**Федеральное государственное автономное учреждение высшего профессионального образования «Национальный исследовательский университет «Высшая школа экономики»**

МИЭМ НИУ ВШЭ

Кафедра «Кибернетика»

**ОТЧЕТ**

**К ЛАБОРАТОРНОЙ РАБОТЕ**

**по дисциплине**

**«Информационная безопасность и защита информации»**

Работу выполнил

студент группы МС-91

А. С. Ковалевский

Москва, 2014

**Условие**

Разработать алгоритмы, реализующие криптографические протоколы (см. вариант) взаимодействия удаленных абонентов на основе асимметричных криптосистем.

Написать программы, реализующие эти протоколы для всех участников. Значения модуля криптосистемы выбирать не менее 50 бит. Для вычислений с большими числами можно использовать специальные программы.

Для проверки чисел на простоту использовать комбинированный алгоритм на основе тестов Леманна или Рабина-Миллера

Хеширование выполнять на основе любого блочного симметричного алгоритма (с использованием алгоритма/программы из предыдущих лаб. работ) по одной из схем, данных в лекциях.

Проверить правильность выполнения протокола для малых значений параметров криптосистемы (контрольный пример).

Продемонстрировать выполнение протокола для больших значений параметров криптосистемы.

**Вариант 5**

Протокол обмена секретным документом комбинированным методом шифрования на основе экспоненциального ключевого обмена по методу Диффи-Хеллмана

**Теория**

**Комбинированный метод шифрования.**

Комбинированный метод шифрования позволяет сочетать преимущества высокой секретности, предоставляемые асимметричными криптоистемами с открытым ключом, с преимуществами высокой скорости работы, присущими симметричным криптосистемам с секретным ключом. При таком подходе криптосистема с открытым ключом применяется для шифрования, передачи и последующего расшифрования только секретного ключа симметричной криптосистемы. А симметричная криптосистема применяется для шифрования и передачи исходного открытого текста. В результате криптосистема с открытым ключом не заменяет симметричную криптосистему с секретным ключом, а лишь дополняет ее, позволяя повысить в целом защищенность передаваемой информации.

Пользователи А и В, использующие комбинированный метод шифрования, имеют каждый по паре асимметричных ключей шифрования

(*К*Ао, *К*Ас) и (*К*Во, *К*Вс).

Если пользователь А хочет передать зашифрованное комбинированным методом сообщение *m* пользователю В, то порядок его действий будет таков.

1. Создать (например, сгенерировать случайным образом) симметричный ключ, называемый в этом методе сеансовым ключом *Ks*.

2. Зашифровать сообщение *m* на сеансовом ключе *Ks*.

3. Зашифровать сеансовый ключ *Ks* на открытом ключе *К*Во пользователя В и своем секретном ключе *К*Ас.

4. Передать по открытому каналу связи в адрес пользователя В зашифрованное сообщение вместе с зашифрованным сеансовым ключом.

Действия пользователя В при получении зашифрованного сообщения и зашифрованного сеансового ключа должны быть обратными:

5. Расшифровать на своем секретном ключе *К*Вс и открытом ключе *К*Ао пользователя А сеансовый ключ *K*s.

6. С помощью полученного сеансового ключа *K*s расшифровать и прочитать сообщение *m*.

При использовании комбинированного метода шифрования можно быть уверенным в том, что только пользователь В сможет правильно расшифровать ключ *K*s и прочитать сообщение *m*.

***Выбор длины ключей в комбированном методе шифрования.*** Таким образом, при комбинированном методе шифрования применяются криптографические ключи как симметричных, так и асимметричных криптосистем. Очевидно, выбор длин ключей для каждого типа криптосистемы следует осуществлять таким образом, чтобы злоумышленнику было одинаково трудно атаковать любой механизм защиты комбинированной криптосистемы.

В таблице 5.1. приведены распространенные длины ключей симметричных и асимметричных криптосистем, для которых трудность атаки полного перебора примерно равна трудности факторизации соответствующих модулей асимметричных криптосистем.

