

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-308Б-21

Студент(ка): А. Ю. Гришин

Преподаватель: А. В. Борисов

Оценка:

Дата: 09.05.2024

Москва, 2024

ОГЛАВЛЕНИЕ

1	Тема	3
2	Задание	3
3	Теория.....	4
	Итерационная модель	4
	Использование блочных симметричных алгоритмов.....	4
	Алгоритм Skein	4
	Алгоритм Threefish	4
	1. Подготовка	5
	2. Дополнение ключа и tweak-значения.....	5
	3. Основной этап	6
	UBI.....	7
4	Ход лабораторной работы	8
	Получение варианта.....	8
	Криптоанализ	9
5	Выводы	11
6	Результаты работы программы	12
7	Список используемой литературы	15
8	Листинг программного кода	15
	skein_test.py.....	15
	threefish_test.py	16
	common.py	16
	skein/skein.py	17
	skein/threefish.py	18

1 Тема

Темой курсовой работы является исследование и анализ алгоритмов функции хеширования в контексте криптографии. Основная цель работы — программная реализация выбранного алгоритма хеширования и его дальнейший анализ с разным числом раундов.

2 Задание

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 16-тиричное число, которое в дальнейшем будет номером варианта.
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.

Как уже было упомянуто ранее, алгоритм Skein реализует итерационную модель хеш-функции с использованием блочного алгоритма. В качестве такого алгоритма был разработан Threefish. Его особенность в

том, что вместо использования S-блоков для замены, он использует операции XOR, сложения по модулю и циклического сдвига.

Алгоритм Threefish предусматривает также задание конфигурационного tweak-значения, которое предназначено для настройки алгоритма путем изменения выходных данных.

В общем виде алгоритм threefish можно представить в виде функции $TF(K, T, A)$, где K — ключ, представляющий слово длиной n бит, T — tweak-значение, представляющее слово длиной 128 бит и A — открытый текст, представляющий слово длиной n бит. Аргумент n может принимать значения 256, 512 и 1024 бит. В результате своей работы алгоритм threefish возвращает слово длиной n бит (так как он является симметричным).

Для увеличения безопасности шифра, на каждом раунде используются разные подключи, а также, после каждого 4-го раунда ключ добавляется к самому состоянию. Последнее также дает алгоритму защиту от методов перебора, обеспечивая более быстрое распространение изменений по всему блоку данных.

Работу алгоритма можно представить в виде следующих этапов

1. Подготовка

В начале своей работы ключ K , tweak-значение T и открытый текст A разбиваются на блоки по 64 бит (собственно, поэтому и значения n должны быть кратны 64):

$$k = (k_1 \ k_2 \ k_3 \ \dots \ k_N), a = (a_1 \ a_2 \ a_3 \ \dots \ a_N), t = (t_1 \ t_2)$$

где $N = \left\lfloor \frac{n}{64} \right\rfloor$.

2. Дополнение ключа и tweak-значения

Далее вектора, полученные путем деления tweak-значения и ключа, дополняются одним элементом:

- $k_{N+1} = C_{240} \oplus k_1 \oplus \dots \oplus k_N$
- $t_3 = t_1 \oplus t_2$

3. Основной этап

После дополнения векторов происходит итерационное выполнение R раундов. Число раундов R зависит от размерности входных данных n и равно 72 для 256 и 412 бит, и 80 - для 1024 бит.

Перед началом выполнения раундов формируется вектор V , представляющий состояние алгоритма на текущем раунде. Вектор V имеет такую же размерность, что ключ и открытый текст, и его начальное значение равно $V = (a_1 \ a_3 \ \dots \ a_N)$

Каждый раунд состоит из следующий шагов:

1. Формируется значение текущего подключа s . Отличие подключа от исходного ключа в том, что оно меняется каждые 4 раунда по следующей формуле

$$s_i = \begin{cases} k_{1+(s+i-1) \bmod (N+1)} & i = \overline{1, N-3} \\ k_{1+(s+i-1) \bmod (N+1)} + t_{1+(s-1) \bmod 3} & i = N-2 \\ k_{1+(s+i-1) \bmod (N+1)} + t_{1+s \bmod 3} & i = N-1 \\ k_{1+(s+i-1) \bmod (N+1)} + s & i = N \end{cases}$$

