Лабораторная работа № 3 по курсу криптографии

Выполнил студент группы М8О-306Б *Дубинин Артем*.

Условие

1. Строку в которой записано своё ФИО подать на вход в хеш-функцию ГОСТ Р 34.11-2012 (Стрибог). Младшие 4 бита выхода интерпретировать как число, которое в дальнейшем будет номером варианта. Процесс выбора варианта требуется отразить в отчёте.

2. Программно реализовать один из алгоритмов функции хеширования в соответствии с номером варианта. Алгоритм содержит в себе несколько раундов.

3. Модифицировать оригинальный алгоритм таким образом, чтобы количество раундов было настраиваемым параметром программы. в этом случае новый алгоритм не будет являться стандартом, но будет интересен для исследования.

4. Применить подходы дифференциального криптоанализа к полученным алгоритмам с разным числом раундов.

5. Построить график зависимости количества раундов и возможности различения отдельных бит при количестве раундов 1,2,3,4,5,... .

6. Сделать выводы.

   Примечание No1. Допустимо использовать сторонние реализации для пункта 1, при условии, что они проходят тесты из стандарта и пригодны для дальнейшей модификации.

   Примечание No2. Если в алгоритме описывается семейство с разными размерами блоков, то можно выбрать любой из них.

Метод решения

Выбор варианта осуществлялся с помощью библиотеки pygost:

```
art@mars:~/study/Cryptography/lab_3$ cat variant.py
from pygost import gost34112012256
print(gost34112012256.new("Дубинин Артем Олегович".encode('utf-8'))
.digest().hex()[-1])
art@mars:~/study/Cryptography/lab_3$ python3 variant.py
6
```

Вариант 6: SHA-2

Для выполнения задания я решил использовать уже хорошо написанный код SHA-2, так как это позволяет примечание №1 к заданию. Так семейство алгоритмов SHA-2 включает несколько(SHA-224, SHA-256, SHA-512 и тд.) было решено взять один из них, а именно SHA-256, так как по примечанию №2 -- это возможно.

Функции

SHA-256 использует 6 логических функций, где каждая функция оперирует 32 битными словами, которые представляются переменными x, y, z. Результат каждой функции тоже 32 битное слово.

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_{0}^{\{256\}}(x) = ROTR^{2}(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$
$$\sum_{1}^{\{256\}}(x) = ROTR^{6}(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$
$$\sigma_0^{\{256\}}(x) = ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x)$$
$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

Константы

SHA-256 использует последовательность из 64 констант(32 битное слово) $K_0^{\{256\}}, K_1^{\{256\}}, ..., K_{63}^{\{256\}}$. Эти константы представляют первые 32 бита дробных частей корней куба первых шестидесяти четырех простых чисел.