Таблица 5.1. Длины ключей для симметричных и асимметричных криптосистем при одинаковой их криптостойкости

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Симметричные криптосистемы | 56 | 64 | 80 | 112 | 128 |
| Асимметричные криптосистемы | 384 | 512 | 786 | 1792 | 2304 |

**Метод экспоненциального ключевого обмена Диффи-Хеллмана**

Одни из авторов идеи криптосистем с открытым ключом, Диффи и Хеллман предложили новую идею - *открытое распределение ключей.*

Они задались вопросом: можно ли организовать такую процедуру взаимодействия абонентов А и В по открытым каналам связи, чтобы решить следующие задачи:

1) вначале у А и В нет никакой общей секретной информации, но в конце процедуры такая общая секретная информация (общий ключ) у А и В появляется, т. е. вырабатывается;

2) пассивный противник, который перехватывает все передачи информации и знает, что хотят получить А и В, тем не менее не может восстановить выработанный общий ключ А и В.

Метод получил название метода экспоненциального ключевого обмена. Он был первой криптосистемой с открытым ключом, хотя для обмена ключами можно использовать любые криптосистемы с открытым ключом, например, тот же алгоритм RSA.

Криптостойкость данного метода определяется трудоемкостью вычисления дискретного логарифма:

*f*(*х*) = *gх* mod *р,*

где *р —* большое простое число, *х —* произвольное натуральное число, *g —* некоторый *примитивный элемент* поля Галуа *GF*(*р*). Общепризнано, что инвертирование функции *gх* mod *р*, т.е. дискретное логарифмирование, является трудной математической задачей.

Сама процедура или *протокол выработки общего ключа* заключается в следующем. Значения *р* и *g* являются общедоступными параметрами протокола. Абоненты A и В независимо друг от друга случайно выбирают по одному большому натуральному числу *x*а и *xb* . Это их секретные ключи. Далее каждый из них вычисляет открытые ключи:

*y*а = *gx*а mod *р* и *у*В = *gxB* mod *р.*

Потом они обмениваются этими элементами по каналу связи. Теперь абонент А, получив *y*В и зная свой секретный элемент *х*а*,* вычисляет новый элемент:

*у*В*x*а mod *р* = (*gxB*) *x*а mod *р.*

Аналогично поступает абонент *В:*

*yAxB* mod *р* = (*gx*а) *xB* mod *р.*

Тем самым у А и В появился общий элемент поля, равный *gx*а *xB.* Этот элемент и объявляется общим ключом абонентов A и В.

Необратимость преобразования в этом случае обеспечивается тем, что достаточно легко вычислить показательную функцию в конечном поле Галуа, состоящем из *р* элементов (*р* — простое число). Обратная задача вычисления *x* из *y* будет достаточно сложной. Если *р* выбрано достаточно правильно, то извлечение логарифма потребует вычислений, пропорциональных L(*р*) = *exр*{ (*ln* *р ln* *ln* *р*)0.5}.

При всей простоте алгоритма Диффи-Хеллмана его недостатком по сравнению с системой RSA является отсутствие гарантированной нижней оценки трудоемкости раскрытия ключа.

**Алгоритм Рабина-Миллера** (Rabin-Miller)

Повсеместно используемым является простой алгоритм, разработанный М. Рабином, частично основанным на идеях Миллера.

