Лабораторная работа № 3 по курсу криптографии

Выполнила студентка группы М8О-307Б *Безлуцкая Елизавета.*

Условие

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. Процесс выбора варианта требуется отразить в отчёте.

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.

Метод решения

Выбор варианта произведен с помощью библиотеки pygost:

```
>>> from pygost import gost34112012256
>>> gost34112012256.new("Безлуцкая Елизавета Николаевна".encode()).digest();
b'\xc3\xcd\xc1\n\x1672%\xa4\x15\x1fn\x89O\x80B.\xb7L\x1as\xb7\x16\xf1\xbb\xa7\x9
7X\xea7\xe3\xc3'
```

Вариант 3: BLAKE Для реализации я выбрала BLAKE-256. Функция работает следующим образом:

1 этап: padding

Сообщение дополняется функцией **padding** данными для кратности 512 битам (64 байтам): сначала – битами, так, что его длина становится по модулю 512 равной 447: сначала добавляется 1, затем необходимое количество нолей. После этого прибавляется ещё одна 1 и 64-битное представление длины сообщения l от старшего бита к младшему. Таким образом, длина сообщения становится кратной 512. **Padding** гарантирует, что длина сообщения станет кратной 512 битам.

2 этап: compress

Затем, блок за блоком, сообщение обрабатывает функция сжатия **compression**. Она принимает на вход:

- Переменные цепочки **h = h0,...,h7** (8 слов);
- Блок сообщения **m = m0,...,m15**;
- Значение соли **s = s0,...,s3**;
- Значение счётчика **t = t0,t1**.

Таким образом, на вход ей подаётся 30 слов (8+16+4+2=30, 30*4 = 120 байт = 960 бит). Возвращает функция сжатия только новое значение переменных цепочки: **h' = h'0,...,h'7**.

Константы
Существуют начальные константы

INITIAL VALUES (IV):
IV0 = 6A09E667 IV1 = BB67AE85
IV2 = 3C6EF372 IV3 = A54FF53A
IV4 = 510E527F IV5 = 9B05688C
IV6 = 1F83D9AB IV7 = 5BE0CD19

16 констант (Первые цифры числа пи):
c0 = 243F6A88 c1 = 85A308D3
c2 = 13198A2E c3 = 03707344
c4 = A4093822 c5 = 299F31D0
c6 = 082EFA98 c7 = EC4E6C89
c8 = 452821E6 c9 = 38D01377
c10 = BE5466CF c11 = 34E90C6C
c12 = C0AC29B7 c13 = C97C50DD
c14 = 3F84D5B5 c15 = B5470917

перестановки 0,...,15 (используются во всех функциях BLAKE):

$$\sigma = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 14 & 10 & 4 & 8 & 9 & 15 & 13 & 6 & 1 & 12 & 0 & 2 & 11 & 7 & 5 & 3 \\ 11 & 8 & 12 & 0 & 5 & 2 & 15 & 13 & 10 & 14 & 3 & 6 & 7 & 1 & 9 & 4 \\ 7 & 9 & 3 & 1 & 13 & 12 & 11 & 14 & 2 & 6 & 5 & 10 & 4 & 0 & 15 & 8 \\ 9 & 0 & 5 & 7 & 2 & 4 & 10 & 15 & 14 & 1 & 11 & 12 & 6 & 8 & 3 & 13 \\ 2 & 12 & 6 & 10 & 0 & 11 & 8 & 3 & 4 & 13 & 7 & 5 & 15 & 14 & 1 & 9 \\ 12 & 5 & 1 & 15 & 14 & 13 & 4 & 10 & 0 & 7 & 6 & 3 & 9 & 2 & 8 & 11 \\ 13 & 11 & 7 & 14 & 12 & 1 & 3 & 9 & 5 & 0 & 15 & 4 & 8 & 6 & 2 & 10 \\ 6 & 15 & 14 & 9 & 11 & 3 & 0 & 8 & 12 & 2 & 13 & 7 & 1 & 4 & 10 & 5 \\ 10 & 2 & 8 & 4 & 7 & 6 & 1 & 5 & 15 & 11 & 9 & 14 & 3 & 12 & 13 & 0 \end{bmatrix}$$

Инициализация

16 переменных, **v0,...,v15**, описывающих текущее состояние v, инициализируются начальными значениями в зависимости от входных данных и представлены в виде матрицы 4x4:

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix} \leftarrow \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{bmatrix}$$

Раундовая функция

После того, как состояние **v** инициализировано, запускается серия из 14 раундов. Раунд — это операция над состоянием v, которая производит вычисления, разбитые на следующие блоки:

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

на r-ом раунде блок вычислений $G_i(a, b, c, d)$ работает следующим образом:
$j \leftarrow \sigma_{r\%10}[2 \cdot i]$
$j \leftarrow \sigma_{r\%10}[2 \cdot i + 1]$
$a \leftarrow a + b + (m_j \oplus c_k)$
$d \leftarrow (d \oplus a) >>> 16$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) >>> 12$
$a \leftarrow a + b + (m_k \oplus c_j)$
$d \leftarrow (d \oplus a) >>> 8$
$c \leftarrow c + d$

$b \leftarrow (b \oplus c) >>> 7$

Последний шаг

После всех раундов новое значение переменных цепочки $h'_0, ..., h'_7$ вычисляется из переменных $v_0, ..., v_{15}$ матрицы состояния, входных переменных h и из соли s:
$h'_0 \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v8$
$h'_1 \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v9$
$h'_2 \leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v10$
$h'_3 \leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v11$
$h'_4 \leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v12$
$h'_5 \leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v13$
$h'_6 \leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v14$
$h'_7 \leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v15$

Реализация алгоритма отчасти была позаимствована из `https://www.springer.com/gp/book/9783662447567`

