

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Криптография»

Студент: Л. Я. Вельтман
Преподаватель: А. В. Борисов
Группа: М8О-307Б
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №3

Задача:

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 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 0: Гост Р 34.11-94.

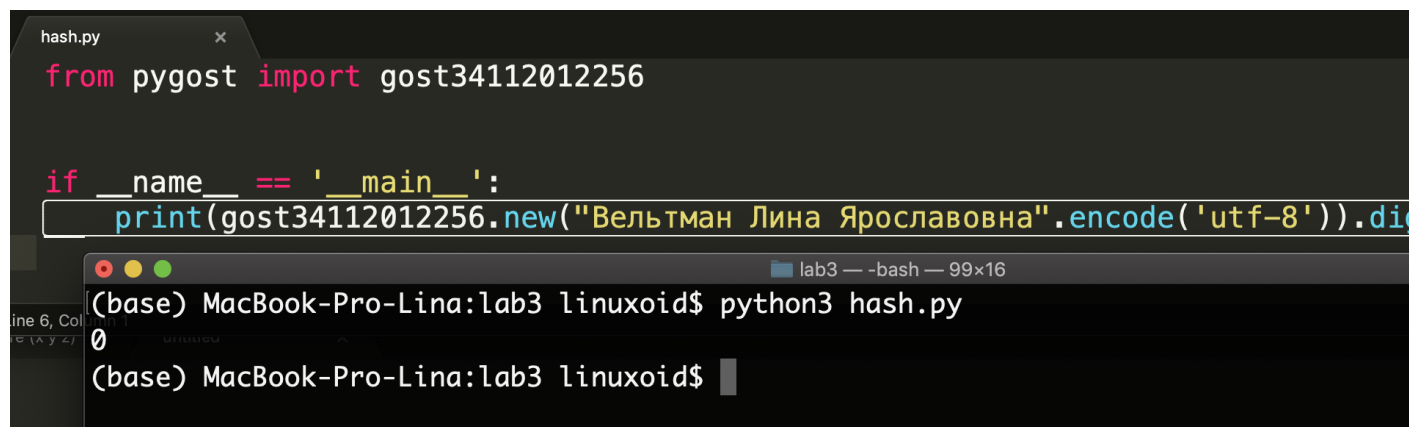
1 Расчёт варианта

Для расчёта варианта была использована сторонняя библиотека **pygost**, которая вычисляет хэш переданной ей строки (в моем случае «Вельтман Лина Ярославовна»)

Исходный код программы:

```
1 from pygost import gost34112012256
2
3
4 if __name__ == '__main__':
5     print(gost34112012256.new("Вельтман ЛинаЯрославовна".encode('utf-8')).digest().hex
      () [-1])
```

Результат вычисления:



The screenshot shows a code editor with a file named `hash.py` containing the following Python code:

```
from pygost import gost34112012256

if __name__ == '__main__':
    print(gost34112012256.new("Вельтман Лина Ярославовна".encode('utf-8')).digest().hex
          () [-1])
```

Below the code editor, a terminal window is open, showing the command `python3 hash.py` being executed. The output of the command is `0`.

Следовательно, мне следует реализовать алгоритм варианта 0:ГОСТ Р 34.11-94.

2 Описание алгоритма

ГОСТ Р 34.11-94 — устаревший российский криптографический стандарт вычисления хеш-функции. В странах СНГ переиздан и используется как межгосударственный стандарт ГОСТ 34.311-95. Стандарт определяет алгоритм и процедуру вычисления хеш-функции для последовательности символов.

Для описания алгоритма хеширования будем использовать следующие обозначения:

- \parallel — слияние (конкатенация) двух блоков в один.
- $+$ — сложение двух блоков длиной 256 бит по модулю 2^{256}
- \oplus — побитное сложение (XOR) двух блоков одинаковой длины.

Далее будем считать, что младший (нулевой) бит в блоке находится справа, старший — слева.

Основой описываемой хеш-функции является шаговая функция хеширования $H_{out} = f(H_{in}, m)$, где H_{in}, H_{out}, m - блоки длины 256 бит.

Входное сообщение M разделяется на блоки $m_n, m_{n-1}, m_{n-2}, \dots, m_1$ по 256 бит. В случае если размер последнего блока m_n меньше 256 бит, то к нему приписываются слева нули для достижения заданной длины блока.

