**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»**

Лабораторная работа № 2  
по курсу «Криптография»

Группа: М8О-312Б-22

Студент(ка): Л. Д, Андрюшин

Преподаватель: А. В. Борисов

Оценка:

Дата: 13.04.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[1 Тема 3](#_Toc158983147)

[2 Задание 3](#_Toc158983148)

[3 Теория 4](#_Toc158983149)

[4 Ход лабораторной работы 6](#_Toc158983150)

[5 Выводы 11](#_Toc158983151)

# **Тема**

Лабораторная работа №2. Работая с хэш-функцией и факторизацией

# **Задание**

Строку в которой записано своё ФИО подать на вход в качестве аргумента хеш-функции ГОСТ Р 34.11-2012 (Стрибог). Младшие 8 бит выхода интерпретировать как число, которое в дальнейшем будет номером варианта от 0 до 255. В отчёт включить снимок экрана с выбором номера варианта, а также описать шаги решения задачи. Задача: разложить каждое из чисел `a` и `b` на нетривиальные сомножители. Ниже представлены 256 вариантов.

# **Теория**

Хеш-функция Стрибог, известная также под обозначением ГОСТ Р 34.11-2012, представляет собой современный отечественный криптографический алгоритм хеширования, предназначенный для формирования компактного и уникального представления входных данных. Алгоритм был разработан для замены устаревшего ГОСТ Р 34.11-94 и официально утверждён как стандарт в Российской Федерации. В его разработке особое внимание уделялось устойчивости к различным видам криптографического анализа, в том числе дифференциальному и линейному, а также к атакам на коллизии и прообразы.

Стрибог работает с входными сообщениями произвольной длины и на выходе формирует хеш фиксированной длины — либо 256 бит, либо 512 бит, в зависимости от выбранного варианта. Алгоритм построен по принципу так называемой компрессионной функции, использующей блочный шифр и обрабатывающей сообщение по частям. Каждый блок данных имеет длину 512 бит, что соответствует 64 байтам. Перед обработкой все сообщение дополняется по определённому правилу: добавляется бит единицы, затем нули до кратности 512, и в конце — длина исходного сообщения в битах. Такое дополнение гарантирует однозначную и корректную обработку любых входных данных.

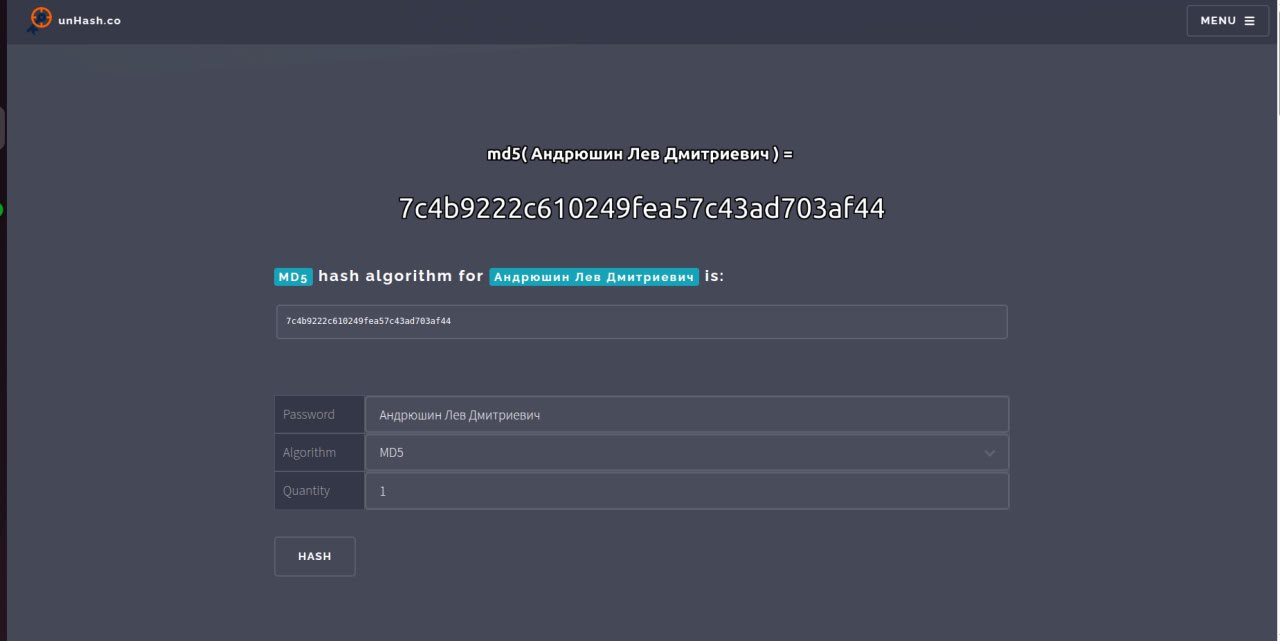
На начальном этапе алгоритм инициализирует внутреннее состояние. В зависимости от того, требуется ли получить хеш длиной 256 или 512 бит, выбирается разное начальное значение. Это внутреннее состояние играет роль как бы стартовой точки, от которой начинается пошаговое преобразование входных данных. Затем каждый блок сообщения последовательно обрабатывается. При этом выполняется ряд преобразований: сначала применяется так называемое S-преобразование (или S-box), заменяющее каждый байт на другой по заранее заданной таблице подстановок, затем перестановка байтов (P-преобразование), и, наконец, линейное преобразование (L-преобразование), которое обеспечивает сильную связность и перемешивание информации между различными битами блока.

