Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

# Отчет по лабораторной работе №5
# “Хэш-функции”

Выполнил:
студент группы 053504
Горожанкин В.О.

Проверил
ассистент кафедры информатики
Лещенко Евгений Александрович

Минск 2023

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Криптографические хеш-функции — это выделенный класс хеш-функций, который имеет определенные свойства, делающие его пригодным для использования в криптографии. ГОСТ 34.11 – стандарт, который определяет алгоритм и 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 (ЭЦП) при передаче, обработке и хранении информации в автоматизированных системах.

Secure Hash Algorithm 1 — алгоритм криптографического хеширования. Описан в RFC 3174. Для входного сообщения произвольной длины алгоритм генерирует 160-битное (20 байт) хеш-значение, называемое также дайджестом сообщения, которое обычно отображается как шестнадцатеричное число длиной в 40 цифр. Используется во многих криптографических приложениях и протоколах.

# 1 ДЕМОНСТРАЦИЯ РАБОТЫ ПРОГРАММЫ

## 1.1 Хеширование SHA1

```
res = SHA1("В чащах юга жил бы цитрус? Да, но фальшивый экземпляр
assert res == "9e32295f8225803bb6d5fdfcc0674616a4413c1b"
print(res)

res = SHA1("The quick brown fox jumps over the lazy dog")
assert res == "2fd4e1c67a2d28fced849ee1bb76e7391b93eb12"
print(res)

res = SHA1("sha")
assert res == "d8f4590320e1343a915b6394170650a8f35d6926"
print(res)

res = SHA1("Sha")
assert res == "ba79baeb9f10896a46ae74715271b7f586e74640"
print(res)

res = SHA1("")
assert res == "da39a3ee5e6b4b0d3255bfef95601890afd80709"
print(res)
```

Рисунок 1 – Результат хеширования строк

Вывод программы:

```
9e32295f8225803bb6d5fdfcc0674616a4413c1b
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
d8f4590320e1343a915b6394170650a8f35d6926
ba79baeb9f10896a46ae74715271b7f586e74640
da39a3ee5e6b4b0d3255bfef95601890afd80709
```

## 1.2 Хеширование ГОСТ 34.11

```
text1 = "This is message, length=32 bytes"
```

e4c945ef73561b0fa964d5877ecd9e0b3f7a1e1f7d0aa5a2d90bf325b511ea38

Рисунок 2 – Запись хеша ГОСТ 34.11

# 2 ОПИСАНИЕ БЛОК-СХЕМЫ АЛГОРИТМА

Блок-схема алгоритма представлена на рисунке 3.