где $s = \left\lfloor \frac{d}{4} \right\rfloor$, а d - номер текущего раунда

2. Далее формируется вектор $e = f(V, s)$ на основе вектора текущего состояния и подключа. Если номер текущего раунда d не кратен 4-м, то $e = V$, в противном случае $e = V + s$. Иными словами, каждый четвертый раунд происходит добавление подключа к состоянию.
3. Далее, используя функцию MIX , формируется вектор g

$$(g_{2i}, g_{2i+1}) = MIX(e_{2i}, e_{2i+1})$$

4. И в конце, на основе матрицы S_p формируется значение следующего состояния

$$V_i = g_{p(i)}, i = \overline{1, N}$$

UBI

Алгоритм Skein использует Threefish в разновидности режима Matyas-Meyer-Oseas, который называется UBI (Unique Block Iteration).

UBI устроен в виде цепочки блоков, которые мы можем представить в виде функции $UBI(A, B, T)$, где A — начальное слово размера n , B — входной текст произвольного размера и T — tweak-значение, которое содержит информацию о количестве обработанных байт, флагах начала и конца цепочки и т. д.

Назначение каждого блока состоит в сжатии входных данных до определенного размера — n байт. Результирующее значение каждый блок высчитывает итерационно следующим образом

$$\begin{aligned} H_0 &= A \\ H_{i+1} &= TF(H_i, T_i, B_i) \end{aligned}$$

Здесь B_i представляет i -й блок сообщения B , полученный после следующих действий.

Если длина сообщения B не кратна 8-ми, то оно дополняется лидирующей единицей и идущими после нее нулями, а также флагу $F_{Final} = 1$ (в противном случае значение этого флага равно нулю).

Далее сообщение B дополняется нулями до тех пор, пока его длина не будет кратна размеру блока для UBI — n . После чего B разбивается на k блоков: B_1, B_2, \dots, B_k .

Tweak-значение для каждой итерации рассчитывается по следующей формуле

$$T + \min\{m, (i + 1)n\} + \alpha_i \cdot 2^{126} + b_i \cdot (F_{First} \cdot 2^{119} + 2^{127})$$

где α_i, b_i — специальные флаги, ответственные за поля First, Final и BitBand.

4 Ход лабораторной работы

Получение варианта

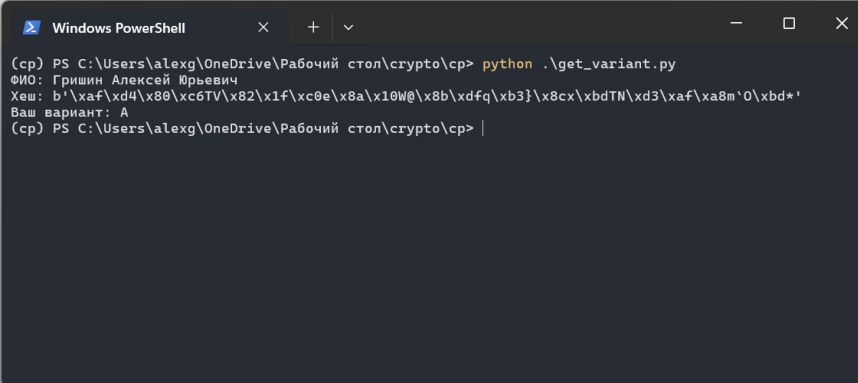
Для получения варианта я воспользовался библиотекой `pygost` на Python. Также, я реализовал небольшую программу, которая по входному ФИО определяет значение хеша и номер варианта по описанному в задании алгоритму.

```
from pygost import gost34112012256

def get_last_bits(b: bytes) -> int:
    return b[-1] & 0b1111

name = input("ФИО: ")
hex_table = '0123456789ABCDEF'
encoded = gost34112012256.new(name.encode()).digest()
variant = get_last_bits(encoded)
print("Хеш:", encoded)
print("Ваш вариант:", hex_table[variant])
```

После запуска программы и ввода своего ФИО я получил следующий результат



```
(cp) PS C:\Users\alevg\OneDrive\Рабочий стол\crypto\cp> python .\get_variant.py
ФИО: Гришин Алексей Юрьевич
Хеш: b'\xaf\xd4\x80\x82\x1f\xce\x8a\x10w\x8b\xdfq\xb3}\x8cx\xbdTN\xd3\xaf\xa8m'0\xbd*'
Ваш вариант: A
(cp) PS C:\Users\alevg\OneDrive\Рабочий стол\crypto\cp> |
```

Собственно, мой вариант оказался вариант «А» - алгоритм Skein

При выполнении лабораторной работы я взял за основу для своей программной реализации библиотеку Python gressify, которая является в свою очередь версией библиотеки ryskein, переписанной на чистый Python.