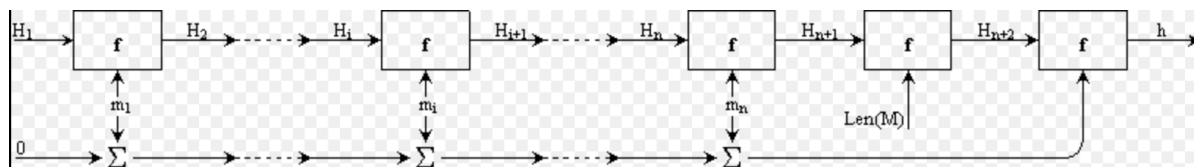
Каждый блок сообщения, начиная с первого подается на шаговую функцию для вычисления промежуточного значения хеш-функции:

$$H_{i+1} = f(H_{i+1}, m_i)$$

После вычисления H_{n+1} конечное значение хеш-функции получают следующим образом:

- $H_{n+2} = f(H_{n+1}, L)$, где L — Длина сообщения M в битах по модулю 2^{256}
- $h = f(H_{n+2}, K)$, где K — Контрольная сумма сообщения M : $m_1 + m_2 + m_3 + \dots + m_n$

h — значение хеш-функции сообщения M



Алгоритм вычисления шаговой функции хеширования: Шаговая функция хеширования f отображает два блока длиной 256 бит в один блок длиной 256 бит: $H_{out} = f(H_{in}, m)$ и состоит из трех частей:

1. Генерирование ключей K_1, K_2, K_3, K_4
2. Шифрующее преобразование — шифрование H_{in} с использованием ключей K_1, K_2, K_3, K_4
3. Перемешивающее преобразование результата шифрования

Генерация ключей: В алгоритме генерации ключей используются:

- Три константы:
 - $C_2 = 0$
 - $C_3 = 0\text{xff}00\text{ffff}000000\text{ffff}0000\text{ff}00\text{ffff}0000\text{ff}00\text{ff}00\text{ff}00\text{ffff}00\text{ff}00\text{ff}00$
 - $C_4 = 0$
- Два преобразования блоков длины 256 бит:
 - Преобразование $A(Y) = A(y_4 \| y_3 \| y_2 \| y_1) = (y_1 \oplus y_2) \| y_4 \| y_3 \| y_2$, где y_1, y_2, y_3, y_4 — подблоки блока Y длины 64 бит.
 - Преобразование $P(Y) = P(y_{32} \| y_{31} \| \dots \| y_1) = y_{\varphi(32)} \| y_{\varphi(31)} \| \dots \| y_{\varphi(1)}$, где $\varphi(i + 1 + 4(k - 1)) = 8i + k, i = 0, \dots, 3, k = 1, \dots, 8$, а $y_{32}, y_{31}, \dots, y_1$ — подблоки блока Y длины 8 бит.

Алгоритм:

1. $U = H_{in}, \quad V = m, \quad W = U \oplus V, \quad K_1 = P(W)$
2. Для $j = 2, 3, 4$ выполняем следующее:
 $U = A(U) \oplus C_j, \quad V := A(A(V)), \quad W := U \oplus V, \quad K_j = P(W)$

Шифрующее преобразование После генерирования ключей происходит шифрование H_{in} по ГОСТ 28147–89 в режиме простой замены на ключах K_i (для $i = 1, 2, 3, 4$), процедуру шифрования обозначим через E (Примечание: функция шифрования E по ГОСТ 28147 шифрует 64 битные данные 256 битным ключом). Для шифрования H_{in} разделяют на четыре блока по 64 бита:

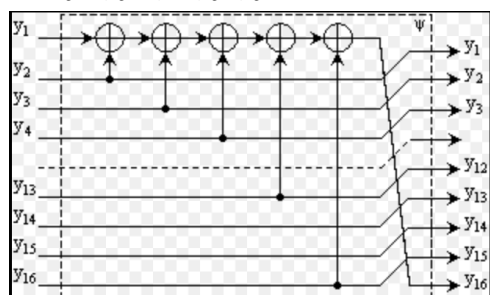
$H_{in} = h_4 \| h_3 \| h_2 \| h_1$ и зашифровывают каждый из блоков:

- $s_1 = E(h_1, K_1)$
- $s_2 = E(h_2, K_2)$
- $s_3 = E(h_3, K_3)$
- $s_4 = E(h_4, K_4)$

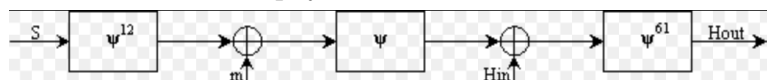
После чего блоки собирают в 256 битный блок: $S = s_4 \| s_3 \| s_2 \| s_1$

Перемешивающее преобразование На последнем этапе происходит перемешивание H_{in} , S и m с применением регистра сдвига, в результате чего получают H_{out} .

