**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Курсовой проект  
по курсу «Криптография»

Группа: М8О-306Б-21

Студент: E. В. Черевичин

Преподаватель: А. В. Борисов

Оценка:

Дата: 11.05.2024

Москва, 2024

**ОГЛАВЛЕНИЕ**

[1 Тема 3](#_Toc162017708)

[2 Задание 3](#_Toc162017709)

[3 Теория 5](#_Toc162017710)

[4 Ход лабораторной работы 6](#_Toc162017711)

[5 Выводы 15](#_Toc162017712)

# **Тема**

Дифференциальный криптоанализ алгоритма хеширования.

# **Задание**

№0. Строку в которой записано своё ФИО подать на вход в хеш-функцию ГОСТ Р 34.11-2012

(Стрибог). Младшие 4 бита выхода интерпретировать как 16-тиричное число, которое в

дальнейшем будет номером варианта.

№1. Программно реализовать один из алгоритмов функции хеширования в соответствии с номером варианта. Алгоритм содержит в себе несколько раундов.

№2. 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.

№3. Применить подходы дифференциального криптоанализа к полученным алгоритмам с разным числом раундов.

№4. Построить график зависимости количества раундов и возможности различения отдельных бит при количестве раундов 1,2,3,4,5,... .

№5. Сделать выводы.

Примечание №1. Допустимо использовать сторонние реализации для пункта 1, при условии,

что они проходят тесты из стандарта и пригодны для дальнейшей модификации.

Примечание №2. Если в алгоритме описывается семейство с разными размерами блоков, то

можно выбрать любой из них.

Приложение №1.

Номер варианта == Алгоритм

0 == ГОСТ Р 34.11-94

1 == ГОСТ Р 34.11-2012 (Стрибог)

2 == Luffa

3 == BLAKE

4 == SHA-0

5 == SHA-1

6 == SHA-2

7 == Keccak

8 == JH

9 == Shabal

A == Skein

B == Blue Midnight Wish-256

C == CubeHash

D == MD5

E == SIMD

F == Whirlpool

Приложение №2.

Процесс выбора варианта, также требуется отразить в отчёте.

В примере ниже описан выбор варианта с помощью библиотеки pygost.

variant.py

from pygost import gost34112012256

try:

    data = "Черевичин Егор Викторович"  *# Unicode string*

    encoded\_data = data.encode("utf-8")  *# Encode the Unicode string to bytes*

    hash\_object = gost34112012256.new(encoded\_data)

    hash\_result = hash\_object.digest()

    print(hash\_result)

except Exception as e:

    print("An error occurred:", e)

Вывод



Следовательно, исследуемый вариант 5(SHA-1)

**3 Теория**

**SHA-1** — **Secure Hash Algorithm 1** — алгоритм криптографического хеширования. Описан в RFC 3174. Для входного сообщения произвольной длины (максимум 2^64 бит, что примерно равно 2 эксабайта) 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. Также рекомендован в качестве основного для государственных учреждений в США. Принципы, положенные в основу SHA-1, аналогичны тем, которые использовались Рональдом Ривестом при проектировании MD4.

**Стрибог** — криптографический алгоритм вычисления хеш-функции с размером блока входных данных 512 бит и размером хеш-кода 256 или 512 бит.

**Дифференциальный криптоанализ** — атака с подобранным открытым текстом. Это означает, что для применения ДК вы должны иметь возможность зашифровать абсолютно любые тексты в абсолютно любом количестве.

**Симметричные криптосистемы** — способ шифрования, в котором для шифрования и расшифрования применяется один и тот же криптографический ключ.

**Блочный шифр** — разновидность симметричного шифра, оперирующего группами бит фиксированной длины — блоками, характерный размер которых меняется в пределах 64‒256 бит.

**Раундом** (или циклом) в криптографии называют один из последовательных шагов обработки данных в алгоритме блочного шифрования

# **4** **Ход лабораторной работы**

Мною было написано 4 программы

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 . Хеш-значением всего сообщения является выход последнего блока.

*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.