Выберите для проверки случайное число *р.* Вычислите *b* – число делений *р-*1 на 2 (т.е. 2*b* – это наибольшая степень числа 2, на которую делится *р-*1). Затем вычислите *т,* такое, что *р =* 1 + 2*b* · *т.*

1) Выберите случайное число *а,* меньшее *р.*

2) Установите *j* = 0 и *z = a*· *m* mod *p.*

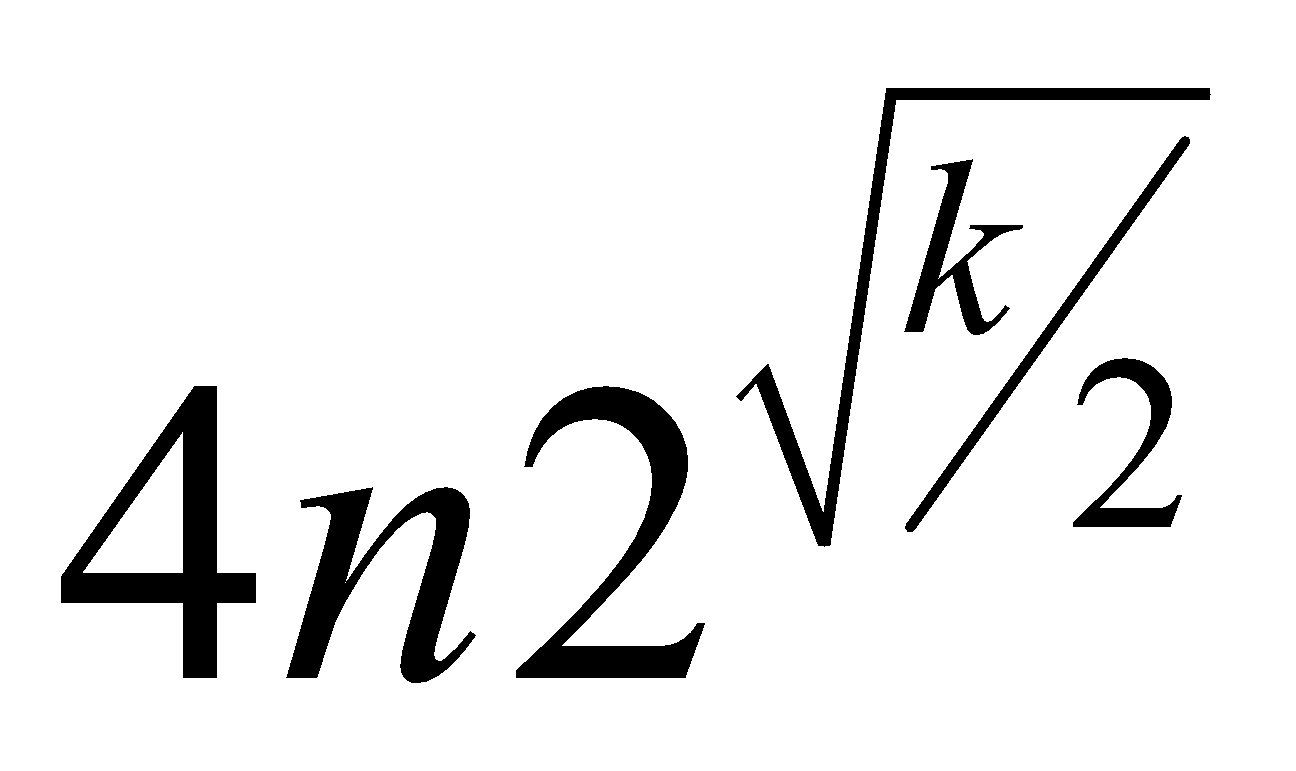
3) Если *z* = 1 или если *z = р-*1, то *р* проходит проверку и может быть простым числом.

4) Если *j*> 0 и *Z* = 1, то *p* не является простым числом.

5) Установите *j= j* + 1. Если *j* < *b* и *z* < *р* - 1, установите *z* = *z2* mod *р* и вернитесь на этап (4). Если *z* = *р* - 1, то *р* проходит проверку и может быть простым числом.

6) Если *j = b* и *z* ≠ *р* - 1, то *р* не является простым числом.