Для описания процесса преобразования сначала необходимо определить функцию ψ , которая производит элементарное преобразование блока длиной 256 бит в блок той же длины: $\psi(Y) = \psi(y_{16} \| y_{15} \| \dots \| y_2 \| y_1) = (y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus y_{13} \oplus y_{16}) \| y_{16} \| y_{15} \| \dots \| y_3 \| y_2$, где $y_{16}, y_{15}, \dots, y_2, y_1$ — подблоки блока Y длины 16 бит.



Перемешивающее преобразование имеет вид $H_{out} = \psi^{61}(H_{in} \oplus \psi(m \oplus \psi^{12}(S)))$, где ψ^i означает суперпозицию $\psi \circ \psi \circ \dots \circ \psi$ длины i . Другими словами, преобразование ψ представляет собой регистр сдвига с линейной обратной связью, а индекс i указывает на количество его раундов.



Параметром используемого в качестве шифрующего преобразования $E(h, K)$ алгоритма ГОСТ 28147-89 является таблица из восьми узлов замены (S-блоков). В своей реализации я применила S-блоки, используемые ЦБ РФ.

Номер S-блока	Значение															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	4	A	9	2	D	8	0	E	6	B	1	C	7	F	5	3
2	E	B	4	C	6	D	F	A	2	3	8	1	0	7	5	9
3	5	8	1	D	A	3	4	2	E	F	C	7	6	0	9	B
4	7	D	A	1	0	8	9	F	E	4	6	C	B	2	5	3
5	6	C	7	1	5	F	D	8	4	A	9	E	0	3	B	2
6	4	B	A	0	7	2	1	D	3	6	8	5	9	C	F	E
7	D	B	4	1	3	F	5	9	0	A	E	7	6	8	2	C
8	1	F	D	0	5	7	A	4	9	2	3	E	6	B	8	C

3 Реализация алгоритма

В моей реализации для хеширования можно хешировать строку или файл. Также есть возможность задать количество раундов.

Исходный код объявления класса из «gost_hash.hpp»:

```
1 #ifndef gost_hash_hpp
2 #define gost_hash_hpp
3
4
5 typedef unsigned char byte;
6
7 typedef byte Block[32]; // Block - массив из 32 байтов (256 бит)
8 typedef byte Block32[4]; // Block32 - массив из 4 байтов (32 бита)
9
10 class Gost {
11 public:
12     Gost();
13     Gost(int user_rounds);
14     ~Gost(){};
15
16     // ГОСТ 28147-89
17     void E(byte D[], byte K[], byte R[]);
18
19     // Функция f в ГОСТ 28147-89
20     void E_f(byte A[], byte K[], byte R[]);
21
22     void f(byte H[], byte M[], byte newH[]);
23
24     // ГОСТ R 34.11-94
25     void A(byte Y[], byte R[]);
26
27     int fi(int arg);
28     void P(byte Y[], byte R[]);
29     void psi(byte arr[]);
30     void psi(byte arr[], int p);
31
32     void hash(byte* buf, int len, byte* result);
33
34 private:
35     int rounds;
36 };
37
38
39 #endif /* gost_hash_hpp */
```

