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. 253503
Минич С.В.

Проверил: ассистент кафедры информатики Герчик А.В.

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В качестве объекта для практического изучения методов криптографического хэширования были выбраны алгоритмы ГОСТ 34.11-2018 и *SHA-1*. Эти хэш-функции обеспечивают преобразование входного сообщения в уникальный фиксированный по длине цифровой отпечаток, который используется для проверки целостности данных и защиты информации. Алгоритм ГОСТ 34.11-2018, разработанный на базе отечественного стандарта, применяет последовательные преобразования *S, P* и *L* и использует специальные матрицы и константы для повышения криптостойкости. *SHA-1*, в свою очередь, является классическим примером хэш-функции, широко применяемой в протоколах передачи данных и цифровых подписях.

Криптостойкость этих алгоритмов обеспечивается сложностью обратного восстановления исходного сообщения по хэшу, что делает их устойчивыми к вычислительным атакам. В отличие от симметричных или асимметричных шифров, хэш-функции не используют ключи для шифрования, а основаны на математических преобразованиях данных, что обеспечивает их эффективность и безопасность при проверке целостности сообщений [1][2].

1 ХОД ВЫПОЛНЕНИЯ РАБОТЫ

В ходе лабораторной работы была реализована проверка работы хэш-функций ГОСТ 34.11-2018 и SHA-1 на языке *Python*. Для алгоритма ГОСТ 34.11-2018 были реализованы основные преобразования: *S*-преобразование, обеспечивающее замену байтов через таблицу *SBOX*; *P*-преобразование, меняющее порядок байтов в блоке; и *L*-преобразование, выполняющее линейное преобразование с использованием заранее заданной матрицы констант.

Для реализации алгоритма *SHA-1* была использована стандартная последовательность шагов: добавление битового дополнения к исходному сообщению, разбиение на блоки по 512 бит, расширение блока до 80 слов и многократное применение логических и циклических операций для вычисления итогового хэша.

Кроме того, в работе была реализована функция *g_function* для ГОСТ 34.11-2018 и функция *e_function*, обеспечивающие комбинированное применение ключа и блока сообщения. Также был реализован порядок работы с ключами через процедуру *key_schedule*, что позволило корректно проводить пошаговое хэширование каждого блока данных.

В практической части лабораторной работы проверялась корректность вычисления хэшей для тестового сообщения. Были получены значения хэшей ГОСТ 34.11-2018 (512 бит и 256 бит) и SHA-1, после чего проводилась проверка изменения хэша при модификации исходного сообщения. Полученные результаты подтвердили чувствительность хэш-функций к изменению входных данных, что демонстрирует их использование для проверки целостности информации. Вывод результата представлен на рисунке 1.

```
PS C:\sem7\MZI> & C:/Users/imsve/AppData/Local/Programs/Python/Python312/python.exe c:/sem7/MZI/lab5/l5.py
Исходное сообщение: 'Test message Test message Test message Test message'
ГОСТ 512: 183519333831304881969150874029932121175497015074779152320902725994865230167353584527359344884832632936181934705
7460207566461
ГОСТ 256: 7fcfbd20ec64f6129aebf9e149f96a0cd52abbfe6c67b212347e3c27dc4ea036
SHA-1: 1245770049134121706589508508935863749432400016004

Измененное сообщение: 'Test message'
ГОСТ 512: 42324789328032216795545452091769413066294314827249918958930432233246799423618705249495891846574572764966871676754
4766091832481
```

Рисунок 1 – Результат выполнения алгоритма

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы был подробно изучен и реализован процесс криптографического хэширования на примере алгоритмов ГОСТ 34.11-2018 и *SHA-1*. На основе теоретических положений алгоритмов была создана программа на *Python*, включающая основные преобразования: *S, P и L* для ГОСТ 34.11-2018, работу с константами и ключевыми функциями *g_function* и *e_function*, а также шаги формирования и обработки блоков данных для *SHA-1*.

Реализация позволила продемонстрировать работу ключевых элементов алгоритмов: преобразование байтов через *SBOX*, перестановку и линейное преобразование блоков, циклические сдвиги и расширение блоков для *SHA-1*, а также чувствительность хэшей к изменениям исходного сообщения. Полученные результаты подтвердили корректность подхода и показали, что хэш-функции обеспечивают надежное формирование цифрового отпечатка данных, пригодного для проверки их целостности.