*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

Код программы sha1.py:

import os

import argparse

ROUNDS = 80

BLOCK\_SIZE = 512

class SHA1:

    def \_\_init\_\_(self, rounds):

        self.h = (0x67452301,

                  0xEFCDAB89,

                  0x98BADCFE,

                  0x10325476,

                  0xC3D2E1F0)

        self.rounds = rounds

    def update(self, msg):

        msg\_bin = ''

        for i in range(len(msg)):

            msg\_bin += '{0:08b}'.format(ord(msg[i]))

        len\_msg = len(msg\_bin)

        msg\_bin += '1'

        while (len(msg\_bin) % 512 != 448):

            msg\_bin += '0'

        msg\_bin += '{0:064b}'.format(len\_msg)

        chunks = self.get\_chunks(msg\_bin)

        for chunk in chunks:

            self.process\_chunk(chunk)

        return self

    def hexdigest(self):

        return '%08x%08x%08x%08x%08x' % self.h

    def get\_chunks(self, msg):

        return [msg[i:i + BLOCK\_SIZE] for i in range(0, len(msg), BLOCK\_SIZE)]

    def process\_chunk(self, chunk):

        h0, h1, h2, h3, h4 = (i for i in self.h)

        w = []

        for i in range(16):

            w.append(int(chunk[i \* 32:i \* 32 + 32], 2))

        for i in range(16, 80):

            w.append(self.rotl(w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16], 1))

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

        for i in range(self.rounds):

            if 0 <= i <= 19:

                f = (b & c) | ((~b) & 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

            a, b, c, d, e = (self.rotl(a, 5) + f + e + k + w[i]) & 0xffffffff, \

                    a, self.rotl(b, 30), c, d

        h0 = (h0 + a) & 0xffffffff

        h1 = (h1 + b) & 0xffffffff

        h2 = (h2 + c) & 0xffffffff

        h3 = (h3 + d) & 0xffffffff

        h4 = (h4 + e) & 0xffffffff

        self.h = (h0, h1, h2, h3, h4)

    @staticmethod

    def rotl(n, k):

        return ((n << k) | (n >> (32 - k))) & 0xffffffff

def sha1(msg, rounds):

    return SHA1(rounds).update(msg).hexdigest()

if \_\_name\_\_ == '\_\_main\_\_':

    flags = argparse.ArgumentParser()

    flags.add\_argument('--rounds', help='Count rounds (<= 80)')

    flags.add\_argument('--input', required=True, help='Input file to hash')

    args = flags.parse\_args()

    if args.rounds:

        rounds = args.rounds

    else:

        rounds = ROUNDS

    filename = args.input

    if os.path.isfile(filename):

        with open(filename, "r") as f:

            text = f.read()

            print(sha1(text, rounds))

    else:

        print("Error, could not find " + filename + " file." )

Так же, программа создающая папку test и файлы, которые хранят строки разной длины

test\_generate.py

import os

import string

import numpy as np

import random

def generate\_random\_string(l : int):

    return ''.join(random.choices(string.ascii\_letters + string.digits,k=l))

def make\_test(test\_count : int):

    if not os.path.exists("./test"):

        os.mkdir("test")

    os.chdir("test")

    for i in range(test\_count):

        filename = f"test\_{i}.txt"

        with open(filename, "w") as f:

            string\_len = np.random.randint(100)

            f.write(generate\_random\_string(string\_len))

    pass

if \_\_name\_\_ == '\_\_main\_\_':

    print("Enter the numbers of test: ")

    n = int(input())

    make\_test(n)

Программа, проверяющая корректность работы алгоритма при помощи сравнивания результатов с встроенной в питон функцией

import unittest

import subprocess

import hashlib

import sha1

ROUNDS = 80

class TestSHA1(unittest.TestCase):

    def test\_comparison(self):

        print('\n>>> test\_comparison')

        for i in range(10):

            filename = f"test/test\_{i}.txt"

            with open(filename, "r") as f:

                data = f.read()

                custom\_sha1 = sha1.sha1(data, ROUNDS)

                lib\_sha1 = hashlib.sha1(data.encode()).hexdigest()

                self.assertEqual(custom\_sha1, lib\_sha1)

                print('custom\_sha1:', custom\_sha1)

                print('lib\_sha1:', lib\_sha1)

                print(f'test\_{i}: success')