Исходный код реализации класса из «gost_hash.cpp»:

```
1 #include <cstdio>
2 #include <memory.h>
```

```

3  #include "gost_hash.hpp"
4
5
6  byte S[8][16] = { // Сблоки-, используемые ЦБРФ
7  { 4, 10, 9, 2, 13, 8, 0, 14, 6, 11, 1, 12, 7, 15, 5, 3},
8  {14, 11, 4, 12, 6, 13, 15, 10, 2, 3, 8, 1, 0, 7, 5, 9},
9  { 5, 8, 1, 13, 10, 3, 4, 2, 14, 15, 12, 7, 6, 0, 9, 11},
10 { 7, 13, 10, 1, 0, 8, 9, 15, 14, 4, 6, 12, 11, 2, 5, 3},
11 { 6, 12, 7, 1, 5, 15, 13, 8, 4, 10, 9, 14, 0, 3, 11, 2},
12 { 4, 11, 10, 0, 7, 2, 1, 13, 3, 6, 8, 5, 9, 12, 15, 14},
13 {13, 11, 4, 1, 3, 15, 5, 9, 0, 10, 14, 7, 6, 8, 2, 12},
14 { 1, 15, 13, 0, 5, 7, 10, 4, 9, 2, 3, 14, 6, 11, 8, 12},
15 };
16
17
18 Gost::Gost() {
19     rounds = 0;
20 }
21
22 Gost::Gost(int user_rounds) {
23     rounds = user_rounds;
24 }
25
26 void Gost::E_f(byte A[], byte K[], byte R[]) {
27     int c = 0; //Складываем по модулю  $2^{32}$ . c - перенос следующего разряд
28     for (int i = 0; i < 4; i++) {
29         c += A[i] + K[i];
30         R[i] = c & 0xFF;
31         c >>= 8;
32     }
33
34     for (int i = 0; i < 8; i++) { // Заменяем хбитные  $4$ - кусочки согласно Сблокам-
35         int x = R[i >> 1] & ((i & 1) ? 0xF0 : 0x0F); // x - хбитный  $4$ - кусочек
36         R[i >> 1] ^= x; // Обнуляем соответствующие биты
37         x >>= (i & 1) ? 4 : 0; // сдвигаем x либона  $0$ , либона  $4$  бита влево
38         x = S[i][x]; // Заменяем согласно Сблоку-
39         R[i >> 1] |= x << ((i & 1) ? 4 : 0); //
40     }
41
42     int tmp = R[3]; // Сдвигаем на  $8$  бит ( $1$  байт) влево
43     R[3] = R[2];
44     R[2] = R[1];
45     R[1] = R[0];
46     R[0] = tmp;
47
48     tmp = R[0] >> 5; // Сдвигаем на  $3$  бита влево
49     for (int i = 1; i < 4; i++) {
50         int nTmp = R[i] >> 5;
51         R[i] = (R[i] << 3) | tmp;

```



```

52     tmp = nTmp;
53 }
54 R[0] = (R[0] << 3) | tmp;
55 }
56
57
58
59 void Gost::E(byte D[], byte K[], byte R[]) {
60     Block32 A, B; //Инициализация блоков A и B
61     for (int i = 0; i < 4; i++) {
62         A[i] = D[i];
63     }
64     for (int i = 0; i < 4; i++) {
65         B[i] = D[i + 4];
66     }
67
68
69     for (int step = 0; step < 4; step++) // K1..K32 идут в прямом порядке - 4 цикла K1..K8
70     for (int i = 0; i < rounds; i += 4) {
71         Block32 tmp;
72         E_f(A, K + i, tmp); // (K + i) - массив K с i-го элемента
73         for (int i = 0; i < 4; i++) {
74             tmp[i] ^= B[i];
75         }
76         memcpy(B, A, sizeof A);
77         memcpy(A, tmp, sizeof tmp);
78     }
79     for (int i = 0; i < 4; i++) {
80         R[i] = B[i];
81     } //Возвращаем результат
82     for (int i = 0; i < 4; i++) {
83         R[i + 4] = A[i];
84     }
85 }
86
87
88 void Gost::A(byte Y[], byte R[]) {
89     for (int i = 0; i < 24; i++) {
90         R[i] = Y[i + 8];
91     }
92     for (int i = 0; i < 8; i++) {
93         R[i + 24] = Y[i] ^ Y[i + 8];
94     }
95 }
96
97
98 int Gost::fi(int arg) { // Функция fi. Отличие от функции в статье - нумерация не 1..32, а
99     0..31
100     int i = arg & 0x03;

