

# Практическая криптография в Java.

## Симметричная криптография

Николай Жишкевич

При создании любой нетривиальной системы или приложения следует уделять большое внимание вопросу защиты, обеспечения надежности работы, а также понимать, что любая система, которая используется в электронной коммерции, потенциально подвержена нападению, попытке ее компрометации.

При создании приложений подобного класса возникает необходимость выполнять защиту информации, передаваемой через общедоступные сети. Общеизвестно, что вопросы защиты информации, передаваемой через Интернет очень важны и в этом направлении уже есть большое количество наработок как в виде стандартов, так и конкретных решений. Передавая информацию через общедоступные сети никогда нельзя быть уверенным, что не существует третьей стороны, которой не станет известна передаваемая информация. По сути дела, принципы функционирования Интернет очень похожи на то, как в школе передается записка с первой парты на последнюю. Возможно, хотя это и не гарантируется, что записка дойдет до адресата, но вот с ее содержимым, я думаю, смогут ознакомиться многие. И как раз для того чтобы наши секретные сведения не стали доступны злоумышленнику существует такая специальная наука как криптография. Надо сказать, что существует и другая наука, которая пытается расшифровать то, что вы прячете от любопытных глаз, а называется эта наука криптоанализ. А вместе они образуют науку криптологию.

В отдельных ситуациях когда не может быть достигнута уверенность в том, что информация будет надежно зашифрована при передаче, можно пойти от другого и попытаться эту информацию скрыть, спрятать, замаскировать под нечто несущественное и не важное, так, чтобы потенциальный злоумышленник просто не обратил на нее внимания. Наиболее широко применяется данный подход при сокрытии информации как дополнительного шума вносимого в аудио-видео материалы,

фотографии. Какой уж нужно обладать фантазией, чтобы в маленьких точках помех и еле заметных искажениях в передаваемом фильме распознать нечто секретное. И называется данная наука — стеганография.

Однако я не ставлю перед собой другую задачу, так что, я не буду долго и нудно рассказывать, что такое криптография. Предполагаю, что вы обладаете некоторым представлением в этой области и умеете программировать на Java, однако, пока вы не знаете, как соединить эти знания и реализовать на Java основные криптографические алгоритмы. По ходу дела я буду употреблять некоторые сложные слова, но также буду стараться их разъяснять. Так, что с этим особых проблем у вас не будет. А вот в чем потребуются серьезные знания так это сам язык. Мы будем решать большинство задач применительно к сетевым приложениям, так что вам дополнительно потребуются знания servlets/JSP/sockets, часть информации мы будем хранить или в файлах XML и базах данных, эти знания вам также понадобятся. Ну и, разумеется, что знания математики в пределах средней школы также приветствуются.

## **1. Первая работающая программа**

Мы начнем с самого простого. Мы напишем работающий кусок кода, а потом попытаемся сделать обобщение. По своему опыту я знаю, что для начинающего очень тяжело выйти на обобщение некоторого факта.

Начнем мы с того, что попытаемся зашифровать некоторый документ, файл, так чтобы никто из тех, кто не знает наш пароль, не мог его прочитать. Прежде всего, давайте решим, как именно мы будем шифровать информацию.

Существует большое количество алгоритмов шифрования (преобразования понятного, или открытого текста, в закрытый или зашифрованный), конечно, существует еще большее количество алгоритмов, которые пока не открыты и ждут своих исследователей. Однако я не советую вам играть в великого математика и ученого и придумывать собственный алгоритм (разумеется, если у вас есть докторская степень в области математики и вы в шутку решаете многомерные интегралы в уме, то это вас не касается).

Однако, следует держать в уме следующий факт. Существует два основных способа защиты информации. Это секретный алгоритм и секретный ключ. В первом случае мы

## *Практическая криптография в Java. Симметричная криптография*

храним в тайне способ, с помощью которого выполняется преобразование информации, а во втором варианте сам способ является известным, а не разглашению подлежит небольшой объем секретной информации, называемый ключом.

Как достаточно подробно объясняется в хороших книжках по криптографии первый способ, секретность алгоритма, не в состоянии принципиально обеспечить безусловной стойкости. Кстати, вот вам одно определение. Безусловная стойкость — это такая стойкость, при которой злоумышленник, даже обладая бесконечными вычислительными ресурсами, не в состоянии прочитать зашифрованный документ. Дело в том, что секретные алгоритмы шифрования, разумеется, не могут быть подвергнуты широким исследованиям, (надеюсь, вы догадываетесь почему) и соответственно существует доля вероятности того, что в алгоритме будет найдено уязвимое место.

Гораздо лучшим подходом является сделать используемый алгоритм общедоступным и открытым для исследования различным организациям и просто увлеченным людям. Второй подход имеет и еще одну притягательную сторону, в случае если злоумышленник раскроет алгоритм шифрования, то для того чтобы снова начать передавать информацию безопасно потребуется новый алгоритм, который либо придется иметь наготове (разумеется, это очень не практично и не удобно), либо придется изобретать новый алгоритм на лету, а потом каким-то образом обмениваться с участниками взаимодействия его идеей, если конечно у всех есть в наличии необходимые ресурсы. Например, можно придумать использовать подстановочную таблицу, когда одни символы заменяются на другие на основании некоторого текста или зависимости, из какой-нибудь редкой антикварной книжки, которой разумеется у вашего собеседника не будет. Во втором случае достаточно будет сменить только секретный ключ, ведь взломан был только он, а алгоритм остается прежним.

Если среди читателей есть те, кто увлекается вопросами защиты информации, то они могут вспомнить один из важнейших принципов, которые сформулировал голландский криптограф Керкхофф (1835-1903).

*«Стойкость шифра, т.е. криптосистемы — набора процедур, управляемых некоторой секретной информацией небольшого объема, должна быть обеспечена в том случае, если криптоаналитику противника известен весь механизм шифрования*

*за исключением секретного ключа — информации управляющей процессом криптографических преобразований».*

В общем, хватит, надеюсь, что вы прониклись пониманием того что первый способ не есть наш путь, а пойдем мы вторым путем. Правда здесь существует еще один подводный камень. Использовать в реальных приложениях те алгоритмы, о которых пишут в книжках про храбрых детективах не стоит. Они хотя и просты, но реальной защиты не обеспечивают. На текущий момент лучше использовать blowfish, twofish, rc5, idea. Всякие там шифры Цезаря, Вернама, вращающиеся квадраты или пляшущих человечков лучше не брать в рассмотрение, разве что только для отработки некоторых вспомогательных систем.

При разработке библиотеки для шифрования информации всегда следует планировать будущую сферу ее применения. В отдельных ситуациях существует необходимость шифровать жестко заданные блоки информации, может быть, будет необходимо защищать файловые или иные потоки ввода-вывода, причем делать это следует максимально прозрачно для других частей системы. В качестве несложной иллюстрации я приведу пример фильтра потоков ввода-вывода в котором вся проходящая информация будет преобразовываться самым простым и известным способом, путем выполнения битовой операции XOR (eXclusive OR) байта данных и байта играющего роль ключа. В данном алгоритме ключ для шифрования данных совпадает с ключом для расшифровки информации. Операция XOR имеет важное свойство  $(A \text{ XOR } B) \text{ XOR } B = A$ . Только не напоминайте мне слова о том, что мы будем рассматривать только надежные криптосистемы, а стойкость алгоритма XOR стремится к нулю.

Помните, что мы рассматриваем данный пример, как один из возможных вариантов создания удобного интерфейса для преобразования информации. Я надеюсь, что вы помните основные понятия фильтров потока. Фильтр это посредник между источником и потребителем информации и может абсолютно прозрачно выполнять преобразования информации.

```
package arti.security;  
  
import java.io.*;
```

## Практическая криптография в Java. Симметричная криптография

```
public class XORInFilter extends FilterInputStream {
    protected byte key ;

    public XORInFilter (InputStream is, byte key){
        super (is);
        this.key = key;
    }

    public int read() throws java.io.IOException {
        int r = super.read();
        if (r == -1) return -1;
        return r ^ key;
    }

    public int read(byte[] parm1, int parm2, int parm3)
        throws java.io.IOException {
        int r , cnt = 0;
        for (int i=parm2; (i < (parm2 + parm3))
            && ((r = read()) != -1); i++,
            cnt++)

            parm1[i] = (byte) r;
        return cnt;
    }

    public int read(byte[] parm1) throws java.io.IOException {
        return read ( parm1 , 0 , parm1.length);
    }
}

package arti.security;

import java.io.*;

public class XOROutFilter extends FilterOutputStream {
    protected byte key;

    public XOROutFilter (OutputStream out , byte key){
        super (out);
        this.key = key;
    }
}
```

## Практическая криптография в Java. Симметричная криптография

```
}

public void write(int b) throws java.io.IOException {
    super.write(b ^ key);
}

public void write(byte[] parm1) throws java.io.IOException {
    write ( parm1 , 0 , parm1.length);
}

public void write(byte[] parm1, int parm2, int parm3)
    throws java.io.IOException
{
    for (int i=parm2; i < parm2 + parm3; i++)
        write (parm1[i]);
}
}
```

И, наконец, я приведу пример фрагмента кода в котором используется данный набор классов фильтров.

```
package arti.security

import java.io.*;

public class usexor {
    public static void main(String[] args) throws IOException{
        System.out.println("Usage:
        java security_1.usexor file.in file.out key");

        if (args.length != 3) return;
        String fIn = args [0];
        String fOut = args [1];
        byte key = Byte.parseByte(args[2]);

        FileInputStream fin = new FileInputStream (fIn);
        FileOutputStream fout = new FileOutputStream (fOut);
        XOROutFilter xout = new XOROutFilter (fout , key);

        byte [] buf = new byte [10000];
```

```
int r = 0;

while ( ( r = fin.read(buf)) > 0){
    xout.write(buf , 0 , r);
}

fin.close();
xout.close();

System.out.println("File was successfully processed");
}
}
```

Разумеется, для практического применения данный алгоритм не подходит. Вообще для оценки стойкости любого алгоритма следует учитывать количество вариантов ключа. В данном случае их всего 255 вариантов (количество значений, которое может принимать переменная типа “byte” - 1). И как вы догадываетесь, перебрать все варианты ключа не составит никакой сложности. Стойкость данного алгоритма можно увеличить путем введения большей длины ключа, блочной обработки информации и т.д. Однако мы договорились, что не будем придумывать собственные алгоритмы, а воспользуемся наработанными решениями.

## **2. Стойкая криптография. Blowfish**

Продолжим наше повествование с того что попытаемся написать собственную версию алгоритма шифрования. На роль кандидата для реализации я выбрал алгоритм Blowfish, хотя данный алгоритм имеет достаточно большой возраст, но пока он надежен и вполне применим для ответственных сфер применения, к тому же у него есть еще один маленький плюс для новичка — его идея и программная реализация очень проста и по силам практически любому начинающему программисту.

