Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Криптография»

Студент: М. А. Бронников
Преподаватель: А. В. Борисов
Группа: М8О-307Б

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №3

Задача:

1. Строку в которой записано своё ФИО подать навход в хеш-функцию ГОСТ Р 34.11-2012 (Стрибог). Младшие 4 бита выхода интерпретировать как число, которое в дальнейшем будет номером варианта.

2. Программно реализовать алгоритм функции хеширования. 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 алгоритму с разным числом раундов.

5. Построить график зависимости количества раундов и возможности различения отдельных бит при различном количестве раундов.

Вариант 7: Keccak.

Выбранная версия: Keccak-1600.

1 Расчёт варианта

Для расчета варианта я написал программу на языке *Python*, которая при помощи сторонней библиотеки **pygost** вычисляет хэш переданной ей строки (в моем случае «Бронников Максим Андреевич»), после чего при помощи операции «побитового И» отделяет последние 4 бита от хэша и выводит их в удобной для восприятия форме.

Исходный код программы:

```
1  from pygost import gost34112012256
2  import sys
3
4  def number_from_str(family):
5      if not isinstance(family, str):
6          return -1
7      # Working only with .encode() method, else TypeError
8      last = gost34112012256.new(family.encode()).digest()[-1]
9      last &= 15
10     if last >= 10:
11         return chr(ord('A') + last - 10)
12     return str(last)
13
14 if __name__ == "__main__":
15     family = "Бронников МаксимАндреевич"
16
17     if len(sys.argv) == 2 and sys.argv[1] == "-i":
18         family = input("Введите вашеФИО: ")
19     elif len(sys.argv) > 1:
20         raise ValueError("Wrong args")
21
22     print("Вариант для" + family + ":", number_from_str(family))
```

Результат вычисления:

```
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ python3 variant_number.py
Вариант для Бронников Максим Андреевич: 7
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ exit
```

Следовательно, мне следует реализовать алгоритм *Keccak*, который имеет 7 различных конфигураций, зависящих от длины состояния. Я решил выбрать наиболее известный и распространённый вариант: *Keccak-1600*.

2 Описание алгоритма

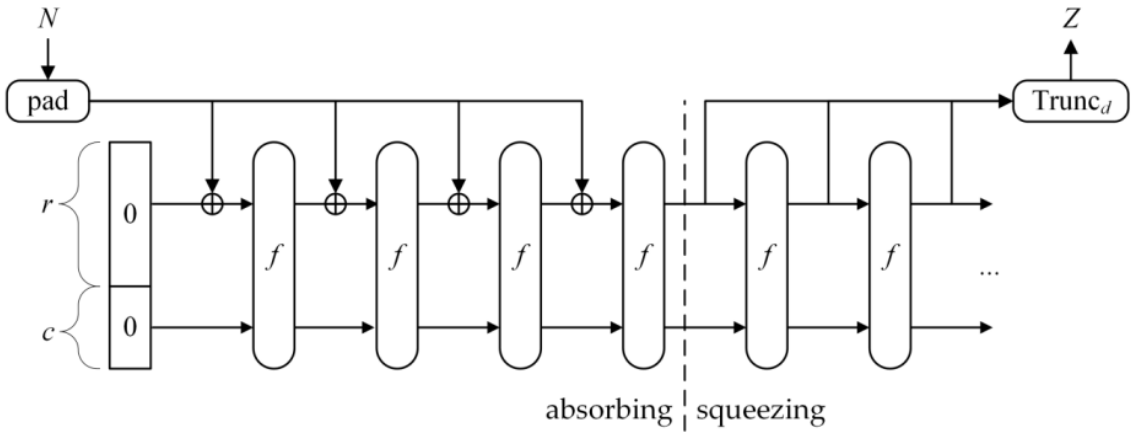
Алгоритм *Keccak* имеет несколько различных конфигураций, зависящих от параметра l : $0 \leq l \leq 6$. При этом длина состояния в битах вычисляется по формуле $b = 25 \cdot 2^l$. Выбранный мной вариант - *Keccak-1600* с $l = 6$, поскольку он наиболее часто ипользуется на практике и хорошо описан в документации[2] что упрощает реализацию. Алгоритм имеет настраиваемые параметры - длину блока **rate**, параметр **capacity**, такой что $capacity + rate = 1600$, а также позволяет настроить количество раундов, длинну выходных данных и разделитель **del**, который будет добавляться к концу последнего блока перед осуществлением *pad10*1*.