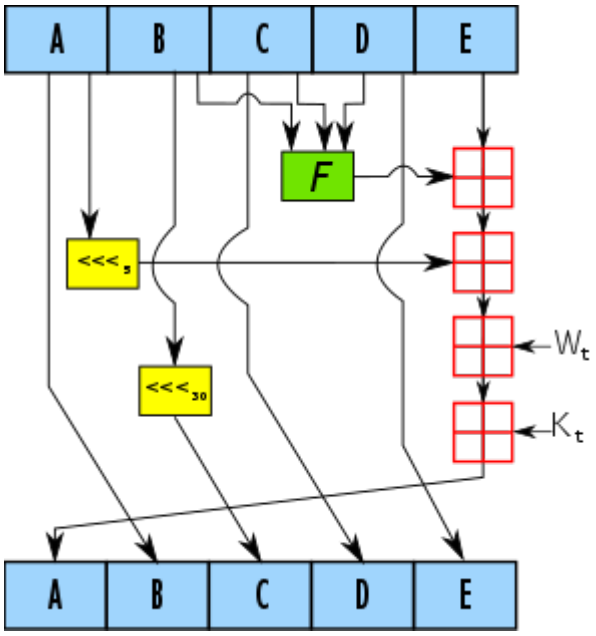


Рисунок 3 – Блок-схема генерации хеша SHA-1

Инициализация

Исходное сообщение разбивается на блоки по 512 бит в каждом. Последний блок дополняется до длины, кратной 512 бит. Сначала добавляется 1 (бит), а потом — нули, чтобы длина блока стала равной 512 — 64 = 448 бит. В оставшиеся 64 бита записывается длина исходного сообщения в битах (в big-endian формате). Если последний блок имеет длину более 447, но менее 512 бит, то дополнение выполняется следующим образом: сначала добавляется 1 (бит), затем — нули вплоть до конца 512-битного блока; после этого создается ещё один 512-битный блок, который заполняется вплоть до 448 бит нулями, после чего в оставшиеся 64 бита записывается длина исходного сообщения в битах (в big-endian формате). Дополнение последнего блока осуществляется всегда, даже если сообщение уже имеет нужную длину.

Инициализируются пять 32-битовых переменных.

A = 0x67452301

B = 0xEFCDAB89

C = 0x98BADCFE

D = 0x10325476

E = 0xC3D2E1F0

Определяются четыре нелинейные операции и четыре константы.

| | | |
|---|---|---|
| $F_t(m,l,k) = (m \wedge l) \vee (\neg m \wedge k)$ | $K_t$ = 0x5A827999 | 0≤t≤19 |
| $F_t(m,l,k) = m \oplus l \oplus k$ | $K_t$ = 0x6ED9EBA1 | 20≤t≤39 |
| $F_t(m,l,k) = (m \wedge l) \vee (m \wedge k) \vee (l \wedge k)$ | $K_t$ = 0x8F1BBCDC | 40≤t≤59 |
| $F_t(m,l,k) = m \oplus l \oplus k$ | $K_t$ = 0xCA62C1D6 | 60≤t≤79 |

Главный цикл

Главный цикл итеративно обрабатывает каждый 512-битный блок. В начале каждого цикла вводятся переменные a, b, c, d, e, которые инициализируются значениями A, B, C, D, E, соответственно. Блок сообщения преобразуется из 16 32-битовых слов

$W_t = M_t$ при 0≤t≤15

$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) << 1$ при 16≤t≤79

```
для t от 0 до 79
    temp = (a<<5) + F_t(b,c,d) + e + W_t + K_t
    e = d
    d = c
    c = b<<30
    b = a
    a = temp
```

где «+» — сложение беззнаковых 32-битных целых чисел с отбрасыванием избытка (33-го бита).

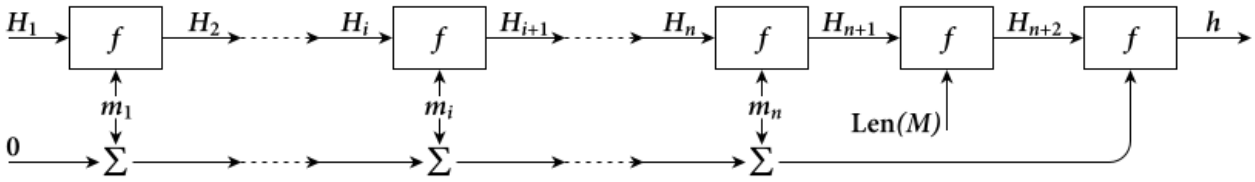


Рисунок 4 – Блок-схема генерации хеша ГОСТ 34.11

**Алгоритм**

1. Инициализация:
    1. $h := H_1$ — Начальное значение хеш-функции. То есть — 256 битовый IV вектор, определяется пользователем.
    2. $\Sigma := 0$ — Контрольная сумма
    3. $L := 0$ — Длина сообщения
2. Функция сжатия внутренних итераций: для i = 1 ... n — 1 выполняем следующее (пока $|M| > 256$):
    1. $h := f(h, m_i)$ — итерация метода последовательного хеширования
    2. $L := L + 256$ — итерация вычисления длины сообщения
    3. $\Sigma := \Sigma + m_i$ — итерация вычисления контрольной суммы
3. Функция сжатия финальной итерации:
    1. $L := L + j\, m_n\, j$ — вычисление полной длины сообщения
    2. $m_n := f0g^{256 - jm_nj} km_n$ — набивка последнего блока
    3. $\Sigma := \Sigma + m_n$ — вычисление контрольной суммы сообщения
    4. $h := f(h, m_n)$
    5. $h := f(h, L)$ — MD — усиление
    6. $h := f(h, \Sigma)$
4. Выход. Значением хеш-функции является h,