Так, я слышу вопрос из зала: а какой алгоритм считается самым крутым, который «ВАЩЕ» не раскрывается, и почему бы не реализовать его и не забивать себе голову множеством алгоритмов.

Вообще то я действительно упоминал понятие вычислительно стойких и безусловно стойких алгоритмов. Blowfish — это представитель вычислительно стойкого алгоритма, т.е. существует теоретическая возможность его вскрытия, однако затраты

времени и сил на это будут настолько велики, что добытая информация к тому времени уже устареет и не оправдает затраченных усилий. Клод Шеннон (кто не знает, именно его считают основоположником теории информации и научного подхода в области ее защиты) в своих трудах ввел понятие стойкости шифра и сказал, что да, действительно существует алгоритм шифрования, который обеспечивает абсолютную (теоретическую) секретность.

Знание шифртекста не позволяет противнику улучшить оценку соответствующего открытого текста.

Говоря проще, если информацию зашифровать с помощью подобного алгоритма, то даже попадание к злоумышленнику шифртекста не дает ему ни одного дополнительного шанса на вскрытие информации.

В качестве примера подобного шифра рассмотрим простейший способ наложения гаммы на информацию (открытый текст) с помощью операции XOR. Да, именно XOR, я не оговорился. Ниже я приведу пример подобного шифрования.

Исходный текст, представленный в виде кодов букв сообщения.

123	56	7	159	234
-----	----	---	-----	-----

В качестве ключа возьмем число 100.

В результате преобразования получаем следующий ряд чисел.

31	92	99	251	142
----	----	----	-----	-----

В таком виде алгоритм не обеспечивает ни вычислительной стойкости, ни тем более стойкости теоретической. Злоумышленник просто переберет все возможные варианты ключей, благо их всего 255. А теперь, давайте сделаем «ход конем», и в качестве ключа возьмем не одно число, а ряд чисел равный по длине самому исходному сообщению, и при этом такой, что бы числа входящие в данный ключ были бы распределены по некоторому случайному закону, или, говоря проще, безо всякой зависимости, так что, зная значения некоторой последовательности чисел нельзя было бы предугадать следующее значение.

Например, если взять следующий ряд чисел.

54	32	45	89	2
----	----	----	----	---

Будет получен следующий зашифрованный текст.

77	24	42	198	232
----	----	----	-----	-----

Давайте ответим сами себе на следующий вопрос: по какому признаку злоумышленник сможет узнать, что шифртекст был вскрыт правильно? Логично предположить, что тогда когда после применения шифра получившийся документ имеет какой-то смысл. И тут возникает удивительная ситуация: в случае использования различных ключей примененных к зашифрованному документу получается очень много осмысленных текстов, каждый из которых вполне может претендовать на роль секретной информации. К сожалению идеального шифра не существует, и хотя данный пример показал, что существуют теоретически невзламываемые шифры, но на практике их применять невозможно, в приведенном выше примере возникает необходимость перед началом сеанса передать партнеру значение секретного ключа, размер которого совпадает с размером исходного сообщения. Правда уже пора задаться следующим вопросом: а раз существует некоторый секретный прием передачи такого большого ключа, то почему бы не воспользоваться им для передачи и самой информации, и не забивать себе голову дополнительными шагами.

### **3. Теория, или как Blowfish устроен изнутри**

На момент написания данного материала алгоритму Blowfish уже исполнилось 10 лет, так он был создан Брюсом Шнейером в 1993 году, и позиционировался как замена окончательно устаревшему на тот момент и дискредитировавшему себя алгоритму DES. Он планировался как аналог и замена ряда других алгоритмов, таких как IDEA, RC5, TripleDES, AES.

Прежде чем я попытаюсь изложить основы тех преобразований, которые заложены в алгоритме Blowfish, следует понять, что Blowfish относится к семейству алгоритмов симметричного блочного шифрования. Кроме семейства блочных алгоритмов существуют также и множество поточных алгоритмов. Главное отличие этих двух направлений в том, что в случае поточного шифрования заранее не известна длина шифруемой последовательности, т.е. элементарное преобразование выполняется над наименьшей единицей информации. Это может быть байт, или бит. А во втором случае длина шифруемой последовательности должна быть известна и кратна размеру

блока. Преобразование выполняется именно над блоком. Например, исходное сообщение длиной 2048 байт разбивается на блоки каждый из которых 4 байта и над этими блоками выполняется криптографическое преобразование. Думающий читатель сразу же возразит, что в реальной жизни такие сообщения, длина которых случайным образом всегда кратна длине блока, не встречаются, будет, разумеется, прав. В таком случае выполняется дополнение исходных данных до длины кратной блоку.

### **3.1. Режимы работы блочного шифра**

Раз уж мы решили создать собственную реализацию алгоритма Blowfish, то я думаю полезным будет рассмотреть режимы в которых может работать данный алгоритм. Выделяют следующие режимы работы, причем важно отметить, что данные режимы характерны не только для Blowfish, но и для любого другого блочного шифра — это режим раздельного преобразования каждого блока в отдельности, а также режим в котором производится сцепление блоков шифртекста.

**ECB — Electronic Code Book Mode.** Самый простой режим в данном случае все блоки, на которые мы разбили исходное сообщение, подвергаются криптопреобразованию независимо друг от друга с помощью одного и того же ключа и никак между собой в общем случае не связываются.

**CBC — Cipher Block Chaining Mode (режим сцепления блоков).** В этом режиме очередной блок открытого текста складывается по модулю 2 с предыдущим блоком шифртекста, после чего подвергается зашифрованию в режиме ECB. Для самого первого блока "предыдущим блоком шифртекста" является либо нулевой блок либо значение вырабатываемое по некоторому случайному закону отправителем сообщения и передаваемое вместе с самим сообщением к получателю.

## **4. Алгоритм работы Blowfish**

Далее пойдет сугубо техническая информация, связанная с созданием собственной реализации данного алгоритма. Начнем наше рассмотрение с простейшего варианта работы шифра в режиме ECB.

Blowfish — является 64-битовым блочным шифром, состоящим из двух частей:

## Практическая криптография в Java. Симметричная криптография

- алгоритм расширения ключа;
- алгоритм непосредственного шифрования/дешифрования.

Дело в том, что зачастую для алгоритма криптопреобразования необходимым для качественной работы является условие, что размер ключа будет очень велик, несколько десятков, сотен, а иногда и тысяч байт. Разумеется, что в практике придумывать ключи подобного размера, занимающие полстраницы книжки, не удобно. Поэтому используют прием, когда пользователь вводит ключ небольшого размера (секретный ключ), а потом на его основе формируется другой ключ, который и используется в самом алгоритме шифрования/дешифрования, такой ключ называют расширенным ключом шифрования/дешифрования. Так вот в алгоритме blowfish к ключу предъявляется требование размера 4168 байт. И как вы догадываетесь, данный ключ формируется на основе секретного ключа, размер которого не превосходит 448 бит. Ключ шифрования представляется в виде нескольких массивов подключей:

- Массив 32-битных констант  $K_1, K_2, K_3, \dots, K_{18}$ .
- Четыре последовательности 32-битовых подключей, каждая из этих последовательностей представлена 256 элементами.

```
Q1(0), Q1(1), ..., Q1(255);
```

```
Q2(0), Q2(1), ..., Q2(255);
```

```
Q3(0), Q3(1), ..., Q3(255);
```

```
Q4(0), Q4(1), ..., Q4(255).
```

По сути своей данный набор констант задает собой большую таблицу отображения входных блоков размером 32-бита в соответствующие им выходные 32-битовые блоки. Алгоритм blowfish построен на основе итеративной криптосхемы Фейстеля, в которой периодически выполняется расчет значения следующей функции.

$$F(x) = ((S1(x1) + S2(x2) \bmod 23^2) \\ \text{XOR } S3(x3)) + S4(x4) \bmod 23^2$$

В данной формуле мы кроме нам уже знакомых таблиц подстановки  $S1(0..255), S2(0..255), S3(0..255), S4(0..255)$  задействованы переменные  $x1, x2, x3, x4$ . Данные величины мы получаем выполняя последовательно следующие разложения. Вначале мы исходный блок размером 64

бита (надеюсь, вы помните, что алгоритм Blowfish относится к классу блочных алгоритмов, и оперирует над блоками размером 64 бита) делим на 2-е части размером 32 бита. Именно такую длину данных мы задали для аргумента функции  $F(x)$ . Здесь  $x$  — половинка блока и имеет размер в 32 бита. А затем мы снова выполнили разложение данного 32 битового блока на 4-е подблока размером 8 бит.

$X = x_1 \mid x_2 \mid x_3 \mid x_4$

В вычислении значения функции  $F(x)$  величины  $x_1, x_2, x_3, x_4$  выступают в роли индексов массивов  $Q_1(x_1), Q_2(x_2), Q_3(x_3), Q_4(x_4)$ .

Алгоритм который иллюстрирует работы криптосхемы Фейстеля приводится ниже.

ВХОДНЫЕ ДАННЫЕ:

Блок входных (открытых) данных  $T = L \mid R$  (операция побитового объединения).

АЛГОРИТМ:

- 1) Объявить переменную-счетчик  $i = 1$ .
- 2) Преобразовать подблок  $L$  и рассчитать значение вспомогательной переменной  $V$  по следующим формулам:

$$L = L \text{ XOR } K(i)$$

$$V = F(L)$$

- 3) Преобразовать блок  $R$  по следующей формуле:

$$R = R \text{ XOR } V$$

- 4) Проверка условия, если  $i = 16$ , тогда перейти к шагу 7

- 5) Увеличить значение переменной  $i$ :

$$i = i + 1$$

и поменять местами значения переменных  $L$  и  $R$

$$R ::= L$$

- 6) Вернуться на шаг 2

## Практическая криптография в Java. Симметричная криптография

7) Преобразовать подблок R

$$R = R \text{ XOR } K(17)$$

8) Преобразовать подблок L

$$L = L \text{ XOR } K(18)$$

9) Стоп преобразования.

**ВЫХОДНЫЕ ДАННЫЕ:**

64-битовый блок закрытого (зашифрованного текста)  $C = L | R$

Для дешифрации используют похожий алгоритм.

**ВХОД:**

64-битовый блок закрытого (зашифрованного) текста  $C = L | R$ .