Цель работы была достигнута: изучение алгоритмов ГОСТ 34.11-2018 и SHA-1 и их программная реализация позволили на практике закрепить понимание принципов работы криптографических хэш-функций, методов защиты информации и основ обеспечения целостности данных.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] ГОСТ 34.11-2018 Информационная технология (ИТ). Криптографическая защита информации. Функция хэширования (с Поправкой) - docs.cntd.ru [Электронный ресурс]. – Режим доступа: https://docs.cntd.ru/document/1200161707. – Дата доступа: 25.10.2025

[2] 6.18 Алгоритм вычисления дайджеста сообщения (SHA1, RFC-3174, сентябрь 2001) [Электронный ресурс]. – Режим доступа: http://book.itep.ru/6/sha1.htm– Дата доступа: 25.10.2025

ЛИСТИНГ ПРОГРАММНОГО КОДА

```
import numpy as np

# https://docs.cntd.ru/document/1200161707
SBOX = [252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250, 218, 35, 197, 4, 77, 233, 119, 240, 219, 147, 46,
153, 186, 23, 54, 241, 187, 20, 205, 95, 193, 249, 24, 101, 90, 226, 92, 239, 33, 129, 28, 60, 66, 139, 1, 142,
79, 5, 132, 2, 174, 227, 106, 143, 160, 6, 11, 237, 152, 127, 212, 211, 31, 235, 52, 44, 81, 234, 200, 72, 171,
242, 42, 104, 162, 253, 58, 206, 204, 181, 112, 14, 86, 8, 12, 118, 18, 191, 114, 19, 71, 156, 183, 93, 135, 21,
161, 150, 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178, 177, 50, 117, 25, 61, 255, 53, 138, 126, 109,
84, 198, 128, 195, 189, 13, 87, 223, 245, 36, 169, 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185, 3, 224,
15, 236, 222, 122, 148, 176, 188, 220, 232, 40, 80, 78, 51, 10, 74, 167, 151, 96, 115, 30, 0, 98, 68, 26, 184,
56, 130, 100, 159, 38, 65, 173, 69, 70, 146, 39, 94, 85, 47, 140, 163, 165, 125, 105, 213, 149, 59, 7, 88, 179,
64, 134, 172, 29, 247, 48, 55, 107, 228, 136, 217, 231, 137, 225, 27, 131, 73, 76, 63, 248, 254, 141, 83, 170,
144, 202, 216, 133, 97, 32, 113, 103, 164, 45, 43, 9, 91, 203, 155, 37, 208, 190, 229, 108, 82, 89, 166, 116,
210, 230, 244, 180, 192, 209, 102, 175, 194, 57, 75, 99, 182]
A_MATRIX_HEX = [
    "8e20faa72ba0b470", "47107ddd9b505a38", "ad08b0e0c3282d1c", "d8045870ef14980e",
    "6c022c38f90a4c07", "3601161cf205268d", "1b8e0b0e798c13c8", "83478b07b2468764",
    "a011d380818e8f40", "5086e740ce47c920", "2843fd2067adea10", "14aff010bdd87508",
    "0ad97808d06cb404", "05e23c0468365a02", "8c711e02341b2d01", "46b60f011a83988e",
    "90dab52a387ae76f", "486dd4151c3dfdb9", "24b86a840e90f0d2", "125c354207487869",
    "092e94218d243cba", "8a174a9ec8121e5d", "4585254f64090fa0", "accc9ca9328a8950",
    "9d4df05d5f661451", "c0a878a0a1330aa6", "60543c50de970553", "302a1e286fc58ca7",
    "18150f14b9ec46dd", "0c84890ad27623e0", "0642ca05693b9f70", "0321658cba93c138",
    "86275df09ce8aaa8", "439da0784e745554", "afc0503c273aa42a", "d960281e9d1d5215",
    "e230140fc0802984", "71180a8960409a42", "b60c05ca30204d21", "5b068c651810a89e",
    "456c34887a3805b9", "ac361a443d1c8cd2", "561b0d22900e4669", "2b838811480723ba",
    "9bcf4486248d9f5d", "c3e9224312c8c1a0", "effa11af0964ee50", "f97d86d98a327728",
    "e4fa2054a80b329c", "727d102a548b194e", "39b008152acb8227", "9258048415eb419d",
    "492c024284fbaec0", "aa16012142f35760", "550b8e9e21f7a530", "a48b474f9ef5dc18",
    "70a6a56e2440598e", "3853dc371220a247", "1ca76e95091051ad", "0edd37c48a08a6d8",
    "07e095624504536c", "8d70c431ac02a736", "c83862965601dd1b", "641c314b2b8ee083"
]

# ------ TRANSFORMATIONS GOST 34.11-2018
def X_transformation(a, b):
    return a ^ b

def S_transformation(a):
    if isinstance(a, int):