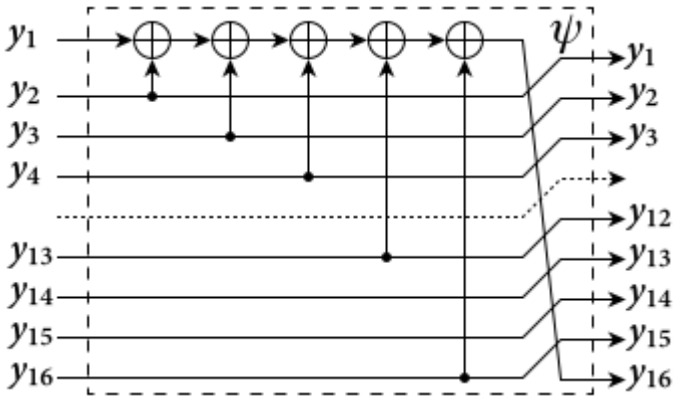


Рисунок 5 – Перемешивающее преобразование ГОСТ 34.11

Генерация ключей

В алгоритме генерации ключей используются:

Два преобразования блоков длины 256 бит:

Преобразование

$A(Y) = A(y_4 \ k \ y_3 \ k \ y_2 \ k \ y_1) = (y_1 \oplus y_2) \ k \ y_4 \ k \ y_3 \ k \ y_2$ — подблоки блока Y длины 64 бит.

Преобразование

$P(Y) = P(y_{32} k y_{31} k \dots k y_1) = y_{\varphi(32)} k y_{\varphi(31)} k \dots k y_{\varphi(1)}$ — подблоки блока Y длины 8 бит.

Три константы:

C2 = 0

C3 = 0xff00ffff000000ffff0000ff00ffff0000ff00ff00ff00ffff00ff00ff00ff00

C4 = 0

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы было реализовано программное средство хеширования с использованием SHA-1 и ГОСТ 34.11.

В итоге, выполнение данной лабораторной работы позволило нам приобрести практические навыки в области хеширования и ознакомиться с принципами работы SHA-1 и ГОСТ 34.11.

Ключевые отличия двух алгоритмов:

ГОСТ 34.11 разработан в России и используется для создания хеш-сумм с использованием блочного алгоритма хеширования. Он использует разные раунды и подстановки, чтобы обеспечить хорошую стойкость к криптоанализу и высокую скорость хеширования.

SHA-1 был разработан США и является частью семейства хеш-функций SHA (Secure Hash Algorithm), созданных Национальным институтом стандартов и технологий (NIST). SHA-1 использует алгоритм Меркла-Дамгора и также был широко использован в прошлом для хеширования данных.

ГОСТ 34.11 может генерировать хеш-значения разной длины, включая 256 бит (ГОСТ Р 34.11-2012).

SHA-1 всегда генерирует 160-битные (20-байтные) хеш-значения.

ГОСТ 34.11 обладает высокой стойкостью к криптоанализу, но его безопасность также зависит от выбора параметров и реализации.

SHA-1 потерял свою стойкость к коллизиям (возможности найти два разных входных сообщения, которые дают одинаковый хеш) и не рекомендуется для использования в криптографических приложениях. SHA-1 считается устаревшим.