В этом тесте вероятность прохождения проверки составным числом убывает быстрее, чем в предыдущих. Гарантируется, что три четверти возможных значений *а* окажутся свидетелями. Это означает, что составное число проскользнет через *t* проверок с вероятностью, не большей 1/4*t*, где *t* - число итераций. На самом деле и эти оценки слишком пессимистичны. Для большинства случайных чисел около 99.9 процентов возможных значений являются свидетелями (Шнайер Б. Прикладная криптография, 2-е изд.: протоколы, алгоритмы, исходные тексты на языке Си).

Существуют более точные оценки (Шнайер Б. Прикладная криптография, 2-е изд.: протоколы, алгоритмы, исходные тексты на языке Си). Для *n*-битового кандидата в простые числа (где *п >*100) вероятность ошибки в одном тесте меньше, чем . И для 256-битового *n* вероятность ошибки в шести тестах меньше, чем 1/251.

***Практические соображения***

В реальных приложениях генерация простых чисел происходит быстро. В книге Шайера Б. «Прикладная криптография» описан следующий алгоритм:

1) Сгенерируйте случайное *n*-битовое число *р.*

2) Установите старший и младший биты равными 1. (Старший бит гарантирует требуемую длину простого числа, а младший бит обеспечивает его нечетность.)

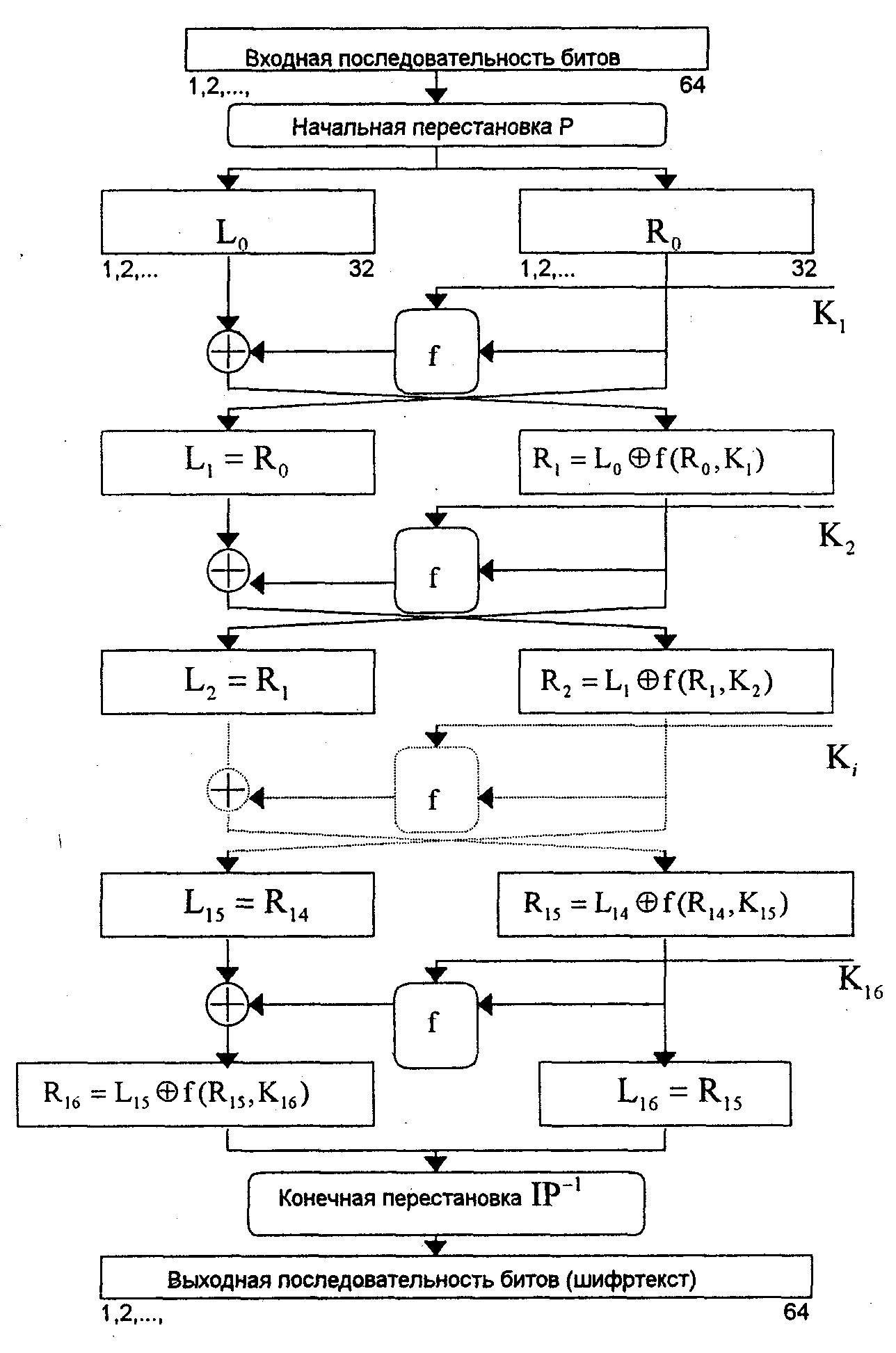
3) Убедитесь, что *р* не делится на небольшие простые числа: 3, 5, 7, 11 и т.д. Во многих реализациях проверяется делимость *р* на все простые числа, меньшие 256. Наиболее эффективной является проверка на делимость для всех простых чисел, меньших 2000.