**АЛГОРИТМ:**

Преобразовать подблок R по следующему закону:

$$R = R \text{ XOR } K(17)$$

2) Преобразовать блок L по следующему закону:

$$L = L \text{ XOR } K(18)$$

3) Объявить переменную-счетчик  $i$  и присвоить ей начальное значение  $i = 16$

4) Вычислить текущее значение переменной  $V$  и изменить блок R по следующему закону:

$$V = F(L)$$

$$R = R \text{ XOR } V$$

5) Преобразовать подблок L по следующему закону:

$$L = L \text{ XOR } K(17 - i)$$

6) Проверка условия  $i = 1$  и если условие истинно, то переход к пункту 9

7) Перемена местами блоков L и R

$L := R$  ;  $R := L$

8) Уменьшить значение переменной  $I = I - 1$  и переход к шагу 4

9) Стоп преобразования

ВЫХОД:

Блок открытого текста  $P = L \parallel R$

Теперь мы знаем все необходимое, чтобы начать программную реализацию алгоритма. Хотя, стоп, мы забыли самое главное. Это алгоритм формирования на основе секретного ключа расширенного ключа, тех самых 4168 байт, представленных в виде массивов Q и K.

## 5. Алгоритм генерации расширенного ключа

Ключи генерируются путем использования алгоритма Blowfish на начальный набор данных задающий случайно распределенных набор чисел. Разумеется, наилучшим образом на эту роль подходит какой-нибудь математический ряд для которого пока не обнаружена закономерность.

Последовательность действий следующая:

Присваиваем подключам  $K(1-18)$ ,  $S1, S2, S3, S4$  начальное значение на основе цифр дробной части числа  $\pi$ . Подобную начальную таблицу обычно задают в виде константы, подробные таблицы начальных значений легко можно найти в Интернет или вытащить из любой программной реализации данного алгоритма.

Затем вам необходимо в цикле пройтись по всем ключам  $K(1-18)$  и выполнить для каждого из них операцию XOR с соответствующими 32 битами ключа. Если длины секретного ключа будет не хватать, то необходимо циклически вернуться на начало ключа, таким образом возникает эквивалентность вычисляемого расширенного ключа для ряда секретных ключей которые являются подмножествами друг друга, например, 1 и 111, или 123 и 123123 и т.д. по аналогии.

Следующим шагом будет выполнение шифрования массивов ключей сначала

$K(1-18)$ , а затем и массивов  $Q_1, Q_2, Q_3, Q_4$ . Важным будет то что начинать процесс шифрования нужно с блока  $P = L \mid R = 0$ .

## 6. Программная реализация алгоритма

Ниже приводится рабочий фрагмент кода, по возможности я попытался добавлять комментарии. С целью удобства код разнесен на несколько классов, выполняющих свое функциональное назначение.

```
package arti.security;

public abstract class BlowFish {
    // список констант задающих параметры алгоритма
    public static final int COUNTROUNDS = 16;
    public static final int SECRETKEYLENGTH = 448 / 8;
    public static final int EXPANDEDKEYLENGTH = 4168 / 8;
    public static final int MINKEYLENGTH = 4;
    public static final int MAXKEYLENGTH = 56;
    public static final int BUFFERSIZE = 1000;
    public static final boolean ENCRYPT = true;
    public static final boolean DECRYPT = false;

    KLine K = null; // объект в котором хранятся группа К-ключей

    Q_1_Grid Q1 = null; // объекты для хранения групп Q1 - ключей
    Q_2_Grid Q2 = null; // объекты для хранения групп Q2 - ключей
    Q_3_Grid Q3 = null; // объекты для хранения групп Q3 - ключей
    Q_4_Grid Q4 = null; // объекты для хранения групп Q4 - ключей

    boolean settedKey = false; // переменная хранящая в себе признак того
    // был ли передан перед непосредственным использованием данного алгоритма
    // секретный ключ

    byte[] UserKey = new byte[SECRETKEYLENGTH]; // массив для хранения переданного
    // пользователем секретного ключа
    int SizeKey = 0; // размер ключа который был передан
    // для обнаружения
    // конца прочитанных байтов может быть добавлен еще один 64 битовый блок
    // типа 1000...000
```

## Практическая криптография в Java. Симметричная криптография

```
/*
    используется 16 раундов шифрования
    длина пользовательского ключа не более 448 бит
    и согласно политике безопасности не менее 16 байт
*/

public BlowFish() {
    settedKey = false;
}
/**
 * Данный метод служит для удаления (заполнения)
 * массива хранящего секретный ключ пользователя нулями
 * @throws SecurityException
 */
public void destroyKey() throws SecurityException {
    int i, n = SizeKey;
    for (i = 0; i < n; i++) {
        UserKey[i] = 0;
    }
}
/**
 * вспомогательная функция для шифрования набора блоков
 * @param countBlocks – количество блоков
 * @param IOBuffer – массив байтов хранящий в себе набор данных блоков
 * важным является то, что предполагается что исходный набор данных
 * в общем случае не кратный размеру блока т.е. 64-бита был уже нормализован
 * или подогнан под универсальную схему
 */
protected void encryptBlocks(int countBlocks, byte[] IOBuffer) {
    int i;
    for (i = 0; i < countBlocks; i++) {
        int t = 8 * i;

        int x0 = ByteToInt(IOBuffer[t + 0]);
        int x1 = ByteToInt(IOBuffer[t + 1]);
        int x2 = ByteToInt(IOBuffer[t + 2]);
        int x3 = ByteToInt(IOBuffer[t + 3]);

        int x4 = ByteToInt(IOBuffer[t + 4]);
        int x5 = ByteToInt(IOBuffer[t + 5]);
    }
}
```

## Практическая криптография в Java. Симметричная криптография

```
int x6 = ByteToInt(IOBuffer[t + 6]);
int x7 = ByteToInt(IOBuffer[t + 7]);

// старшие биты расположены раньше
// установлены только младшие 8 бит
// Left – старшая часть
int Left = x0 << 24;
Left = Left | (x1 << 16);
Left = Left | (x2 << 8);
Left = Left | (x3);

int Right = x4 << 24;
Right = Right | (x5 << 16);
Right = Right | (x6 << 8);
Right = Right | (x7);

long C = Encrypt(Left, Right);

x7 = (byte) (C & 0xFFL);
C >>>= 8;

x6 = (byte) (C & 0xFFL);
C >>>= 8;

x5 = (byte) (C & 0xFFL);
C >>>= 8;

x4 = (byte) (C & 0xFFL);
C >>>= 8;
// -----//
x3 = (byte) (C & 0xFFL);
C >>>= 8;

x2 = (byte) (C & 0xFFL);
C >>>= 8;

x1 = (byte) (C & 0xFFL);
C >>>= 8;

x0 = (byte) (C & 0xFFL);
```

## Практическая криптография в Java. Симметричная криптография

```
C >>>= 8;
// и положим все назад, где взяли
IOBuffer[t] = (byte) x0;
IOBuffer[t + 1] = (byte) x1;
IOBuffer[t + 2] = (byte) x2;
IOBuffer[t + 3] = (byte) x3;

IOBuffer[t + 4] = (byte) x4;
IOBuffer[t + 5] = (byte) x5;
IOBuffer[t + 6] = (byte) x6;
IOBuffer[t + 7] = (byte) x7;
}
}
/**
 * Вспомогательная функция решает задачу приведения массива подлежащих
 * шифрованию данных к общему виду, т.е. кратно 64 битам
 * @param count — количество байтов подлежащих шифрованию
 * @param IOBuffer — ссылка на массив байтов для шифрования
 * @return — количество нормализованных байтов
 * для нормализации к исходной последовательности добавляется новый блок
 * или выполняется дополнение последнего неполного блока
 * последовательностью вида 10000...
 * @throws SecurityException
 */
protected int normalizeBlock(int count, byte[] IOBuffer) throws
    SecurityException {
    try {
        int oldcount = count;

        if ( (count % 8) != 0) { // дополняем до кратной 8 последний блок

            while ( (count % 8) != 0) { // если идти от нуля
                IOBuffer[count] = (byte) 0x00;
                count++;
            }

            IOBuffer[oldcount] = (byte) 0x80; // 128 в десятичной т.е. устанавливаем
            //седьмой бит в 1
            return count;
        }
    }
}
```

## Практическая криптография в Java. Симметричная криптография

```
    }

    IOBuffer[count] = (byte) 0x80;
    count++;
    while ( (count % 8) != 0) {
        IOBuffer[count] = (byte) 0x00;
        count++;
    }

    return oldcount + 8;
} //try
catch (java.lang.Exception e) {
    throw new SecurityException("error while normalize:" + e);
}
}
/**
 * вспомогательная функция выполняет задачу дешифровки данных
 * данные должны быть нормализованы под размер кратный размеру блока
 * @param countBlocks – количество блоков информации
 * @param IOBuffer – массив блоков для дешифровки
 */
protected void decryptBlocks(int countBlocks, byte[] IOBuffer) {
    int i;
    for (i = 0; i < countBlocks; i++) {
        int t = 8 * i;
        int x0 = ByteToInt(IOBuffer[t + 0]);
        int x1 = ByteToInt(IOBuffer[t + 1]);
        int x2 = ByteToInt(IOBuffer[t + 2]);
        int x3 = ByteToInt(IOBuffer[t + 3]);

        int x4 = ByteToInt(IOBuffer[t + 4]);
        int x5 = ByteToInt(IOBuffer[t + 5]);
        int x6 = ByteToInt(IOBuffer[t + 6]);
        int x7 = ByteToInt(IOBuffer[t + 7]);

        // старшие биты расположены раньше
        // установлены только младшие 8 бит
        // left – старшая часть

        int Left = x0 << 24;
```

## Практическая криптография в Java. Симметричная криптография

```
Left = Left | (x1 << 16);
Left = Left | (x2 << 8);
Left = Left | (x3);

int Right = x4 << 24;
Right = Right | (x5 << 16);
Right = Right | (x6 << 8);
Right = Right | (x7);

long C = Decrypt(Left, Right);

x7 = (byte) (C & 0xFFL);
C >>>= 8;

x6 = (byte) (C & 0xFFL);
C >>>= 8;

x5 = (byte) (C & 0xFFL);
C >>>= 8;

x4 = (byte) (C & 0xFFL);
C >>>= 8;
// -----//
x3 = (byte) (C & 0xFFL);
C >>>= 8;

x2 = (byte) (C & 0xFFL);
C >>>= 8;

x1 = (byte) (C & 0xFFL);
C >>>= 8; // C == 0

x0 = (byte) (C & 0xFFL);
C >>>= 8;
// и положим все назад, где взяли
IOBuffer[t] = (byte) x0;
IOBuffer[t + 1] = (byte) x1;
IOBuffer[t + 2] = (byte) x2;
IOBuffer[t + 3] = (byte) x3;
```

