

1. Цель работы

Реализовать алгоритм шифрования Twofish.

2. Описание алгоритма

2.1. Twofish

Twofish — итеративный блочный алгоритм с длиной информационного блока 128 бит. Длина используемого ключа — 128, 192 или 256 бит. В основе данного алгоритма лежит сеть Файстела. Отличительной особенностью является наличие циклических сдвигов. Количество раундов — 16. На вход алгоритма подается 128 бит данных. Эти данные разделяются на 4 блока по 32 бита. Затем они складываются по модулю два с четырьмя 32-битными ключами. Данная процедура называется входным отбеливанием (input whitening). Далее эти данные попадают в цикл преобразований, длящийся 16 раундов, где два левых слова (по 32 бита каждое) подаются на вход функции F . Результаты преобразований складываются по модулю два с правыми словами. Правые и левые части меняются местами, и происходит еще 15 раундов. В заключение производится дополнительная перестановка, а на выходе данные складываются по модулю два с 32-битными ключами (выходное отбеливание или output whitening). Данную процедуру можно представить также в виде следующих операций. Пусть p_0, p_1, \dots, p_{15} — байты 128-битного блока исходного сообщения, P_0, P_1, P_2, P_3 — 32-битные слова, составленные из блоков исходной информации:

$$P_i = \sum_{j=0}^3 p_{4i+j} \cdot 2^{8j}, i = 0, \dots, 3$$

Результат первоначального сложения с блоками ключа:

$$R_{0i} = P_i \otimes K_i, i = 0, \dots, 3$$

Далее выполняются преобразования в 16 циклах ($r = 0, \dots, 15$):

$$(F_{r,0}, F_{r,1}) = F(R_{r,0}, R_{r,1}, r);$$

$$R_{r+1,0} = \text{ЦСП}(R_{r,2} \otimes F_{r,0}, 1);$$

$$R_{r+1,1} = \text{ЦСЛ}(R_{r,3}, 1) \otimes F_{r,1};$$

$$R_{r+1,2} = R_{r,0};$$

$$R_{r+1,3} = R_{r,1};$$

где ЦСЛ и ЦСП — функции циклического сдвига влево и вправо, соответственно (второй аргумент указывает, на сколько бит необходимо сдвинуть)

Выходное отбеливание:

$$C_i = R_{16,(i+2) \bmod 4} \otimes K_{i+4}, i = 0, \dots, 3$$

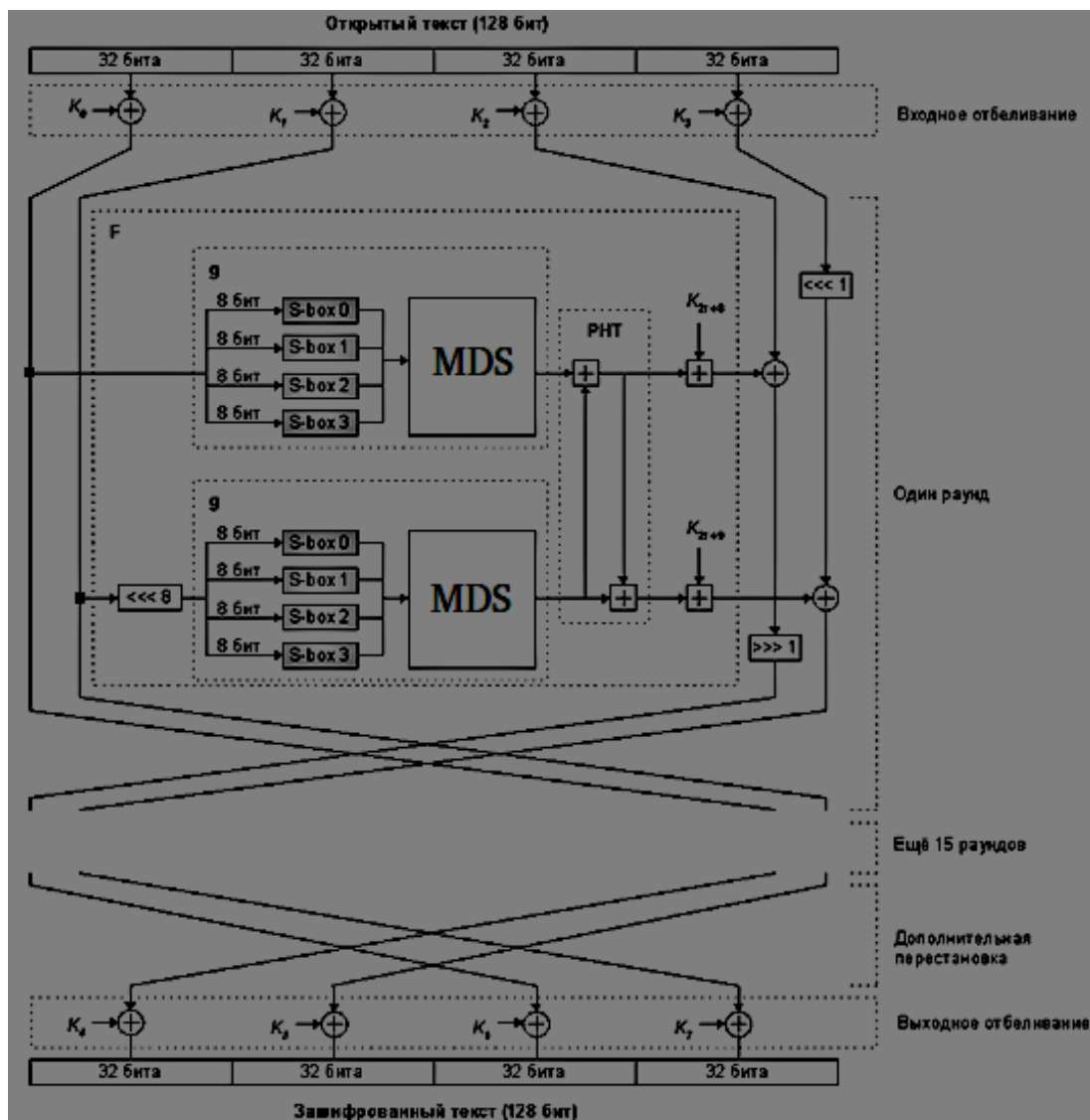


Рис. 1 Схема алгоритма Twofish

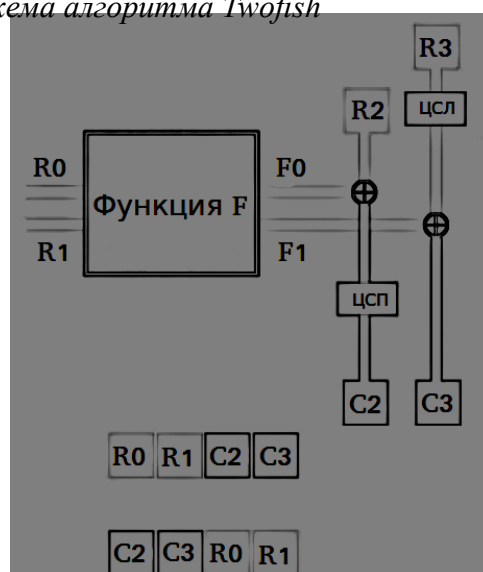


Рис. 2 Схема одного раунда

2.2. Функция F

Функция F — 64-битная перестановка, зависящая от ключа. Аргументы ее — два 32-битных слова R_0 и R_1 и номер раунда r , который определяет выбор подключей. R_0 подвергается преобразованию, определяемому функцией g , результатом является блок T_0 . R_1 сначала сдвигается на 8 бит влево, а затем также преобразуется в функции g , результатом является блок T_1 . Далее T_0 и T_1 трансформируются с помощью псевдоадаморова преобразования (ПАП или РНТ — Pseudo-Hadamar Transform) и складываются по модулю 2^{32} с соответствующими блоками ключа, которые определяются номером раунда. Результатом функции F будут являться блоки F_0 и F_1 . Формально функцию F можно записать в виде:

$$T_0 = g(R_0);$$

$$T_1 = g(\text{ЦСЛ}(R_1, 8));$$

$$F_0 = (T_0 + T_1 + K_{2r+8}) \bmod 2^{32};$$

$$F_1 = (T_0 + 2T_1 + K_{2r+9}) \bmod 2^{32};$$

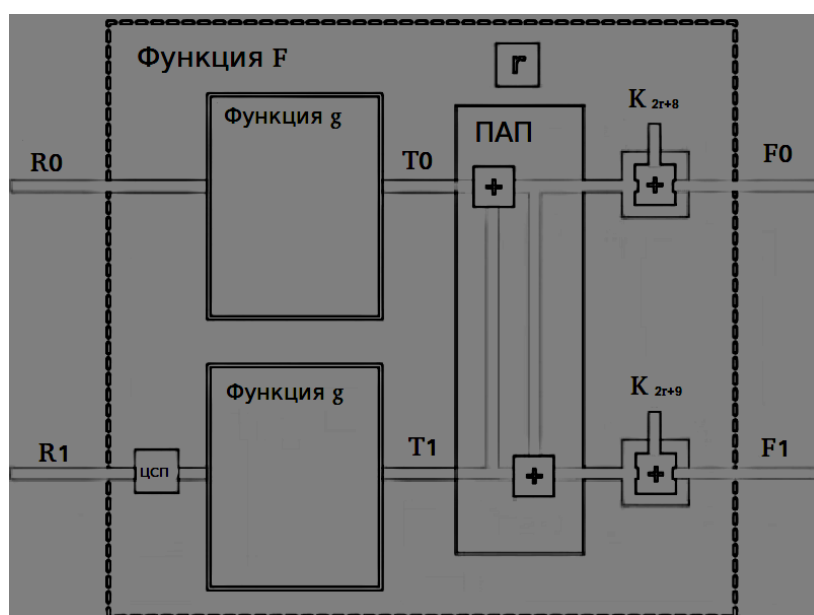


Рис. 3 Схема функции F

2.3. Преобразование g

Данное преобразование является «сердцем» алгоритма Twofish. Исходный 32-битный блок делится на четыре байта. Далее каждый байт подвергается преобразованию в соответствующей S-Box, на выходе которого также 8-битный блок. Эти блоки представляются как вектор длины 4 над $GF(2^8)$ с примитивным многочленом $m(x) = x^8 + x^6 + x^5 + x^3 + 1$, который умножается на МДР-матрицу (все операции выполняются в поле $GF(2^8)$). МДР-матрица является порождающей матрицей

специального класса помехоустойчивых кодов, делимых кодов с максимальным расстоянием. После умножения на матрицу получается 32-битное слово, которое и является результатом преобразования g . Формально преобразование g можно записать следующим образом:

$$x_i = \lfloor X/2^{8i} \rfloor \bmod 2^8, i = 0, \dots, 3;$$

$$y_i = s_i[x_i], i = 0, \dots, 3;$$

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \text{МДР} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix};$$

$$\text{МДР} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix};$$

где элементы матрицы МДР представлены в шестнадцатеричной системе счисления.

