m);
    const t_xor_h = xor(t, h);
    const G = xor(t_xor_h, m);

    return G;
}

export function sha1(message) {
    const K = [
        0x5A827999,
        0x6ED9EBA1,
        0x8F1BBCDC,

```



```

    0xCA62C1D6
];

let H = [
    0x67452301,
    0xEFCDAB89,
    0x98BADCFE,
    0x10325476,
    0xC3D2E1F0
];

let bytes = stringToBytes(message);
const originalBitLength = bytes.length * 8;

const padded = new Uint8Array(bytes.length + 1);
padded.set(bytes);
padded[bytes.length] = 0x80;
bytes = padded;

const zeroPadding = (56 - (bytes.length % 64) + 64) % 64;
const temp = new Uint8Array(bytes.length + zeroPadding);
temp.set(bytes);
bytes = temp;

const lengthBytes = new Uint8Array(8);
const length = originalBitLength;
for (let i = 0; i < 8; i++) {
    lengthBytes[7 - i] = (length >>> (i * 8)) & 0xFF;
}

const finalBytes = new Uint8Array(bytes.length + 8);
finalBytes.set(bytes);
finalBytes.set(lengthBytes, bytes.length);
bytes = finalBytes;

for (let i = 0; i < bytes.length; i += 64) {
    const block = bytes.slice(i, i + 64);
    const W = new Array(80);

    for (let t = 0; t < 16; t++) {
        W[t] = (
            (block[t * 4] << 24) |
            (block[t * 4 + 1] << 16) |
            (block[t * 4 + 2] << 8) |
            (block[t * 4 + 3])
        ) >>> 0;
    }

    for (let t = 16; t < 80; t++) {
        W[t] = rotateLeft(W[t - 3] ^ W[t - 8] ^ W[t - 14] ^ W[t - 16], 1);
    }

    let [a, b, c, d, e] = H;

    for (let t = 0; t < 80; t++) {
        let f, k;

        if (t < 20) {
            f = (b & c) | (~b & d);
            k = K[0];
        } else if (t < 40) {
            f = b ^ c ^ d;
            k = K[1];
        } else if (t < 60) {

```

```

        f = (b & c) | (b & d) | (c & d);
        k = K[2];
    } else {
        f = b ^ c ^ d;
        k = K[3];
    }

    const temp = (rotateLeft(a, 5) + f + e + k + W[t]) >>> 0;
    e = d;
    d = c;
    c = rotateLeft(b, 30);
    b = a;
    a = temp;
}

H[0] = (H[0] + a) >>> 0;
H[1] = (H[1] + b) >>> 0;
H[2] = (H[2] + c) >>> 0;
H[3] = (H[3] + d) >>> 0;
H[4] = (H[4] + e) >>> 0;
}

const resultBytes = wordsToBytes(H);
return bytesToHex(resultBytes);
}

```