```

```

100     int k = arg >> 2;
101     k++;
102     return (i << 3) + k - 1;
103 }
104
105
106 void Gost::P(byte Y[], byte R[]) {
107     for (int i = 0; i < 32; i++) { R[i] = Y[fi(i)]; }
108 }
109
110 void Gost::psi(byte arr[]) {
111     byte y16[] = {0, 0};
112     y16[0] ^= arr[ 0]; y16[1] ^= arr[ 1];
113     y16[0] ^= arr[ 2]; y16[1] ^= arr[ 3];
114     y16[0] ^= arr[ 4]; y16[1] ^= arr[ 5];
115     y16[0] ^= arr[ 6]; y16[1] ^= arr[ 7];
116     y16[0] ^= arr[24]; y16[1] ^= arr[25];
117     y16[0] ^= arr[30]; y16[1] ^= arr[31];
118     for (int i = 0; i < 30; ++i) {
119         arr[i] = arr[i + 2];
120     }
121     arr[30] = y16[0]; arr[31] = y16[1];
122 }
123
124
125 void Gost::psi(byte arr[], int p) {
126     while (p--) {
127         psi(arr);
128     }
129 }
130
131 void Gost::f(byte H[], byte M[], byte newH[]) { // Функция f
132     Block C[4];
133     memset(C, 0, sizeof C);
134     C[2][ 0] = 0x00;
135     C[2][ 1] = 0xFF;
136     C[2][ 2] = 0x00;
137     C[2][ 3] = 0xFF;
138     C[2][ 4] = 0x00;
139     C[2][ 5] = 0xFF;
140     C[2][ 6] = 0x00;
141     C[2][ 7] = 0xFF;
142     C[2][ 8] = 0xFF;
143     C[2][ 9] = 0x00;
144     C[2][10] = 0xFF;
145     C[2][11] = 0x00;
146     C[2][12] = 0xFF;
147     C[2][13] = 0x00;
148     C[2][14] = 0xFF;

```

```

149 C[2][15] = 0x00;
150 C[2][16] = 0x00;
151 C[2][17] = 0xFF;
152 C[2][18] = 0xFF;
153 C[2][19] = 0x00;
154 C[2][20] = 0xFF;
155 C[2][21] = 0x00;
156 C[2][22] = 0x00;
157 C[2][23] = 0xFF;
158 C[2][24] = 0xFF;
159 C[2][25] = 0x00;
160 C[2][26] = 0x00;
161 C[2][27] = 0x00;
162 C[2][28] = 0xFF;
163 C[2][29] = 0xFF;
164 C[2][30] = 0x00;
165 C[2][31] = 0xFF;
166
167 Block U, V, W, K[4], tmp;
168 memcpy(U, H, sizeof U);
169 memcpy(V, M, sizeof V);
170 for (int i = 0; i < 32; i++) {
171     W[i] = U[i] ^ V[i];
172 }
173 P(W, K[0]);
174
175 for (int step = 1; step < 4; step++) {
176     A(U, tmp);
177     for (int i = 0; i < 32; i++) {
178         U[i] = tmp[i] ^ C[step][i];
179     }
180     A(V, tmp);
181     A(tmp, V);
182     for (int i = 0; i < 32; i++) {
183         W[i] = U[i] ^ V[i];
184     }
185     P(W, K[step]);
186 }
187
188 Block S;
189 for (int i = 0; i < rounds; i += 8) { //32
190     E(H + i, K[i >> 3], S + i);
191 }
192
193 psi(S, 12);
194 for (int i = 0; i < 32; i++) {
195     S[i] ^= M[i];
196 }
197

```

```

198     psi(S, 1 );
199     for (int i = 0; i < 32; i++) {
200         S[i] ^= H[i];
201     }
202
203     psi(S, 61);
204     memcpy(newH, S, sizeof S);
205 }
206
207
208 void Gost::hash(byte* buf, int len, byte* result) {
209     Block block, Sum, L, H, newH;
210     int pos = 0, posIB = 0;
211
212     memset(Sum, 0, sizeof Sum);
213     memset(H, 0, sizeof H);
214
215     while ((posIB < len) || pos) {
216         if (posIB < len) {
217             block[pos++] = buf[posIB++];
218         } else {
219             block[pos++] = 0;
220         }
221
222         if (pos == 32) {
223             pos = 0;
224
225             int c = 0;
226             for (int i = 0; i < 32; i++) {
227                 c += block[i] + Sum[i];
228                 Sum[i] = c & 0xFF;
229                 c >>= 8;
230             }
231
232             f(H, block, newH);
233             memcpy(H, newH, sizeof newH);
234         }
235     }
236
237     memset(L, 0, sizeof L);
238     int c = len << 3;
239     for (int i = 0; i < 32; i++) {
240         L[i] = c & 0xFF;
241         c >>= 8;
242     }
243     f(H, L, newH);
244     memcpy(H, newH, sizeof newH);
245     f(H, Sum, newH);
246     memcpy(result, newH, sizeof newH);