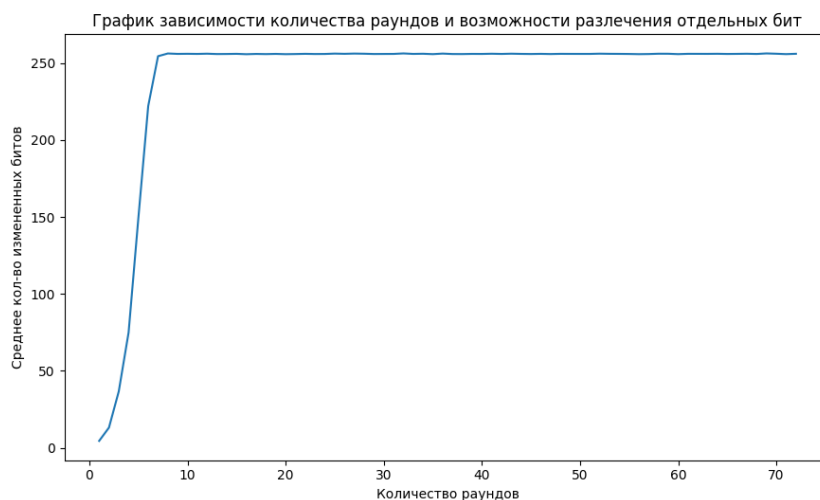
Конечно, такая реализация значительно проигрывает по скорости работы библиотеке ryskein, однако тот факт, что исходный код написан на Python, позволило облегчить процесс интеграции настройки количества раундов в исходный код.

Далее, я упростил код, оставив только реализацию с 512-битным представлением состояния внутри. Дальнейший криптоанализ проводился конкретно с этой версией алгоритма Skein.

Криптоанализ

Перед началом анализа алгоритма я учел, что Skein использует режим UBI, который также вносит свой вклад в лавинный эффект. Поэтому я решил сначала провести анализ алгоритма Threefish отдельно. В спецификации алгоритма Threefish было заявлено, что для крипто стойкости алгоритма достаточно 72 итераций для входных сообщений размера 256 и 512 бит, и 80 итераций для размера в 1024 бит. Поэтому я решил провести анализ алгоритм с варьированием количества раундов от 1 до 72-х. На каждой итерации я генерировал 100 случайных строк длины 512 бит. Далее, я генерировал 512 строк, представляющих исходную строку с одним измененным битом. Для каждой пары строк я высчитывал значение алгоритма Threefish с идентичным значением ключа. После обработки всех строк происходил

расчёт среднего количества измененных битов во входном сообщении. Ниже представлен график изменения этого значения в зависимости от количества раундов

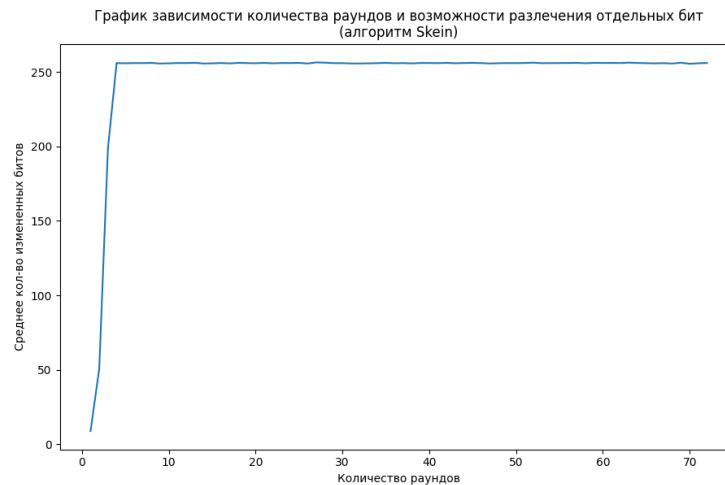


Данный график демонстрирует, как меняет хеш при минимальном изменении исходного сообщения. Учитывая, что размер выходного сообщения равен 512 битам, мы можем увидеть, что, начиная с 10 раундов, во выходном сообщении изменяется половина битов, что говорит нам о ярко выраженном лавинном эффекте. Конечно, такой подход не дает полную информацию о криптостойкости алгоритма, однако мы можем утверждать, что криптоаналитику, при таком количестве раундов, уже будет сложно восстановить входное сообщение, имея лишь информацию только о выходном.