4) Выполните тест Рабина-Миллера для некоторого случайного *а.* Если *р* проходит тест, сгенерируйте другое случайное *а* и повторите проверку. Выбирайте небольшие значения *а* для ускорения вычислений. Выполните пять тестов. (Одного может показаться достаточным, но выполните пять.) Если *р* не проходит одной из проверок, сгенерируйте другое *р* и попробуйте снова.

Иначе, можно не генерировать *р* случайным образом каждый раз, но последовательно перебирать числа, начиная со случайно выбранного до тех пор, пока не будет найдено простое число.

Этап 3 не является обязательным, но это хорошая идея. Проверка, что случайное нечетное *р* не делится на 3, 5 и 7 отсекает 54% нечетных чисел еще до этапа 4. Проверка делимости на все простые числа, меньшие 100, убирает 76% нечетных чисел, проверка делимости на все простые числа, меньшие 256, убирает 80% нечетных чисел. В общем случае, доля нечетных кандидатов, которые не делятся ни на одно простое число, меньшее *m*, равна 1.12/1*n* *m.* Чем больше проверяемое *m*, тем больше предварительных вычислений нужно выполнить до теста Рабина-Миллера.

**Структура симметричного алгоритма шифрования DES**



**Практика**

При выполнении лабораторной работы были разработаны следующие классы:

* Alice и Bob – классы-имитаторы передающей и принимающей сторон.
* DesEncrypter – класс необходимый реализации DES-алгоритма шифрования документа;
* Dialog – по сути класс-контроллер, который осуществляет связь между Alice и Bob.
* Functions – класс с необходимыми фукнциями для реализации задачи.
* Main – класс для инициализации “диалога”. Помимо этого в этом классе выполняем DES-шифрование.

**Alice.java (исходный код в приложении 1а)**

Класс Alice.java хранит публичные и закрытые ключи и параметры, необходимые для передачи данных другому классу, а также общий секретный ключ.

**Bob.java (исходный код в приложении 1b)**

См. Описание Alice.java

**DesEncrypter.java (исходный код в приложении 2)**

Класс DesEncrypter.javaреализует симметричный алгоритм шифрования DES. Содержит методы для шифрования и дешифрования данных.

**Functions.java (исходный код в приложении 3)**

Класс, реализующий методы для генерации простых чисел, возведения в степень по модулю, проверки числа на простоту, согласно тесту Рабина-Миллера.

**Dialog.java (исходный код в приложении 4)**

Класс-контроллер Diaolg.java использует методы классов Functions, Alice и Bob для передачи необходимых данных между этими классами. Также реализует тестовый метод для сравнения ключей.