Алгоритм *Keccak-1600* имеет массив-состояние **state** длинной 1600 бит и работает по принципу «губки», что позволяет поделить его на 2 этапа:

1. *«Впитывание»*. Алгоритм последовательно считывает блоки по **rate** бит. Каждый блок *впитывается* в массив **state** при помощи операции *XOR*, после чего вызывается основная преобразующая функция алгоритма **Keccak-f**. Данное действие повторяется пока не будет считан все входные данные. К последнему считанному блоку данных делается дополнение по правилу *pad10*1* - к блоку добавляется еденичный бит, после него необходимое число нулей с единицей на конце, чтобы последний блок делился нацело на **rate**.

2. *«Выжимание»*. Из массива **state** берётся необходимое число бит, в зависимости от заданной длины хэша, однако если заднная длина превышает **rate**, то в таком случае последовательно достаётся **rate** бит и вызывается функция **Keccak-f**, пока не будет «выжато» необходимое количество данных.

Иллюстрация работы «губки»:



Функция **Keccak-f** преобразует массив-состояние **state** выполняя на каждый свой вызов n_r раунов(в стандартной реализации $n_r = 24$), каждый из которых состоит из 5 последовательных шагов:

$$round = \iota \circ \chi \circ \rho \circ \pi \circ \theta$$

Введем следующую запись для доступа к битам состояния *state*:

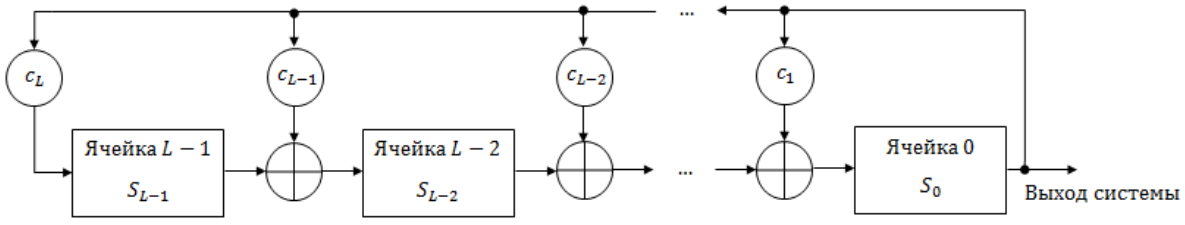
$$a[x][y][z] = state[64(x \bmod 5 + y \bmod 5) + z \bmod 64]$$

Описание шагов раунда:

- θ: $a[x][y][z] \leftarrow a[x][y][z] \oplus \sum_{i=0}^{4} a[x-1][i][z] \oplus \sum_{i=0}^{4} a[x+1][i][z-1]$
- π: $a[x][y][z] \leftarrow a[x][y][z - \frac{(t+1)(t+2)}{2}]$, где $0 \le t < 24$ и $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$
- ρ: $a[X][Y][z] \leftarrow a[x][y][z]$, где $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} X \\ Y \end{pmatrix}$
- χ: $a[x][y][z] \leftarrow a[x][y][z] \oplus (a[x+1][y][z] \oplus 1) \otimes a[x+2][y][z]$
- ι: $a[x][y][z] \leftarrow a[x][y][z] \oplus RC[i_r][x][y][z]$, где $RC[i_r]$ - константа значение каждого бита которой равно 0, кроме 7 бит, которые зависят от номера раунда i_r и удовлетворяют условию:

  $$RC[i_r][0][0][2^j - 1] = (x^{j+7i_r} \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x \ \forall j : 0 \le j \le 6,$$

  где x в поле Галуа из 2 элементов (1 и 0). Это значение ищется при помощи функции регистра сдвига с линейной обратной связью в конфигурации Галуа[1] с регистром S длины $L = 8$ бит и начальным состоянием регистра $S_0 = 1$, $S_1 = 1$, $S_2 = 1$, $S_3 = 0$, $S_4 = 1$, $S_5 = 0$, $S_6 = 0$, $S_7 = 0$, чтобы первый вызов функции регистра дал результат $x^0 = 1$ до вычета по модулю x.