## Практическая криптография в Java. Симметричная криптография

```
        IOBuffer[t + 4] = (byte) x4;
        IOBuffer[t + 5] = (byte) x5;
        IOBuffer[t + 6] = (byte) x6;
        IOBuffer[t + 7] = (byte) x7;
    }

}

/**
 * Вспомогательная функция выполняет роль обратную той которую выполняла
 * функция normalizeBlock
 * мы должны избавиться от добавки к информации в виде набора 10000...
 * @param count – количество байтов информации
 * @param IOBuffer – массив данных
 * @return – размер информации без добавленного хвоста
 * @throws SecurityException
 */
protected int deNormalizeBlock(int count, byte[] IOBuffer) throws
    SecurityException {
    try {
        while (IOBuffer[count - 1] != (byte) 0x80) {
            count--;
        }
        return count - 1;
    }
    catch (java.lang.Exception e) {
        throw new SecurityException("error while normalize:" + e);
    }
}

public void setKey(byte[] key) throws SecurityException {
    settedKey = false;

    K = new KLine();
    Q1 = new Q_1_Grid();
    Q2 = new Q_2_Grid();
    Q3 = new Q_3_Grid();
    Q4 = new Q_4_Grid();

    if (key == null) {
        throw new SecurityException("Ключ не был передан и инициализирован
```

```
(расширен)");
    }
    // проверка длины ключа на минимальные размеры
    if (key.length < MINKEYLENGTH) {
        throw new SecurityException("Длина ключа не должна быть менее чем " +
MINKEYLENGTH );
    }

    if (key.length > MAXKEYLENGTH) {
        throw new SecurityException("Длина ключа не должна превышать " +
MAXKEYLENGTH );
    }
    settedKey = true;
    int i;
    SizeKey = key.length;
    for (i = 0; i < SizeKey; i++) {
        UserKey[i] = key[i];
    }
}

protected long Encrypt(int Left, int Right) {
// данная функция по сути выполняет именно преобразование
//  $F(x) = ((S1(x1) + S2(x2) \bmod 23^2) \text{ XOR } S3(x3)) + S4(x4) \bmod 23^2$ 
    int i, V, W;

    for (i = 0; i < 16; i++) {
        Left = Left ^ K.KLine[i];
        V = F(Left);

        Right = Right ^ V;
        W = Left;
        Left = Right;
        Right = W;
    }

    W = Left;
    Left = Right;
    Right = W;

    Right = K.KLine[17 - 1] ^ Right;
```

## Практическая криптография в Java. Симметричная криптография

```
Left = K.KLine[18 - 1] ^ Left;

return
    (IntToLong(Left) << 32)
    | IntToLong(Right);
}

protected long Decrypt(int Left, int Right) {
    int i, V, W;

    for (i = 17; i >= 2; i--) {
        Left = Left ^ K.KLine[i];
        Right = F(Left) ^ Right;

        W = Left;
        Left = Right;
        Right = W;
    }

    W = Left;
    Left = Right;
    Right = W;

    Right = Right ^ K.KLine[1];
    Left = Left ^ K.KLine[0];

    return
        (IntToLong(Left) << 32) |
        IntToLong(Right);
}

public void initKey() throws SecurityException {
    if (settedKey == false) {
        throw new SecurityException("Ключ не был передан");
    }
}

// XOR-им P последовательность с ключем пользователя
```

## Практическая криптография в Java. Симметричная криптография

```
int i, j = 0;

for (i = 0; i < 18; i++) {
    byte x0 = (byte) UserKey[ (j) % SizeKey];
    byte x1 = (byte) UserKey[ (1 + j) % SizeKey];
    byte x2 = (byte) UserKey[ (2 + j) % SizeKey];
    byte x3 = (byte) UserKey[ (3 + j) % SizeKey];

    j += 4;
    j = (j % SizeKey);

    int X = (ByteToInt(x0) << 24) |
            (ByteToInt(x1) << 16) |
            (ByteToInt(x2) << 8) |
            ByteToInt(x3);

    K.KLine[i] = K.KLine[i] ^ X;
}

int Left = 0;
int Right = 0;

// первый запуск шифрования
for (i = 0; i < 18 - 1; i += 2) {

    long C = Encrypt(Left, Right); // шифруем сначала нулевую строку
    // а потом результат предыдущего шифрования,
    // попутно изменяя K(1 -: 18) последовательность ключей
    Left = (int) (C >>> 32); // берем старшие 32 бита
    Right = (int) C; // берем младшие 32 бита

    K.KLine[i] = Left;
    K.KLine[i + 1] = Right;
}

// для первой последовательности Q

for (i = 0; i <= 255 - 1; i += 2) {
    long C = Encrypt(Left, Right);
```

## Практическая криптография в Java. Симметричная криптография

```
    Left = (int) (C >>> 32); // берем старшие 32 бита
    Right = (int) C; // берем младшие 32 бита
    Q1.QGrid[i] = Left;
    Q1.QGrid[i + 1] = Right;

}

// для второй последовательности Q

for (i = 0; i <= 255 - 1; i += 2) {
    long C = Encrypt(Left, Right);
    Left = (int) (C >>> 32); // берем старшие 32 бита
    Right = (int) C; // берем младшие 32 бита
    Q2.QGrid[i] = Left;
    Q2.QGrid[i + 1] = Right;

}

// для третьей последовательности Q

for (i = 0; i <= 255 - 1; i += 2) {
    long C = Encrypt(Left, Right);
    Left = (int) (C >>> 32); // берем старшие 32 бита
    Right = (int) C; // берем младшие 32 бита
    Q3.QGrid[i] = Left;
    Q3.QGrid[i + 1] = Right;

}

// для четвертой последовательности Q

for (i = 0; i <= 255 - 1; i += 2) {
    long C = Encrypt(Left, Right);
    Left = (int) (C >>> 32); // берем старшие 32 бита
    Right = (int) C; // берем младшие 32 бита
    Q4.QGrid[i] = Left;
    Q4.QGrid[i + 1] = Right;

}

// Ура мы расширили ключ, хотя его еще в идеале следует проверить
```

## Практическая криптография в Java. Симметричная криптография

```
//на стойкость
if (weekKey()) {
    throw new arti.security.SecurityException(
        "Сгенерированный ключ не прошел теста
        на стойкость и рекомендуется его заменить");
}
}

protected boolean weekKey() {
    int i, j, n, old;
    n = 256 - 1; // последний элемент не берем, т.к. не с чем его сравнивать

    for (i = 0; i < n; i++) {
        old = Q1.QGrid[i];
        for (j = i + 1; j < 256; j++) {
            if (Q1.QGrid[j] == old) {
                return true; // Ключ будет считаться слабым если два элемента
                // массивов подстановок Q[i] == Q[j]
            }
        }
    }

    for (i = 0; i < n; i++) {
        old = Q2.QGrid[i];
        for (j = i + 1; j < 256; j++) {
            if (Q2.QGrid[j] == old) {
                return true;
            }
        }
    }

    for (i = 0; i < n; i++) {
        old = Q3.QGrid[i];
        for (j = i + 1; j < 256; j++) {
            if (Q3.QGrid[j] == old) {
                return true;
            }
        }
    }
}
```

## Практическая криптография в Java. Симметричная криптография

```
for (i = 0; i < n; i++) {
    old = Q4.QGrid[i];
    for (j = i + 1; j < 256; j++) {
        if (Q4.QGrid[j] == old) {
            return true;
        }
    }
}

return false; // Если текст был пройден то
              // ключ можно использовать для шифрования/дешифрования
} //end of function

// --- Набор дополнительных функций
// используемых в преобразованиях -----

protected int F(int X) {

    int x0 = (X & 0xFF);
    X = X >>> 8;

    int x1 = (X & 0xFF);
    X = X >>> 8;

    int x2 = (X & 0xFF);
    X = X >>> 8;

    int x3 = (X & 0xFF);
    X = X >>> 8; // X == 0

    long X3 = IntToLong(Q4.QGrid[x0]);
    long X2 = IntToLong(Q3.QGrid[x1]);
    long X1 = IntToLong(Q2.QGrid[x2]);
    long X0 = IntToLong(Q1.QGrid[x3]);

    long temp_32 = (X1 % 32); // !
    long temp_64 = IntToLong( (int) (temp_32 + X0));
    long temp_128 = temp_64 ^ X2;
    long temp_256 = X3 % 32;
    long temp_512 = IntToLong( (int) (temp_256 + temp_128));
```

```
        return (int) (temp_512);
    }

//Функции для работы с битами и преобразования
// типов данных между собой с избеганием побочных эффектов
-----
// под побочным эффектом понимают размножение знака
// при поднятии типа например от
// байта к int или от int к long
    static protected int ByteToInt(byte b) {
        return ( (int) b) & 0x000000FF;
    }

    static protected int ByteToInt(int b) {
        return b & 0x000000FF;
    }

    static protected long ByteToLong(int b) {
        return b & 0x0000000000000000FFL;
    }

    static protected long IntToLong(int i) {
        return ( (long) i) & 0x00000000FFFFFFFFL;
    }

    static protected long ByteToLong(byte b) {
        return ( (long) b) & 0x0000000000000000FFFL;
    }

    // end of class
}

package arti.security;

public class BlowFishArray
    extends BlowFish {
    /**
     * Данный класс является расширением класса
     * Blowfish и служит для выполнения
```

## Практическая криптография в Java. Симметричная криптография

```
* конкретного преобразования над массивом байтов
*/
boolean Mode = true;
int actualSize = 0;
// реальный размер массива байтов
// без всяких нормализаций и дополнений

byte[] Array = null;

public BlowFishArray() {
    settedKey = false;
}

// передача массива подлежащего обработке
public void setArray(byte[] arr) throws SecurityException {
    Array = new byte[arr.length + 8];
    for (int i = 0; i < arr.length; i++) {
        Array[i] = arr[i];
    }
    actualSize = arr.length;
}

// возврат массива после преобразования
public byte[] getArray() {
    byte[] ArrRet = new byte[actualSize];
    for (int i = 0; i < actualSize; i++) {
        ArrRet[i] = Array[i];
    }
    return ArrRet;
}

public void start(boolean b) throws SecurityException {
    Mode = b; // режим шифровать или дешифровать
    if (Array == null) {
        throw new SecurityException("Нет массива
        данных для обработки");
    }

    if (Mode == BlowFish.ENCRYPT) {
        StartEncrypt();
    }
}
```

```
        else {
            StartDecrypt();
        }
    }
    // шифрование данных включает в себя
    //прежде всего нормализацию и непосредственно шифрование
    protected void StartEncrypt() throws SecurityException {
        actualSize = normalizeBlock(actualSize, Array);
        encryptBlocks(actualSize / 8, Array); // число блоков
    }

    // дешифрование состоит из дешифровки и денормализации размера блоков
    protected void StartDecrypt() throws SecurityException {
        decryptBlocks(actualSize / 8, Array);
        actualSize = deNormalizeBlock(actualSize, Array);
    }

    public int getActualSize() {
        return actualSize;
    }

    // end of class
}

package arti.security;

import java.io.*;

public class BlowFishFile
    extends BlowFish {
    /**
     * Расширение класса blowfish для работы
     * с файлами на вход подается файл для
     * шифровки или дешифровки а также имя
     * файла куда будет сохранен результат преобразования
     */
    FileInputStream fin = null;
    FileOutputStream fout = null;
```