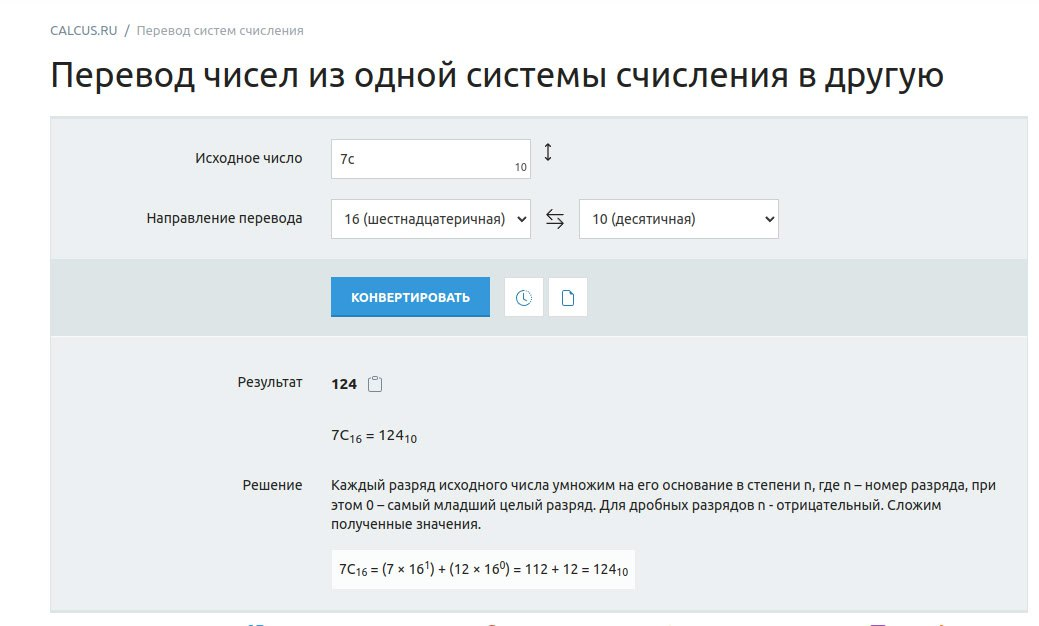
Цепочка этих преобразований повторяется на каждом шаге обработки каждого блока сообщения. Благодаря этому обеспечивается высокая степень диффузии: любое минимальное изменение во входных данных приводит к совершенно другому хешу, что соответствует важнейшему требованию криптографических хеш-функций — эффекту лавины.

В отличие от многих зарубежных алгоритмов, таких как SHA-2 или SHA-3, Стрибог был создан с нуля и не является модификацией уже существующих решений. Это делает его особенно интересным для изучения, так как он опирается на уникальные принципы построения. Одним из преимуществ алгоритма является его эффективность и возможность аппаратной реализации, что делает его пригодным для использования в смарт-картах, криптографических токенах и других устройствах с ограниченными вычислительными ресурсами.