3 Реализация алгоритма

Для реализации я использовал язык *C++*, поскольку в нем типы данных строго ограничены по количеству используемой памяти и есть такие типы данных как *uint8_t* для удобной работы с байтами и *uint64_t* для одновременного проведения операций с блоками по 64 бит, которые хорошо описаны в документации[2] по алгоритму.

Перед началом реализации я проверил порядок нумерации байт в своем компьютере и обнаружил что у меня аппаратно реализован *little-endian* порядок.

Код для проверки на языке C:

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
  uint16_t x = 1; /* 0000 0001 */
  printf("%s\n", *((uint8_t *) &x) == 0 ? "big-endian" : "little-endian");
  return 0;
}
```

Исходя из этого я следует упомянуть следующие *особенности реализации:*

1. Поскольку у меня на компьютере *little-endian* порядок, а биты в описании алгоритма нумеруются как *Least Significant Bit first*, я принял решение размещать байты в **state** без изменения порядка следования бит, а при работе с линиями $a[x][y]$ по 64 бита считывать их в переменные *uint64_t* простым разименовыванием по указателю, и нумеровать биты в перменной как *LSB first* что позволяет получать доступ к ним при поомощью простейших побитовых операций.

2. При указанном выше приеме следует помнить, что единица, которая добавляется в конец последнего блока при дополнении будет стоять на позиции *MSB*, что реализуется добавлением в конец байта $0x80$.

3. Если указанный пользователем **del** имеет единичный бит на позиции *MSB*, следует вызвать **Keccak-f** перед дополнением.

4. Начальное состояние регистра для подсчета констант раунда должно быть $0xB8$, чтобы при выполнении первого вызова получить состояние $0x01$.

Моя реализация имеет возможность хэширования не только входящих строк, но и файлов, а также предостовляет возможность ручного управления алгоритмом. Есть возможность задать количество раундов и разделитель для дополнения.

Исходный код объявления класса из «Keccak.h»:

```
#ifndef KECCAK_H
#define KECCAK_H

#include <string>
#include <iostream>
#include <vector>
#include <inttypes.h>
#include <exception>
#include <fstream>
#include <cstring>

using namespace std;

#define MAX_BUFFER_SIZE 4096

class Keccak1600{
public:
    Keccak1600();
    Keccak1600(uint16_t r, uint16_t c);
    Keccak1600(uint16_t r, uint16_t c, uint8_t del);
    Keccak1600(uint16_t r, uint16_t c, uint8_t del, uint8_t rnds);

    void set_delimeter(uint8_t del);
    void set_output_size(uint16_t bytesize);
    void set_nrounds(uint8_t rnds);

    void push_bytes(const uint8_t* bytes, uint32_t bytesize);
    void get_hash(uint8_t* hash);

    void hash_compute(const uint8_t* input, uint8_t bytesize, uint8_t* output);
    void string_hash_to_vec(const string& input, vector<uint8_t>& output);
    void file_hash(const char* filename, vector<uint8_t>& output);
private:
    uint64_t a(uint8_t x, uint8_t y);
    uint64_t* A(uint8_t x, uint8_t y);
    uint64_t ROT(uint64_t line, uint8_t num);

    void Keccak_f();
    void Tetta();
    void PiRo();
    void Hi();
    void Li();
    uint64_t RC();
    bool LFSR();

    uint8_t LFSRState;
    const uint8_t lenght = 6;
    const uint16_t bits = 1600;
```

```
    uint16_t capacity; // in bytes
    uint16_t rate; // in bytes
    uint8_t delimiter;
    uint8_t state[200];
    uint16_t readed_bytes;
    uint16_t output_size;
    uint16_t nrounds;
};

#endif
```

Исходный код реализации класса из «Keccak.cpp»:

```
#include "Keccak.h"

Keccak1600::Keccak1600(){
    delimiter = 0x01;
    output_size = 64;
    rate = 72;
    capacity = 128;
    nrounds = 24;
    readed_bytes = 0;
    memset(state, 0, 200 * sizeof(uint8_t));
}

Keccak1600::Keccak1600(uint16_t r, uint16_t c, uint8_t del){
    if(r & 7){
        throw exception();
    }
    if(c & 7){
        throw exception();
    }
    rate = r >> 3;
    capacity = c >> 3;
    delimiter = del;
    output_size = 64;
    nrounds = 24;
    readed_bytes = 0;
    if(((rate + capacity) << 3) != bits){
        throw exception();
    }
    memset(state, 0, 200 * sizeof(uint8_t));
}