## Практическая криптография в Java. Симметричная криптография

```
DataInputStream din = null;
DataOutputStream dout = null;

String FSourceName = null; // имя файла источника данных для
криптопреобразования
String FDestinationName = null; // имя файла приемника результата преобразования

boolean Mode = true;

// размер массива будет больше чем
// планируется для того чтобы в нем поместились
// данные еще одного блока для нормализации размера файла,
// помните, что blowfish – блочный
// алгоритм шифрования с размером блока 64 бита
byte[] IOBuffer = new byte[BUFFERSIZE * 8 + 8];

public BlowFishFile(String From, String To) {
    FSourceName = From; // имя файла источника
    FDestinationName = To; // имя файла приемника
    settedKey = false; // признак того был ли инициализирован ключ
шифрования/дешифрования
}

public void start(boolean b) throws SecurityException {
    Mode = b; // режим шифровать или дешифровать
    if (Mode == BlowFish.ENCRYPT) {
        StartEncrypt();
    }
    else {
        StartDecrypt();
    }
}

protected void StartEncrypt() throws SecurityException {
    if (FSourceName == null || FDestinationName == null) {
        throw new SecurityException("Не указаны файлы
источники данных и приемник данных");
    }
    try {
```

## Практическая криптография в Java. Симметричная криптография

```
fin = new FileInputStream(FSourceName);
din = new DataInputStream(fin);

fout = new FileOutputStream(FDestinationName);
dout = new DataOutputStream(fout);
int count = 0;

File Fl = new File(FSourceName);
long available = Fl.length();
Fl = null;

int Steps = (int) available / (8 * BUFFERSIZE) + 1;

while ( (count = din.read(IOBuffer, 0, 8 * BUFFERSIZE)) != -1) {
    // оставим про запас еще 8 байт
    // количество прочитанных байтов, а не блоков по 64 бита

    available -= count;

    if (available == 0) { // последний блок нормализовать
        count = normalizeBlock(count, IOBuffer);
    }

    encryptBlocks(count / 8, IOBuffer); // число блоков

    dout.write(IOBuffer, 0, count); // зашифрованные данные
    //на выход
}
din.close();
dout.close();
}
catch (java.io.IOException e) {
    System.out.println(e);
    throw new SecurityException("Ошибка на стадии
    шифрования данных "+e);
}
}

protected void StartDecrypt() throws SecurityException {
```

## Практическая криптография в Java. Симметричная криптография

```
try {
    fin = new FileInputStream(FSourceName);
    din = new DataInputStream(fin);

    fout = new FileOutputStream(FDestinationName);
    dout = new DataOutputStream(fout);

    int count = 0;

    File Fl = new File(FSourceName);
    long available = Fl.length();
    Fl = null;

    int Steps = (int) available / (8 * BUFFERSIZE) + 1;

    while ( (count = din.read(IOBuffer, 0, 8 * BUFFERSIZE)) != -1) {
        if ( (count % 8) != 0) {
            throw new SecurityException("Ошибка в файле содержится
            не полное число блоков");
        }

        decryptBlocks(count / 8, IOBuffer);

        available -= count;

        // опять все данные в байтах
        if (available == 0) {
            count = deNormalizeBlock(count, IOBuffer);
            // теперь в счетчике число актуальных байт
            //без хвоста
        }
        dout.write(IOBuffer, 0, count);
    }

    din.close();
    dout.close();
}
catch (java.io.IOException e) {
    System.out.println(e);
    throw new SecurityException("Ошибка ввода вывода на стадии
```

## Практическая криптография в Java. Симметричная криптография

```
        дешифровки данных");
    }

}

// end of class
}

package arti.security;

import java.io.*;

public class KLine {
    public int KLine[] = {
        0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344, 0xa4093822, 0x299f31d0,
        0x082efa98, 0xec4e6c89, 0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
        0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917, 0x9216d5d9, 0x8979fb1b
    };

    public void print(PrintStream d) throws IOException {

        for (int i = 0; i < KLine.length; i++) {
            d.println(BlowFish.IntToLong(KLine[i]));
        }
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof KLine)) {
            return false;
        }
        int i, n = KLine.length;

        if (n != 18) {
            System.out.println("Длина ключа не совпадает с заданной");
        }
        KLine k = (KLine) obj;

        for (i = 0; i < n; i++) {
            if (KLine[i] != k.KLine[i]) {
```

## Практическая криптография в Java. Симметричная криптография

```
        return false;
    }
}
return true;
}
}

package arti.security;

import java.io.*;

public class Q_1_Grid {

    public void print(OutputStream d) throws IOException {
        DataOutputStream dout = new DataOutputStream(d);
        for (int i = 0; i < QGrid.length; i++) {
            dout.writeInt(QGrid[i]);
        }
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Q_1_Grid)) {
            return false;
        }
        int i, n = QGrid.length;

        if (n != 256) {
            System.out.println("Длина ключа не совпадает
            с заданной -Q0-");
        }

        Q_1_Grid k = (Q_1_Grid) obj;

        for (i = 0; i < n; i++) {
            if (QGrid[i] != k.QGrid[i]) {
                return false;
            }
        }
        return true;
    }
}
```

```
public int QGrid[] = {  
    0xd1310ba6,  
    0x98dfb5ac,  
    0x2ffd72db,  
    0xd01adfb7,  
    0xb8e1afed,  
    0x6a267e96,  
    0xba7c9045,  
    0xf12c7f99,  
    0x24a19947,  
    0xb3916cf7,  
    0x0801f2e2,  
    0x858efc16,  
    0x636920d8,  
    0x71574e69,  
    0xa458fea3,  
    0xf4933d7e,  
    0x0d95748f,  
    0x728eb658,  
    0x718bcd58,  
    0x82154aee,  
    0x7b54a41d,  
    0xc25a59b5,  
    0x9c30d539,  
    0x2af26013,  
    0xc5d1b023,  
    0x286085f0,  
    0xca417918,  
    0xb8db38ef,  
    0x8e79dcb0,  
    0x603a180e,  
    0x6c9e0e8b,  
    0xb01e8a3e,  
    0xd71577c1,  
    0xbd314b27,  
    0x78af2fda,  
    0x55605c60,  
    0xe6525f3,  
    0xaa55ab94,
```

## Практическая криптография в Java. Симметричная криптография

```
0x57489862,  
0x63e81440,  
0x55ca396a,  
0x2aab10b6,  
0xb4cc5c34,  
0x1141e8ce,  
0xa15486af,  
0x7c72e993,  
0xb3ee1411,  
0x636fbc2a,  
0x2ba9c55d,  
0x741831f6,  
0xce5c3e16,  
0x9b87931e,  
0xafd6ba33,  
0x6c24cf5c,  
0x7a325381,  
0x28958677,  
0x3b8f4898,  
0x6b4bb9af,  
0xc4bfe81b,  
0x66282193,  
0x61d809cc,  
0xfb21a991,  
0x487cac60,  
0x5dec8032,  
0xef845d5d,  
0xe98575b1,  
0xdc262302,  
0xeb651b88,  
0x23893e81,  
0xd396acc5,  
0x0f6d6ff3,  
0x83f44239,  
0x2e0b4482,  
0xa4842004,  
0x69c8f04a,  
0x9e1f9b5e,  
0x21c66842,  
0xf6e96c9a,
```

```
0x670c9c61,  
0xabd388f0,  
0x6a51a0d2,  
0xd8542f68,  
0x960fa728,  
0xab5133a3,  
0x6eef0b6c,  
0x137a3be4,  
    0xba3bf050,  
0x7efb2a98,  
0xa1f1651d,  
0x39af0176,  
0x66ca593e,  
0x82430e88,  
0x8cee8619,  
0x456f9fb4,  
0x7d84a5c3,  
0x3b8b5ebe,  
0xe06f75d8,  
0x85c12073,  
0x401a449f,  
0x56c16aa6,  
0x4ed3aa62,  
0x363f7706,  
0x1bfedf72,  
0x429b023d,  
0x37d0d724,  
0xd00a1248,  
0xdb0fead3,  
0x49f1c09b,  
0x075372c9,  
0x80991b7b,  
0x25d479d8,  
0xf6e8def7,  
0xe3fe501a,  
0xb6794c3b,  
0x976ce0bd,  
0x04c006ba,  
0xc1a94fb6,  
0x409f60c4,
```

## Практическая криптография в Java. Симметричная криптография

```
0x5e5c9ec2,  
0x196a2463,  
0x68fb6faf,  
0x3e6c53b5,  
0x1339b2eb,  
0x3b52ec6f,  
0x6dfc511f,  
0x9b30952c,  
0xcc814544,  
0xaf5ebd09,  
0xbee3d004,  
0xde334afd,  
0x660f2807,  
0x192e4bb3,  
0xc0cba857,  
0x45c8740f,  
0xd20b5f39,  
0xb9d3fbdb,  
0x5579c0bd,  
0x1a60320a,  
0xd6a100c6,  
0x402c7279,  
0x679f25fe,  
0xfb1fa3cc,  
0x8ea5e9f8,  
    0xdb3222f8,  
0x3c7516df,  
0xfd616b15,  
0x2f501ec8,  
0xad0552ab,  
0x323db5fa,  
0xfd238760,  
0x53317b48,  
0x3e00df82,  
0x9e5c57bb,  
0xca6f8ca0,  
0x1a87562e,  
0xdf1769db,  
0xd542a8f6,  
    0x287effc3,
```