# **Ход лабораторной работы**

В начале лабораторной работы я определили свой вариант:





Мой вариант:

a[124]=72757483262684987131898580874446707971478300749640534583584864087274478493863

b[124]=32317006071311007300714876688669951960444102669715484032130345427524655138867890893197201411522913463688717960921898019494119559150490921095088153164924225020603615488430913119461968461022585082547691599645445077738968377465019316678641058866433970409858878566293638797251481898277228159274965524429317908080083903023339216881948915346620403152836793109257796649966895095535308356847014896197607809000924088956968626483660214506119879472028818001626534905609907095163270742695728880509218944157788436400353752892829712638035054446033265131554461141003657055757762286066943695370592144742305800429861934725638537744659

В начале я написал простой код на питоне, который должен был найти сомножители чисел a и b:

from sympy import isprime, factorint

a = 72757483262684987131898580874446707971478300749640534583584864087274478493863

b = 32317006071311007300714876688669951960444102669715484032130345427524655138867890893197201411522913463688717960921898019494119559150490921095088153164924225020603615488430913119461968461022585082547691599645445077738968377465019316678641058866433970409858878566293638797251481898277228159274965524429317908080083903023339216881948915346620403152836793109257796649966895095535308356847014896197607809000924088956968626483660214506119879472028818001626534905609907095163270742695728880509218944157788436400353752892829712638035054446033265131554461141003657055757762286066943695370592144742305800429861934725638537744659

print("Факторизация a:")

print(factorint(a))

print("\nФакторизация b:")

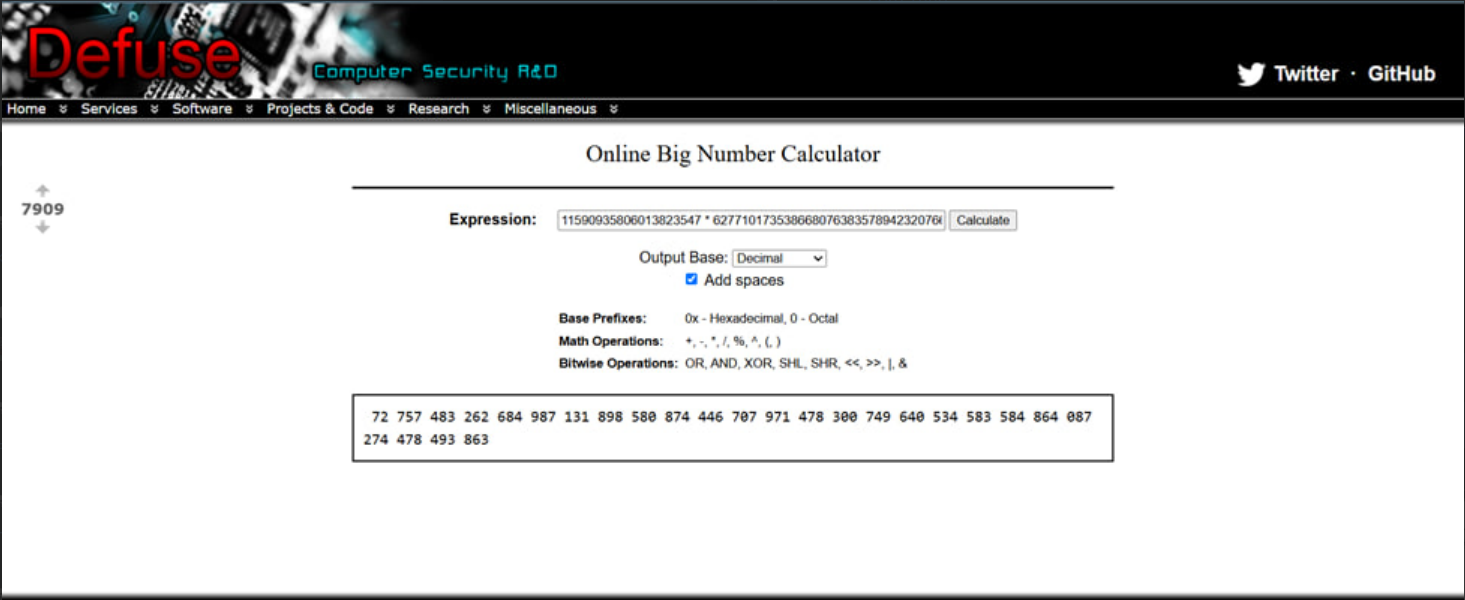
print(factorint(b))