        b = bytearray(a.to_bytes(64, 'big'))
    else:
        b = bytearray(a)[:64]
    b = b.rjust(64, b'\x00')[:64]
    for i in range(64):
        b[i] = SBOX[b[i]]
    return int.from_bytes(b, 'big')

def P_transformation(a):
    if isinstance(a, int):
        b = bytearray(a.to_bytes(64, 'big'))
    else:
        b = bytearray(a)
    b = b.rjust(64, b'\x00')[:64]
    res = bytearray(64)
    for i in range(64):
        ind = (i % 8) * 8 + (i // 8)
        res[i] = b[ind]
    return int.from_bytes(res, 'big')

def L_transformation(a):
    if isinstance(a, int):
        full = a.to_bytes(64, 'big')
    else:
        full = bytes(a)
    full = full.rjust(64, b'\x00')[:64]

    out_words = []
    A_rows = [int(x, 16) for x in A_MATRIX_HEX]

    for block_idx in range(8):
        word_bytes = full[block_idx*8:(block_idx+1)*8]
        word = int.from_bytes(word_bytes, 'big')
        result_word = 0
        for j in range(64):
            if (word >> (63 - j)) & 1:
                result_word ^= A_rows[j]
        out_words.append(result_word.to_bytes(8, 'big'))

    result = b''.join(out_words)
    return int.from_bytes(result, 'big')

# ------ COMPRESSION GOST 34.11-2018
C                                                                                                                =
["b1085bda1ecadae9ebcb2f81c0657c1f2f6a76432e45d016714eb88d7585c4fc4b7ce09192676901a2422a08a460d31505767436cc744
d23dd806559f2a64507",
     "6fa3b58aa99d2f1a4fe39d460f70b5d7f3feea720a232b9861d55e0f16b501319ab5176b12d699585cb561c2db0aa7ca55dda21bd
7cbcd56e679047021b19bb7",
```

```
    "f574dcac2bce2fc70a39fc286a3d843506f15e5f529c1f8bf2ea7514b1297b7bd3e20fe490359eb1c1c93a376062db09c2b6f443867adb31991e96f50aba0ab2",
    "ef1fdfb3e81566d2f948e1a05d71e4dd488e857e335c3c7d9d721cad685e353fa9d72c82ed03d675d8b71333935203be3453eaa193e837f1220cbebc84e3d12e",
    "4bea6bacad4747999a3f410c6ca923637f151c1f1686104a359e35d7800fffbdbfcd1747253af5a3dfff00b723271a167a56a27ea9ea63f5601758fd7c6cfe57",
    "ae4faeae1d3ad3d96fa4c33b7a3039c02d66c4f95142a46c187f9ab49af08ec6cffaa6b71c9ab7b40af21f66c2bec6b6bf71c57236904f35fa68407a46647d6e",
    "f4c70e16eeaac5ec51ac86febf240954399ec6c7e6bf87c9d3473e33197a93c90992abc52d822c3706476983284a05043517454ca23c4af38886564d3a14d493",
    "9b1f5b424d93c9a703e7aa020c6e41414eb7f8719c36de1e89b4443b4ddbc49af4892bcb929b069069d18d2bd1a5c42f36acc2355951a8d9a47f0dd4bf02e71e",
    "378f5a541631229b944c9ad8ec165fde3a7d3a1b258942243cd955b7e00d0984800a440bdbb2ceb17b2b8a9aa6079c540e38dc92cb1f2a607261445183235adb",
    "abbedea680056f52382ae548b2e4f3f38941e71cff8a78db1fffe18a1b3361039fe76702af69334b7a1e6c303b7652f43698fad1153bb6c374b4c7fb98459ced",
    "7bcd9ed0efc889fb3002c6cd635afe94d8fa6bbbebab07612001802114846679 8a1d71efea48b9caefbacd1d7d476e98dea2594ac06fd85d6bcaa4cd81f32d1b",
    "378ee767f11631bad21380b00449b17acda43c32bcdf1d77f82012d430219f9b5d80ef9d1891cc86e71da4aa88e12852faf417d5d9b21b9948bc924af11bd720"]

def key_schedule(key, i):
    key = key ^ int(C[i], 16)
    key = S_transformation(key)
    key = P_transformation(key)
    key = L_transformation(key)
    return key

def e_function(key, m):
    state = key ^ m
    for i in range(12):
        state = S_transformation(state)
        state = P_transformation(state)
        state = L_transformation(state)
        key = key_schedule(key, i)
        state = state ^ key
    return state

def g_function(N, m, h):
    key = h ^ N
    key = S_transformation(key)
    key = P_transformation(key)
    key = L_transformation(key)
    t = e_function(key, m)
    t = h ^ t