Исходный код

blake256.h

```c
#ifndef BLAKE256_BLAKE256_H
#define BLAKE256_BLAKE256_H

#include <string.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

uint32_t BUFFER_SIZE = 1024;

#define U8TO32_BIG(p) \
(((uint32_t)((p)[0]) << 24) | ((uint32_t)((p)[1]) << 16) | \
((uint32_t)((p)[2]) << 8) | ((uint32_t)((p)[3]) ))

#define U32TO8_BIG(p, v) \
(p)[0] = (uint8_t)((v) >> 24);(p)[1] = (uint8_t)((v) >> 16);\
(p)[2] = (uint8_t)((v) >> 8);(p)[3] = (uint8_t)((v) );

typedef struct {
    uint32_t h[8], s[4], t[2];
    int buflen, nullt;
    uint8_t buf[64];
} state256;

const uint8_t SIGMA[][16] = {
```

```
        {0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15},
        {14, 10, 4,  8,  9,  15, 13, 6,  1,  12, 0,  2,  11, 7,  5,  3},
        {11, 8,  12, 0,  5,  2,  15, 13, 10, 14, 3,  6,  7,  1,  9,  4},
        {7,  9,  3,  1,  13, 12, 11, 14, 2,  6,  5,  10, 4,  0,  15, 8},
        {9,  0,  5,  7,  2,  4,  10, 15, 14, 1,  11, 12, 6,  8,  3,  13},
        {2,  12, 6,  10, 0,  11, 8,  3,  4,  13, 7,  5,  15, 14, 1,  9},
        {12, 5,  1,  15, 14, 13, 4,  10, 0,  7,  6,  3,  9,  2,  8,  11},
        {13, 11, 7,  14, 12, 1,  3,  9,  5,  0,  15, 4,  8,  6,  2,  10},
        {6,  15, 14, 9,  11, 3,  0,  8,  12, 2,  13, 7,  1,  4,  10, 5},
        {10, 2,  8,  4,  7,  6,  1,  5,  15, 11, 9,  14, 3,  12, 13, 0},
        {0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15},
        {14, 10, 4,  8,  9,  15, 13, 6,  1,  12, 0,  2,  11, 7,  5,  3},
        {11, 8,  12, 0,  5,  2,  15, 13, 10, 14, 3,  6,  7,  1,  9,  4},
        {7,  9,  3,  1,  13, 12, 11, 14, 2,  6,  5,  10, 4,  0,  15, 8},
        {9,  0,  5,  7,  2,  4,  10, 15, 14, 1,  11, 12, 6,  8,  3,  13},
        {2,  12, 6,  10, 0,  11, 8,  3,  4,  13, 7,  5,  15, 14, 1,  9}};

const uint32_t U[16] = {
        0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,
        0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,
        0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
        0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917};

static const uint8_t PADDING[129] = {
        0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

uint32_t ROUNDS = 14;

void Blake256Compress(state256 *S, const uint8_t *block) {
    uint32_t v[16], m[16], i;
#define ROT(x, n) (((x)<<(32-n))|( (x)>>(n)))
#define G(a, b, c, d, e) \
v[a] += (m[SIGMA[i][e]] ^ U[SIGMA[i][e+1]]) + v[b]; \
v[d] = ROT( v[d] ^ v[a],16); \
v[c] += v[d]; \
v[b] = ROT( v[b] ^ v[c],12); \
v[a] += (m[SIGMA[i][e+1]] ^ U[SIGMA[i][e]])+v[b]; \
v[d] = ROT( v[d] ^ v[a], 8); \
v[c] += v[d]; \
v[b] = ROT( v[b] ^ v[c], 7);
    for (i = 0; i < 16; ++i) m[i] = U8TO32_BIG(block + i * 4);
    for (i = 0; i < 8; ++i) v[i] = S->h[i];
    v[8] = S->s[0] ^ U[0];
    v[9] = S->s[1] ^ U[1];
```

```
78      v[10] = S->s[2] ^ U[2];
79      v[11] = S->s[3] ^ U[3];
80      v[12] = U[4];
81      v[13] = U[5];
82      v[14] = U[6];
83      v[15] = U[7];
84
85      if (!S->nullt) {
86          v[12] ^= S->t[0];
87          v[13] ^= S->t[0];
88          v[14] ^= S->t[1];
89          v[15] ^= S->t[1];
90      }
91      for (i = 0; i < ROUNDS; ++i) {
92          G(0, 4, 8, 12, 0);
93          G(1, 5, 9, 13, 2);
94          G(2, 6, 10, 14, 4);
95          G(3, 7, 11, 15, 6);
96          G(0, 5, 10, 15, 8);
97          G(1, 6, 11, 12, 10);
98          G(2, 7, 8, 13, 12);
99          G(3, 4, 9, 14, 14);
100     }
101     for (i = 0; i < 16; ++i) S->h[i % 8] ^= v[i];
102     for (i = 0; i < 8; ++i) S->h[i] ^= S->s[i % 4];
103 }
104
105 void Blake256Init(state256 *S) {
106     S->h[0] = 0x6a09e667;
107     S->h[1] = 0xbb67ae85;
108     S->h[2] = 0x3c6ef372;
109     S->h[3] = 0xa54ff53a;
110     S->h[4] = 0x510e527f;
111     S->h[5] = 0x9b05688c;
112     S->h[6] = 0x1f83d9ab;
113     S->h[7] = 0x5be0cd19;
114     S->t[0] = S->t[1] = S->buflen = S->nullt = 0;
115     S->s[0] = S->s[1] = S->s[2] = S->s[3] = 0;
116 }
117
118 void Blake256Update(state256 *S, uint8_t *in, uint64_t inlen) {
119     int left = S->buflen;
120     int fill = 64 - left;
121     if (left && (inlen >= fill)) {
122         memcpy((void *) (S->buf + left), (void *) in, fill);
123         S->t[0] += 512;
124         if (S->t[0] == 0) S->t[1]++;
125         Blake256Compress(S, S->buf);
126         in += fill;
127         inlen -= fill;
128         left = 0;
```

```
129     }
130     while (inlen >= 64) {
131         S->t[0] += 512;
132         if (S->t[0] == 0) S->t[1]++;
133         Blake256Compress(S, in);
134         in += 64;
135         inlen -= 64;
136     }
137     if (inlen > 0) {
138         memcpy((void *) (S->buf + left), (void *) in, (size_t) inlen);
139         S->buflen = left + (int) inlen;
140     } else S->buflen = 0;
141 }
142 
143 void Blake256Final(state256 *S, uint8_t *out) {
144     uint8_t msglen[8], zo = 0x01, oo = 0x81;
145     uint32_t lo = S->t[0] + (S->buflen << 3), hi = S->t[1];
146     if (lo < (S->buflen << 3)) hi++;
147     U32TO8_BIG(msglen + 0, hi);
148     U32TO8_BIG(msglen + 4, lo);
149     if (S->buflen == 55) {
150         S->t[0] -= 8;
151         Blake256Update(S, &oo, 1);
152     } else {
153         if (S->buflen < 55) {
154             if (!S->buflen) S->nullt = 1;
155             S->t[0] -= 440 - (S->buflen << 3);
156             Blake256Update(S, PADDING, 55 - S->buflen);
157         } else {
158             S->t[0] -= 512 - (S->buflen << 3);
159             Blake256Update(S, PADDING, 64 - S->buflen);
160 
161             S->t[0] -= 440;
162             Blake256Update(S, PADDING + 1, 55);
163             S->nullt = 1;
164         }
165         Blake256Update(S, &zo, 1);
166         S->t[0] -= 8;
167     }
168     S->t[0] -= 64;
169     Blake256Update(S, msglen, 8);
170     U32TO8_BIG(out + 0, S->h[0]);
171     U32TO8_BIG(out + 4, S->h[1]);
172     U32TO8_BIG(out + 8, S->h[2]);
173     U32TO8_BIG(out + 12, S->h[3]);
174     U32TO8_BIG(out + 16, S->h[4]);
175     U32TO8_BIG(out + 20, S->h[5]);
176     U32TO8_BIG(out + 24, S->h[6]);
177     U32TO8_BIG(out + 28, S->h[7]);
178 }
179 
```

```
180 void Blake256Hash(uint8_t *out, uint8_t *in, uint64_t inlen) {
181     state256 S;
182     Blake256Init(&S);
183     Blake256Update(&S, in, inlen);
184     Blake256Final(&S, out);
185 }
186 
187 #endif //BLAKE256_BLAKE256_H
```