Keccak1600::Keccak1600(uint16_t r, uint16_t c){
    if(r & 7){
        throw exception();
    }
    if(c & 7){
        throw exception();
```

```
38      }
39      rate = r >> 3;
40      capacity = c >> 3;
41      delimiter = 0x01;
42      output_size = 64;
43      nrounds = 24;
44      readed_bytes = 0;
45      if(((rate + capacity) << 3) != bits){
46          throw exception();
47      }
48      memset(state, 0, 200 * sizeof(uint8_t));
49  }
50
51  Keccak1600::Keccak1600(uint16_t r, uint16_t c, uint8_t del, uint8_t rnds){
52      if(r & 7){
53          throw exception();
54      }
55      if(c & 7){
56          throw exception();
57      }
58      rate = r >> 3;
59      capacity = c >> 3;
60      delimiter = del;
61      output_size = 64;
62      nrounds = rnds;
63      readed_bytes = 0;
64      if(((rate + capacity) << 3) != bits){
65          throw exception();
66      }
67      memset(state, 0, 200 * sizeof(uint8_t));
68  }
69
70  void Keccak1600::set_delimeter(uint8_t del){
71      delimiter = del;
72  }
73
74  void Keccak1600::set_nrounds(uint8_t rnds){
75      nrounds = rnds;
76  }
77
78  void Keccak1600::set_output_size(uint16_t bytesize){
79      if(bytesize < 1){
80          throw exception();
81      }
82      output_size = bytesize;
83  }
84
85  uint64_t Keccak1600::a(uint8_t x, uint8_t y){
86      return *((uint64_t*)&state[sizeof(uint64_t)*(5*y + x)]);
```

```
87  }
88
89  uint64_t* Keccak1600::A(uint8_t x, uint8_t y){
90      return (uint64_t*)&state[sizeof(uint64_t)*(5*y + x)];
91  }
92
93  uint64_t Keccak1600::ROT(uint64_t line, uint8_t num){
94      return (line << num) ^ (line >> (64 - num));
95  }
96
97  bool Keccak1600::LFSR(){
98      if(LFSRState & 0x80){
99          LFSRState <<= 1;
100         LFSRState ^= 0x71;
101         return true;
102     }
103     LFSRState <<= 1;
104     return false;
105 }
106
107 uint64_t Keccak1600::RC(){
108     uint64_t constant = 0;
109     for(uint8_t j = 0; j <= lenght; ++j){
110         if(LFSR()){
111             constant |= (uint64_t)1 << ((1 << j) - 1);
112         }
113     }
114     return constant;
115 }
116
117 void Keccak1600::Tetta(){
118     uint64_t C[5];
119     for(uint8_t x = 0; x < 5; ++x){
120         C[x] = a(x, 0);
121         for(uint8_t y = 1; y < 5; ++y){
122             C[x] ^= a(x, y);
123         }
124     }
125     for(uint8_t x = 0; x < 5; ++x){
126         uint64_t D = C[x - 1 >= 0 ? x - 1 : 4];
127         D ^= ROT(C[x + 1 < 5 ? x + 1 : 0], 1);
128         for(uint8_t y = 0; y < 5; ++y){
129             *A(x, y) = a(x, y) ^ D;
130         }
131     }
132 }
133
134 void Keccak1600::PiRo(){
135     uint8_t x = 1, y = 0;
```