Проведем теперь анализ самого алгоритма Skein. Для анализа я использовал те же идеи, что и при анализе алгоритма Threefish:

- варьировалось количество раундов от 1 до 72-х с шагом 1
- для каждого количества раундов генерировалось по 100 сообщений с варьируемой длиной от 200 до 500 битов
- для каждой строки генерировались парные строки путем изменения одного бита

- для каждой пары сообщений высчитывалось значение хеша, после чего производился расчет количества отличающихся битов
- после обработки всех строк высчитывалось среднее количество отличающихся битов.



Как мы можем увидеть, минимальное количество раундов, которое потребовалось для изменения половины битов в выходном сообщении, уже немного меньше по сравнению с тем, которое потребовалось для алгоритма Threefish. Связано это с тем, что алгоритм Skein помимо использования Threefish, использует UBI, состоящую из 3-х блоков, что также оказывает влияние на лавинный эффект.

5 Выводы

В ходе выполнения курсовой работы я осуществил программную реализацию криптографического алгоритма хеширования Skein. Алгоритм Skein оказался довольно интересным объектом для рассмотрения, так как его структура отличается от привычных структур других алгоритмов, но в то же время является лаконичной и простой.

Также, я провел криптоанализ алгоритма, выявив закономерность между количеством изменяющихся битов в выходном сообщении с количеством раундов. Я на практике убедился в важности большого

количества раундов, так как это как минимум обеспечивает лавинный эффект.