## Практическая криптография в Java. Симметричная криптография

```
0xac6732c6,  
0x8c4f5573,  
0x695b27b0,  
0xbbca58c8,  
0xe1ffa35d,  
0xb8f011a0,  
0x10fa3d98,  
0xfd2183b8,  
0x4afcb56c,  
0x2dd1d35b,  
0x9a53e479,  
0xb6f84565,  
0xd28e49bc,  
0x4bfb9790,  
0xe1ddf2da,  
0xa4cb7e33,  
0x62fb1341,  
0xcee4c6e8,  
0xef20cada,  
0x36774c01,  
0xd07e9efe,  
0x2bf11fb4,  
0x95dbda4d,  
0xae909198,  
0xeaad8e71,  
0x6b93d5a0,  
0xd08ed1d0,  
0xafc725e0,  
0x8e3c5b2f,  
0x8e7594b7,  
0x8ff6e2fb,  
0xf2122b64,  
0x8888b812,  
0x900df01c,  
0x4fad5ea0,  
0x688fc31c,  
0xd1cff191,  
0xb3a8c1ad,  
0x2f2f2218,  
0xbe0e1777,
```

```
0xea752dfe,  
0x8b021fa1,  
0xe5a0cc0f,  
0xb56f74e8,  
0x18acf3d6,  
0xce89e299,  
0xb4a84fe0,  
0xfd13e0b7,  
0x7cc43b81,  
0xd2ada8d9,  
0x165fa266,  
0x80957705,  
0x93cc7314,  
0x211a1477,  
0xe6ad2065,  
0x77b5fa86,  
    0xc75442f5,  
0xfb9d35cf,  
0xebcdaf0c,  
0x7b3e89a0,  
0xd6411bd3,  
0xae1e7e49,  
0x00250e2d,  
0x2071b35e,  
0x226800bb,  
0x57b8e0af,  
0x2464369b,  
0xf009b91e,  
0x5563911d,  
0x59dfa6aa,  
0x78c14389,  
0xd95a537f,  
0x207d5ba2,  
0x02e5b9c5,  
0x83260376,  
0x6295cfa9,  
0x11c81968,  
0x4e734a41,  
0xb3472dca,  
0x7b14a94a,
```

```
    0x1b510052,  
    0x9a532915,  
    0xd60f573f,  
    0xbc9bc6e4,  
    0x2b60a476,  
    0x81e67400,  
    0x08ba6fb5,  
    0x571be91f,  
    0xf296ec6b,  
    0x2a0dd915,  
    0xb6636521,  
    0xe7b9f9b6,  
    0xff34052e,  
    0xc5855664,  
    0x53b02d5d,  
    0xa99f8fa1,  
    0x08ba4799,  
    0x6e85076a  
};  
  
}  
  
package arti.security;  
  
import java.io.*;  
  
public class Q_2_Grid {  
  
    public void print(OutputStream d) throws IOException {  
        DataOutputStream dout = new DataOutputStream(d);  
        for (int i = 0; i < QGrid.length; i++) {  
            dout.writeInt(QGrid[i]);  
        }  
    }  
  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Q_2_Grid)) {  
            return false;  
        }  
        int i, n = QGrid.length;
```

```
if (n != 256) {
    System.out.println("Длина ключа не
        совпадает с заданной -Q1-");
}
Q_2_Grid k = (Q_2_Grid) obj;

for (i = 0; i < n; i++) {
    if (QGrid[i] != k.QGrid[i]) {
        return false;
    }
}
return true;
}

public int QGrid[] = {
    0x4b7a70e9,
    0xb5b32944,
    0xdb75092e,
    0xc4192623,
    0xad6ea6b0,
    0x49a7df7d,
    0x9cee60b8,
    0x8fedb266,
    0xecaa8c71,
    0x699a17ff,
    0x5664526c,
    0xc2b19ee1,
    0x193602a5,
    0x75094c29,
    0xa0591340,
    0xe4183a3e,
    0x3f54989a,
    0x5b429d65,
    0x6b8fe4d6,
    0x99f73fd6,
    0xa1d29c07,
    0xefef830f5,
    0x4d2d38e6,
```

## Практическая криптография в Java. Симметричная криптография

```
0xf0255dc1,  
0x4cdd2086,  
0x8470eb26,  
0x6382e9c6,  
0x021ecc5e,  
0x09686b3f,  
    0x3ebaefc9,  
0x3c971814,  
0x6b6a70a1,  
0x687f3584,  
0x52a0e286,  
0xb79c5305,  
0xaa500737,  
0x3e07841c,  
0x7fdeae5c,  
0x8e7d44ec,  
0x5716f2b8,  
0xb03ada37,  
0xf0500c0d,  
0xf01c1f04,  
0x0200b3ff,  
0xae0cf51a,  
0x3cb574b2,  
0x25837a58,  
0xdc0921bd,  
0xd19113f9,  
0x7ca92ff6,  
0x94324773,  
0x22f54701,  
0x3ae5e581,  
0x37c2dadc,  
0xc8b57634,  
0x9af3dda7,  
0xa9446146,  
0x0fd0030e,  
0xecc8c73e,  
0xa4751e41,  
0xe238cd99,  
0x3bea0e2f,  
0x3280bba1,
```

## Практическая криптография в Java. Симметричная криптография

```
0x183eb331,  
0x4e548b38,  
0x4f6db908,  
0x6f420d03,  
0xf60a04bf,  
0x2cb81290,  
0x24977c79,  
0x5679b072,  
0xbcaf89af,  
0xde9a771f,  
0xd9930810,  
0xb38bae12,  
0xdccf3f2e,  
0x5512721f,  
0x2e6b7124,  
0x501adde6,  
0x9f84cd87,  
0x7a584718,  
0x7408da17,  
0xbc9f9abc,  
0xe94b7d8c,  
0xec7aec3a,  
0xdb851dfa,  
    0x63094366,  
0xc464c3d2,  
0xef1c1847,  
0x3215d908,  
0xdd433b37,  
0x24c2ba16,  
0x12a14d43,  
0x2a65c451,  
0x50940002,  
0x133ae4dd,  
0x71dff89e,  
0x10314e55,  
0x81ac77d6,  
0x5f11199b,  
0x043556f1,  
0xd7a3c76b,  
0x3c11183b,
```

## Практическая криптография в Java. Симметричная криптография

```
0x5924a509,  
0xf28fe6ed,  
0x97f1fbfa,  
0x9ebabf2c,  
0x1e153c6e,  
0x86e34570,  
0xae96fb1,  
0x860e5e0a,  
0x5a3e2ab3,  
0x771fe71c,  
0x4e3d06fa,  
0x2965dcb9,  
0x99e71d0f,  
0x803e89d6,  
0x5266c825,  
0x2e4cc978,  
0x9c10b36a,  
0xc6150eba,  
0x94e2ea78,  
0xa5fc3c53,  
0x1e0a2df4,  
0xf2f74ea7,  
0x361d2b3d,  
0x1939260f,  
0x19c27960,  
0x5223a708,  
0xf71312b6,  
0xebadfe6e,  
0xeac31f66,  
0xe3bc4595,  
0xa67bc883,  
0xb17f37d1,  
0x018cff28,  
0xc332ddef,  
0xbe6c5aa5,  
0x65582185,  
0x68ab9802,  
0xeecea50f,  
0xdb2f953b,  
0x2aef7dad,
```

## Практическая криптография в Java. Симметричная криптография

```
0x5b6e2f84,  
0x1521b628,  
0x29076170,  
0xecdd4775,  
0x619f1510,  
0x13cca830,  
0xeb61bd96,  
0x0334fe1e,  
0xaa0363cf,  
0xb5735c90,  
0x4c70a239,  
0xd59e9e0b,  
0xcbaade14,  
0xeecc86bc,  
    0x60622ca7,  
0x9cab5cab,  
0xb2f3846e,  
0x648b1eaf,  
0x19bdf0ca,  
0xa02369b9,  
0x655abb50,  
0x40685a32,  
0x3c2ab4b3,  
0x319ee9d5,  
0xc021b8f7,  
0x9b540b19,  
0x875fa099,  
0x95f7997e,  
0x623d7da8,  
0xf837889a,  
0x97e32d77,  
0x11ed935f,  
0x16681281,  
0x0e358829,  
0xc7e61fd6,  
0x96dedfa1,  
0x7858ba99,  
0x57f584a5,  
0x1b227263,  
0x9b83c3ff,
```

## Практическая криптография в Java. Симметричная криптография

```
0x1ac24696,  
0xcdb30aeb,  
0x532e3054,  
0x8fd948e4,  
0x6dbc3128,  
0x58ebf2ef,  
0x34c6ffea,  
0xfe28ed61,  
0xee7c3c73,  
0x5d4a14d9,  
0xe864b7e3,  
0x42105d14,  
0x203e13e0,  
0x45eee2b6,  
0xa3aaabea,  
0xdb6c4f15,  
0xfacb4fd0,  
0xc742f442,  
0xef6abbb5,  
0x654f3b1d,  
0x41cd2105,  
0xd81e799e,  
0x86854dc7,  
0xe44b476a,  
0x3d816250,  
0xcf62a1f2,  
0x5b8d2646,  
0xfc8883a0,  
0xc1c7b6a3,  
0x7f1524c3,  
0x69cb7492,  
  0x47848a0b,  
0x5692b285,  
0x095bbf00,  
0xad19489d,  
0x1462b174,  
0x23820e00,  
0x58428d2a,  
0x0c55f5ea,  
0x1dadf43e,
```

## Практическая криптография в Java. Симметричная криптография

```
    0x233f7061,  
    0x3372f092,  
    0x8d937e41,  
    0xd65fecf1,  
    0x6c223bdb,  
    0x7cde3759,  
    0xcbee7460,  
    0x4085f2a7,  
    0xce77326e,  
    0xa6078084,  
    0x19f8509e,  
    0xe8efd855,  
    0x61d99735,  
    0xa969a7aa,  
    0xc50c06c2,  
    0x5a04abfc,  
    0x800bcadc,  
    0x9e447a2e,  
    0xc3453484,  
    0xfdd56705,  
    0x0e1e9ec9,  
    0xdb73dbd3,  
    0x105588cd,  
    0x675fda79,  
    0xe3674340,  
    0xc5c43465,  
    0x713e38d8,  
    0x3d28f89e,  
    0xf16dff20,  
    0x153e21e7,  
    0x8fb03d4a,  
    0xe6e39f2b,  
    0xdb83adf7  
};  
  
}  
  
package arti.security;  
  
import java.io.*;
```

```
public class Q_3_Grid {
    public void print(OutputStream d) throws IOException {
        DataOutputStream dout = new DataOutputStream(d);
        for (int i = 0; i < QGrid.length; i++) {
            dout.writeInt(QGrid[i]);
        }
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Q_3_Grid)) {
            return false;
        }
        int i, n = QGrid.length;

        if (n != 256) {
            System.out.println("Длина ключа не
                совпадает с заданной -Q3-");
        }
        Q_3_Grid k = (Q_3_Grid) obj;

        for (i = 0; i < n; i++) {
            if (QGrid[i] != k.QGrid[i]) {
                return false;
            }
        }
        return true;
    }

    public int QGrid[] = {
        0xe93d5a68,
        0x948140f7,
        0xf64c261c,
        0x94692934,
        0x411520f7,
        0x7602d4f7,
        0xbc46b2e,
        0xd4a20068,
        0xd4082471,
```

