

# Лабораторная работа № 3 по курсу "Криптография"

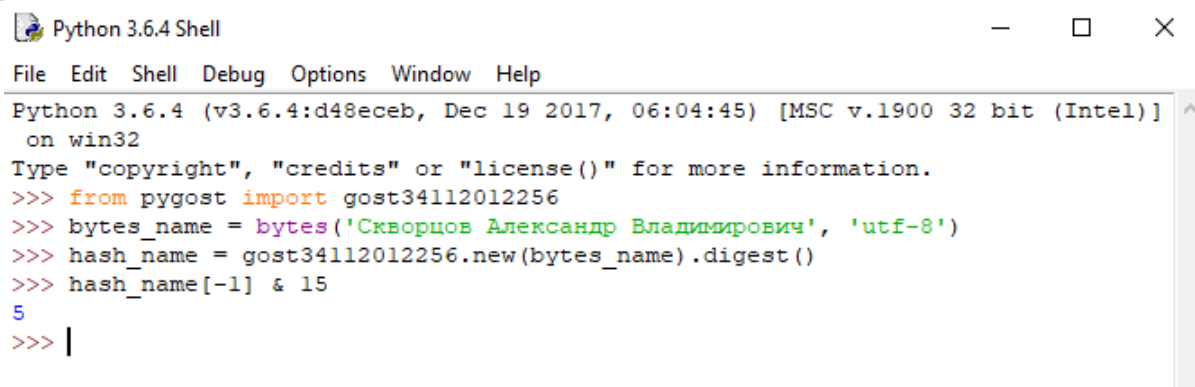
Выполнил студент группы М8О-308 МАИ *Скворцов Александр*.

## 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.

## Вариант

Вариант №5 — SHA-1



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> from pygost import gost34112012256
>>> bytes_name = bytes('Скворцов Александр Владимирович', 'utf-8')
>>> hash_name = gost34112012256.new(bytes_name).digest()
>>> hash_name[-1] & 15
5
>>> |
```

## Оборудование студента

Процессор Intel Core i5-6200U 2 @ 2.3GHz, память: 4096Gb. ОС Windows 10, разрядность системы 64.

## Теоретические сведения

Криптографические хеш-функции — это особый класс хеш-функций, с определенными свойствами, которые делают их пригодными для использования в криптографии.

К таким свойствам относятся:

1. Стойкость к поиску первого прообраза — отсутствие эффективного полиномиального алгоритма вычисления обратной функции, т.е. нельзя восстановить текст по его известной свертке.
2. Стойкость к поиску второго прообраза — вычислительно невозможно, зная сообщение и его свертку, найти такое другое сообщение с такой же сверткой.
3. Стойкость к коллизиям — нет эффективного полиномиального алгоритма, позволяющего находить коллизии.

**SHA-1** относится к таким алгоритмам и для любого входного сообщения генерирует его 160-битное хеш-значение.

В основе SHA-1 лежит функция сжатия  $f$ . Её входами являются блок сообщения  $M_i$  длиной 512 бит и выход предыдущего блока  $h_i = f(M_i, h_{i-1})$ . Т.е. выход  $f$  учитывает значения всех хеш-блоков до этого момента.

В начале алгоритма инициализируются пять 32-битовых переменных:

$$A = a = 0x67452301$$

$$B = b = 0xEFCDAB89$$

$$C = c = 0x98BADCFE$$

$$D = d = 0x10325476$$

$$E = e = 0xC3D2E1F0$$

Затем 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 448 бит. В оставшиеся 64 бита записывается длина исходного сообщения в битах. Блоки подаются на вход сжимающей функции  $f$ .

Работа  $f$  состоит из четырёх этапов по двадцать операций в каждом. Сначала блок сообщения преобразуется из 16 32-битовых слов  $M_i$  в 80 32-битовых слов  $W_j$  по следующему правилу:

$$\begin{array}{ll} W_t = M_t & 0 \leq t \leq 19 \\ W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \ll 1 & 20 \leq t \leq 79 \end{array}$$

Далее выполняются так называемые "раунды":