6 Результаты работы программы

```
(.venv) PS D:\Desktop\crypto\prog2> pypy .\skein_test.py
```

```
rounds: 1, iteration: 100 / 100 mean: 8.85675
rounds: 2, iteration: 100 / 100 mean: 50.7095
rounds: 3, iteration: 100 / 100 mean: 198.563
rounds: 4, iteration: 100 / 100 mean: 256.0295
rounds: 5, iteration: 100 / 100 mean: 255.8925
rounds: 6, iteration: 100 / 100 mean: 256.037
rounds: 7, iteration: 100 / 100 mean: 256.0235
rounds: 8, iteration: 100 / 100 mean: 256.13725
rounds: 9, iteration: 100 / 100 mean: 255.72925
rounds: 10, iteration: 100 / 100 mean: 255.823
rounds: 11, iteration: 100 / 100 mean: 256.0675
rounds: 12, iteration: 100 / 100 mean: 256.0385
rounds: 13, iteration: 100 / 100 mean: 256.2095
rounds: 14, iteration: 100 / 100 mean: 255.73
rounds: 15, iteration: 100 / 100 mean: 255.83275
rounds: 16, iteration: 100 / 100 mean: 256.076
rounds: 17, iteration: 100 / 100 mean: 255.81025
rounds: 18, iteration: 100 / 100 mean: 256.1815
rounds: 19, iteration: 100 / 100 mean: 256.00425
rounds: 20, iteration: 100 / 100 mean: 255.926
rounds: 21, iteration: 100 / 100 mean: 256.15825
rounds: 22, iteration: 100 / 100 mean: 255.8695
rounds: 23, iteration: 100 / 100 mean: 256.08075
rounds: 24, iteration: 100 / 100 mean: 256.0395
rounds: 25, iteration: 100 / 100 mean: 256.1505
rounds: 26, iteration: 100 / 100 mean: 255.71925
rounds: 27, iteration: 100 / 100 mean: 256.45775
rounds: 28, iteration: 100 / 100 mean: 256.29825
rounds: 29, iteration: 100 / 100 mean: 255.9855
rounds: 30, iteration: 100 / 100 mean: 255.965
rounds: 31, iteration: 100 / 100 mean: 255.7805
rounds: 32, iteration: 100 / 100 mean: 255.74625
rounds: 33, iteration: 100 / 100 mean: 255.8435
rounds: 34, iteration: 100 / 100 mean: 255.969
rounds: 35, iteration: 100 / 100 mean: 256.177
rounds: 36, iteration: 100 / 100 mean: 255.9195
rounds: 37, iteration: 100 / 100 mean: 256.02875
rounds: 38, iteration: 100 / 100 mean: 255.814
rounds: 39, iteration: 100 / 100 mean: 256.0605
rounds: 40, iteration: 100 / 100 mean: 256.067
```

```
rounds: 41, iteration: 100 / 100 mean: 255.9925
rounds: 42, iteration: 100 / 100 mean: 256.1845
rounds: 43, iteration: 100 / 100 mean: 255.9015
rounds: 44, iteration: 100 / 100 mean: 256.0585
rounds: 45, iteration: 100 / 100 mean: 256.15425
rounds: 46, iteration: 100 / 100 mean: 256.02125
rounds: 47, iteration: 100 / 100 mean: 255.7395
rounds: 48, iteration: 100 / 100 mean: 255.9325
rounds: 49, iteration: 100 / 100 mean: 256.0325
rounds: 50, iteration: 100 / 100 mean: 255.989
rounds: 51, iteration: 100 / 100 mean: 256.10075
rounds: 52, iteration: 100 / 100 mean: 256.28925
rounds: 53, iteration: 100 / 100 mean: 255.9475
rounds: 54, iteration: 100 / 100 mean: 256.021
rounds: 55, iteration: 100 / 100 mean: 256.0385
rounds: 56, iteration: 100 / 100 mean: 256.08825
rounds: 57, iteration: 100 / 100 mean: 256.16725
rounds: 58, iteration: 100 / 100 mean: 255.90025
rounds: 59, iteration: 100 / 100 mean: 256.17725
rounds: 60, iteration: 100 / 100 mean: 256.129
rounds: 61, iteration: 100 / 100 mean: 256.13475
rounds: 62, iteration: 100 / 100 mean: 256.047
rounds: 63, iteration: 100 / 100 mean: 256.32375
rounds: 64, iteration: 100 / 100 mean: 256.081
rounds: 65, iteration: 100 / 100 mean: 255.96575
rounds: 66, iteration: 100 / 100 mean: 255.82625
rounds: 67, iteration: 100 / 100 mean: 255.97625
rounds: 68, iteration: 100 / 100 mean: 255.70075
rounds: 69, iteration: 100 / 100 mean: 256.2835
rounds: 70, iteration: 100 / 100 mean: 255.55425
rounds: 71, iteration: 100 / 100 mean: 255.88925
rounds: 72, iteration: 100 / 100 mean: 256.09775
```

```
(.venv) PS D:\Desktop\crypto\prog2> pypy .\threefish_test.py
```

```
rounds: 1, iteration: 100 / 100 mean: 4.4925
rounds: 2, iteration: 100 / 100 mean: 13.1275
rounds: 3, iteration: 100 / 100 mean: 36.688359375
rounds: 4, iteration: 100 / 100 mean: 74.934453125
rounds: 5, iteration: 100 / 100 mean: 149.6203125
rounds: 6, iteration: 100 / 100 mean: 222.127265625
rounds: 7, iteration: 100 / 100 mean: 254.504453125
rounds: 8, iteration: 100 / 100 mean: 256.2684375
rounds: 9, iteration: 100 / 100 mean: 256.004375
rounds: 10, iteration: 100 / 100 mean: 256.0684375
rounds: 11, iteration: 100 / 100 mean: 255.9890625
rounds: 12, iteration: 100 / 100 mean: 256.130625
rounds: 13, iteration: 100 / 100 mean: 255.909765625
rounds: 14, iteration: 100 / 100 mean: 255.947890625
rounds: 15, iteration: 100 / 100 mean: 256.04328125
rounds: 16, iteration: 100 / 100 mean: 255.79703125
rounds: 17, iteration: 100 / 100 mean: 255.977734375
rounds: 18, iteration: 100 / 100 mean: 255.871796875
```