$$Z = \sum_{i=0}^3 z_i \cdot 2^{8i};$$

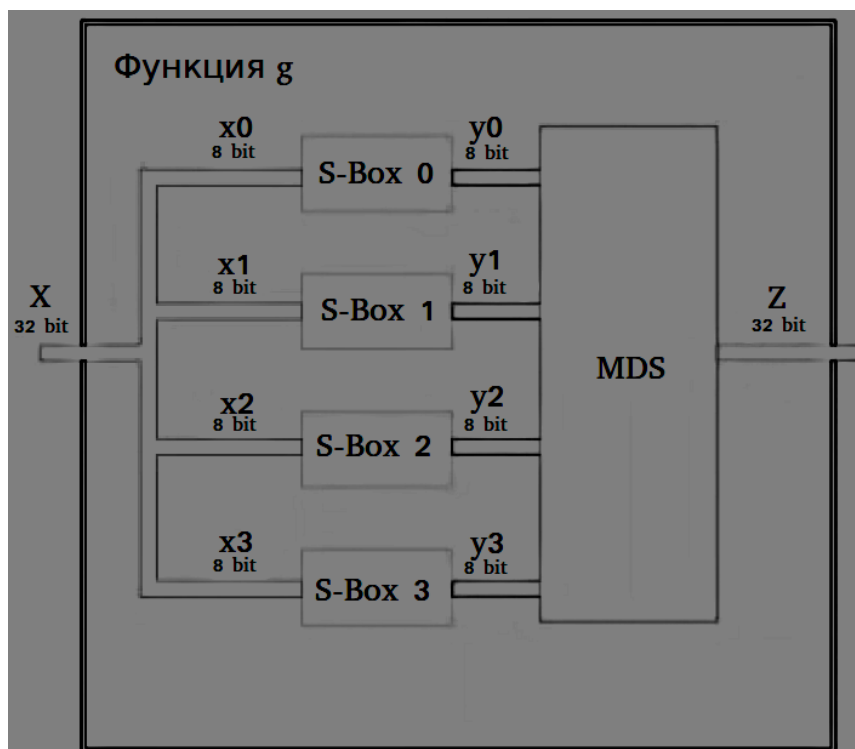


Рис. 4 Схема преобразования g

2.4. Генерация ключей

Процедура генерации ключей обеспечивает создание 40 слов расширенного ключа и четырех зависимых от ключа S -Box, используемых в функции g . Если длина введенного

ключа не совпадает с одним из трех возможных, то ключ дополняется до ближайшего нулевыми битами.

Исходный ключ делится на равные части по 8 байт. Каждая часть умножается на фиксированную матрицу, полученную из порождающей матрицы помехоустойчивого кода Рида-Соломона над $GF(2^8)$ с примитивным многочленом $m(x) = x^8 + x^6 + x^3 + x^2 + 1$. В результате умножения будет получен вектор, который можно интерпретировать как 32-битный блок. Таким образом будут получены, так называемые, S-ключи, которые фиксированы на протяжении всего алгоритма.

$$PC = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix};$$

где элементы PC-матрицы представлены в шестнадцатеричной системе счисления.

