Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа №2

**Информационные сети. Основы безопасности. Информационные сети. Основы безопасности.**

Тема:   
**«Идентификация и аутентификация пользователей. Протокол Kerberos»**.

Выполнил студент гр. 753504

Осипик И.Ф.

Проверил

Протько М.И.

Минск, 2020

# 1. Введение

Целью данной лабораторной было реализовать программные средства протокола распределения ключей Kerberos вместе с процедурой, которая реализует Алгоритм DES.

**ЗАДАНИЕ:**

1. Изучить теоретические сведения.
2. Создать приложение, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

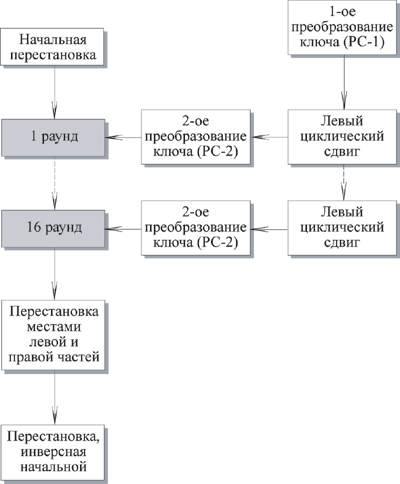
В интерфейсе приложения должны быть наглядно представлены:

* Исходные данные протокола (модули, ключи, секретные данные и т.п.);
* Данные, передаваемые по сети каждой из сторон;
* Проверки, выполняемые каждым из участников.

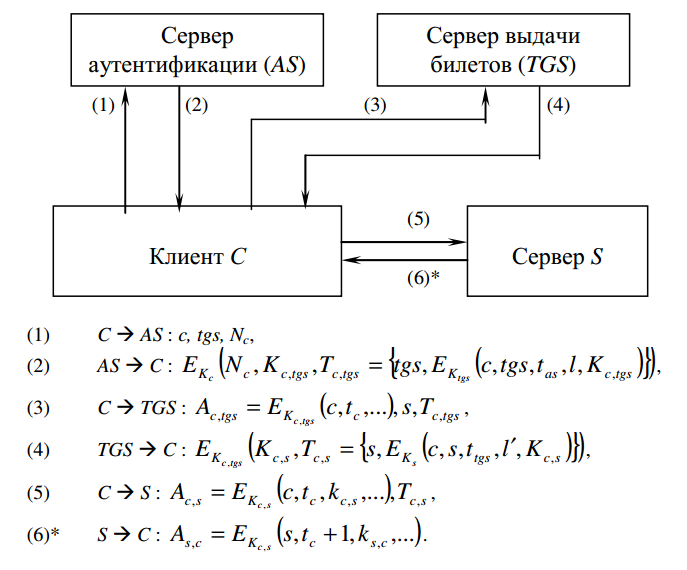
Процесс взаимодействия между сторонами протокола может быть реализован при помощи буферных переменных. Также необходимо выделить каждый из этапов протоколов для того, чтобы его можно было отделить от остальных.

# 2**. Блок-схемы алгоритмов**

**Блок-схема алгоритма DES:**



**Блок-схема протокола Kerberos:**



# 3. Теоретические сведения

**Протокол Kerberos**

Протокол Kerberos является одной из реализаций протокола аутентификации с использованием третьей стороны, призванной уменьшить количество сообщений, которыми обмениваются стороны.

Протокол Kerberos, достаточно гибкий и имеющий возможности тонкой настройки под конкретные применения, существует в нескольких версиях. Мы рассмотрим упрощенный механизм аутентификации, реализованный с помощью протокола Kerberos версии 5 (рис. 1):



Рисунок 1. Схема протокола Kerberos

Прежде всего стоит сказать, что при использовании Kerberos нельзя напрямую получить доступ к какому-либо целевому серверу. Чтобы запустить собственно процедуру аутентификации, необходимо обратиться к специальному серверу аутентификации с запросом, содержащим логин пользователя. Если сервер не находит автора запроса в своей базе данных, запрос отклоняется. В противном случае сервер аутентификации работает по следующему рабочему процессу:

**Рабочий этап:**

Пусть клиент C собирается начать взаимодействие с сервером SS (англ. *Service* *Server* - *сервер*, предоставляющий сетевые сервисы). В несколько упрощенном виде, протокол предполагает следующие шаги:

1. **C -> AS: {c}.**

Клиент C посылает серверу аутентификации AS свой идентификатор c (идентификатор передается открытым текстом).