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2.
```

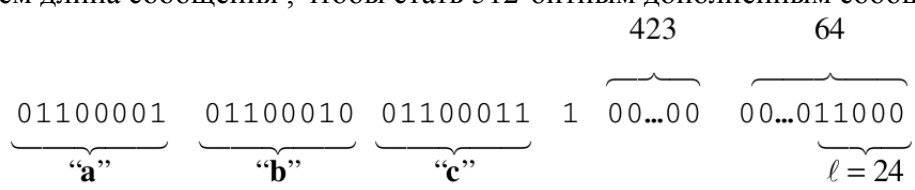
Предпроцессинг

Предварительная обработка должна выполняться до начала вычисления хеша. Эта предварительная обработка состоит из трех этапов: дополнение сообщения M, разбор дополненного сообщения в блоки сообщений и установка начального значения хеш-функции $H^{(0)}$.

Дополнение сообщения Сообщение M должно быть дополнено до начала вычисления хеша. Цель этого дополнения - убедиться, что заполненное сообщение кратно 512 или 1024 битам, в зависимости от алгоритма.

Допустим длинна сообщения, M это l бит. Добавим бит «1» в конец сообщения, за которым следует k нулевых битов, где k - наименьшее неотрицательное решение уравнения l + 1

+ k = 448 mod 512. Затем добавьте 64-битный блок, который равен числу l, выраженному в двоичном представлении. Например, (8-битное ASCII) сообщение «abc» имеет длину 8 · 3 = 24, поэтому сообщение дополняется одним битом, затем 448 - (24 + 1) = 423 нулевых бита, а затем длина сообщения , чтобы стать 512-битным дополненным сообщением.



Длина дополненного сообщения теперь должна быть кратна 512 битам.

Парсинг дополненного сообщения

После того как сообщение дополнено, оно должно быть проанализировано в N m-битных блоков, прежде чем начнется вычисление хеша.

Для SHA-1 и SHA-256 заполненное сообщение анализируется на N 512-битных блоков, $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Поскольку 512 бит входного блока могут быть выражены как шестнадцать 32-битных слов, первые 32 бита блока сообщения i обозначаются как $M_0^{(i)}$, следующие 32 бита - это $M_1^{(i)}$ и так далее до ${}_15^{(i)}$.

Установка начального значения хеш-функции

Перед началом вычисления хеша для каждого из алгоритмов защищенного хеша необходимо установить начальное хеш-значение $H^{(0)}$. Размер и количество слов в $H^{(0)}$ зависит от размера дайджеста сообщения.

Для SHA-256 начальное хеш-значение $H^{(0)}$ должно состоять из следующих восьми 32-битных слов в шестнадцатеричном виде:

$H_0^{(0)}$ = 6a09e667
$H_1^{(0)}$ = bb67ae85
$H_2^{(0)}$ = 3c6ef372
$H_3^{(0)}$ = a54ff53a
$H_4^{(0)}$ = 510e527f
$H_5^{(0)}$ = 9b05688c
$H_6^{(0)}$ = 1f83d9ab
$H_7^{(0)}$ = 5be0cd19.

Алгоритм

SHA-256 может использоваться для хеширования сообщения M длиной l бит, где $0 \leq l < 2^{64}$. Алгоритм использует 1) расписание сообщений из шестидесяти четырех 32-битных слов, 2) восемь рабочих переменных по 32 бита каждая и 3) значение хеш-функции восьми 32-битных слов. Конечным результатом SHA-256 является 256-битный дайджест сообщения.

Слова расписания сообщений помечены $W_0, W_1, ..., W_{63}$. Восемь рабочих переменных помечены как a, b, c, d, e, f, g и h. Слова значения хэша помечены как $H_0^{(i)}, H_1^{(i)}, ..., H_7^{(i)}$, кото-