## Практическая криптография в Java. Симметричная криптография

```
0x3320f46a,  
0x43b7d4b7,  
0x500061af,  
0x1e39f62e,  
0x97244546,  
0x14214f74,  
    0xbf8b8840,  
0x4d95fc1d,  
0x96b591af,  
0x70f4ddd3,  
0x66a02f45,  
0xbfb09ec,  
0x03bd9785,  
0x7fac6dd0,  
0x31cb8504,  
0x96eb27b3,  
0x55fd3941,  
0xda2547e6,  
0xabca0a9a,  
0x28507825,  
    0x530429f4,  
0x0a2c86da,  
0xe9b66dfb,  
0x68dc1462,  
0xd7486900,  
0x680ec0a4,  
0x27a18dee,  
0x4f3ffea2,  
0xe887ad8c,  
0xb58ce006,  
0x7af4d6b6,  
0xaace1e7c,  
0xd3375fec,  
0xce78a399,  
0x406b2a42,  
0x20fe9e35,  
0xd9f385b9,  
0xee39d7ab,  
0x3b124e8b,  
0x1dc9faf7,
```

## Практическая криптография в Java. Симметричная криптография

```
0x4b6d1856,  
0x26a36631,  
0xae397b2,  
0x3a6efa74,  
0xdd5b4332,  
0x6841e7f7,  
0xca7820fb,  
0xfb0af54e,  
0xd8feb397,  
0x454056ac,  
0xba489527,  
0x55533a3a,  
0x20838d87,  
0xfe6ba9b7,  
0xd096954b,  
0x55a867bc,  
0xa1159a58,  
0xcc92963,  
0x99e1db33,  
0xa62a4a56,  
0x3f3125f9,  
0x5ef47e1c,  
0x9029317c,  
0xfd8e802,  
0x04272f70,  
0x80bb155c,  
0x05282ce3,  
0x95c11548,  
0xe4c66d22,  
0x48c1133f,  
0xc70f86dc,  
0x07f9c9ee,  
0x41041f0f,  
0x404779a4,  
0x5d886e17,  
0x325f51eb,  
0xd59bc0d1,  
    0xf2bcc18f,  
0x41113564,  
0x257b7834,
```

```
0x602a9c60,  
0xdff8e8a3,  
0x1f636c1b,  
0x0e12b4c2,  
0x02e1329e,  
0xaf664fd1,  
0xcad18115,  
0x6b2395e0,  
0x333e92e1,  
0x3b240b62,  
0xeebeeb922,  
0x85b2a20e,  
0xe6ba0d99,  
0xde720c8c,  
0x2da2f728,  
0xd0127845,  
0x95b794fd,  
0x647d0862,  
0xe7ccf5f0,  
0x5449a36f,  
0x877d48fa,  
0xc39dfd27,  
0xf33e8d1e,  
0x0a476341,  
0x992eff74,  
0x3a6f6eab,  
0xf4f8fd37,  
0xa812dc60,  
0xa1ebddf8,  
0x991be14c,  
0xdb6e6b0d,  
0xc67b5510,  
0x6d672c37,  
0x2765d43b,  
0xdc0e804,  
0xf1290dc7,  
0xcc00ffa3,  
0xb5390f92,  
0x690fed0b,  
0x667b9ffb,
```

## Практическая криптография в Java. Симметричная криптография

```
0xcedb7d9c,  
0xa091cf0b,  
0xd9155ea3,  
0xbb132f88,  
0x515bad24,  
0x7b9479bf,  
0x763bd6eb,  
0x37392eb3,  
0xcc115979,  
0x8026e297,  
0xf42e312d,  
0x6842ada7,  
0xc66a2b3b,  
0x12754ccc,  
    0x782ef11c,  
0x6a124237,  
0xb79251e7,  
0x06a1bbe6,  
0x4bfb6350,  
0x1a6b1018,  
0x11caedfa,  
0x3d25bdd8,  
0xe2e1c3c9,  
0x44421659,  
0x0a121386,  
0xd90cec6e,  
0xd5abea2a,  
0x64af674e,  
    0xda86a85f,  
0xbebfe988,  
0x64e4c3fe,  
0x9dbc8057,  
0xf0f7c086,  
0x60787bf8,  
0x6003604d,  
0xd1fd8346,  
0xf6381fb0,  
0x7745ae04,  
0xd736fccc,  
0x83426b33,
```

## Практическая криптография в Java. Симметричная криптография

```
0xf01eab71,  
0xb0804187,  
0x3c005e5f,  
0x77a057be,  
0xbde8ae24,  
0x55464299,  
0xbf582e61,  
0x4e58f48f,  
0xf2ddfda2,  
0xf474ef38,  
0x8789bdc2,  
0x5366f9c3,  
0xc8b38e74,  
0xb475f255,  
0x46fcd9b9,  
0x7aeb2661,  
0x8b1ddf84,  
0x846a0e79,  
0x915f95e2,  
0x466e598e,  
0x20b45770,  
0x8cd55591,  
0xc902de4c,  
0xb90bace1,  
0xbb8205d0,  
0x11a86248,  
0x7574a99e,  
0xb77f19b6,  
0xe0a9dc09,  
0x662d09a1,  
0xc4324633,  
0xe85a1f02,  
0x09f0be8c,  
0x4a99a025,  
0x1d6efe10,  
0x1ab93d1d,  
0x0ba5a4df,  
0xa186f20f,  
0x2868f169,  
0xdc7da83,
```

## Практическая криптография в Java. Симметричная криптография

```
0x573906fe,  
0xa1e2ce9b,  
0x4fcd7f52,  
0x50115e01,  
0xa70683fa,  
    0xa002b5c4,  
0x0de6d027,  
0x9af88c27,  
0x773f8641,  
0xc3604c06,  
0x61a806b5,  
0xf0177a28,  
0xc0f586e0,  
0x006058aa,  
0x30dc7d62,  
0x11e69ed7,  
0x2338ea63,  
0x53c2dd94,  
0xc2c21634,  
0xbbcbee56,  
0x90bcb6de,  
0xebfc7da1,  
0xce591d76,  
0x6f05e409,  
0x4b7c0188,  
0x39720a3d,  
0x7c927c24,  
0x86e3725f,  
0x724d9db9,  
0x1ac15bb4,  
0xd39eb8fc,  
0xed545578,  
0x08fca5b5,  
0xd83d7cd3,  
0x4dad0fc4,  
0x1e50ef5e,  
0xb161e6f8,  
0xa28514d9,  
0x6c51133c,  
0x6fd5c7e7,
```

## Практическая криптография в Java. Симметричная криптография

```
    0x56e14ec4,  
    0x362abfce,  
    0xddc6c837,  
    0xd79a3234,  
    0x92638212,  
    0x670efa8e,  
    0x406000e0  
};  
  
}  
  
package arti.security;  
  
import java.io.*;  
  
public class Q_4_Grid {  
  
    public void print(OutputStream d) throws IOException {  
        DataOutputStream dout = new DataOutputStream(d);  
        for (int i = 0; i < QGrid.length; i++) {  
            dout.writeInt(QGrid[i]);  
        }  
    }  
  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Q_4_Grid)) {  
            return false;  
        }  
        int i, n = QGrid.length;  
  
        if (n != 256) {  
            System.out.println("Ошибка длина ключа  
не соответствует заданной -Q4-");  
        }  
        Q_4_Grid k = (Q_4_Grid) obj;  
  
        for (i = 0; i < n; i++) {  
            if (QGrid[i] != k.QGrid[i]) {  
                return false;  
            }  
        }  
    }  
}
```

```
    }  
    return true;  
}  
  
public int QGrid[] = {  
    0x3a39ce37,  
    0xd3faf5cf,  
    0xabc27737,  
    0x5ac52d1b,  
    0x5cb0679e,  
    0x4fa33742,  
    0xd3822740,  
    0x99bc9bbe,  
    0xd5118e9d,  
    0xbf0f7315,  
    0xd62d1c7e,  
    0xc700c47b,  
    0xb78c1b6b,  
    0x21a19045,  
    0xb26eb1be,  
    0x6a366eb4,  
    0x5748ab2f,  
    0xbc946e79,  
    0xc6a376d2,  
    0x6549c2c8,  
    0x530ff8ee,  
    0x468dde7d,  
    0xd5730a1d,  
    0x4cd04dc6,  
    0x2939bbdb,  
    0xa9ba4650,  
    0xac9526e8,  
    0xbe5ee304,  
    0xa1fad5f0,  
    0x6a2d519a,  
    0x63ef8ce2,  
    0x9a86ee22,  
    0xc089c2b8,  
    0x43242ef6,  
    0xa51e03aa,  
}
```

## Практическая криптография в Java. Симметричная криптография

```
0x9cf2d0a4,  
0x83c061ba,  
0x9be96a4d,  
0x8fe51550,  
0xba645bd6,  
0x2826a2f9,  
0xa73a3ae1,  
0x4ba99586,  
0xef5562e9,  
0xc72fefd3,  
0xf752f7da,  
0x3f046f69,  
0x77fa0a59,  
0x80e4a915,  
0x87b08601,  
0x9b09e6ad,  
0x3b3ee593,  
0xe990fd5a,  
0x9e34d797,  
0x2cf0b7d9,  
0x022b8b51,  
0x96d5ac3a,  
0x017da67d,  
0xd1cf3ed6,  
0x7c7d2d28,  
0x1f9f25cf,  
0xadf2b89b,  
0x5ad6b472,  
0x5a88f54c,  
0xe029ac71,  
0xe019a5e6,  
0x47b0acfd,  
0xed93fa9b,  
0xe8d3c48d,  
0x283b57cc,  
0xf8d56629,  
0x79132e28,  
0x785f0191,  
0xed756055,  
0xf7960e44,
```

## Практическая криптография в Java. Симметричная криптография

```
0xe3d35e8c,  
0x15056dd4,  
0x88f46dba,  
0x03a16125,  
0x0564f0bd,  
0xc3eb9e15,  
0x3c9057a2,  
0x97271aec,  
0xa93a072a,  
0x1b3f6d9b,  
0x1e6321f5,  
    0xf59c66fb,  
0x26dcf319,  
0x7533d928,  
0xb155fdf5,  
0x03563482,  
0x8aba3cbb,  
0x28517711,  
0xc20ad9f8,  
0xabcc5167,  
0xccad925f,  
0x4de81751,  
0x3830dc8e,  
0x379d5862,  
0x9320f991,  
0xea7a90c2,  
0xfb3e7bce,  
0x5121ce64,  
0x774fbe32,  
0xa8b6e37e,  
0xc3293d46,  
0x48de5369,  
0x6413e680,  
0xa2ae0810,  
0xdd6db224,  
0x69852dfd,  
0x09072166,  
0xb39a460a,  
0x6445c0dd,  
0x586cdecf,
```