**Main.java (исходный код в приложении 5)**

Основная задача этого класса, зашифровать с помощью полученного от классов Alice и Bob секретного ключа данные методами класса DesEncrypter.java, также реализовано расшифровка этих данных в отдельный файл.

**Приложение 1а**

import java.math.BigInteger;  
public class Alice {  
 public static BigInteger *q*; // Закрытый ключ А  
 public static BigInteger *p*; // Открытый параметр  
 private static BigInteger *a*; // Закрытый ключ - Х1  
 private static BigInteger *alice*; // Открытый ключ У1  
 private static BigInteger *secret*; // Общий секретный ключ Z  
  
 public static BigInteger getSecret() {  
 return *secret*;  
 }  
  
 public static void setSecret(BigInteger secret) {  
 Alice.*secret* = secret;   
 }  
  
 public static BigInteger get\_q() {  
 return *q*;  
 }  
  
 public static void set\_q(BigInteger q) {  
 Alice.*q* = q;   
 }  
  
 public static BigInteger get\_p() {  
 return *p*;  
 }  
  
 public static void set\_p(BigInteger p) {  
 Alice.*p* = p;   
 }  
  
 public static BigInteger get\_a() {  
 return *a*;  
 }  
  
 public static void set\_a(BigInteger a) {  
 Alice.*a* = a;   
 }  
  
 public static BigInteger getAlice() {  
 return *alice*;  
 }  
  
 public static void setAlice(BigInteger alice) {  
 Alice.*alice* = alice;   
 }  
  
 public static BigInteger get\_B\_fromBob(){  
 BigInteger bob = Bob.*getBob*();  
 return bob;  
 }  
}

**Приложение 1b**

import java.math.BigInteger;  
  
public class Bob {  
 private static BigInteger *b*; // Закрытый ключ Х2  
 private static BigInteger *bob*; //Открытый ключ У2  
 public static BigInteger *p*; // Открытый параметр  
 public static BigInteger *q*; // Открытый параметр  
 private static BigInteger *secret*; // Общий секретный ключ Z  
  
 public static BigInteger getSecret() {  
 return *secret*;  
 }  
  
 public static void setSecret(BigInteger secret) {  
 Bob.*secret* = secret;   
 }  
  
 public static BigInteger get\_b() {  
 return *b*;  
 }  
  
 public static void set\_b(BigInteger b) {  
 Bob.*b* = b;   
 }  
  
 public static BigInteger get\_p() {  
 return *p*;  
 }  
  
 public static void set\_p(BigInteger p) {  
 Bob.*p* = p;   
 }  
  
 public static BigInteger get\_q() {  
 return *q*;  
 }  
  
 public static void set\_q(BigInteger q) {  
 Bob.*q* = q;   
 }  
  
 public static BigInteger getBob() {  
 return *bob*;  
 }  
  
 public static void setBob(BigInteger bob) {  
 Bob.*bob* = bob;   
 }  
  
 public static BigInteger get\_A\_fromAlice(){  
 BigInteger alice = Alice.*getAlice*();  
 return alice;  
 }  
  
 public static BigInteger get\_q\_fromAlice(){  
 BigInteger q = Alice.*get\_q*();  
 *set\_q*(q);  
 return q;  
 }  
 public static BigInteger get\_p\_fromAlice(){  
 BigInteger p = Alice.*get\_p*();  
 *set\_p*(p);  
 return p;  
 }  
}

**Приложение 2**

import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
  
import javax.crypto.Cipher;  
import javax.crypto.CipherInputStream;  
import javax.crypto.CipherOutputStream;  
import javax.crypto.SecretKey;  
import javax.crypto.SecretKeyFactory;  
import javax.crypto.spec.DESKeySpec;  
  
public class DesEncrypter {  
  
  
 public static void encrypt(String key, InputStream is, OutputStream os) throws Throwable {  
 *encryptOrDecrypt*(key, Cipher.*ENCRYPT\_MODE*, is, os);  
 }  
  
 public static void decrypt(String key, InputStream is, OutputStream os) throws Throwable {  
 *encryptOrDecrypt*(key, Cipher.*DECRYPT\_MODE*, is, os);  
 }  
  
 public static void encryptOrDecrypt(String key, int mode, InputStream is, OutputStream os) throws Throwable {  
  