рый будет содержать начальное значение хеш-функции $H^{(0)}$, замененное каждым последующим промежуточным значением хеш-функции (после обработки каждого блока сообщения), $H^{(i)}$ и заканчивая конечным значением хеш-функции $H^{(N)}$. SHA-256 также использует два временных слова, $T_1 T_2$.

SHA-256 Препроцессинг

- Дополним сообщение M, в соответствии с алгоритмом указанным выше.
- Спарсим дополненное сообщение в N 512-битных блок.
- Заполним начальный хэш

SHA-256 Вычисление хэша

В вычислении хэша SHA-256 используются функции и константы, ранее определенные. Сложение (+) выполняется по модулю 2^32.

После завершения предварительной обработки каждый блок сообщений, $M^{(1)}, M^{(2)}, ..., M^{(N)}$, обрабатывается по порядку,используя следующие шаги:

for i = 1 to N:

1. Подготовим таблицу сообщений, W_t :

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

2. Инициализируем восемь рабочих переменных a, b, c, d, e, f, g и h с помощью $(i-1)^-$ хеш-значения:

$$a = H_0^{(i-1)}$$
$$b = H_1^{(i-1)}$$
$$c = H_2^{(i-1)}$$
$$d = H_3^{(i-1)}$$
$$e = H_4^{(i-1)}$$
$$f = H_5^{(i-1)}$$
$$g = H_6^{(i-1)}$$
$$h = H_7^{(i-1)}$$

3. For t = 0 to 63:

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$
$$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

4. Вычислим i промежуточное хеш-значение H^i:

$$H_0^{(i)} = a + H_0^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)}$$
$$H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_7^{(i)} = h + H_7^{(i-1)}$$

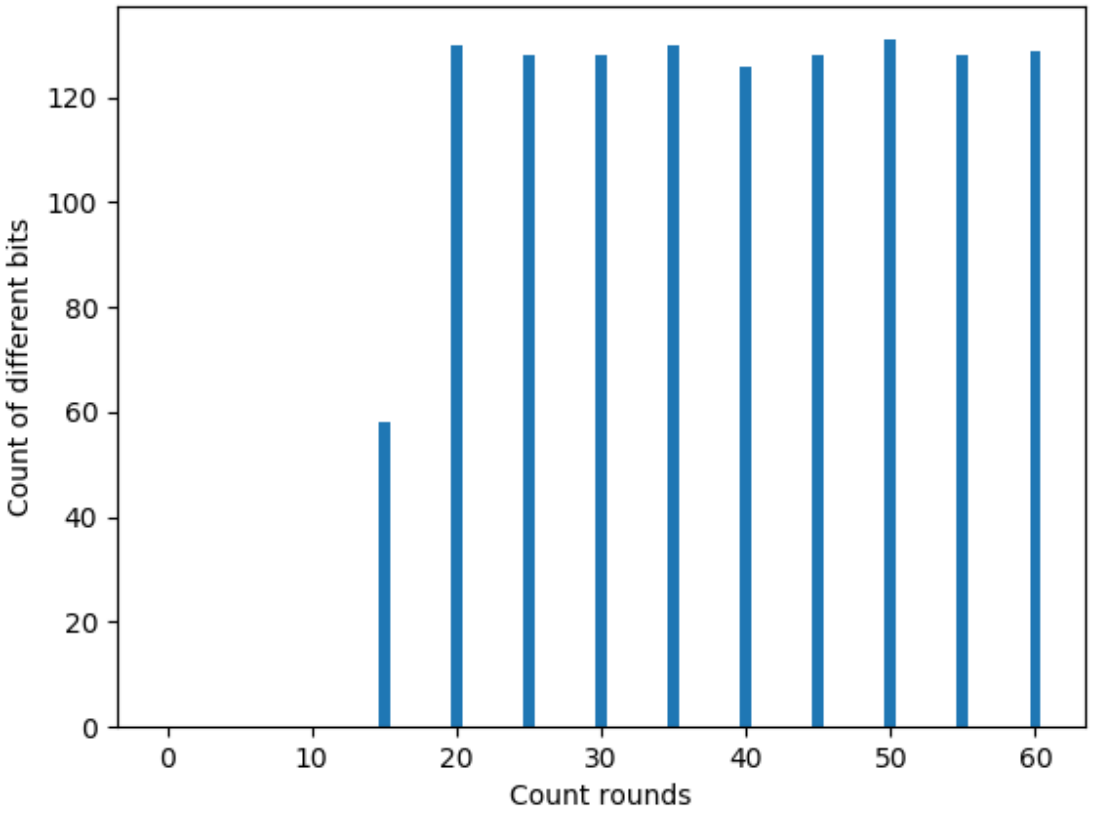
После повторения шагов с одного по четыре в общей сложности N раз (то есть после обработки $M^{(N)}$) результирующий дайджест сообщения M, состоящий из 256 битов, равен

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)} .$$

Реализация проходит тесты. Значение написанной программы сравниваются со стандартной python'a(hashlib).

Далее были применены подходы дифференциального криптоанализа к полученной хеш-функции. Я генерирую строку, создаю ее копию с инвертированным последним битом. Затем считаю хеши обеих строк для количества раундов в интервале от 0 до 60 с шагом 5. В конце считаю количество различных бит в полученных хешах. Для более честного анализа данная процедура была проведена 10 раз и взяты средние значения количества измененных бит для каждого теста с раундами.

Кол-во раундов	Число изменных бит
5	0
10	0
15	58
20	130
25	128
30	128
35	130
40	126
45	128
50	131
55	128
60	129



Такой анализ позволяет увидеть, насколько меняется хеш при минимальном изменении исходного сообщения. Учитывая, что итоговое значение имеет длину 256 бит, нетрудно заметить, что примерно с 25 раунда меняется примерно половина бит хеша, что не может нас не радовать с точки зрения криптографии. Можно предположить, что SHA-1 удовлетворяет ла-

винному критерию. Конечно, сложно утверждать, что полученные результаты хоть сколько-нибудь значимы, потому что было проведено слишком мало тестов.

Результат работы программы