# ПРИЛОЖЕНИЕ А
# (обязательное)
# Листинг программного кода

```
from bitarray import bitarray



def before_coding(text):
    file = 0
    bitArray = bitarray(endian="big")
    if type(text) == str:
        bitArray.frombytes(text.encode("utf-8"))
    else:
        bitArray.frombytes(text)
        file = 1
    return file, bitArray



def appendPaddingBytes(bitArray):
    bitArray.append(1)
    while len(bitArray) % 512 != 448:
        bitArray.append(0)
    return bitarray(bitArray, endian="big")



def appendLength(bitArray, length):
    bitArray.extend(bin(length)[2:].zfill(64))
    return bitArray



def fromBitsToInt(bitArray):
    X = []
    for i in range(len(bitArray)//32):
```

```
        tmp = bitArray[:32]
        X.append(int.from_bytes(tmp.tobytes(), byteorder="big"))
        bitArray = bitArray[32:]
    return X


def mainProcces(intArray):
    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476
    h4 = 0xC3D2E1F0
    F1 = lambda x, y, z: (x & y) | (~x & z)
    F2 = lambda x, y, z: x ^ y ^ z
    F3 = lambda x, y, z: (x & y) | (x & z) | (y & z)
    rotateLeft = lambda x, n: (x << n) | (x >> (32 - n))
    modularAdd = lambda a, b: (a + b) % pow(2, 32)

    for i in range(len(intArray)//16):
        X = intArray[0:16]
        intArray = intArray[16:]

        w = [0 for w in range(80)]
        for k in range(80):
            if 0 <= k <= 15:
                w[k] = X[k]
            if 16 <= k <= 79:
                w[k] = rotateLeft((w[k-3] ^ w[k-8] ^ w[k-14] ^ w[k-16]), 1) %
pow(2, 32)

        a = h0
        b = h1
        c = h2
        d = h3
```

```
        e = h4

        for j in range(4 * 20):
            if 0 <= j <= 19:
                k = 0x5A827999
                temp = F1(b, c, d)
            elif 20 <= j <= 39:
                k = 0x6ED9EBA1
                temp = F2(b, c, d)
            elif 40 <= j <= 59:
                k = 0x8F1BBCDC
                temp = F3(b, c, d)
            elif 60 <= j <= 79:
                k = 0xCA62C1D6
                temp = F2(b, c, d)

            temp = modularAdd(rotateLeft(a, 5), temp)
            temp = modularAdd(temp, e)
            temp = modularAdd(temp, k)
            temp = modularAdd(temp, w[j])
            e = d
            d = c
            c = rotateLeft(b, 30)
            b = a
            a = temp
        h0 = modularAdd(h0, a)
        h1 = modularAdd(h1, b)
        h2 = modularAdd(h2, c)
        h3 = modularAdd(h3, d)
        h4 = modularAdd(h4, e)
    return h0, h1, h2, h3, h4
```

```
def SHA1(text):
    file, bitArray = before_coding(text)
    length = len(bitArray) % pow(2, 64)
    step1 = appendPaddingBytes(bitArray)
    step2 = appendLength(step1, length)

    intArray = fromBitsToInt(step2)
    h0, h1, h2, h3, h4 = mainProcces(intArray)
    h0res = bin(h0)[2:].zfill(32)
    h1res = bin(h1)[2:].zfill(32)
    h2res = bin(h2)[2:].zfill(32)
    h3res = bin(h3)[2:].zfill(32)
    h4res = bin(h4)[2:].zfill(32)
    hAll = h0res + h1res + h2res + h3res + h4res
    tmp = bitarray(endian="big")
    tmp.extend(hAll)
    res = ''
    for i in range(len(tmp)//4):
        res+=hex(int.from_bytes(tmp[:4].tobytes(), byteorder="big"))[2:][:1]
        tmp = tmp[4:]
    return res
from gost89 import GOST_28147
def toLittle(K) -> object:
    res = ''
    for i in range(len(K) // 8):
        res += K[len(K) - 8:len(K)]
        K = K[0:len(K) - 8]
    return res

def before_coding(text):
    if type(text) == str:
        text = text.encode("utf-8")
    else:
```

```
        text = text