## Практическая криптография в Java. Симметричная криптография

```
0x1c20c8ae,  
0x5bbef7dd,  
0x1b588d40,  
0xccd2017f,  
0x6bb4e3bb,  
0xdda26a7e,  
0x3a59ff45,  
0x3e350a44,  
0xbscb4cdd5,  
0x72eacea8,  
0xfa6484bb,  
0x8d6612ae,  
0xbf3c6f47,  
0xd29be463,  
0x542f5d9e,  
0xaec2771b,  
0xf64e6370,  
0x740e0d8d,  
0xe75b1357,  
0xf8721671,  
0xaf537d5d,  
0x4040cb08,  
0x4eb4e2cc,  
0x34d2466a,  
0x0115af84,  
0xe1b00428,  
0x95983a1d,  
0x06b89fb4,  
0xce6ea048,  
0x6f3f3b82,  
0x3520ab82,  
0x011a1d4b,  
0x277227f8,  
0x611560b1,  
0xe7933fdc,  
0xbb3a792b,  
0x344525bd,  
0xa08839e1,  
0x51ce794b,  
0x2f32c9b7,
```

## Практическая криптография в Java. Симметричная криптография

```
0xa01fbac9,  
0xe01cc87e,  
 0xbcc7d1f6,  
0xcf0111c3,  
0xa1e8aac7,  
0x1a908749,  
0xd44fbd9a,  
0xd0dadecb,  
0xd50ada38,  
0x0339c32a,  
0xc6913667,  
0x8df9317c,  
0xe0b12b4f,  
0xf79e59b7,  
0x43f5bb3a,  
0xf2d519ff,  
0x27d9459c,  
0xbf97222c,  
0x15e6fc2a,  
0x0f91fc71,  
0x9b941525,  
0xfae59361,  
0xceb69ceb,  
0xc2a86459,  
0x12baa8d1,  
0xb6c1075e,  
0xe3056a0c,  
0x10d25065,  
0xcb03a442,  
0xe0ec6e0e,  
0x1698db3b,  
0x4c98a0be,  
0x3278e964,  
0x9f1f9532,  
0xe0d392df,  
0xd3a0342b,  
0x8971f21e,  
0x1b0a7441,  
0x4ba3348c,  
0xc5be7120,
```

## Практическая криптография в Java. Симметричная криптография

```
0xc37632d8,  
0xdf359f8d,  
0x9b992f2e,  
0xe60b6f47,  
0x0fe3f11d,  
0xe54cda54,  
0x1edad891,  
0xce6279cf,  
0xcd3e7e6f,  
0x1618b166,  
0xfd2c1d05,  
0x848fd2c5,  
0xf6fb2299,  
0xf523f357,  
0xa6327623,  
0x93a83531,  
0x56cccd02,  
0xacf08162,  
0x5a75ebb5,  
    0x6e163697,  
0x88d273cc,  
0xde966292,  
0x81b949d0,  
0x4c50901b,  
0x71c65614,  
0xe6c6c7bd,  
0x327a140a,  
0x45e1d006,  
0xc3f27b9a,  
0xc9aa53fd,  
0x62a80f00,  
0xbb25bfe2,  
0x35bdd2f6,  
0x71126905,  
0xb2040222,  
0xb6cbcf7c,  
0xcd769c2b,  
0x53113ec0,  
0x1640e3d3,  
0x38abbd60,
```

```
0x2547adf0,  
0xba38209c,  
0xf746ce76,  
0x77afa1c5,  
0x20756060,  
0x85cbfe4e,  
0x8ae88dd8,  
0x7aaaf9b0,  
0x4cf9aa7e,  
0x1948c25c,  
0x02fb8a8c,  
0x01c36ae4,  
0xd6ebe1f9,  
0x90d4f869,  
0xa65cdea0,  
0x3f09252d,  
0xc208e69f,  
0xb74e6132,  
0xce77e25b,  
0x578fdfe3,  
0x3ac372e6  
};  
}
```

А теперь самое главное: демонстрация работы данного алгоритма. С целью этого я разработал несложный интерфейс формы служащей для шифрования/дешифрования файлов. Ниже я приведу пример интерфейса пользователя, который я разработал для данной иллюстрации.

И наконец исходный код программы создающей данную форму.

```
package arti.security;  
  
import javax.swing.*.*;  
import java.awt.*.*;  
import java.awt.event.*.*;  
  
public class FrmBlowfishFile  
    extends JFrame {
```

```
GridBagLayout gridBagLayout1 = new GridBagLayout();
JTextField txtFileName = new JTextField();
JButton btnSelectInFile = new JButton();
JButton btnSelectOutFile = new JButton();
JTextField txtFileNameOut = new JTextField();
JRadioButton radioCrypt = new JRadioButton();
JRadioButton radioDecrypt = new JRadioButton();
JLabel labMes = new JLabel();
JLabel jLabel2 = new JLabel();
JButton btnGo = new JButton();
JPasswordField txtPassword = new JPasswordField();
ButtonGroup bgMode = new ButtonGroup();

public FrmBlowfishFile() {
    try {
        jbInit();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    FrmBlowfishFile frmBlowfishFile = new FrmBlowfishFile();
    frmBlowfishFile.setSize(400, 200);
    frmBlowfishFile.show();
}

private void jbInit() throws Exception {
    txtFileName.setText("Имя входного файла ");
    setTitle("Blowfish File Cryptooperations");
    this.getContentPane().setLayout(gridBagLayout1);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    btnSelectInFile.setText("Выбрать");
    btnSelectInFile.addActionListener(
        new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btnSelectInFile_actionPerformed(e);
            }
        }
    );
};
```

```
btnSelectOutFile.setText("Выбрать");
btnSelectOutFile.addActionListener(
    new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            btnSelectOutFile_actionPerformed(e);
        }
    });
txtFileOutName.setText("Имя выходного файла ");
radioCrypt.setText("Шифровать");
radioDecrypt.setText("Дешифровать");
labMes.setHorizontalAlignment(SwingConstants.CENTER);
labMes.setText("Выберите режим работы:
    шифрование или дешифрование");
jLabel2.setText("Секретный ключ");
btnGo.setText("Старт");
btnGo.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        btnGo_actionPerformed(e);
    }
});
txtPassword.setAlignmentX( (float) 0.5);
txtPassword.setToolTipText("");
txtPassword.setText("");
this.getContentPane().add(txtFileInName,
    new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(btnSelectInFile,
    new GridBagConstraints(1, 0, 1, 1, 1.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.NONE,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(btnSelectOutFile,
    new GridBagConstraints(1, 1, 1, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.NONE,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(txtFileOutName,
    new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(radioCrypt,
```

## Практическая криптография в Java. Симметричная криптография

```
        new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.NONE,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(radioDecrypt,
    new GridBagConstraints(1, 3, 1, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.NONE,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(labMes,
    new GridBagConstraints(0, 2, 2, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(jLabel2,
    new GridBagConstraints(0, 4, 1, 1, 1.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.NONE,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(btnGo,
    new GridBagConstraints(0, 5, 2, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
    new Insets(0, 0, 0, 0), 0, 0));
this.getContentPane().add(txtPassword,
    new GridBagConstraints(1, 4, 1, 1, 0.0, 0.0
    , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
    new Insets(0, 0, 0, 0), 0, 0));
bgMode.add(radioCrypt);
bgMode.add(radioDecrypt);
}

void btnSelectInFile_actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showOpenDialog(FrmBlowfishFile.this);
    if (returnVal == JFileChooser.APPROVE_OPTION) {

        txtFileName.setText(chooser.getSelectedFile().getAbsolutePath());
    }
}

void btnSelectOutFile_actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showSaveDialog(FrmBlowfishFile.this);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
```

```
        txtFileOutName.setText(chooser.getSelectedFile().getAbsolutePath());
    }
}

void btnGo_actionPerformed(ActionEvent e) {
    if (radioCrypt.isSelected()) {
        BlowFishFile bf = new BlowFishFile(txtFileInName.getText(),
                                           txtFileOutName.getText());

        try {
            bf.setKey(txtPassword.getText().getBytes());
            bf.initKey();
        }
        catch (SecurityException ex) {
            JOptionPane.showMessageDialog(this,
                "Ошибка на стадии инициализации ключа:\n" + ex);
            return;
        }
        try {
            bf.start(BlowFish.ENCRYPT);
            JOptionPane.showMessageDialog(this,
                "Операция шифрования успешно завершена");
        }
        catch (SecurityException ex1) {
            JOptionPane.showMessageDialog(this,
                "Ошибка на стадии шифрования:\n" + ex1);
            return;
        }
    }

    else
    if (radioDecrypt.isSelected()) {
        BlowFishFile bf = new BlowFishFile(txtFileInName.getText(),
                                           txtFileOutName.getText());

        try {
            bf.setKey(txtPassword.getText().getBytes());
            bf.initKey();
        }
        catch (SecurityException ex) {
            JOptionPane.showMessageDialog(this,
```

```
        "Ошибка на стадии инициализации ключа :\n" + ex);
        return;
    }
    try {
        bf.start(BlowFish.DECRYPT);
        JOptionPane.showMessageDialog(this,
            "Операция дешифрования успешно завершена ");
    }
    catch (SecurityException ex1) {
        JOptionPane.showMessageDialog(this,
            "Ошибка на стадии дешифрования:\n" + ex1);
        return;
    }
}
}
```

## **7. Стоит ли изобретать велосипед?**

Разумеется, существует большое количество реализаций Blowfish и многих других криптоалгоритмов. Некоторые из них являются коммерческими, другие бесплатными и свободны для использования. Однако не следует рассматривать наши усилия как бесполезные и повторяющие чужие наработки. Любые усилия которые мы затрачиваем изменяют нас, дают возможность приобрести новые знания и расширить собственный кругозор. Если поискать в сети, то можно найти достаточное количество сайтов компаний предлагающих свои решения. Однако чтобы называть себя действительно хорошим специалистом, следует попробовать сделать что-то свое.

## **8. Заключение**

Подведем итог тому, что мы узнали. Мы разобрали роль которую играют современные криптографические алгоритмы при построении реальных приложений. Мы узнали о задачах решаемых науками входящими в состав криптологии. мы рассмотрели прием максимально прозрачной интеграции криптоалгоритмов в вашу систему, с минимальными затратами на изменение существующего кода. Мы рассмотрели один из алгоритмов появившихся за последние годы: Blowfish и создали его практическую

реализацию. Следующим шагом нашего повествования будет введение в методы асимметричной криптографии.

## **9. Ресурсы**

<http://java.sun.com/products/jce/index-14.html> — где вы еще узнаете о модели криптографии как не тут.

<http://alexeenko.prima.susu.ac.ru> — просто и доступно о разных алгоритмах. Есть реализация на C/C++.

<http://www.docs.h1.ru/crypt.html> — читать, читать, и еще раз читать.

<http://www.google.com> — искать и находить, в сети очень много документации в том числе на русском, правда я в основном пользуюсь иноязычными ресурсами, так свою первую реализацию blowfish я сделал анализируя исходные коды на ASM, с комментариями на немецком языке.