 DESKeySpec dks = new DESKeySpec(key.getBytes());  
 SecretKeyFactory skf = SecretKeyFactory.*getInstance*("DES");  
 SecretKey desKey = skf.generateSecret(dks);  
 Cipher cipher = Cipher.*getInstance*("DES"); // DES/ECB/PKCS5 Padding for SunJCE  
  
 if (mode == Cipher.*ENCRYPT\_MODE*) {  
 cipher.init(Cipher.*ENCRYPT\_MODE*, desKey);  
 CipherInputStream cis = new CipherInputStream(is, cipher);  
 *doCopy*(cis, os);  
 } else if (mode == Cipher.*DECRYPT\_MODE*) {  
 cipher.init(Cipher.*DECRYPT\_MODE*, desKey);  
 CipherOutputStream cos = new CipherOutputStream(os, cipher);  
 *doCopy*(is, cos);  
 }  
 }  
  
 public static void doCopy(InputStream is, OutputStream os) throws IOException {  
 byte[] bytes = new byte[64];  
 int numBytes;  
 while ((numBytes = is.read(bytes)) != -1) {  
 os.write(bytes, 0, numBytes);  
 }  
 os.flush();  
 os.close();  
 is.close();  
 }  
  
}

**Приложение 3**

package com.alfadeprojects;  
  
import java.math.BigInteger;  
import java.text.Bidi;  
import java.util.Random;  
  
/\*\*  
 \* Created by macbookpro on 19.11.14.  
 \*/  
public class Functions {  
 static Random *rnd* = new Random();  
  
 public static int p\_LengthGen(){  
 int n;  
 do {  
 n = *rnd*.nextInt(1000);  
 } while(n < 50);  
 return n;  
 }  
  
 // Генерируем простое 50-тибитовое большое число  
 public static BigInteger primeNumberGen(){  
 BigInteger p;  
 do{  
 p = new BigInteger(*p\_LengthGen*(), 1, *rnd*);  
 }while(p.bitLength() < 50);   
 return p;  
 }  
 public static int naturalNumberGen(){  
 Random rnd = new Random();  
 int x = rnd.nextInt(50);  
 System.*out*.println("x = " + x);  
 return x;  
 }  
 public static BigInteger powMod(BigInteger q, BigInteger a, BigInteger p){  
 BigInteger pm = q.modPow(a, p);  
 return pm;  
 }  
 // Тест Рабина-Миллера используется в методе isProbablePrime языка Java  
 public static void testRM(BigInteger p) {  
 int certainty = 948272723;  
 System.*out*.println(p.toString() + " is " + (p.isProbablePrime(certainty) ? "probably prime" : "composite"));  
 }  
}

**Приложение 4**

import java.math.BigInteger;  
  
public class Dialog {  
  
 public static boolean dialog(){  
 Alice.*set\_p*(Functions.*primeNumberGen*());  
 System.*out*.println("Alice P = "+Alice.*get\_p*() + "\nP.bitlength() = " + Alice.*get\_p*().bitLength());  
 Functions.*testRM*(Alice.*get\_p*());

do{  
 Alice.*set\_q*(Functions.*primeNumberGen*());  
 } while(Alice.*get\_q*().bitLength() >= Alice.*get\_p*().subtract(BigInteger.*ONE*).bitLength());  
 System.*out*.println("Alice Q = " + Alice.*get\_q*() + "\n Q.bitlength() = " + Alice.*get\_q*().bitLength());  
 do{  
 Alice.*set\_a*(Functions.*primeNumberGen*());  
 } while(Alice.*get\_a*().bitLength() >= Alice.*get\_p*().bitLength());  
 System.*out*.println("Alice A = " + Alice.*get\_a*() + "\n A.bitlength() = " + Alice.*get\_a*().bitLength());  
  