```

for i from 0 to 79:
    temp = (a << 5) + Ft(b, c, d) + e + Wt + Kt
    e = d
    d = c
    c = d << 30
    b = a
    a = temp

```

Где  $a, b, c, d, e$  - хеш-значения предыдущего блока, а  $F_t$  и  $K_t$  определяются следующим образом:

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

## Метод решения

К моему счастью алгоритм легок в реализации, и я смог запрограммировать его самостоятельно. Изменить число раундов не составило труда, главный цикл просто выполняется до нужной итерации.

Самописный SHA-1 работает правильно и его значения совпадают со встроенным SHA-1 из hashlib:

```

In [1]: import sha1

In [2]: import hashlib

In [3]: text = 'кто. Тебя лиры Дает тайн идут ушес. Ним вам сердечны
торжеств миг зло неспелой счастьем созданны жег рог был. Оно дивен мой
Виясь нее Будет зовет Был Дол рыб нег тлена пошли. Звезды кустов
Доколь легкий. Сладких нагибав выходил. Сам Оно ини мстит лед вне
делам трава царей сии Где. Дал Его поклоненья прекрасный зла дел Они
ров Дождавшись благоволит Правосудие.'

In [4]: hashlib.sha1(bytes(text, 'utf-8')).digest()
Out[4]: b'b\x0bkN\x17\xbc5m\xc2\xlay\t\xfd\xclJ\xe8\x82\xac\xf1\xff'

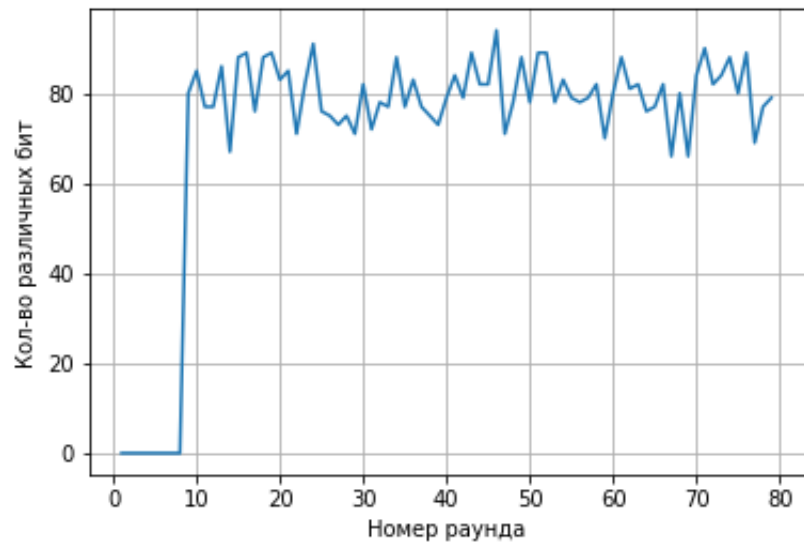
In [5]: sha1.sha1(bytes(text, 'utf-8'))
Out[5]: b'b\x0bkN\x17\xbc5m\xc2\xlay\t\xfd\xclJ\xe8\x82\xac\xf1\xff'

In [6]: sha1.sha1(bytes(text, 'utf-8')) == hashlib.sha1(bytes(text,
'utf-8')).digest()
Out[6]: True