Данный код легко нашел сомножители числа a:

Факторизация a:

{11590935806013823547: 1, 6277101735386680763835789423207666416102355444464034513029: 1}

Вот проверка, показывающая что код действительно нашел правильные сомножители:



При попытки посчитать сомножители b код зависает. Число b слишком больше для подбора сомножителей наивным способом, поэтому нужно пробовать решить данную задачу иначе. В pdf файле с лабораторной работой представлено аж 256 чисел b. Я предположил, что они сгенерированы по одному принципу. Тогда я написал код, который открывает наш pdf файл, считывает оттуда все числа b[i], и для каждого из них считает НОД с моим b[124]. Как только НОД оказывался больше единицы - это означало, что и моё число, и другое число из файла делятся на одно и то же число. А значит, это и есть нетривиальный сомножитель b[124]. Когда такой сомножитель нашёлся, я просто разделил b[124] на него и получил второй сомножитель. Таким образом, число b удалось разложить, и задача была решена.

Вот код, который считывает значения b из pdf файла:

import fitz

import re

doc = fitz.open("lab2.pdf")

text = ""

for page in doc:

    text += page.get\_text()

pattern = r"b\[(\d+)\]=((?:\d+\n?)+)"

matches = re.findall(pattern, text)

with open("b.txt", "w") as out:

    for index, number in matches:

        number\_clean = number.replace("\n", "").strip()

        out.write(f"b[{index}]={number\_clean}\n")

А вот код, который непосредственно ищет сомножители для числа b:

import re

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

b\_numbers = []

with open("b.txt", "r") as f:

    for line in f:

        match = re.match(r"b\[(\d+)\]=(\d+)", line)

        if match:

            index = int(match.group(1))

            value = int(match.group(2))

            b\_numbers.append((index, value))

target\_index = 124

b124 = b\_numbers[target\_index][1]

factors = []

for idx, value in b\_numbers:

    if idx == target\_index:

        continue

    common\_gcd = gcd(b124, value)

    if common\_gcd > 1:

        factors.append((common\_gcd, idx))

print(f"\nНайдено {len(factors)} общих сомножителей для b[{target\_index}]:")

for factor, idx in factors:

    print(f"НОД(b[{target\_index}], b[{idx}]) = {factor}")

if factors:

    factor = factors[0][0]

    other = b124 // factor

    print(f"\nРазложение b[{target\_index}]:")

    print(f"  b[{target\_index}] = {factor} × {other}")

Вот вывод данного кода. Как видно из данного вывода, b[124] имеет сомножители с двумя другими числами b:

vingael@vingael-IdeaPad-Gaming-3-15ACH6:~/Popa/python/Биткоин$ /bin/python3 /home/vingael/Popa/python/Биткоин/lab2\_22.py

Найдено 2 общих сомножителей для b[124]:

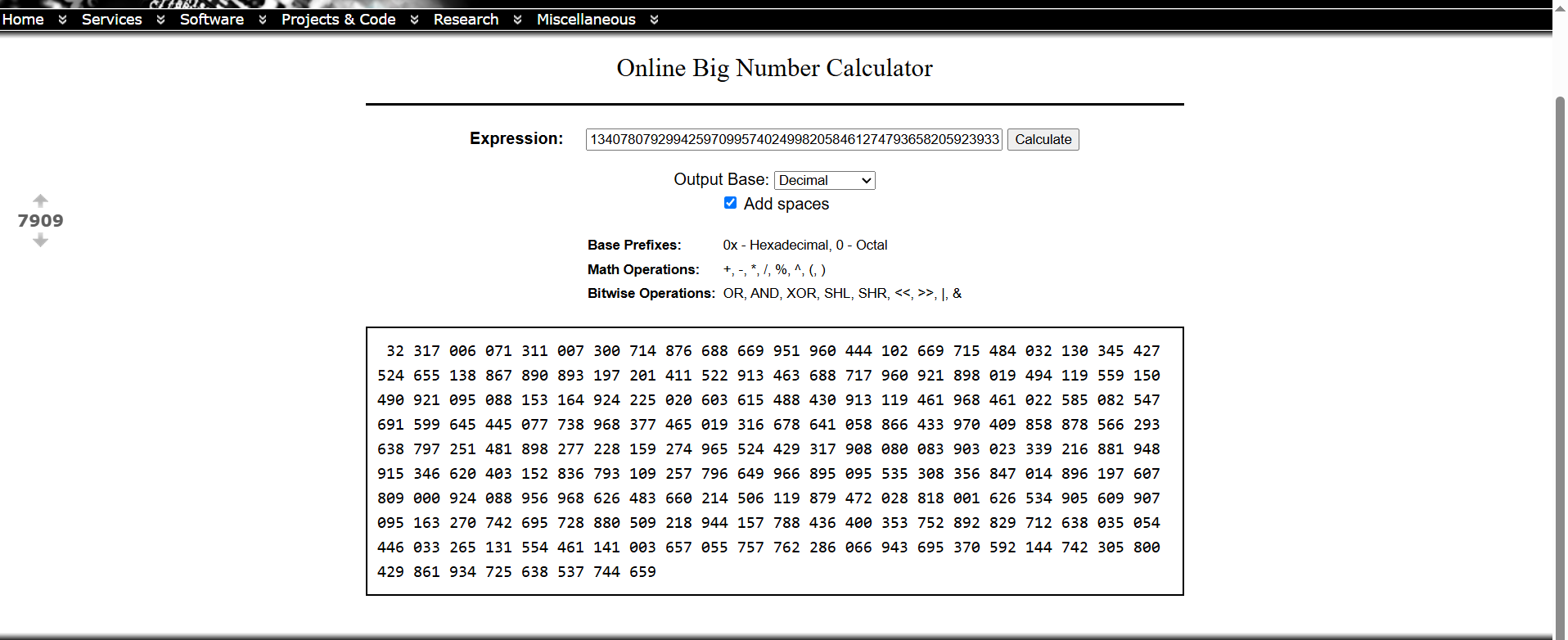
НОД(b[124], b[38]) = 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649329061289

НОД(b[124], b[111]) = 2410312426921032588580116606028314112912093247945688951359675039065257391591803200669085024107346049663448766280888004787862416978794958324969612987890774651455213339381625224770782077917681499676845543137387820057597345857904599109461387122099507964997815641342300677629473355281617428411794163967785870370368969109221591943054232011562758450080579587850900993714892283476646631181515063804873375182260506246992837898705971012525843324401232986857004760990922331

Разложение b[124]:

b[124] = 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649329061289 × 2410312426921032588580116606028314112912093247945688951359675039065257391591803200669085024107346049663448766280888004787862416978794958324969612987890774651455213339381625224770782077917681499676845543137387820057597345857904599109461387122099507964997815641342300677629473355281617428411794163967785870370368969109221591943054232011562758450080579587850900993714892283476646631181515063804873375182260506246992837898705971012525843324401232986857004760990922331

Вот проверка, которая показывает, что мы нашли правильные сомножители:



# **Выводы**

В ходе выполнения лабораторной работы были изучены и практически применены методы криптографического хеширования, а также алгоритмы факторизации больших чисел. На первом этапе была использована хеш-функция ГОСТ Р 34.11-2012 (Стрибог) для вычисления контрольной суммы от строки, содержащей ФИО. Младшие 8 бит хеша были интерпретированы как целое число, которое определило номер варианта задачи. Это число определило конкретную пару значений a и b, для которых требовалось разложение на нетривиальные сомножители.

Значение a было успешно разложено с помощью стандартных методов факторизации, реализованных на языке Python. Однако разложение числа b вызвало затруднение из-за его большого размера. Это побудило к разработке дополнительного алгоритма, основанного на нахождении общего делителя (НОД) между b и другими числами того же типа, с использованием алгоритма Евклида. Такой подход позволил найти хотя бы один нетривиальный сомножитель числа b и восстановить его разложение.

Таким образом, в процессе работы были закреплены навыки использования криптографических хеш-функций, анализа битовых представлений данных, работы с большими числами и применения алгоритмов для факторизации. Также была проведена проверка корректности найденных сомножителей с помощью внешних онлайн-инструментов для арифметики больших чисел. Лабораторная работа показала, что даже в условиях ограниченных ресурсов и невозможности применения универсальных библиотек можно найти эффективные пути решения поставленной криптографической задачи.

# **Список используемой литературы**

<https://gostcrypto.readthedocs.io/en/latest/>

<https://home.cs.colorado.edu/~srirams/courses/csci2824-spr14/gmpTutorial.html>