```
136     uint64_t line = a(x, y);
137     for(uint16_t t = 0; t < 24; ++t){
138         uint8_t Y = ((x << 1) + 3 * y) % 5;
139         x = y;
140         y = Y;
141         uint64_t newline = a(x, y);
142         *A(x, y) = ROT(line, ((t + 1)*(t + 2) >> 1) % 64);
143         line = newline;
144     }
145 }
146 
147 void Keccak1600::Hi(){
148     for(uint8_t y = 0; y < 5; ++y){
149         uint64_t C[5];
150         for(uint8_t x = 0; x < 5; ++x){
151             C[x] = a(x, y);
152         }
153         for(uint8_t x = 0; x < 5; ++x){
154             *A(x, y) = C[x] ^ (~C[x + 1 < 5 ? x + 1 : x - 4] & C[x + 2 < 5 ? x + 2 : x
                - 3]);
155         }
156     }
157 }
158 
159 void Keccak1600::Li(){
160     *A(0, 0) ^= RC();
161 }
162 
163 void Keccak1600::Keccak_f(){
164     LFSRState = 0xB8;
165     for(uint8_t i = 0; i < nrounds; ++i){
166         Tetta();
167         PiRo();
168         Hi();
169         Li();
170     }
171 }
172 
173 void Keccak1600::push_bytes(const uint8_t* bytes, uint32_t bytesize){
174     for(uint32_t i = 0; i < bytesize; ++i){
175         state[readed_bytes++] ^= bytes[i];
176         if(readed_bytes == rate){
177             Keccak_f();
178             readed_bytes = 0;
179         }
180     }
181 }
182 
183 void Keccak1600::get_hash(uint8_t* hash){
```

```
184     state[readed_bytes++] ^= delimiter;
185     if(readed_bytes == rate && (delimiter & 0x80)){
186         Keccak_f();
187     }
188     state[rate - 1] ^= 0x80;
189     readed_bytes = 0;
190     Keccak_f();
191 
192     uint16_t in_rate = 0;
193     for(uint16_t i = 0; i < output_size; ++i){
194         hash[i] = state[in_rate++];
195         if(in_rate == rate){
196             Keccak_f();
197             in_rate = 0;
198         }
199     }
200 }
201 
202 void Keccak1600::hash_compute(const uint8_t* input, uint8_t bytesize, uint8_t* output)
        {
203     memset(state, 0, 200 * sizeof(uint8_t));
204     readed_bytes = 0;
205     push_bytes(input, bytesize);
206     get_hash(output);
207 }
208 
209 void Keccak1600::string_hash_to_vec(const string& input, vector<uint8_t>& output){
210     output.resize(output_size);
211     hash_compute((const uint8_t*)input.c_str(), input.size(), output.data());
212 }
213 
214 void Keccak1600::file_hash(const char* filename, vector<uint8_t>& output){
215     memset(state, 0, 200 * sizeof(uint8_t));
216     readed_bytes = 0;
217     uint8_t buffer[MAX_BUFFER_SIZE];
218     ifstream is(filename, ios::in | ios::binary);
219 
220     if(!is){
221         cout << "Wrong path to file!" << endl;
222         throw exception();
223     }
224 
225     while(is && is.peek() != EOF){
226         is.read((char*)buffer, MAX_BUFFER_SIZE);
227         push_bytes(buffer, is.gcount());
228     }
229     output.resize(output_size);
230     get_hash(output.data());
231 }
```

Демонстрация работы алгоритма:

```
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ ls
documentation  Keccak          task.pdf  variant_number.py
filehash.cpp   stringhash.cpp  test.c
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ g++ -std=c++11 -Wall -pedantic
Keccak/Keccak.cpp stringhash.cpp -o run
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ ./run
String "Бронников Максим Андреевич" Hash:
========================================
0x4e 0xe7 0xb8 0x45 0x83 0xce 0x7b 0xff 0xf1 0x28 0x04 0xf7 0x4b 0x47 0x86
0x2b 0x5b 0xc5 0x67 0x2a 0xdd 0x7a 0xcc 0x86 0xcd 0x8c 0xfb 0xdc 0x93 0x15
0x24 0xed 0x0f 0xc8 0x9f 0xf6 0xe0 0x82 0xb2 0x14 0xb6 0xed 0xfd 0xd8 0xcd
0xac 0xc4 0x86 0xd2 0x9c 0x9c 0x97 0xab 0x13 0x6e 0xd1 0xba 0xaa 0x42 0x54
0x34 0xf1 0x55 0xd7
========================================
Program ends! Made by Bronnikov Max!
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ g++ -std=c++11 -Wall -pedantic
Keccak/Keccak.cpp filehash.cpp -o run
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ ./run
File "documentation/Keccak-reference-3.0.pdf" Hash:
========================================
0x3c 0xf3 0xb4 0xee 0x4e 0xb0 0x5c 0x55 0x4b 0x0a 0x98 0x84 0x6f 0xe5 0x05
0x46 0xa4 0xf0 0xe8 0x0c 0xb5 0x54 0x80 0xf0 0xd4 0xe6 0x28 0x13 0x51 0x1c
0x0b 0x0f 0x01 0x2a 0x03 0x8b 0x73 0x0b 0x3b 0xdc 0xdf 0xa8 0x05 0xcd 0x3e
0x50 0xf5 0x4c 0xae 0x38 0xfb 0x56 0x65 0x11 0x80 0x2f 0x32 0xb8 0x55 0xb9
0x83 0xeb 0x8a 0xad
========================================
Program ends! Made by Bronnikov Max!
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$
```

В тестах использовался разделитель $0x06$ для проверки работы в сравнении с классическим алгоритмом *SHA-3*, который отличается от стандартонго *Keccak* добавлением битов 01 перед дополнением. Для стандартного *Keccak-1600* используйте $0x01$.

Тесты сравнивались со значениями, полученными в онлайн *SHA-3* генераторах.

Весь код проекта вместе с кодами для демонстрации работы алгоритма можно найти в моём *GitHub* репозитории: `https://github.com/Bronnikoff/Cryptography/tree/master/lab3`

4 Анализ алгоритма

Для анализа работы своего алгоритма я воспользовался методом, который называется «Дифференциальный криптоанализ». Он позволяет оценить качество алгоритма при различном количестве раундов.

При анализе я рассмотрел как изменение количества раундов алгоритма влияет на дифференциальную разность выходного хэша, а именно посчитал количество бит выходного хэша, которое различается для двух текстов с разницей в 1 бит.

Для простоты эксперимента я выбрал дифференциал $\Delta X = 1$ и проводил анализ для количества раундов, варьирующегося от 1 до классических для *Keccak-1600* 24 раундов.

Для этого я генерировал случайным способом $n = 100$ различных текстов X_i, $\forall i : 0 \leq i < n$, и для каждого текста X_i произвел следующую последовательность действий:

1. Сгенерировал 2-ой текст $\overline{X_i} = X_i \oplus \Delta X$, отличающийся от исходного текста X_i лишь 1 битом.

2. Для каждого анализируемого количества раундов $\forall j : 1 \leq j \leq 24$:

   - Вычисляем значение хэшэй $Y_i^j = H^j(X_i)$ и $\overline{Y_i^j} = H^j(\overline{X_i})$ при помощи нашей хэш-функции H^j с настраиваемым количеством раундов j.
   - Вычисляем их дифференциал $\Delta Y_i^j = Y_i^j \oplus \overline{Y_i^j}$.
   - Подсчитываем количество битов $c_i^j = \sum_{k=0}^{|\Delta Y_i^j|-1} \Delta Y_i^j[k]$, которые различны у двух хэшей Y_i^j и $\overline{Y_i^j}$.

После подсчитаем среднее количество битов $c^j = \frac{\sum_{i=0}^{n-1}(c_i^j)}{n}$, которое оказалось различным для каждого анализируемого количества раундов $\forall j : 1 \leq j \leq 24$ алгоритма H^j.

Результат вычисления:

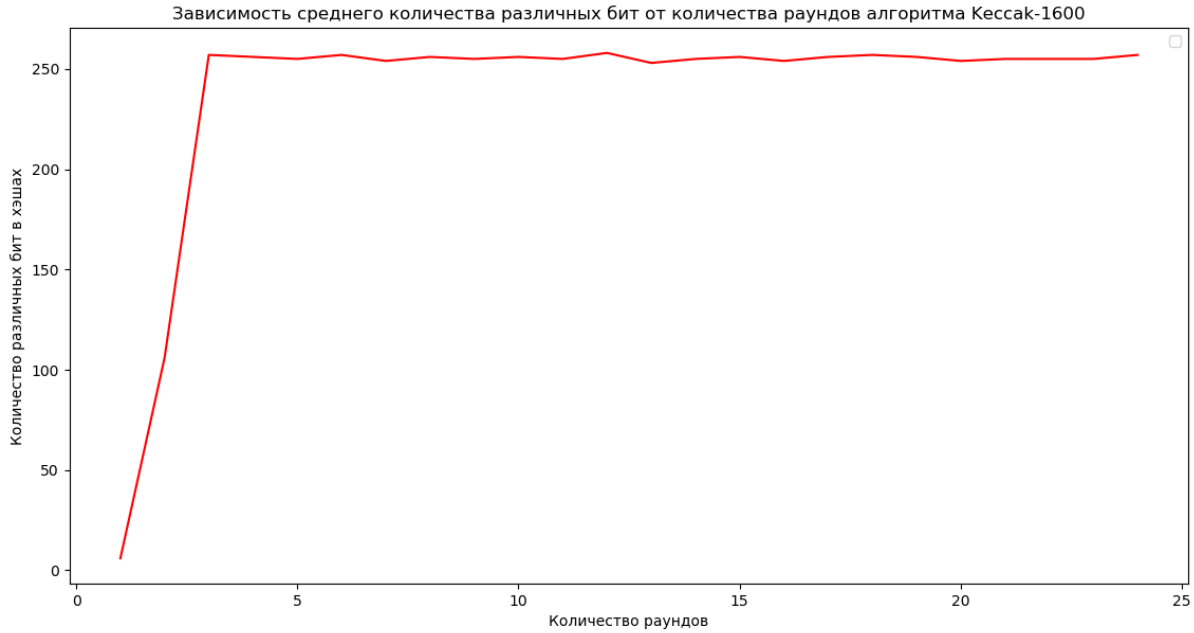
```
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ g++ -std=c++11 -Wall -pedantic
Keccak/Keccak.cpp diff_analysis.cpp -o run
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ ./run
Differential analysis counter started!