1. **AS -> C: {{TGT}KAS\_TGS, KC\_TGS}KC,**

где:

* KC - основной ключ C;
* KC\_TGS - ключ, выдаваемый C для доступа к серверу выдачи разрешений *TGS*;
* {TGT} - *Ticket* Granting *Ticket* - билет на доступ к серверу выдачи разрешений

{TGT} = {c, *tgs*, t1, p1, KC\_TGS}, где *tgs* - идентификатор сервера выдачи разрешений, t1 - отметка времени, p1 - *период действия* билета.

Запись \{ \cdot \} K_{X}здесь и далее означает, что содержимое фигурных скобок зашифровано на ключе KX (Алгоритм шифрования приводится ниже).

На этом шаге сервер аутентификации AS, проверив, что клиент C имеется в его базе, возвращает ему билет для доступа к серверу выдачи разрешений и ключ для взаимодействия с сервером выдачи разрешений. Вся посылка зашифрована на ключе клиента C. Таким образом, даже если на первом шаге взаимодействия идентификатор с послал не клиент С, а нарушитель X, то полученную от AS посылку X расшифровать не сможет.

Получить доступ к содержимому билета TGT не может не только нарушитель, но и клиент C, т.к. билет зашифрован на ключе, который распределили между собой сервер аутентификации и сервер выдачи разрешений.

1. **C ->** ***TGS*: {TGT}KAS\_TGS, {Aut1} KC\_TGS, {ID}**

где {Aut1} - аутентификационный блок - Aut1 = {с, t2}, t2 - метка времени; ID - идентификатор запрашиваемого сервиса (в частности, это может быть идентификатор сервера SS).

Клиент C на этот раз обращается к серверу выдачи разрешений ТGS. Он пересылает полученный от AS билет, зашифрованный на ключе KAS\_TGS, и аутентификационный блок, содержащий идентификатор c и метку времени, показывающую, когда была сформирована посылка. Сервер выдачи разрешений расшифровывает билет TGT и получает из него информацию о том, кому был выдан билет, когда и на какой срок, ключ шифрования, сгенерированный сервером AS для взаимодействия между клиентом C и сервером *TGS*. С помощью этого ключа расшифровывается аутентификационный блок. Если метка в блоке совпадает с меткой в билете, это доказывает, что посылку сгенерировал на самом деле С (ведь только он знал ключ KC\_TGS и мог правильно зашифровать свой идентификатор). Далее делается проверка времени действия билета и времени отправления посылки **3**). Если проверка проходит и действующая в системе политика позволяет клиенту С обращаться к клиенту SS, тогда выполняется шаг **4**).

1. ***TGS* -> C: {{*****TGS*}KTGS\_SS, KC\_SS} KC\_TGS,**

где KC\_SS - ключ для взаимодействия C и SS, {*TGS*} - *Ticket* Granting Service - билет для доступа к SS (обратите внимание, что такой же аббревиатурой в описании протокола обозначается и сервер выдачи разрешений).  
{*TGS*} = {с, ss, t3, p2, KC\_SS}.

Сейчас сервер выдачи разрешений *TGS* посылает клиенту C ключ шифрования и билет, необходимые для доступа к серверу SS. Структура билета такая же, как на шаге 2): идентификатор того, кому выдали билет; идентификатор того, для кого выдали билет; отметка времени; *период действия*; ключ шифрования.

1. **C -> SS: {*****TGS*}KTGS\_SS, {Aut2} KC\_SS**

где Aut2 = {c, t4}.

Клиент C посылает билет, полученный от сервера выдачи разрешений, и свой аутентификационный блок серверу SS, с которым хочет установить сеанс защищенного взаимодействия. Предполагается, что SS уже зарегистрировался в системе и распределил с сервером *TGS* ключ шифрования KTGS\_SS. Имея этот ключ, он может расшифровать билет, получить ключ шифрования KC\_SS и проверить подлинность *отправителя сообщения*.

1. **SS -> C: {t4+1} KC\_SS**

Смысл последнего шага заключается в том, что теперь уже SS должен доказать C свою подлинность. Он может сделать это, показав, что правильно расшифровал предыдущее сообщение. Вот поэтому, SS берет отметку времени из аутентификационного блока C, изменяет ее заранее определенным образом (увеличивает на 1), шифрует на ключе KC\_SS и возвращает C.

Если все шаги выполнены правильно и все проверки прошли успешно, то стороны взаимодействия C и SS, во-первых, удостоверились в подлинности друг друга, а во-вторых, получили *ключ* шифрования для защиты сеанса связи - *ключ* KC\_SS.

Нужно отметить, что в процессе сеанса работы клиент проходит шаги 1) и 2) только один раз. Когда нужно получить билет на *доступ* к другому серверу (назовем его SS1), клиент С обращается к серверу выдачи разрешений *TGS* с уже имеющимся у него билетом, т.е. протокол выполняется начиная с шага 3).