```

Демонстрация работы алгоритма:

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ g++ -std=c++11 gost_hash.cpp main.cpp
-o begin
```

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ ./begin
```

```
This is message,length=32 bytes
```

```
0xE64BEE65EF2EA691B86DD9BE3E00F3AAC5C6EF8F7BF84A6693BBB910A5EAAED1
```

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ cat input.txt
```

```
32
```

```
73657479622032333D6874676E656C202C6567617373656D2073692073696854
```

```
50
```

```
7365747962203035203D206874676E656C20736168206567617373656D206C616E696769726F2065687420
```

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ ./begin input.txt
```

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ cat output.txt
```

```
0xE64BEE65EF2EA691B86DD9BE3E00F3AAC5C6EF8F7BF84A6693BBB910A5EAAED1
```

```
0x789B571365162C497E0AC490F32BC6141732FC6BC23AF0003ECE3A8F179A526
```

4 Анализ алгоритма

Для анализа работы своего алгоритма я использовала «Дифференциальный криптоанализ». Он позволяет оценить качество алгоритма при различном количестве раундов.

Нужно было рассмотреть как влияет изменение количества раундов алгоритма на дифференциальную разность выходного хэша. Подсчитала количество бит выходного хэша, которое различается для двух текстов с разницей в 1 бит.

В данном случае я выбрала дифференциал $\Delta X = 1$ и проводила анализ для количества раундов, варьирующегося от 1 до классических для *ГОСТ Р 34.11-94* 32 раундов. (с шагом 8)

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

1. Сгенерировала 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 $\forall j : 1 \leq j \leq 32$:
 - 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 $\forall j : 1 \leq j \leq 32$ алгоритма H^j .

Результат вычисления:

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ g++ -std=c++11 gost_hash.cpp diff_crypto_analy
-o start
(base) MacBook-Pro-Lina:lab3 linuxoid$ ./start
Differential crypto analysis
```

```
Number of test strings: 100
Will explore round numbers from 1 to 32
statistics will be written into file: statistics
```

```
(base) MacBook-Pro-Lina:lab3 linuxoid$ cat statistics
8 128
16 128
24 129
32 127
```

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

Теперь построим график зависимости среднего количества различных битов c^j от количества раундов алгоритма j :



Исходный код отрисовки графика на языке Python:

```
1 | import matplotlib.pyplot as plt
2 |
```

```

3 filename = "statistics"
4
5 rounds = []
6 bits = []
7
8 # Read data:
9 with open(filename, "r") as f:
10     for line in f:
11         nums = line.split()
12         rounds.append(int(nums[0]))
13         bits.append(int(nums[1]))
14
15 #plotting:
16
17 plt.plot(rounds, bits, color = "r")
18 plt.title("Зависимость среднего количества различных бит от количества раундов алгоритма ГОСТР
19          34.11-94")
20 plt.xlabel("Количество раундов")
21 plt.ylabel("Количество различных бит в хэшах")
22
23 plt.legend()
24 plt.show()

```


5 Выводы

Сделав данную лабораторную работу, я получила опыт в реализации криптографического стандарта вычисления хэш-функции. Это был уже устаревший российский криптографический стандарт ГОСТ 34.11-94. Также я познакомилась с дифференциальным криптоанализом и применить его для анализа своего алгоритма.

Это лабораторная работа показалась мне сложнее предыдущих. Не сразу поняла, как мне лучше задавать раунды, так как в ГОСТе 34.11-94 он один, поэтому пришлось задавать раунды для алгоритма (ГОСТ 89), который реализован внутри стандарта 94 года. Также полезно было вновь вспомнить работу с битами: их сдвигами, сложение по модулю 2^{32} .

Список литературы

- [1] *Gost P 34.11-94*
URL: https://ru.wikipedia.org/wiki/ГОСТ_P_34.1194 (дата обращения: 10.04.2020).
- [2] *Gost 28147-89 reference*
URL: https://ru.wikipedia.org/wiki/ГОСТ_2814789 (дата обращения: 10.04.2020).
- [3] *Gost 28147-89 reference*
URL: https://ru.wikipedia.org/wiki/Криптоанализ_ГОСТ_2814789 (дата обращения: 10.04.2020).