    b = hex(int.from_bytes(text, "big"))[2:]
    res1 = ''
    for i in range(len(b)):
        res1+=bin(int(b[i], 16))[2:].zfill(4)
    return res1
def transA(block):
    y = []
    for i in range(4):
        y.append(block[:64])
        block = block[64:]
    return y[1] + y[2] + y[3] + bin(int(y[0], 2) ^ int(y[1],
2))[2:].zfill(64)


def transP(block):
    fi = [0 for i in range(32)]
    for i in range(4):
        for k in range(1, 9):
            fi[i + 1 + 4*(k-1) - 1] = 8*i + k - 1
    Y = []
    for i in range(len(block)//8):
        Y.append(block[:8])
        block = block[8:]
    res = ''
    for i in range(len(Y)):
        res+=Y[fi[i]] #Y[fi[len(Y)-i-1]]
    return res



def genKeys(Hin, m):
    C = [0,
0x00ff00ff00ff00ffff00ff00ff00ff0000ffff00ff0000ffff000000ffff00ff, 0]
    U = Hin
    V = m
```

```
    W = bin(int(U, 2) ^ int(V, 2))[2:].zfill(256)
    K = []
    K.append(transP(W))
    for i in range(1, 4):
        U = bin(int(transA(U), 2) ^ C[i-1])[2:].zfill(256)
        V = transA(transA(V))
        W = bin(int(U, 2) ^ int(V, 2))[2:].zfill(256)
        K.append(transP(W))
    K1 = K[0]
    res = ''
    for i in range(len(K1)//4):
        res+=hex(int(K1[:4], 2))[2:][:1]
        K1 = K1[4:]
    return K
def transShifr(Hin, K):
    h = []
    for i in range(len(Hin)//64):
        h.append(Hin[:64])
        Hin = Hin[64:]
    S = ''
    for i in range(4):
        S+=GOST_28147(K[i], h[i], 1)
    K1 = S
    res = ''
    for i in range(len(K1)//4):
        res+=hex(int(K1[:4], 2))[2:][:1]
        K1 = K1[4:]
    return S


def psi(block):
    Y = []
    for i in range(len(block) // 16):
```

```
        Y.append(block[:16])
        block = block[16:]
    tmp = bin(int(Y[0], 2) ^ int(Y[1], 2))[2:].zfill(16)
    tmp = bin(int(tmp, 2) ^ int(Y[2], 2))[2:].zfill(16)
    tmp = bin(int(tmp, 2) ^ int(Y[3], 2))[2:].zfill(16)
    tmp = bin(int(tmp, 2) ^ int(Y[12], 2))[2:].zfill(16)
    tmp = bin(int(tmp, 2) ^ int(Y[15], 2))[2:].zfill(16)
    res = ''
    for i in range(1, 16, 1):
        res+=Y[i]
    res = res + tmp
    return res


def transShuffle(Hin, S, m):
    Hout = S
    for i in range(12):
        Hout = psi(Hout)
    Hout = bin(int(Hout, 2) ^ int(m, 2))[2:].zfill(256)
    Hout = psi(Hout)
    Hout = bin(int(Hin, 2) ^ int(Hout, 2))[2:].zfill(256)
    for i in range(61):
        Hout = psi(Hout)
    K1 = Hout
    res = ''
    for i in range(len(K1)//4):
        res+=hex(int(K1[:4], 2))[2:][:1]
        K1 = K1[4:]
    return Hout
def funcF(Hin, m):
    K = genKeys(Hin, m)
    shifr = transShifr(Hin, K)
    shuffle = transShuffle(Hin, shifr, m)
```

```
    return shuffle
def GOST341194(text, Hin):
    bitArray = before_coding(text)
    Sum = '0'.zfill(256)
    L = '0'.zfill(256)

    for i in range(0, len(bitArray) - 255, 256):
        m = bitArray[:256]
        bitArray = bitArray[256:]
        Hin = funcF(Hin, m)
        Sum = bin((int(Sum, 2) + int(m, 2)) % pow(2, 256))[2:].zfill(256)
        L = bin(((int(L, 2) + 256) % pow(2, 256)))[2:].zfill(256)



    if len(bitArray) > 0:
        length = len(bitArray)
        while len(bitArray) != 256:
            bitArray = bitArray + '0'
        L = bin(((int(L, 2) + length) % pow(2, 256)))[2:].zfill(256)
        Hin = funcF(Hin, bitArray)
        Sum = bin((int(Sum, 2) + int(bitArray, 2)) % pow(2,
256))[2:].zfill(256)


    L = toLittle(L)
    Hin = funcF(Hin, L)
    Hin = funcF(Hin, Sum)


    resHex = ''
    for i in range(len(Hin)//4):
        resHex+=hex(int(Hin[:4], 2))[2:][:1]
        Hin = Hin[4:]
    return resHex
```