    g = t ^ m
    return g

# ------ hash-func GOST 34.11-2018
def hash_gost(m, output=512):
    if output == 512:
        h = bytes([0x00] * 64)
    elif output == 256:
        h = bytes([0x01] * 64)
    h = int.from_bytes(h, 'big')

    N = 0x00
    E = 0x00

    if isinstance(m, str):
        m = bytearray(m.encode('utf-8'))
    elif isinstance(m, int):
        m = bytearray(m.to_bytes(64, 'big'))

    message_bits = len(m) * 8
    while message_bits >= 512:
        m_block = m[-64:]
        m_int = int.from_bytes(m_block, 'big')
        h = g_function(N, m_int, h)
        N = (N + 512) % (2**512)
        E = (E + m_int) % (2**512)
        m = m[:-64]
        message_bits = len(m) * 8

    m_with_padding = bytearray(64)
    m_with_padding[-len(m):] = m
    one_byte_pos = 64 - len(m) - 1
    m_with_padding[one_byte_pos] = 0x80  # 0x80 = 10000000 в двоичной
    m_int = int.from_bytes(m_with_padding, 'big')

    h = g_function(N, m_int, h)
    N = (N + len(m)*8) % (2**512)
    E = (E + m_int) % (2**512)
    h = g_function(0, h, N)
    h = g_function(0, h, E)

    if output == 256:
        h = h.to_bytes(64,'big').hex()
        h = h[:64]

    return h
```

```
# ------ TRANSFORMATIONS SHA-1
# http://book.itep.ru/6/sha1.htm

import struct

def rotate_left(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(message):
    if isinstance(message, str):
        message = message.encode('utf-8')

    msg_len = len(message)
    message += b'\x80'

    while (len(message) % 64) != 56:
        message += b'\x00'

    message += (msg_len * 8).to_bytes(8, 'big')

    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476
    h4 = 0xC3D2E1F0

    for i in range(0, len(message), 64):
        block = message[i:i+64]

        w = [int.from_bytes(block[i*4:(i+1)*4], 'big') for i in range(16)]

        for t in range(16, 80):
            w.append(rotate_left(w[t-3] ^ w[t-8] ^ w[t-14] ^ w[t-16], 1))

        a, b, c, d, e = h0, h1, h2, h3, h4

        for t in range(80):
            if 0 <= t <= 19:
                K = 0x5A827999
                f = (b & c) | ((~b) & d)
            elif 20 <= t <= 39:
                K = 0x6ED9EBA1
                f = b ^ c ^ d
            elif 40 <= t <= 59:
                K = 0x8F1BBCDC
                f = (b & c) | (b & d) | (c & d)
            else:
                K = 0xCA62C1D6
                f = b ^ c ^ d

            temp = (rotate_left(a, 5) + f + e + w[t] +K) & 0xffffffff
            e = d
            d = c
            c = rotate_left(b, 30)
            b = a
            a = temp

        h0 = (h0 + a) & 0xffffffff
        h1 = (h1 + b) & 0xffffffff
        h2 = (h2 + c) & 0xffffffff
        h3 = (h3 + d) & 0xffffffff
        h4 = (h4 + e) & 0xffffffff

    return (h0 << 128) | (h1 << 96) | (h2 << 64) | (h3 << 32) | h4

def verify_file_integrity(actual_hash, expected_hash):
    return actual_hash.lower() == expected_hash.lower()


if __name__ == "__main__":
    test_file_content = "Test message Test message Test message Test message"

    gost512_hash = hash_gost(test_file_content, 512)
    gost256_hash = hash_gost(test_file_content, 256)
    sha1_hash = sha1(test_file_content)

    print(f"Исходное сообщение: '{test_file_content}'")
    print(f"ГОСТ 512: {gost512_hash}")
    print(f"ГОСТ 256: {gost256_hash}")
    print(f"SHA-1: {sha1_hash}")


    modified_content = "Test message"
    modified_gost512 = hash_gost(modified_content, 512)

    print(f"\nИзмененное сообщение: '{modified_content}'")
    print(f"ГОСТ 512: {modified_gost512}")
```