**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](#_Toc166253634)

[2 Задание 3](#_Toc166253635)

[3 Теория 4](#_Toc166253636)

[Итерационная модель 4](#_Toc166253637)

[Использование блочных симметричных алгоритмов 4](#_Toc166253638)

[Алгоритм Skein 5](#_Toc166253639)

[Алгоритм Threefish 5](#_Toc166253640)

[1. Подготовка 6](#_Toc166253641)

[2. Дополнение ключа и tweak-значения 6](#_Toc166253642)

[3. Основной этап 6](#_Toc166253643)

[UBI 7](#_Toc166253644)

[4 Ход лабораторной работы 8](#_Toc166253645)

[Получение варианта 8](#_Toc166253646)

[Криптоанализ 10](#_Toc166253647)

[5 Выводы 12](#_Toc166253648)

[6 Результаты работы программы 12](#_Toc166253649)

[7 Список используемой литературы 15](#_Toc166253650)

[8 Листинг программного кода 15](#_Toc166253651)

[skein\_test.py 15](#_Toc166253652)

[threefish\_test.py 16](#_Toc166253653)

[common.py 17](#_Toc166253654)

[skein/skein.py 17](#_Toc166253655)

[skein/threefish.py 19](#_Toc166253656)

# **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.

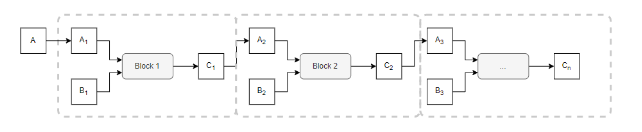
# **Задание**

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.

# **Теория**

## **Итерационная модель**

Одной из моделей, по которой строятся хеш-функции, является итерационная. Как видно из названия, она заключается в итерационном способе вычисления хеша. При такой модели алгоритм хеширования состоит из определенного количества одинаковых блоков, на вход которым подается пара из двух текстов , а на выходе получается текст , который передается на вход следующему блоку и так далее.



Касаемо размера блоков, стоит отметить, что каждый блок принимает на вход текст длины , а на выходе выдает блок длины . Блок играет роль константы и подается на вход первому блоку, остальные же блоки представляют результат работы блоков на прошлой итерации и имеют размер .

Блоки имеют размер и получаются в результате расширения входного сообщения , имеющего длину , до длины, кратной , и последующего деления.

## **Использование блочных симметричных алгоритмов**

В качестве блоков можно использовать и блочные симметричные алгоритмы (как мы увидим далее в алгоритме Skein). При таком подходе текущий блок входного сообщения целесообразно подавать в качестве ключа алгоритму, а - в качестве входного сообщения. Таким образом, стойкость полученной модели зависит от выбора блочного алгоритма.

## **Алгоритм Skein**

Алгоритм Skein был разработан группой авторов под руководством Брюса Шнайера, который был основателем алгоритмов Blowfish, twofish. Этот алгоритм был финалистом в конкурсе NIST, но уступил алгоритму Keccak.

При разработке алгоритма авторы придерживались следующих идей:

* использовать как можно меньше памяти
* устойчивость к атакам
* простота реализации
* оптимизация под 64-разрядные процессоры

## **Алгоритм Threefish**

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 , где — ключ, представляющий слово длиной бит, — tweak-значение, представляющее слово длиной 128 бит и — открытый текст, представляющий слово длиной бит. Аргумент может принимать значения 256, 512 и 1024 бит. В результате своей работы алгоритм threefish возвращает слово длиной 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 , tweak-значение и открытый текст разбиваются на блоки по 64 бит (собственно, поэтому и значения должны быть кратны 64):

где .

### **2. Дополнение ключа и tweak-значения**

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

### **3. Основной этап**

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

Перед началом выполнения раундов формируется вектор , представляющий состояние алгоритма на текущем раунде. Вектор имеет такую же размерность, что ключ и открытый текст, и его начальное значение равно

Каждый раунд состоит из следующий шагов:

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

где , а - номер текущего раунда

1. Далее формируется вектор на основе вектора текущего состояния и подключа. Если номер текущего раунда не кратен 4-м, то , в противном случае . Иными словами, каждый четвертый раунд происходит добавление подключа к состоянию.
2. Далее, используя функцию , формируется вектор
3. И в конце, на основе матрицы $p$ формируется значение следующего состояния

## **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 , где — начальное слово размера , — входной текст произвольного размера и — 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 — байт. Результирующее значение каждый блок высчитывает итерационно следующим образом

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

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

Далее сообщение дополняется нулями до тех пор, пока его длина не будет кратна размеру блока для UBI — . После чего разбивается на блоков: .

Tweak-значение для каждой итерации рассчитывается по следующей формуле

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

# **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'

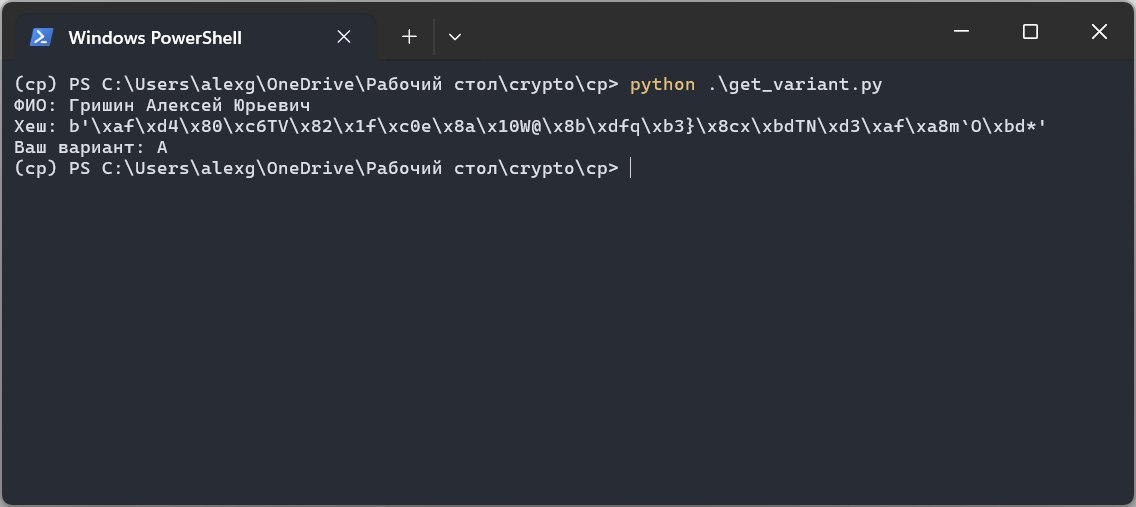
encoded = gost34112012256.new(name.encode()).digest()