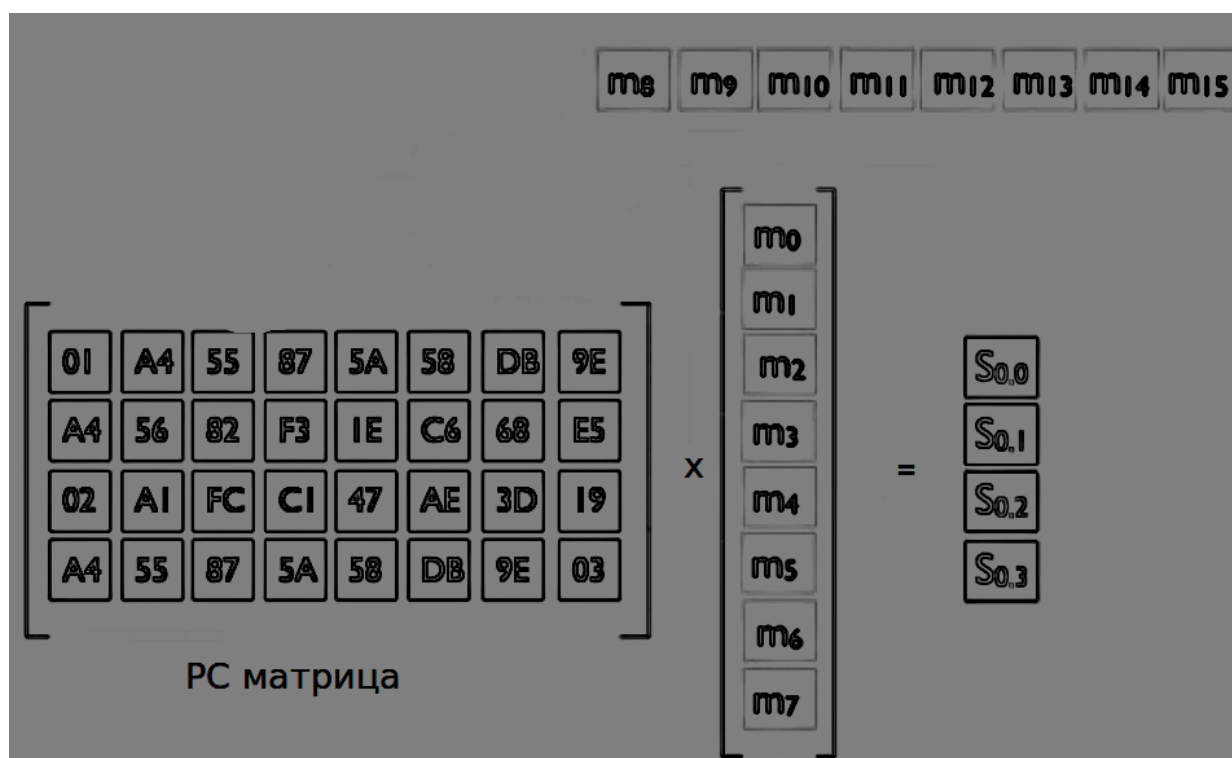


Рис. 5 Пример получения S-ключей для исходного ключа с длиной 128 бит

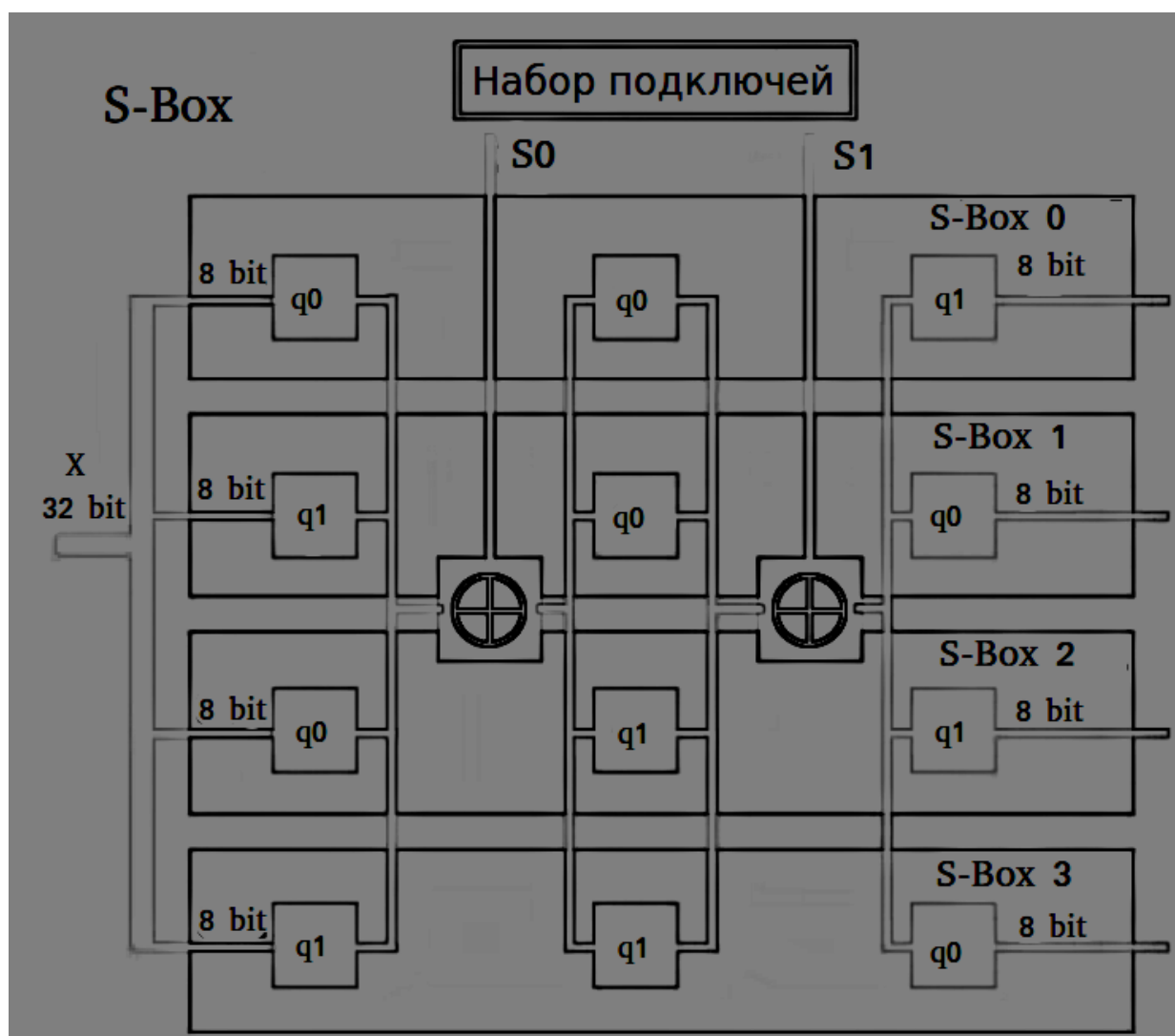


Рис. 6 Применение S-ключей

Чтобы получить, так называемые, *М*-ключи, необходимо разделить исходный ключ на равные части по 4 байта и пронумеровать по порядку, начиная с нуля. Четные части необходимо объединить в один блок, а нечетные в другой (См. пример на рис. 7). *М*-ключи будут применены для формирования раундовых ключей K . На рис. 8 показана полная схема функции F , где h — формирование раундовых ключей K (четные и нечетные ключи формируются по-разному — разница в *М*-ключах и сдвигах); g — преобразование g , описанное ранее, где используются *S*-ключи. Данный пример приведен для 128-битного ключа. Если ключ будет длиннее, то увеличится количество перестановок в соответствии с рис. 9, где L_i — либо *S*-ключ, либо *М*-ключ.