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

И последняя программа, которая проводит дифференциальный криптоанализ, путем сравнивания работы алгоритма тестового слова и того же слова только с 1 случайно измененным битом

differential\_cryptanalysis.py

import random

import string

import logging

import bitarray

import matplotlib.pyplot as plt

import sha1

def get\_random\_string(N=50):

    return ''.join(random.choice(string.ascii\_letters + string.digits) for \_ in range(N))

def flip\_random\_bit(input\_string):

    index = random.randint(0, len(input\_string) - 1)

    char\_code = ord(input\_string[index])

    bit\_position = random.randint(0, 7)

    new\_char\_code = char\_code ^ (1 << bit\_position)

    new\_char = chr(new\_char\_code)

    return input\_string[:index] + new\_char + input\_string[index+1:]

def bitcount(n):

    return bin(n).count('1')

def mean(l):

    return [int(sum(i)/len(i)) for i in zip(\*l)]

if \_\_name\_\_ == '\_\_main\_\_':

    logging.basicConfig(filename="cryptanalysis.log", level=logging.INFO)

    test\_byte\_diff = []

    for i in range(0, 10):

        logging.info(f"TEST NUMBER: {i + 1}")

        filename =  f"test/test\_{i}.txt"

        with open(filename, "r") as f:

            data = f.read()

        changed\_data = flip\_random\_bit(data)

        logging.info(f"CURENT STRING: {data}")

        logging.info(f"DIFFER STRING: {changed\_data}")

        diffs = []

        for count\_round in range(0,80 + 1, 5):

            logging.info(f"NUMBER OF ROUNDS: {count\_round}")

            hash1 = sha1.sha1(data, count\_round)

            hash2 = sha1.sha1(changed\_data, count\_round)

            logging.info(f"ORIGINAL HASH:  {hash1}")

            logging.info(f"CHANGED HASH:   {hash2}")

            diff\_number = bitcount(int(hash1, 16) ^ int(hash2, 16))

            diffs.append(diff\_number)

            logging.info("NUMBER OF DIFFERENT BITS: ".format(diff\_number))

        logging.info("------------")

        test\_byte\_diff.append(diffs)

    rounds\_count = [i for i in range(0, 80 + 1, 5)]

    mean\_diffs = mean(test\_byte\_diff)

    plt.bar(rounds\_count, mean\_diffs, align='center')

    plt.xlabel('Count rounds')

    plt.ylabel('Count of different bits')

    plt.show()

**Проверка алгоритма на правильность работы:**

PS D:\crypto\_kp> python .\test\_generate.py

Enter the numbers of test:

10

PS D:\crypto\_kp> python .\test\_sha1.py

>>> test\_comparison

custom\_sha1: 95cb0bfd2977c761298d9624e4b4d4c72a39974a

lib\_sha1: 95cb0bfd2977c761298d9624e4b4d4c72a39974a

test\_0: success

custom\_sha1: a8a863508f2eaef56eb7cf34ecca39ae71aaaae2

lib\_sha1: a8a863508f2eaef56eb7cf34ecca39ae71aaaae2

test\_1: success

custom\_sha1: 88ab4649985aa002555589b0a76c97789761b3b2

lib\_sha1: 88ab4649985aa002555589b0a76c97789761b3b2

test\_2: success

custom\_sha1: 5c9b601998a6b9e2f2c7eee96f0d74c6e645fd09

lib\_sha1: 5c9b601998a6b9e2f2c7eee96f0d74c6e645fd09

test\_3: success

custom\_sha1: 747599c50846fcfe29063d3616f4817605e6c4c1

lib\_sha1: 747599c50846fcfe29063d3616f4817605e6c4c1

test\_4: success

custom\_sha1: 5fdf739792e436a0a463ca83a76f18b8b9bb7e5a

lib\_sha1: 5fdf739792e436a0a463ca83a76f18b8b9bb7e5a

test\_5: success

custom\_sha1: 81d7afeec3817aabb5e0f5d5ea0852246b32fff0

lib\_sha1: 81d7afeec3817aabb5e0f5d5ea0852246b32fff0

test\_6: success

custom\_sha1: a648646894c1f5652f48954becba0ef24942f911

lib\_sha1: a648646894c1f5652f48954becba0ef24942f911

test\_7: success

custom\_sha1: 8163868d39d9c8dc19077587b1c3ccd0512d374f

lib\_sha1: 8163868d39d9c8dc19077587b1c3ccd0512d374f

test\_8: success

custom\_sha1: 77062696dbc24449dbb402b604f6af5d4ccad9ec

lib\_sha1: 77062696dbc24449dbb402b604f6af5d4ccad9ec

test\_9: success

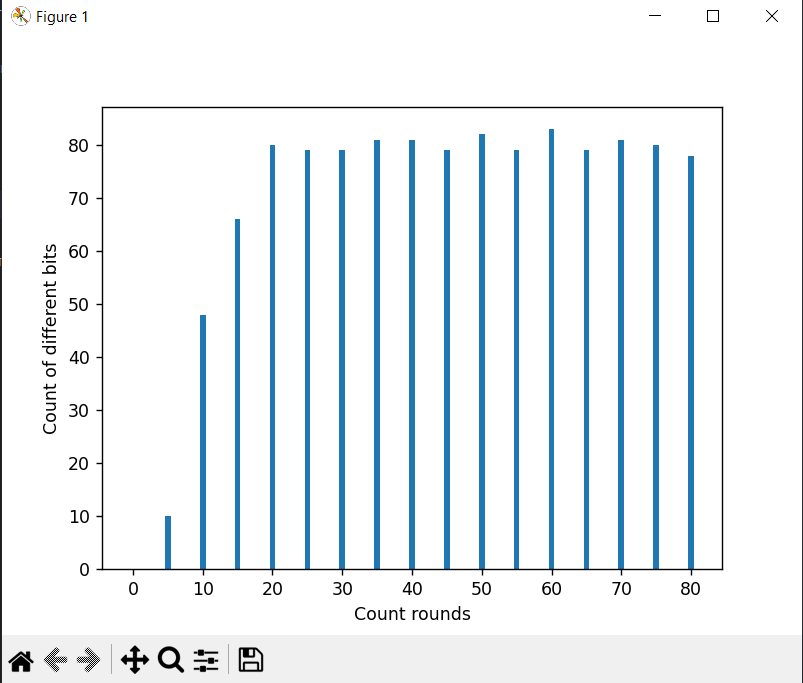
.

----------------------------------------------------------------------

Ran 1 test in 0.065s

OK

**Вывод differential\_cryptanalysis.py**



Такой анализ позволяет заметить, насколько резко меняется хеш при даже небольшом изменении исходного сообщения. Учитывая, что итоговое значение составляет 160 бит, можно отметить, что около половины бит хеша меняется примерно после 15 раундов, что представляет интерес с точки зрения криптографии. Можно предположить, что SHA-1 удовлетворяет критерию лавинного эффекта. Однако, стоит отметить, что результаты этого анализа могут быть не слишком значимыми из-за недостаточного количества проведенных тестов, в целях демонстрации пришлось ограничится 10 тестовыми строками.

Весь код с выводом программ(файл: cryptanalysis.log) так же будет представлен в моем github: <https://github.com/LAshinCHE/crypto_kp>

# **Выводы**

В ходе выполнения курсового проекта я погрузилась в мир криптографических хеш-функций, в частности, в SHA-1. Теперь я могу безопасно и почти без риска коллизий генерировать хеши. SHA-1 - это классика, но существует множество других более надежных, быстрых и устойчивых к атакам хеш-функций. Так же, я познакомился с тем как можно анализировать криптографические хеш-функции и написал код, который проводит дифференциальный криптоанализ для SHA-1.