main.c

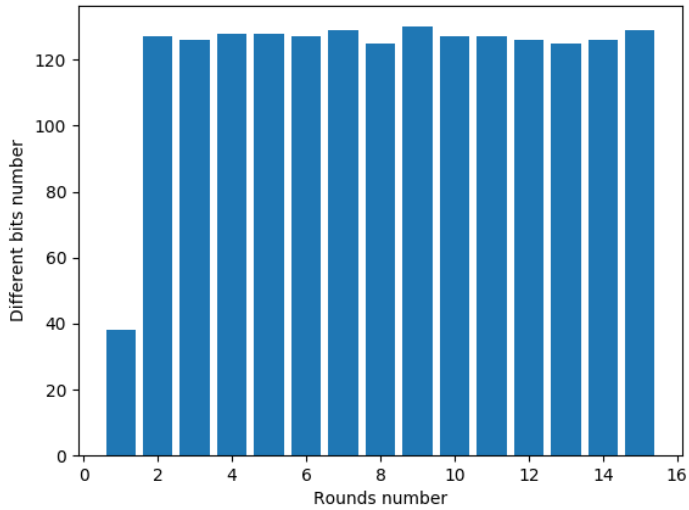
```
1  #include "blake256.h"
2  
3  int main() {
4      printf("Input message: ");
5      uint8_t data[BUFFER_SIZE];
6  
7      uint32_t i = 0;
8      uint8_t c = getchar();
9      while (c != '\n') {
10         data[i] = c;
11         ++i;
12         c = getchar();
13     }
14 
15     uint8_t digest[33];
16     Blake256Hash(digest, data, i);
17 
18     printf("Digest: ");
19     for (int i = 0; i < 32; ++i) {
20         printf("%02x", digest[i]);
21     }
22 
23     return 0;
24 }
```

Консоль

```
->blake256 ./blake
Input message:
Digest: 716f6e863f744b9ac22c97ec7b76ea5f5908bc5b2f67c61510bfc4751384ea7a
->blake256 ./blake
Input message: The quick brown fox jumps over the lazy dog
Digest: 7576698ee9cad30173080678e5965916adbb11cb5245d386bf1ffda1cb26c9d7
->blake256 ./blake
Input message: BLAKE
Digest: 07663e00cf96fbc136cf7b1ee099c95346ba3920893d18cc8851f22ee2e36aa6
```

Далее был произведен дифференциальный криптоанализ к полученной хеш-функции. Он основан на анализе пар сообщений, между которыми существует определенная разность. Я произвожу хеширование с количеством раундом от 1 до 15. На каждом шаге для начала генерирую сообщение-строку, затем получаю другую строку инвертированием последнего бита исходной. Считаю хеши строк, считаю количество отличающихся в них битов. Проделываю данную операцию 10 раз и считаю среднюю разницу для каждого количества раундов.