Рис. 7 Пример получения M-ключей

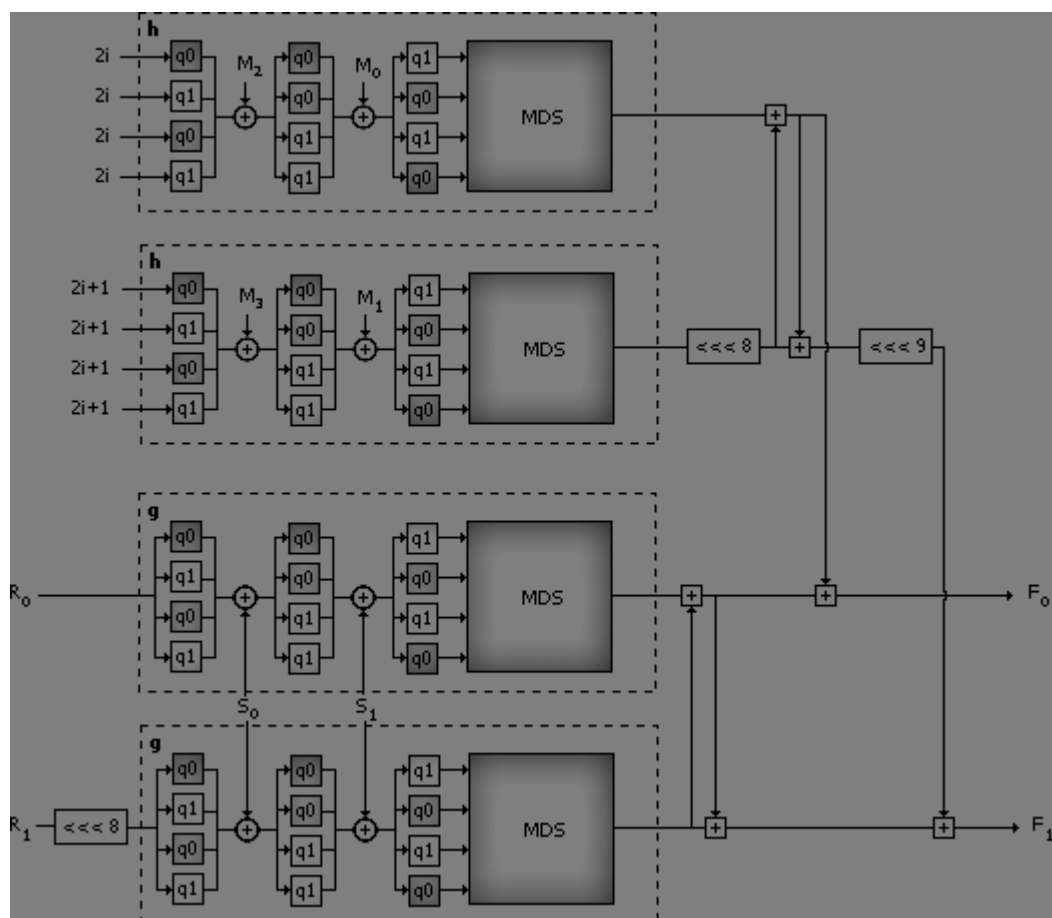


Рис. 8 Полная схема функции F

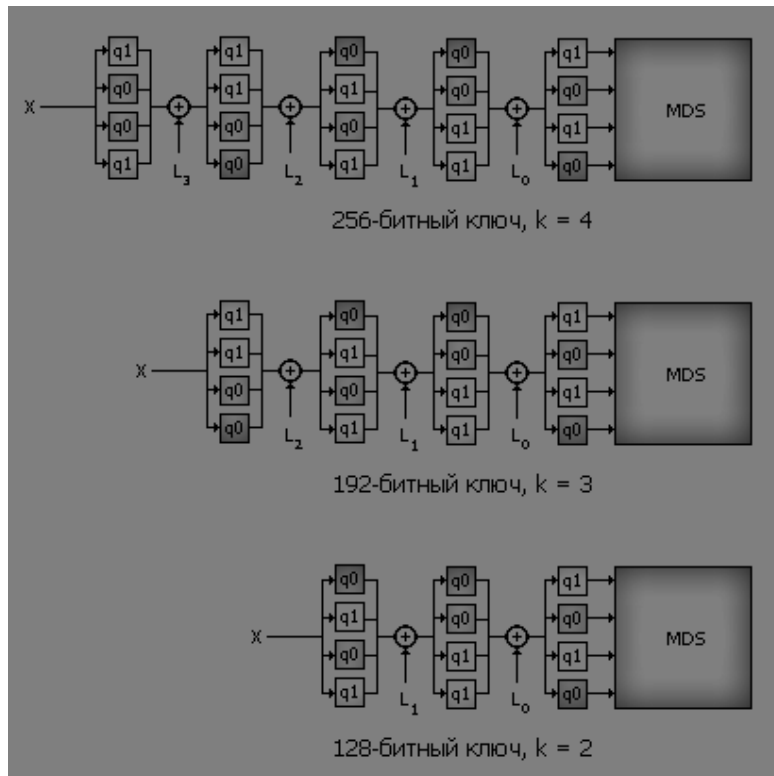


Рис. 9 Перестановки в зависимости от длины исходного ключа

2.5. Перестановки q_0 и q_1

q_0 и q_1 — фиксированные перестановки 8 битов входного байта x .

Байт x разбивается на две 4-битные половинки a_0 и b_0 , над которыми необходимо произвести следующие операции:

$$\begin{aligned}
 a_0 &= x/16; \\
 b_0 &= x \bmod 16; \\
 a_1 &= a_0 \otimes b_0; \\
 b_1 &= a_0 \otimes \text{ЦСП}_4(b_0, 1) \otimes 8a_0 \bmod 16; \\
 a_2 &= t_0[a_1]; \\
 b_2 &= t_1[b_1]; \\
 a_3 &= a_2 \otimes b_2; \\
 b_3 &= a_2 \otimes \text{ЦСП}_4(b_2, 1) \otimes 8a_2 \bmod 16; \\
 a_4 &= t_2[a_3]; \\
 b_4 &= t_3[b_3]; \\
 y &= 16b_4 + a_4;
 \end{aligned}$$

Результатом преобразований будет y . Здесь ЦСП_4 — 4-битный циклический сдвиг вправо. На рис. 10 приведена схема q -перестановки. q_0 и q_1 отличаются лишь в фиксированных таблицах со строками t_0, t_1, t_2, t_3 .

Для q_0 векторы имеет вид:

$t_0 = [8\ 1\ 7\ D\ 6\ F\ 3\ 2\ 0\ B\ 5\ 9\ E\ C\ A\ 4]$
 $t_1 = [E\ C\ B\ 8\ 1\ 2\ 3\ 5\ F\ 4\ A\ 6\ 7\ 0\ 9\ D]$
 $t_2 = [B\ A\ 5\ E\ 6\ D\ 9\ 0\ C\ 8\ F\ 3\ 2\ 4\ 7\ 1]$
 $t_3 = [D\ 7\ F\ 4\ 1\ 2\ 6\ E\ 9\ B\ 3\ 0\ 8\ 5\ C\ A]$

Для q_1 векторы имеет вид:

$t_0 = [2\ 8\ B\ D\ F\ 7\ 6\ E\ 3\ 1\ 9\ 4\ 0\ A\ C\ 5]$
 $t_1 = [1\ E\ 2\ B\ 4\ C\ 3\ 7\ 6\ D\ A\ 5\ F\ 9\ 0\ 8]$
 $t_2 = [4\ C\ 7\ 5\ 1\ 6\ 9\ A\ 0\ E\ D\ 8\ 2\ B\ 3\ F]$
 $t_3 = [B\ 9\ 5\ 1\ C\ 3\ D\ E\ 6\ 4\ 7\ F\ 2\ 0\ 8\ A]$