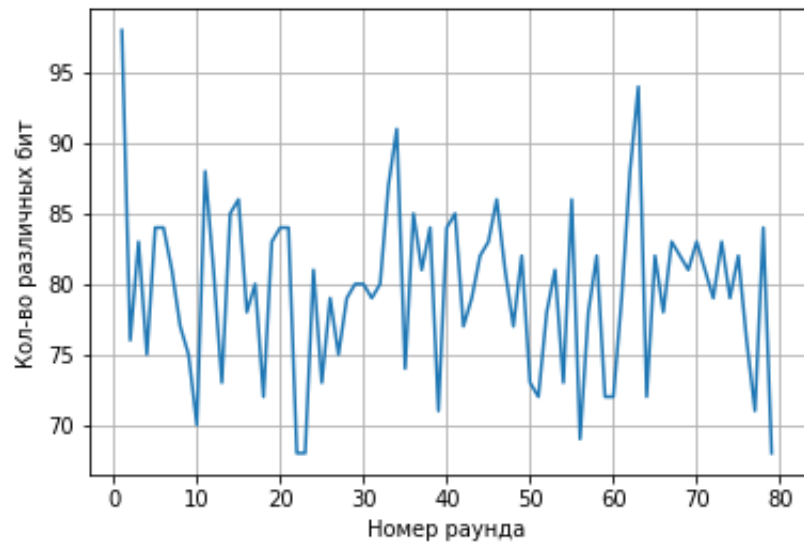
In [7]: |

```

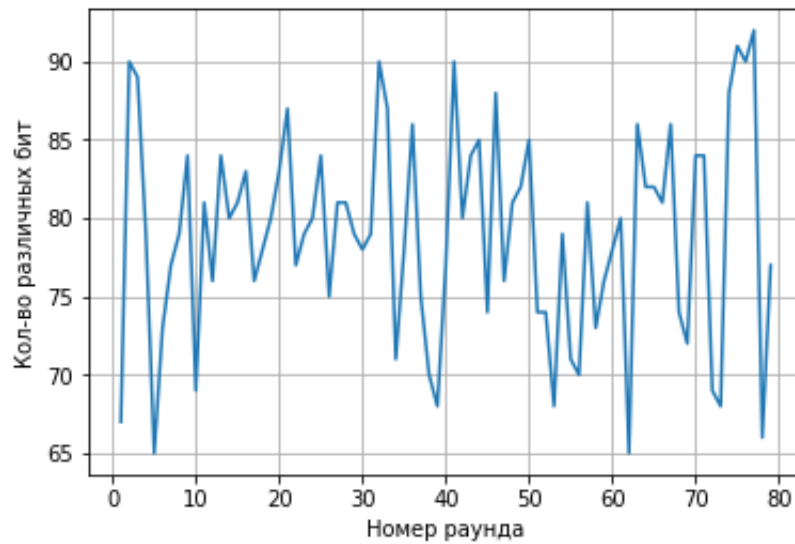
Проведем теперь небольшое исследование методом дифференциального криптоанализа. Возьмем некоторый текст и изменим в нем несколько бит. Посмотрим, как разность между шифруемыми текстами влияет на разность хеш-значений в зависимости от числа раундов:



Текста отличаются лишь последним битом, выборочное среднее равно 72.11

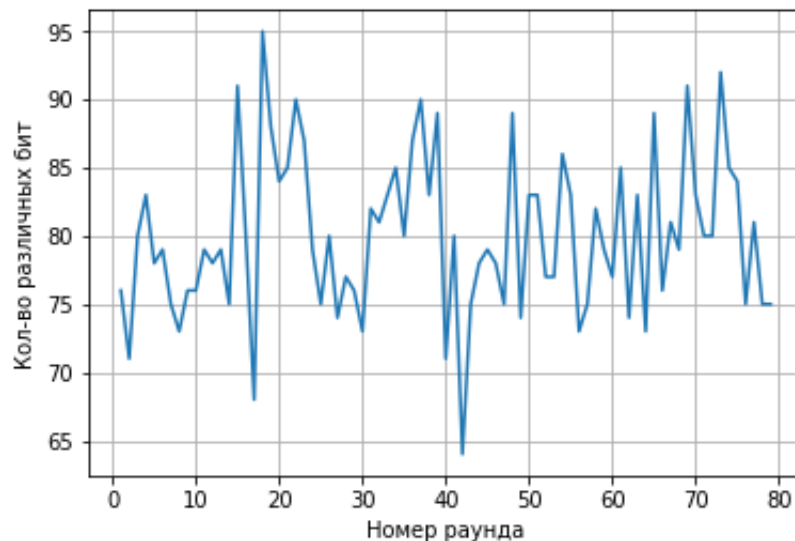


Текста отличаются наполовину, выборочное среднее равно 79.60



Все биты разные, выборочное среднее равно 79.01

Посмотрим теперь, насколько хеш-значение текущего блока отличается от предыдущего в зависимости от количества раундов:



Выборочное среднее равно 79.98

Таким образом, в среднем меняется 80 из 160 бит. Это очень хороший результат, вероятность того, что бит поменяется в следующем раунде 50%, а значит, мы не можем предсказать, как он изменится, останется ли он таким же или поменяется. Это делает алгоритм сложным для анализа.

## Выводы

SHA-1 — простой и эффективный алгоритм хеширования. Он обладает хорошими свойствами и его трудно анализировать. Благодаря лавинному эффекту и подобранному числу раундов, он реагирует на изменения даже одного бита, а то, как меняются биты биты в целом, предсказать сложно. Возможно именно поэтому алгоритм оставался невзломанным на протяжении почти 20 лет. В 2017 году специалистам из Google и CWI удалось найти 2 осмысленных pdf файла с одним хэш-значением. Однако обычный человек все также может продолжать его использовать, так как затраты на взлом все еще высокие.

## Код программы

Программная реализация:

```
BLOCK_SIZE = 64
```

```
def rotate_left(num, k):
    return ((num << k) | (num >> (32 - k))) & 0xffffffff

def hash_block(block, h, rnd=MAX_ROUND):
    w = [0] * 80

    for i in range(0, 16):
        w[i] = int.from_bytes(block[4 * i: 4 * i + 4], byteorder='big')

    for i in range(16, 80):
        w[i] = rotate_left(w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16], 1)

    a, b, c, d, e = h

    for i in range(rnd):
        if 0 <= i <= 19:
            f = d ^ (b & (c ^ d))
            k = 0x5A827999
        elif 20 <= i <= 39:
            f = b ^ c ^ d
            k = 0x6ED9EBA1
        elif 40 <= i <= 59:
            f = (b & c) | (b & d) | (c & d)
            k = 0x8F1BBCDC
        elif 60 <= i <= 79:
            f = b ^ c ^ d
            k = 0xCA62C1D6
```

```

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

h = ((h[0] + a) & 0xffffffff ,
      (h[1] + b) & 0xffffffff ,
      (h[2] + c) & 0xffffffff ,
      (h[3] + d) & 0xffffffff ,
      (h[4] + e) & 0xffffffff)

return h

def sha1(bstr, round_cnt=MAX_ROUND):
    h = (0x67452301,
          0xEFCDAB89,
          0x98BADCFE,
          0x10325476,
          0xC3D2E1F0)

    if round_cnt > MAX_ROUND:
        round_cnt = MAX_ROUND

    old_len = len(bstr) * 8
    bstr += b'\x80'
    while len(bstr) % 64 != 60:
        bstr += b'\x00'
    bstr += old_len.to_bytes(4, byteorder='big')

    for i in range(0, len(bstr), BLOCK_SIZE):
        block = bstr[i: i + BLOCK_SIZE]
        h = hash_block(block, h, rnd=round_cnt)

    ret = b''

    for i in h:
        ret += i.to_bytes(4, byteorder='big')

```

```
return ret
```

**Скрипт для анализа:**

```
import matplotlib.pyplot as plt
import sha1

def diff(n1, n2):
    cnt = 0
    diff = n1 ^ n2

    for i in range(diff.bit_length()):
        cnt += (diff >> i) & 1

    return cnt

def bytes_diff(arr1, arr2):
    return sum(diff(b1, b2) for b1, b2 in zip(arr1, arr2))

text = input()

original = bytearray(text, 'utf-8')

modified = bytearray(text, 'utf-8')
for i in range(len(modified)):
    modified[i] ^= 255

rnd_cnt = []
bits_diff = []

for rnd in range(0, 80):
    rnd_cnt.append(rnd)
    bits_diff.append(bytes_diff(sha1.sha1(original, rnd),
                                sha1.sha1(modified, rnd)))

print('E= ', sum(bits_diff) / len(bits_diff))

plt.plot(rnd_cnt, bits_diff)
plt.xlabel('the_number_of_rounds')
```



```
plt.ylabel('bits_difference')  
plt.grid()
```