rounds: 19, iteration: 100 / 100 mean: 256.00234375
rounds: 20, iteration: 100 / 100 mean: 255.822578125
rounds: 21, iteration: 100 / 100 mean: 255.914609375
rounds: 22, iteration: 100 / 100 mean: 256.046171875
rounds: 23, iteration: 100 / 100 mean: 255.92875
rounds: 24, iteration: 100 / 100 mean: 255.962421875
rounds: 25, iteration: 100 / 100 mean: 256.192421875
rounds: 26, iteration: 100 / 100 mean: 256.05125
rounds: 27, iteration: 100 / 100 mean: 256.20765625
rounds: 28, iteration: 100 / 100 mean: 256.111171875
rounds: 29, iteration: 100 / 100 mean: 255.93625
rounds: 30, iteration: 100 / 100 mean: 255.967578125
rounds: 31, iteration: 100 / 100 mean: 255.991640625
rounds: 32, iteration: 100 / 100 mean: 256.30265625
rounds: 33, iteration: 100 / 100 mean: 255.995703125
rounds: 34, iteration: 100 / 100 mean: 256.097421875
rounds: 35, iteration: 100 / 100 mean: 255.84421875
rounds: 36, iteration: 100 / 100 mean: 256.20203125
rounds: 37, iteration: 100 / 100 mean: 255.923046875
rounds: 38, iteration: 100 / 100 mean: 255.89484375
rounds: 39, iteration: 100 / 100 mean: 256.0059375
rounds: 40, iteration: 100 / 100 mean: 255.975859375
rounds: 41, iteration: 100 / 100 mean: 256.107109375
rounds: 42, iteration: 100 / 100 mean: 255.980703125
rounds: 43, iteration: 100 / 100 mean: 256.133515625
rounds: 44, iteration: 100 / 100 mean: 256.00875
rounds: 45, iteration: 100 / 100 mean: 255.9321875
rounds: 46, iteration: 100 / 100 mean: 256.0309375
rounds: 47, iteration: 100 / 100 mean: 255.9109375
rounds: 48, iteration: 100 / 100 mean: 256.0384375
rounds: 49, iteration: 100 / 100 mean: 256.01234375
rounds: 50, iteration: 100 / 100 mean: 256.002265625
rounds: 51, iteration: 100 / 100 mean: 255.997578125
rounds: 52, iteration: 100 / 100 mean: 256.144921875
rounds: 53, iteration: 100 / 100 mean: 256.052734375
rounds: 54, iteration: 100 / 100 mean: 256.018203125
rounds: 55, iteration: 100 / 100 mean: 255.955234375
rounds: 56, iteration: 100 / 100 mean: 255.837421875
rounds: 57, iteration: 100 / 100 mean: 255.91421875
rounds: 58, iteration: 100 / 100 mean: 256.111640625
rounds: 59, iteration: 100 / 100 mean: 256.09328125
rounds: 60, iteration: 100 / 100 mean: 255.850234375
rounds: 61, iteration: 100 / 100 mean: 256.051171875
rounds: 62, iteration: 100 / 100 mean: 256.037578125
rounds: 63, iteration: 100 / 100 mean: 256.0215625
rounds: 64, iteration: 100 / 100 mean: 256.075703125
rounds: 65, iteration: 100 / 100 mean: 255.961796875
rounds: 66, iteration: 100 / 100 mean: 256.017578125
rounds: 67, iteration: 100 / 100 mean: 256.100078125
rounds: 68, iteration: 100 / 100 mean: 255.96015625
rounds: 69, iteration: 100 / 100 mean: 256.304921875
rounds: 70, iteration: 100 / 100 mean: 256.1046875

```
rounds: 71, iteration: 100 / 100 mean: 255.853203125
rounds: 72, iteration: 100 / 100 mean: 256.071875
```

7 Список используемой литературы

- Обзорная статья про алгоритм Skein на Habr: <https://habr.com/ru/articles/531140/>
- Описание алгоритма Skein: <https://www.pgpru.com/novosti/2008/heshfunkcijaskkeiniblochnyjshifrtthreefish>
- Официальный сайт алгоритма Skein: <http://www.skein-hash.info/>
- Статья Брюса Шнайера про алгоритм Skein: <https://www.schneier.com/skein.html>