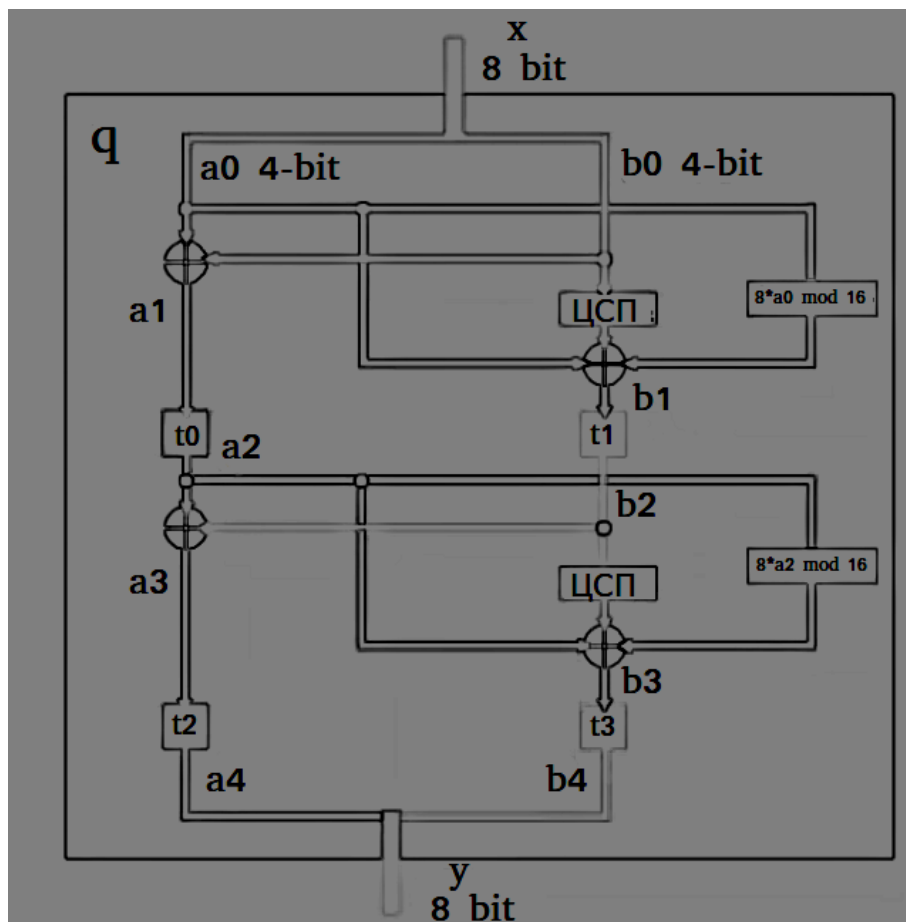


Рис. 10 Схема q -перестановки

3. Описание реализации алгоритма

3.1. Класс Twofish

Данный класс является основным. В конструктор класса передаются ключ и данные. Содержит методы:

```
public void encrypt(String filename);
```

Метод для шифрования. В параметрах указывается название файла, куда записать результат. Далее представлен код с выполнением 16 раундов.

```

for (int round = 0; round < this.ROUND_COUNT; round++) {
    if (round % 2 == 0) {
        int[] F = FFunction.run(R[0], R[1], round);
        R[2] = CyclicShift.toRight((F[0] ^ R[2]), Integer.SIZE, 1);
        R[3] = F[1] ^ (CyclicShift.toLeft(R[3], Integer.SIZE, 1));
    } else {
        int[] F = FFunction.run(R[2], R[3], round);
        R[0] = CyclicShift.toRight((F[0] ^ R[0]), Integer.SIZE, 1);
        R[1] = F[1] ^ (CyclicShift.toLeft(R[1], Integer.SIZE, 1));
    }
}
}

```

```

public void decrypt(String filename);

```

Метод для расшифрования. В качестве параметра указывается название файла, куда необходимо записать результат. Отличие в цикле лишь в том, что раунды выполняются с конца, чтобы ключи применялись в обратном порядке.

3.2. Классы InputWhitening/OutputWhitening

Данные классы предназначены для выполнения входных и выходных преобразований, т.е. для входных блоков данных выполняется XOR с соответствующими алгоритму подключами.

Для InputWhitening:

```

for (int i = 0; i < INT_BLOCK_COUNT; i++) {
    result[i] = dataBlock[i] ^ Twofish.getKSubkey(i);
}

```

Для OutputWhitening:

```

for (int i = 0; i < INT_BLOCK_COUNT; i++) {
    result[i] = dataBlock[i] ^ Twofish.getKSubkey(4+i);
}

```

3.3. Класс FFunction

Класс является аналогом функции F раунда в алгоритме. Имеет статический метод, принимающий на вход блоки R_0 и R_1 , а также номер раунда. $MODUL = 0x100000000L$.

```

public static int[] run(int R0, int R1, int round) {
    int[] result = new int[RESULT_BLOCK_COUNT];
    R1 = CyclicShift.toLeft(R1, Integer.SIZE, 8);
    int T0 = GFunction.run(R0);
    int T1 = GFunction.run(R1);
    PHT pht = new PHT(T0, T1);
}

```

```

        pht.directTransformation();
        int A = pht.getA();
        int B = pht.getB();
        result[0] = (int)((A+Twofish.getKSubkey(2*round+8))%MODUL);
        result[1] = (int)((B+Twofish.getKSubkey(2*round+9))%MODUL);
        return result;
    }

```

3.4. Класс GFunction

Класс является аналогом функции G, применяемой в функции F, описанной ранее. Имеет статический метод, на вход которого подается 32-битный блок.

```

public static int run(int x) {
    byte[] bytes = Splitter.run(x);
    SBox sBox = new SBox(x);
    bytes = sBox.S_run();
    int result = MDSMultiply.run(bytes);
    return result;
}

```

На выходе также 32-битный блок.

3.5. Класс PHT

Класс — аналог криптоадамарового преобразования. Основной метод — метод, выполняющий данное преобразование.

```

public void directTransformation() {
    int _a = (int)((long)(a + b) % MODUL);
    int _b = (int)((long)(a + 2 * b) % MODUL);
    a = _a;
    b = _b;
}

```

Числа a и b передаются в конструктор класса. MODUL = 0x100000000L.

3.6. Класс HFunction

В данном классе методом keygen() генерируются раундовые, так называемые, K-подключи. Количество K-подключей равно 40. В одном цикле формируются четный и нечетный подключи.

```

private void keygen() {
    for (int k = 0; k < KEYS_COUNT; k += 2) {
        int even = 0x01010101 * k;
        int odd = 0x01010101 * (k + 1);
    }
}

```

```

SBox evenSBox = new SBox(even);
SBox oddSBox = new SBox(odd);

byte[] evenBytes = evenSBox.M_run(SBox.M_EVEN);
byte[] oddBytes = oddSBox.M_run(SBox.M_ODD);

int evenMDSres = MDSMultiply.run(evenBytes);
int oddMDSres = MDSMultiply.run(oddBytes);

PHT pht = new PHT(evenMDSres,
CyclicShift.toLeft(oddMDSres, Integer.SIZE, 8));
pht.directTransformation();

KKey[k] = pht.getA();
KKey[k+1] = CyclicShift.toLeft(pht.getB(), Integer.SIZE, 9);
}
}

```

3.7. Класс QFunction

В данном классе реализованы подстановки. В классе хранятся фиксированные таблицы q_0 и q_1 . Далее производится алгоритм, описанный в п.2.5.