Кол-во раундов	Число изменных бит
1	38
2	127
3	126
4	128
5	128
6	127
7	129
8	125
0 9	130
10	127
11	127
12	126
13	125
14	126
15	129



Такой анализ позволяет увидеть, насколько меняется хеш при минимальном изменении исходного сообщения. При 256-битном хеше уже со второго раунда меняется половина бит!

Исходный код

da.c

```
#include "blake256.h"

uint16_t SIZE = 101;
uint16_t IT_NUM = 10;

void str_to_bin(uint8_t in[], uint8_t bin[][9]) {

    for (int i = 0; i < SIZE; ++i) {
        for (int j = 7; j >= 0; --j) {
            if (in[i] & (1 << j)) {
                bin[i][7 - j] = '1';
            } else {
                bin[i][7 - j] = '0';
            }
            bin[i][8] = '\0';
        }
    }
}

void rand_str(char *dest, size_t length) {
    char charset[] = "0123456789"
                     "abcdefghijklmnopqrstuvwxyz"
                     "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    while (length-- > 0) {
        size_t index = (double) rand() / RAND_MAX * (sizeof charset - 1);
        *dest++ = charset[index];
    }
    *dest = '\0';
}

int main() {
    for (int i = 1; i <= 15; ++i) {
        printf("_____\n");
        ROUNDS = i;
        printf("Number of rounds: %d\n", ROUNDS);

        uint32_t diff = 0;
        for (int j = 0; j < IT_NUM; ++j) {
            uint8_t data1[SIZE];
            rand_str(data1, SIZE - 1);

            char bin[SIZE][9];
            str_to_bin(data1, bin);
            if (bin[SIZE - 2][7] == '0') {
                bin[SIZE - 2][7] = '1';
            } else {
                bin[SIZE - 2][7] = '0';
```

```
49              }
50
51              uint8_t data2[SIZE];
52              for (int l = 0; l < SIZE; ++l) {
53                  char c = strtol(bin[l], 0, 2);
54                  data2[l] = c;
55              }
56
57              uint8_t digest1[33];
58              Blake256Hash(digest1, data1, SIZE);
59              char binDigest1[33][9];
60              str_to_bin(digest1, binDigest1);
61
62              uint8_t digest2[33];
63              Blake256Hash(digest2, data2, SIZE);
64              char binDigest2[33][9];
65              str_to_bin(digest2, binDigest2);
66
67              for (int l = 0; l < 32; ++l) {
68                  for (int k = 0; k < 9; ++k) {
69                      if (binDigest1[l][k] != binDigest2[l][k]) {
70                          ++diff;
71                      }
72                  }
73              }
74          }
75          printf("Number of different bits: %d\n", diff / IT_NUM);
76      }
77
78      return 0;
79  }
```

Консоль

```
->blake256 ./da
_____________________________________
Number of rounds: 1
Number of different bits: 38
_____________________________________
Number of rounds: 2
Number of different bits: 127
_____________________________________
Number of rounds: 3
Number of different bits: 126
_____________________________________
Number of rounds: 4
Number of different bits: 128
```

```
Number of rounds: 5
Number of different bits: 128
```

```
Number of rounds: 6
Number of different bits: 127
```

```
Number of rounds: 7
Number of different bits: 129
```

```
Number of rounds: 8
Number of different bits: 125
```

```
Number of rounds: 9
Number of different bits: 130
```

```
Number of rounds: 10
Number of different bits: 127
```

```
Number of rounds: 11
Number of different bits: 127
```

```
Number of rounds: 12
Number of different bits: 126
```

```
Number of rounds: 13
Number of different bits: 125
```

```
Number of rounds: 14
Number of different bits: 126
```

```
Number of rounds: 15
Number of different bits: 129
```

Выводы

Хеш-функция BLAKE является достаточно криптостойкой, а именно обеспечивает стойкость к атакам второго рода, защиту от коллизий. Алгоритм сжатия является модифицированной версией хорошо параллелизируемого поточного шифра ChaCha, чья безопасность тщательно проанализирована. Интересно, что дифференциальный анализ показал значительные результаты уже на втором раунде, хотя SHA-1 для этого требует, например, не менее 20 раундов, что говорит о явном преимуществе BLAKE. Следует отметить, что на практике используется высокоразвитая версия BLAKE – BLAKE2, где главным образом улучшено быстродействие.