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



Собственно, мои вариантом оказался вариант «А» - алгоритм Skein

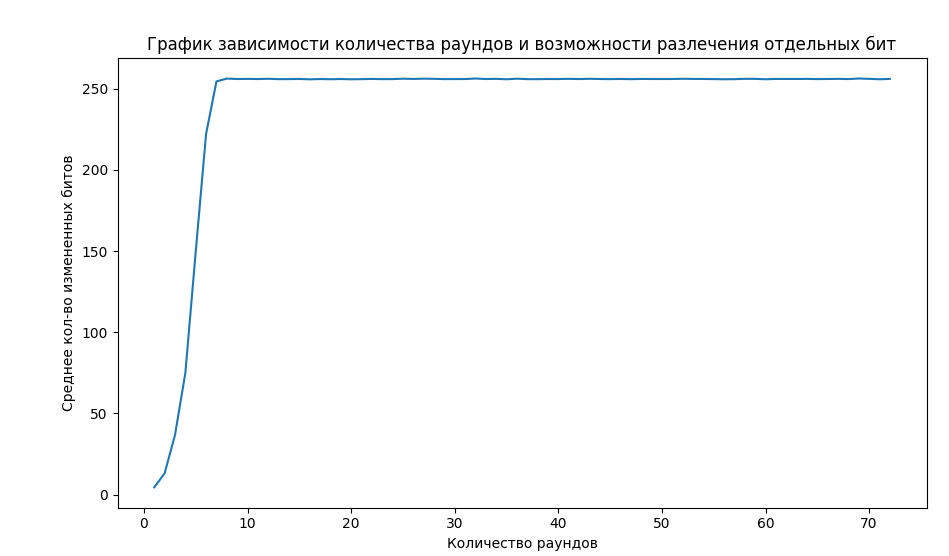
При выполнении лабораторной работы я взял за основу для своей программной реализации библиотеку Python gressify, которая является в свою очередь версией библиотеки pyskein, переписанной на чистый Python.

Конечно, такая реализация значительно проигрывает по скорости работы библиотеке pyskein, 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**

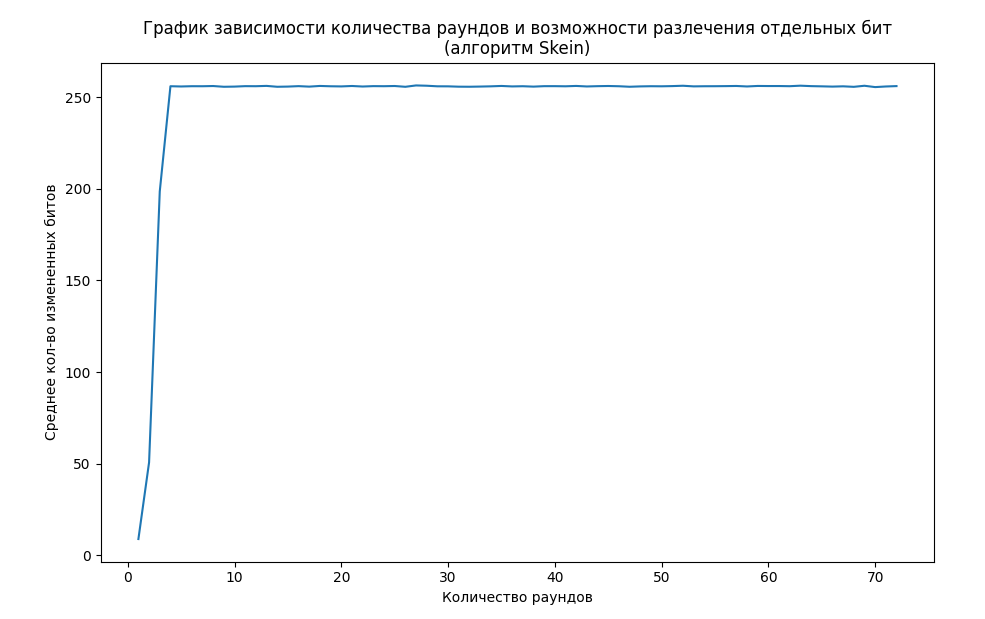
Перед началом анализа алгоритма я учел, что Skein использует режим UBI, который также вносит свой вклад в лавинный эффект. Поэтому я решил сначала провести анализ алгоритма Threefish отдельно. В спецификации алгоритма Threefish было заявлено, что для крипто стойкости алгоритма достаточно 72 итераций 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.

# **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.

# **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

# **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/heshfunkcijaskeiniblochnyjjshifrthreefish>
* Официальный сайт алгоритма Skein: <http://www.skein-hash.info/>
* Статья Брюса Шнайера про алгоритм Skein: <https://www.schneier.com/skein.html>

# **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,0,0,0,0,0,0))

        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))