3.8. Класс KeyShedule

Класс генерирует еще, так называемые, S- и M-подключи. Здесь же хранится матрица Рида-Соломона.

```

private void MKeyGen(byte[] globalKey) {
    int size = globalKeySize / M_BLOCK_BYTE_COUNT;
    MSubkey = new byte[size][BLOCK_COUNT];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < M_BLOCK_BYTE_COUNT; j++) {
            MSubkey[i][j] = globalKey[M_BLOCK_BYTE_COUNT * i + j];
        }
    }
}

private void SKeyGen(byte[] globalKey) {
    int size = globalKeySize / S_BLOCK_BYTE_COUNT;
    SSubkey = new byte[size][BLOCK_COUNT];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < BLOCK_COUNT; j++) {
            int tmp = 0;
            for (int k = i * S_BLOCK_BYTE_COUNT; k < (i + 1) *
                S_BLOCK_BYTE_COUNT; k++) {
                tmp = tmp ^ PolynomMultiply.run(RS[j][k % S_BLOCK_BYTE_COUNT],
                    ((int) globalKey[k] & 0xff));
            }
            SSubkey[i][j] = (byte) PolynomModul.run(tmp, MODUL);
        }
    }
}

```

```

    }
}

```

3.9. Класс MDSMultiply

Класс выполняет умножение вектора из байтов на фиксированную MDS матрицу. Результатом является 32-битный блок.

```

public static int run(byte[] bytes) {
    byte[] result = new byte[BYTE_COUNT];
    for (int i = 0; i < BYTE_COUNT; i++) {
        int tmp = 0;
        for (int j = 0; j < BYTE_COUNT; j++) {
            tmp = tmp ^ PolynomMultiply.run(MDS[i][j],
                ((int)bytes[j] & 0xff));
        }
        result[i] = (byte) PolynomModul.run(tmp, MODUL);
    }
    return Combiner.run(result);
}

```

3.10. Класс PolynomModul

Класс вычисляет модуль по многочлену. В статический метод передается многочлен, представленный в виде числа, составленного по соответствующим коэффициентам многочлена. Также передается многочлен, по модулю которого необходимо вычислить результат. Модуль представлен так же — в виде коэффициентов. Вычисление реализовано с помощью операции битовых сдвигов и умножений на маску.

```

public static int run(int polynom, int modul);

```

3.11. Класс PolynomMultiply

Класс выполняет умножение многочленов, которые передаются в статический метод в виде их коэффициентов.

```

public static int run(int x, int y);

```

3.12. Класс Splitter

Класс предназначен для разбиения одного 32-битного блока на четыре 8-битных блока.

```

public static byte[] run(int x) {
    byte[] bytes = new byte[INT_BYTE_COUNT];
    for (int i = 0; i < INT_BYTE_COUNT; i++) {
        bytes[i] = (byte)((x >> (Byte.SIZE * i)) & MASK);
    }
    return bytes;
}

```

3.13. Класс Combiner

Класс предназначен для объединения четырех 8-битных блоков в один 32-битный блок.

```
public static int run(byte[] bytes) {
    int result = 0;
    for (int i = 0; i < INT_BYTE_COUNT; i++) {
        result += ((int) bytes[i] & MASK) << (Byte.SIZE * i);
    }
    return result;
}
```

3.14. Класс CyclicShift

Класс выполняет циклические сдвиги на произвольное количество бит.

Причем, число может быть разной длины, не обязательно равным одному из типов данных.

```
public static int toRight(int num, int numSize, int bitCount);
public static int toLeft(int num, int numSize, int bitCount);
```

3.15. Класс Key

Класс выполняет генерацию ключа или считывание его из файла.

```
public Key(String filename);
```

Конструктор считывает ключ из указанного файла.

```
public Key(int keySize, String outputFile);
```

Данный конструктор генерирует ключ указанной длины и записывает его в указанный файл.

3.16. Класс Data

Класс предназначен для чтения данных из файла. В конструктор передается имя файла, содержащего данные. Содержит методы:

```
public int[] getBlock();
```

Метод возвращает 128 бит данных, которые необходимы для работы алгоритма. Если данных не хватает, что бывает в конце, дописываются нули.

```
public byte getFirstByte();
```

Метод возвращает первый байт. Это необходимо для расшифрования, чтобы игнорировать биты, записанные при шифровании.

4. Примеры работы программы

Перед тем, чтобы использовать алгоритм, был проведен тест автора (рис. 11), указанный на официальном сайте, чтобы убедиться в правильности работы программы.

```
Algorithm Name:      TWOFISH
Principal Submitter: Bruce Schneier, Counterpane Systems

=====

KEYSIZE=128

KEY=00000000000000000000000000000000

;
;makeKey:  Input key      --> S-box key      [Encrypt]
;          00000000 00000000 --> 00000000
;          00000000 00000000 --> 00000000
;          Subkeys
;          52C54DDE 11F0626D  Input whiten
;          7CAC9D4A 4D1B4AAA
;          B7B83A10 1E7D08EB  Output whiten
;          EE9C341F CFE14BE4
;          F98FFE99 9C5B3C17  Round subkeys
;          15A48310 342A4D81
;          424D89FE C14724A7
;          311B834C FDE87320
;          3302778F 26CD67B4
;          7A6C6362 C2BAF60E
;          3411B994 D972C87F
;          84ADB1EA A7DEE434
;          54D2960F A2F7CAA8
;          A6B8FF8C 8014C425
;          6A748D1C EDBAF720
;          928EF78C 0338EE13
;          9949D6BE C8314176
;          07C07D68 ECAE7EA7
;          1FE71844 85C05C89
;          F298311E 696EA672
```

Рис. 11 Тест автора и результат работы программы

После того, как тест прошел успешно, попробуем зашифровать картинку — белый прямоугольник формата BMP. Размер изображения составил 186174 байта. Алгоритм шифрования выдал файл формата BIN размерностью 186177 байт. Различие в размерах можно объяснить следующим образом. Т.к. на вход алгоритма необходимо подать 128 бит, а остаток от деления размера входного изображения на 16 (128 бит — это 16 байт) равен 14. Т.е. последний блок будет равен 14 байт или 112 бит. В алгоритме в подобном случае, дописываются недостающие байты (в данном случае 2 байта). К тому же, необходимо указать, сколько байт было дописано. На это выделяется ровно один байт в начале файла. Итого, получаем, что размер зашифрованного файла равен $186174 + 2 + 1 = 186177$ байт. Когда расшифруем файл, получаем файл исходного размера. На рис. 12 показаны данные результаты.