```
art@mars:~/study/Cryptography/lab_3$ tail -11 sha256.py
    vectors = [
        '',
        'abc',
        'The quick brown fox jumped over the lazy dog',
        'The quick brown fox jumped over the lazy dog.',
        "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
    ]
    for i in vectors:
        assert hashlib.sha256(i.encode()).hexdigest() == sha256(i, 64)

    print("all tests passed")
art@mars:~/study/Cryptography/lab_3$ python3 sha256.py
all tests passed
```

Выводы

До этого мне приходилось сталкиваться только с алгоритмом хэширования md5 и то очень поверхностно. Сейчас же я понял, как работают алгоритмы хэширования в целом. В частности было очень интересно понять, как работает функция sha-256. Мне показалось, что sha-256 похож на стиральную машину, у которой изначально есть переменные $H_1, H_2, ..., H_7$, которая получает кусок сообщения и перемешивает биты и меняет сами переменные $H_1, H_2, ..., H_7$ и делает это детерменированно. Так же я наткнулся на видео известно канала по математики 3Blue1Brown, который на примере sha-256 красиво обьяснял насколько перебором будет сложно подобрать хэш, если бы почти у каждого человека на планете земля был бы компьютер мощностью серверов гугл и таких планет, как земля было бы много, все равно бы понадобилось бы 507 миллиардов лет.

Листинг программного кода

sha256.py

```
class sha2_32(object):
    ''' Superclass for both 32 bit SHA2 objects (SHA224 and SHA256) '''

    def __init__(self, message, rounds):
        self.rounds = rounds
        length = bin(len(message) * 8)[2:].rjust(64, "0")
```

```
        while len(message) > 64:
            self._handle(''.join(bin(i)[2:].rjust(8, "0")
                                 for i in message[:64]))
            message = message[64:]
        message = ''.join(bin(i)[2:].rjust(8, "0") for i in message) + "1"
        message += "0" * ((448 - len(message) % 512) % 512) + length
        for i in range(len(message) // 512):
            self._handle(message[i * 512:i * 512 + 512])

    def _handle(self, chunk):

        rrot = lambda x, n: (x >> n) | (x << (32 - n))
        w = []

        k = [
            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]

        for j in range(len(chunk) // 32):
            w.append(int(chunk[j * 32:j * 32 + 32], 2))

        for i in range(16, 64):
            s0 = rrot(w[i - 15], 7) ^ rrot(w[i - 15], 18) ^
            (w[i - 15] >> 3)
            s1 = rrot(w[i - 2], 17) ^ rrot(w[i - 2], 19) ^
            (w[i - 2] >> 10)
            w.append((w[i - 16] + s0 + w[i - 7] + s1) & 0xffffffff)
```

```
        a = self._h0
        b = self._h1
        c = self._h2
        d = self._h3
        e = self._h4
        f = self._h5
        g = self._h6
        h = self._h7

        for i in range(self.rounds):
            s0 = rrot(a, 2) ^ rrot(a, 13) ^ rrot(a, 22)
            maj = (a & b) ^ (a & c) ^ (b & c)
            t2 = s0 + maj
            s1 = rrot(e, 6) ^ rrot(e, 11) ^ rrot(e, 25)
            ch = (e & f) ^ ((~ e) & g)
            t1 = h + s1 + ch + k[i] + w[i]