8 Листинг программного кода

skein_test.py

```
from skein import Skein512
from common import random_bytes_string, invert_bit, diff
import json

DATASET_SIZE = 100
ROUNDS = range(1, 73)
MESSAGE_SIZE = 105

def encrypt(msg: bytes, rounds: int) -> bytes:
    sk = Skein512(rounds=rounds, msg=msg)
    return sk.digest()

def main():
    data = []

    for rounds in ROUNDS:
        result = []

        for t in range(DATASET_SIZE):
            print(f'\rrounds: {rounds}, iteration: {t + 1} / {DATASET_SIZE}', end='')
            a = random_bytes_string(MESSAGE_SIZE)

            for i in range(0, len(a) * 8, 4):
                b = invert_bit(a, i)
                hash1 = encrypt(a, rounds)
                hash2 = encrypt(b, rounds)
                result.append(diff(hash1, hash2))

        mean = sum(result) / len(result)
        print(f" mean: {mean}")
        data.append({"rounds": rounds, "bits_changed": mean})

    with open("skein.json", "w") as file:
        file.write(json.dumps(data, indent=4))
```

```
if __name__ == "__main__":
    main()
```

threefish_test.py

```
from common import random_bytes_string, invert_bit, diff
import json
from skein import Threefish512, bytes2words, words2bytes

DATASET_SIZE = 100
ROUNDS = range(1, 73)

def encrypt(msg: bytes, rounds: int) -> bytes:
    th = Threefish512(rounds=rounds)
    th.prepare_key()
    th.prepare_tweak()
    return words2bytes(th.encrypt_block(bytes2words(msg)))

def main():
    data = []

    for rounds in ROUNDS:
        result = []

        for t in range(DATASET_SIZE):
            print(f'\rrounds: {rounds}, iteration: {t + 1} / {DATASET_SIZE}', end='')
            a = random_bytes_string(64)

            for i in range(0, len(a) * 8, 4):
                b = invert_bit(a, i)
                hash1 = encrypt(a, rounds)
                hash2 = encrypt(b, rounds)
                result.append(diff(hash1, hash2))

            mean = sum(result) / len(result)
            print(f" mean: {mean}")
            data.append({"rounds": rounds, "bits_changed": mean})

        with open("threefish.json", "w") as file:
            file.write(json.dumps(data, indent=4))

if __name__ == "__main__":
    main()
```

common.py

```
from random import randint

def random_bytes_string(n: int) -> bytes:
    return bytes([randint(0, 255) for _ in range(n)])

def invert_bit(a: bytes, i: int) -> bytes:
    b = list(a)
```



```

    b[i // 8] ^= 1 << (i % 8)
    return bytes(b)

def diff(a: bytes, b: bytes) -> int:
    assert len(a) == len(b)
    diff_bits = 0

    for x, y in zip(a, b):
        d = x ^ y
        while d:
            diff_bits += d & 1
            d >>= 1

    return diff_bits

def to_bits(a: bytes) -> str:
    return "".join(bin(x)[2:].zfill(8) for x in a)

```

skein/skein.py

```

import array
import binascii
import os

from .threefish import (add64, bigint, bytes2words, Threefish512, words,
                        words2bytes, words_format, xrange,
                        zero_bytes, zero_words)

empty_bytes = array.array('B').tobytes()

class Skein512(object):
    block_size = 64
    block_bits = 512
    block_type = {'key': 0,
                  'nonce': 0x5400000000000000,
                  'msg': 0x7000000000000000,
                  'cfg_final': 0xc400000000000000,
                  'out_final': 0xff00000000000000}

    def __init__(self, rounds: int, msg="", digest_bits=512, key=None, block_type='msg'):
        self.tf = Threefish512(rounds=rounds)
        if key:
            self.digest_bits = 512
            self._start_new_type('key')
            self.update(key)
            self.tf.key = bytes2words(self.final(False))
        self.digest_bits = digest_bits
        self.digest_size = (digest_bits + 7) >> 3
        self._start_new_type('cfg_final')
        b = words2bytes((0x133414853, digest_bits, 0, 0, 0, 0, 0))
        self._process_block(b, 32)
        self._start_new_type(block_type)
        if msg:
            self.update(msg)

    def _start_new_type(self, block_type):
        self.buf = empty_bytes