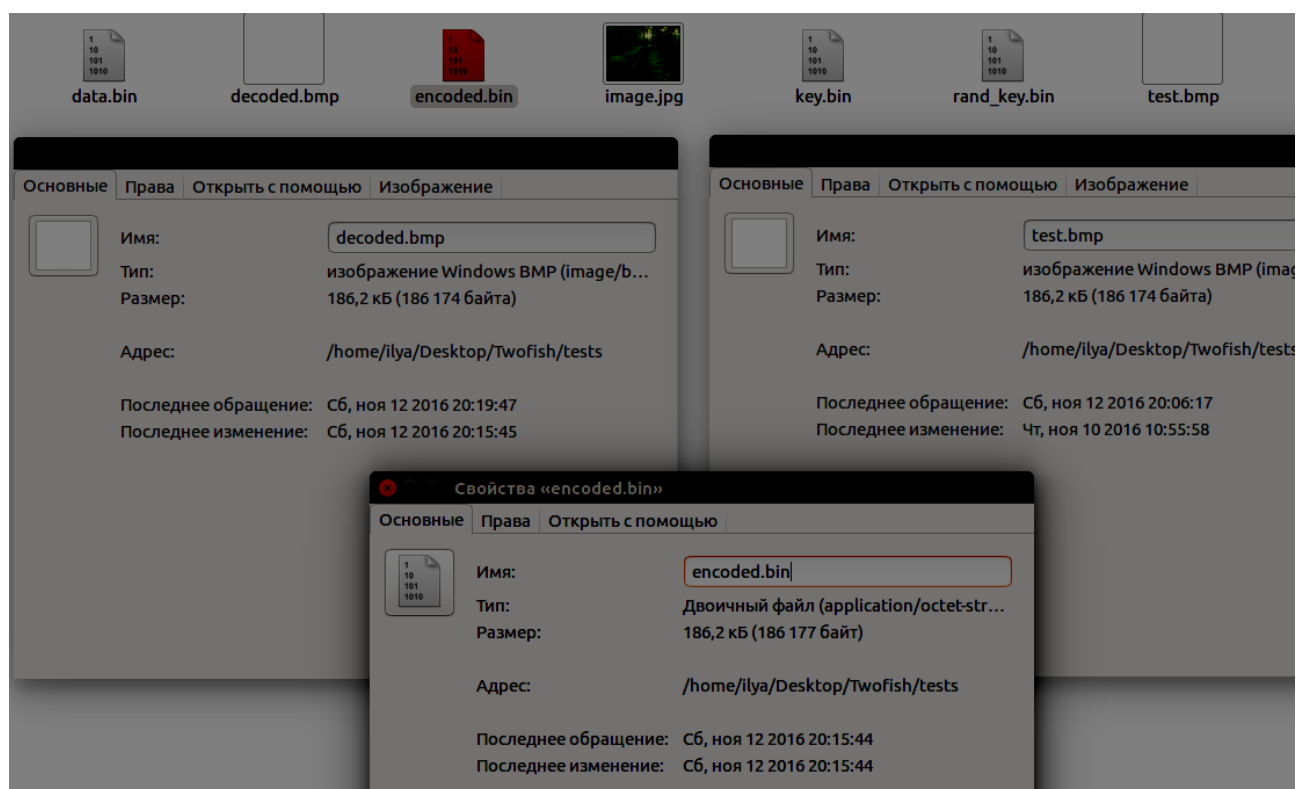


Рис. 12 Сравнение размеров файлов

5. Анализ алгоритма

На 2008 год лучшим вариантом криптоанализа Twofish является вариант усечённого дифференциального криптоанализа, который был опубликован Shiho Moriai и Yiqun Lisa Yin в Японии в 2000 году. Они показали, что для нахождения необходимых дифференциалов требуется 2^{51} подобранных открытых текстов. Тем не менее исследования носили теоретический характер, никакой реальной атаки проведено не было. В своём блоге создатель Twofish Брюс Шнайер утверждает, что в реальности провести такую атаку невозможно.

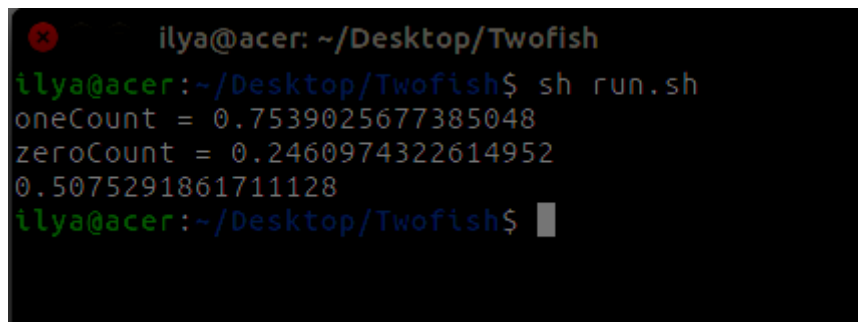
Также, стоит отметить, что для данного алгоритма отсутствуют слабые ключи.

Для примера с изображением были посчитаны коэффициент корреляции для входного и выходного потока алгоритма шифрования, а также распределение «0» и «1» в выходном потоке. На рис. 13 представлены данные результаты. Под коэффициентом корреляции понимается похожесть данных, где 1 — данные совпадают, 0 — данные никак не связаны. В распределении представлены нормированные значения количества «0» и «1».

Количество единиц составило примерно 75,4%

Количество нулей — 24,6%

Коэффициент корреляции — 0,507

A terminal window with a dark background. The title bar shows a red close button and the text 'ilya@acer: ~/Desktop/Twofish'. The terminal content shows a shell prompt 'ilya@acer:~/Desktop/Twofish\$' followed by the command 'sh run.sh'. The output consists of three lines of floating-point numbers: 'oneCount = 0.7539025677385048', 'zeroCount = 0.2460974322614952', and '0.5075291861711128'. The prompt 'ilya@acer:~/Desktop/Twofish\$' is shown again with a cursor.

```
ilya@acer: ~/Desktop/Twofish
ilya@acer:~/Desktop/Twofish$ sh run.sh
oneCount = 0.7539025677385048
zeroCount = 0.2460974322614952
0.5075291861711128
ilya@acer:~/Desktop/Twofish$
```

Рис. 13 Статистические характеристики