Number of test strings: 100
Will explore round numbers from 1 to 24
Stat will write in file: analis.stat

Program made by Max Bronnikov
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$ cat analis.stat
1 6
2 106
3 257
4 256
5 255
6 257
7 254
8 256
9 255
10 256
11 255
12 258
13 253
14 255
15 256
16 254
17 256
18 257
19 256
20 254
21 255
22 255
23 255
24 257
(base) max@max-Lenovo-B50-30:~/Cryptography/lab3$
```

Здесь первый столбец - количество раундов алгоритма, второй столбец - среднее количество различных бит для n случайных тестовых строк.

И, наконец, построим график зависимости среднего количества различных битов c^j от количества раундов алгоритма j:



Исходный код отрисовки графика на языке Python:

```
import matplotlib.pyplot as plt

filename = "analis.stat"

rounds = []
bits = []

with open(filename, "r") as f:
    for line in f:
        nums = line.split()
        rounds.append(int(nums[0]))
        bits.append(int(nums[1]))

plt.plot(rounds, bits, color = "r")
plt.title("Зависимость Keccak-1600")
plt.xlabel("Количество раундов")
plt.ylabel("Количество бит")

plt.legend()
plt.show()
```

5 Выводы

Выполнив третью лабораторную работу по курсу «Криптография», я получил новые знания о таком понятии, как *Криптографическая хэш-функция*. Мне пришлось узнать и разобраться в том, как они реализуются и даже довелось самостоятельно реализовать один из самых современных алгоритмов хэширования - *Keccak*, который лег в основу стандарта *SHA-3*. Также мне довелось познакомится с новым и несколько пугающим на первый взгляд явлением - дифференциальным криптоанализом и даже применить его простейшие элементы для анализа своего алгоритма.

Эта лабораторная оказалась самой трудной из тех, что я делал до этого, поскольку в ней пришлось изучить множество новых для себя понятий, таких как *Least Significant Bit*, *little-ending*, *регистр линейного сдвига с обратной связью*, а также прочитать много документации на английском языке.

Однако несмотря на все сложности и неприятности я не пожалел о том, что мне пришлось столкнуться с этой темой, ведь она оставила у меня огромный багаж новых знаний, а также позволила в очередной раз перебороть себя на пути к своей цели стать хорошим программистом.

Список литературы

[1] *Регистр сдвига с линейной обратной связью*
URL: `https://ru.wikipedia.org/wiki/Регистр_сдвига_с_линейной_обратной_связью` (дата обращения: 04.04.2020).

[2] *Keccak reference*
URL: `https://keccak.team/files/Keccak-reference-3.0.pdf` (дата обращения: 02.04.2020).

[3] *Весь исходный код*
URL: `https://github.com/Bronnikoff/Cryptography/tree/master/lab3` (дата обращения: 06.04.2020).