 Alice.*setAlice*(Functions.*powMod*(Alice.*get\_q*(), Alice.*get\_a*(), Alice.*get\_p*())); // A = (g^a) mod p  
 System.*out*.println("Alice ALICE = " + Alice.*getAlice*() + "\nALICE.bitlength() = " + Alice.*getAlice*().bitLength());  
 Bob.*get\_p\_fromAlice*();  
 System.*out*.println("(gfa)Bob P = " + Bob.*get\_p*());  
 Bob.*get\_q\_fromAlice*();  
 System.*out*.println("(gfa)Bob Q = " + Bob.*get\_q*());  
 Bob.*get\_A\_fromAlice*();  
 System.*out*.println("(gfa)Bob ALICE = " + Bob.*get\_A\_fromAlice*());  
  
 do{  
 Bob.*set\_b*(Functions.*primeNumberGen*());  
 } while(Bob.*get\_b*().bitLength() >= Bob.*get\_p*().subtract(BigInteger.*ONE*).bitLength());  
 System.*out*.println("Bob B = " + Bob.*get\_b*() + "\nB.bitlength() = "+Bob.*get\_b*().bitLength());  
  
 Bob.*setBob*(Functions.*powMod*(Bob.*get\_q*(), Bob.*get\_b*(), Bob.*get\_p*())); // B = (g^b) mod p  
 System.*out*.println("Bob BOB = " + Bob.*getBob*() + "\nBOB.bitlength() = "+Bob.*getBob*().bitLength());  
  
 Alice.*setSecret*(Functions.*powMod*(Alice.*get\_B\_fromBob*(), Alice.*get\_a*(), Alice.*get\_p*()));  
 System.*out*.println("Alice SECRETKEY = " + Alice.*getSecret*() + "\nSECRETKEY.bitlength() = " + Alice.*getSecret*().bitLength());  
 Bob.*setSecret*(Functions.*powMod*(Bob.*get\_A\_fromAlice*(), Bob.*get\_b*(), Bob.*get\_p*()));  
 System.*out*.println("Bob SECRETKEY = " + Bob.*getSecret*() + "\nSECRETKEY.bitlength() = " + Bob.*getSecret*().bitLength());  
 return *isEqual*(Alice.*getSecret*(), Bob.*getSecret*());  
 }  
  
 public static boolean isEqual(BigInteger alice, BigInteger bob){  
 if (Alice.*getSecret*().toString().equals(Bob.*getSecret*().toString())){  
 System.*out*.println("Алиса и Боб имеют одинаковые ключи.");  
 return true;  
 } else{  
 System.*out*.println("Алиса и Боб имеют разные ключи.");  
 return false;  
 }  
 }  
}

**Приложение 5**

import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.math.BigInteger;  
import java.security.SecureRandom;  
import java.util.Random;  
import com.alfadeprojects.Functions;  
  
public class Main {  
 private static String *secretKey*;  
 public static void main(String[] args) {  
 if (Dialog.*dialog*() == true){  
 *secretKey* = Alice.*getSecret*().toString();  
 }  
 System.*out*.println("Секретный ключ в виде строки : " + *secretKey*);  
 try{  
 FileInputStream fis = new FileInputStream("Data/original.txt");  
 FileOutputStream fos = new FileOutputStream("Data/encrypted.txt");  
 DesEncrypter.*encrypt*(*secretKey*, fis, fos);  
  
 FileInputStream fis2 = new FileInputStream("Data/encrypted.txt");  
 FileOutputStream fos2 = new FileOutputStream("Data/decrypted.txt");  
 DesEncrypter.*decrypt*(*secretKey*, fis2, fos2);  
 } catch(Throwable e){  
 e.printStackTrace();  
 }  
 }  
}