            h = g
            g = f
            f = e
            e = (d + t1) & 0xffffffff
            d = c
            c = b
            b = a
            a = (t1 + t2) & 0xffffffff

        self._h0 = (self._h0 + a) & 0xffffffff
        self._h1 = (self._h1 + b) & 0xffffffff
        self._h2 = (self._h2 + c) & 0xffffffff
        self._h3 = (self._h3 + d) & 0xffffffff
        self._h4 = (self._h4 + e) & 0xffffffff
        self._h5 = (self._h5 + f) & 0xffffffff
        self._h6 = (self._h6 + g) & 0xffffffff
        self._h7 = (self._h7 + h) & 0xffffffff

    def hexdigest(self):
        return ''.join(hex(i)[2:].rjust(8, "0")
                       for i in self._digest())

    def digest(self):
        hexdigest = self.hexdigest()
        return bytes(int(hexdigest[i * 2:i * 2 + 2], 16)
```

```
                    for i in range(len(hexdigest) // 2))


class SHA256(sha2_32):
    _h0, _h1, _h2, _h3, _h4, _h5, _h6, _h7 = (
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19)

    def _digest(self):
        return (self._h0, self._h1, self._h2, self._h3,
                self._h4, self._h5, self._h6, self._h7)


def sha256(message, rounds):
    ''' Return hexdigest '''
    encode_message = str.encode(message)
    hash_algo = SHA256(encode_message, rounds)
    return hash_algo.hexdigest()


if __name__ == '__main__':
    # m = input()
    # r = int(input())
    # print(sha256(m, r))

    import hashlib

    vectors = [
        '',
        'abc',
        'The quick brown fox jumped over the lazy dog',
        'The quick brown fox jumped over the lazy dog.',
        "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
    ]
    for i in vectors:
        assert hashlib.sha256(i.encode()).hexdigest() == sha256(i, 64)

    print("all tests passed")
```

---

diff_anal.py

```
import random
import string
```

```
import logging

import bitarray
import matplotlib.pyplot as plt

import sha256


CNT_TESTS = 11


def get_random_string(N=50):
    return ''.join(random.choice(string.ascii_letters + string.digits) for


def change_one_bit(msg):
    ba = bitarray.bitarray()
    ba.frombytes(msg.encode('ascii'))
    last_bit = ba[-1]
    new_last = bitarray.bitarray('0') if last_bit else bitarray.bitarray('1
    ba = ba[:-1]
    ba += new_last
    return bitarray.bitarray(ba.tolist()).tobytes().decode('ascii')


def bitcount(n):
    return bin(n).count('1')


def mean(list_):
    return [int(sum(i)/len(i)) for i in zip(*list_)]


if __name__ == '__main__':
    logging.basicConfig(filename="analysis.log", level=logging.INFO)

    all_diffs = []
    cnt_rounds = [i for i in range(0, 65, 5)]
    for i in range(1, CNT_TESTS):
        logging.info("Test #{0}".format(i))
        input1 = get_random_string()
        input2 = change_one_bit(input1)
```

```
        logging.info("Input string:   {0}".format(input1))
        logging.info("Changed string: {0}".format(input2))

        diffs = []
        rounds = range(0, 65, 5)
        for i in rounds:
            logging.info("Count rounds: {0}".format(i))
            output1 = sha256.sha256(input1, i)
            output2 = sha256.sha256(input2, i)
            logging.info("Output original:  {0}".format(output1))
            logging.info("Output changed:   {0}".format(output2))
            res = bitcount(int(output1, 16) ^ int(output2, 16))
            diffs.append(res)
            logging.info("Count of different bits: {0}".format(res))
        logging.info("------------")
        all_diffs.append(diffs)

mean_diffs = mean(all_diffs)
for i, j in zip(cnt_rounds, mean_diffs):
    print("Count rounds: {0}".format(i))
    print("Count of different bits: {0}".format(j))
    print("------------")

plt.bar(cnt_rounds, mean_diffs, align='center')
plt.xlabel('Count rounds')
plt.ylabel('Count of different bits')
plt.show()
```