```

```

self.tf.tweak = words([0, self.block_type[block_type]])

def _process_block(self, block, byte_count_add):
    block_len = len(block)
    for i in xrange(0, block_len, 64):
        w = bytes2words(block[i:i+64])
        self.tf.tweak[0] = add64(self.tf.tweak[0], byte_count_add)
        self.tf.prepare_tweak()
        self.tf.prepare_key()
        self.tf.key = self.tf.encrypt_block(w)
        self.tf._feed_forward(self.tf.key, w)
        # set second tweak value to ~SKEIN_T1_FLAG_FIRST:
        self.tf.tweak[1] ^= bigint(0xbfffffffffffffff)

def update(self, msg):
    self.buf += msg
    buflen = len(self.buf)
    if buflen > 64:
        end = -(buflen % 64) or (buflen-64)
        data = self.buf[0:end]
        self.buf = self.buf[end:]
        try:
            self._process_block(data, 64)
        except:
            print(len(data))
            print(binascii.b2a_hex(data))

def final(self, output=True):
    self.tf.tweak[1] |= bigint(0x8000000000000000) # SKEIN_T1_FLAG_FINAL
    buflen = len(self.buf)
    self.buf += zero_bytes[:64-buflen]

    self._process_block(self.buf, buflen)

    if not output:
        hash_val = words2bytes(self.tf.key)
    else:
        hash_val = empty_bytes
        self.buf = zero_bytes[:]
        key = self.tf.key[:] # temporary copy
        i=0
        while i*64 < self.digest_size:
            self.buf = words_format[1].pack(i) + self.buf[8:]
            self.tf.tweak = [0, self.block_type['out_final']]
            self._process_block(self.buf, 8)
            n = self.digest_size - i*64
            if n >= 64:
                n = 64
            hash_val += words2bytes(self.tf.key)[0:n]
            self.tf.key = key
            i+=1
        return hash_val

digest = final

```

skein/threefish.py

```

ROT = bytelist((46, 36, 19, 37,
                 33, 27, 14, 42,

```

```

        17, 49, 36, 39,
        44, 9, 54, 56,
        39, 30, 34, 24,
        13, 50, 10, 17,
        25, 29, 39, 43,
        8, 35, 56, 22))

PERM = bytelist(((0,1),(2,3),(4,5),(6,7),
                 (2,1),(4,7),(6,5),(0,3),
                 (4,1),(6,3),(0,5),(2,7),
                 (6,1),(0,7),(2,5),(4,3)))

class Threefish512(object):
    def __init__(self, rounds: int, key: bytes | None = None, tweak: bytes | None = None):
        self._rounds = rounds

        if key:
            self.key = bytes2words(key)
            self.prepare_key()

        else:
            self.key = words(zero_words[:] + [0])

        if tweak:
            self.tweak = bytes2words(tweak, 2)
            self.prepare_tweak()

        else:
            self.tweak = zero_words[:3]

    def prepare_key(self) -> None:
        final = reduce(xor, self.key[:8]) ^ SKEIN_KS_PARITY

        try:
            self.key[8] = final

        except IndexError:
            self.key = words(list(self.key) + [final])

    def prepare_tweak(self) -> None:
        final = self.tweak[0] ^ self.tweak[1]

        try:
            self.tweak[2] = final

        except IndexError:
            self.tweak = words(list(self.tweak) + [final])

    def encrypt_block(self, plaintext: bytes):
        key = self.key
        tweak = self.tweak
        state = words(list(imap(add64, plaintext, key[:8])))
        state[5] = add64(state[5], tweak[0])
        state[6] = add64(state[6], tweak[1])

        for round in range(1, self._rounds + 1):
            r = (round - 1) // 4 + 1
            s = [16, 0][r % 2]
            t = ((round - 1) % 4) * 4

            for i in range(t, t + 4):

```

```

        m, n = PERM[i]
        state[m] = add64(state[m], state[n])
        state[n] = RotL_64(state[n], ROT[i+s])
        state[n] = state[n] ^ state[m]

    if round % 4 == 0:
        for y in range(8):
            state[y] = add64(state[y], key[(r + y) % 9])
            state[5] = add64(state[5], tweak[r % 3])
            state[6] = add64(state[6], tweak[(r+1) % 3])
            state[7] = add64(state[7], r)

    return state

def _feed_forward(self, state, plaintext):
    state[:] = list(map(xor, state, plaintext))

```