В алгоритме Kerberos могут применяться различные алгоритмы блочного симметричного шифрования. Для целей настоящей работы будем использовать алгоритм DES:

**Алгоритм DES Основные сведения**

Одной из наиболее известных криптографических систем с закрытым ключом является DES – Data Encryption Standard. Эта система первой получила статус государственного стандарта в области шифрования данных. Она разработана специалистами фирмы IBM и вступила в действие в США 1977 году. Алгоритм DES по-прежнему широко применяется и заслуживает внимания при изучении блочных шифров с закрытым ключом.

Стандарт DES построен на комбинированном использовании перестановки, замены и гаммирования. Шифруемые данные должны быть представлены в двоичном виде.

*DES* является классической *сетью Фейстеля* с двумя ветвями. Данные шифруются 64-битными блоками, используя 56-битный ключ. Алгоритм преобразует за несколько *раундов* 64-битный вход в 64-битный выход. Длина ключа равна 56 битам. Процесс шифрования состоит из четырех этапов. На первом из них выполняется начальная перестановка (*IP*) 64-битного исходного текста (забеливание), во время которой биты переупорядочиваются в соответствии со стандартной таблицей. Следующий этап состоит из 16 *раундов* одной и той же функции, которая использует операции сдвига и подстановки. На третьем этапе левая и правая половины выхода последней (16-й) итерации меняются местами. Наконец, на четвертом этапе выполняется перестановка IP-1 результата, полученного на третьем этапе. Перестановка IP-1 инверсна начальной перестановке.

**Шифрование**

**Начальная перестановка**

Начальная перестановка и её инверсия определяются стандартной таблицей. Если М – это произвольные 64 бита, то X = IP (M)-переставленные 64 бита. Если применить обратную функцию перестановки Y = IP-1 (X) = IP-1 (IP(M)), то получится первоначальная последовательность бит.

**Последовательность преобразований отдельного раунда**

Теперь рассмотрим последовательность преобразований, используемую в каждом *раунде*.

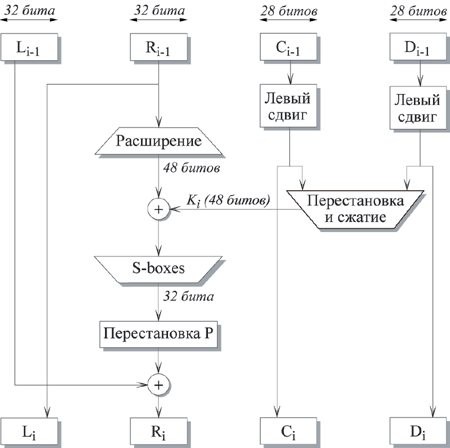


Рисунок 3. I-ый раунд DES

64-битный входной блок проходит через 16 *раундов*, при этом на каждой итерации получается промежуточное 64-битное значение. Левая и правая части каждого промежуточного значения трактуются как отдельные 32-битные значения, обозначенные *L* и *R*. Каждую итерацию можно описать следующим образом:

Li = Ri-1

Ri = Li-1  F(Ri-1, Ki)

Где обозначает операцию XOR.

Таким образом, выход левой половины *Li* равен входу правой половины *Ri-1*. Выход правой половины *Ri*является результатом применения операции XOR к *Li-1* и функции *F*, зависящей от *Ri-1* и *Ki*.

Рассмотрим функцию *F* более подробно.

*Ri*, которое подается на вход функции *F*, имеет длину 32 бита. Вначале Ri расширяется до 48 бит, используя таблицу, которая определяет перестановку плюс расширение на 16 бит. Расширение происходит следующим образом. 32 бита разбиваются на группы по 4 бита и затем расширяются до 6 бит, присоединяя крайние биты из двух соседних групп. Например, если часть входного сообщения

. . . efgh ijkl mnop . . .

то в результате расширения получается сообщение

. . . defghi hijklm lmnopq . . .

После этого для полученного 48-битного значения выполняется операция XOR с 48-битным *подключом Ki*. Затем полученное 48-битное значение подается на вход функции подстановки, результатом которой является 32-битное значение.

Подстановка состоит из восьми *S-boxes,* каждый из которых на входе получает 6 бит, а на выходе создает 4 бита. Эти преобразования определяются специальными таблицами. Первый и последний биты входного значения *S-box* определяют номер строки в таблице, средние 4 бита определяют номер столбца. Пересечение строки и столбца определяет 4-битный выход. Например, если входом является 011011, то номер строки равен 01 (строка 1) и номер столбца равен 1101 (столбец 13). Значение в строке 1 и столбце 13 равно 5, т.е. выходом является 0101.

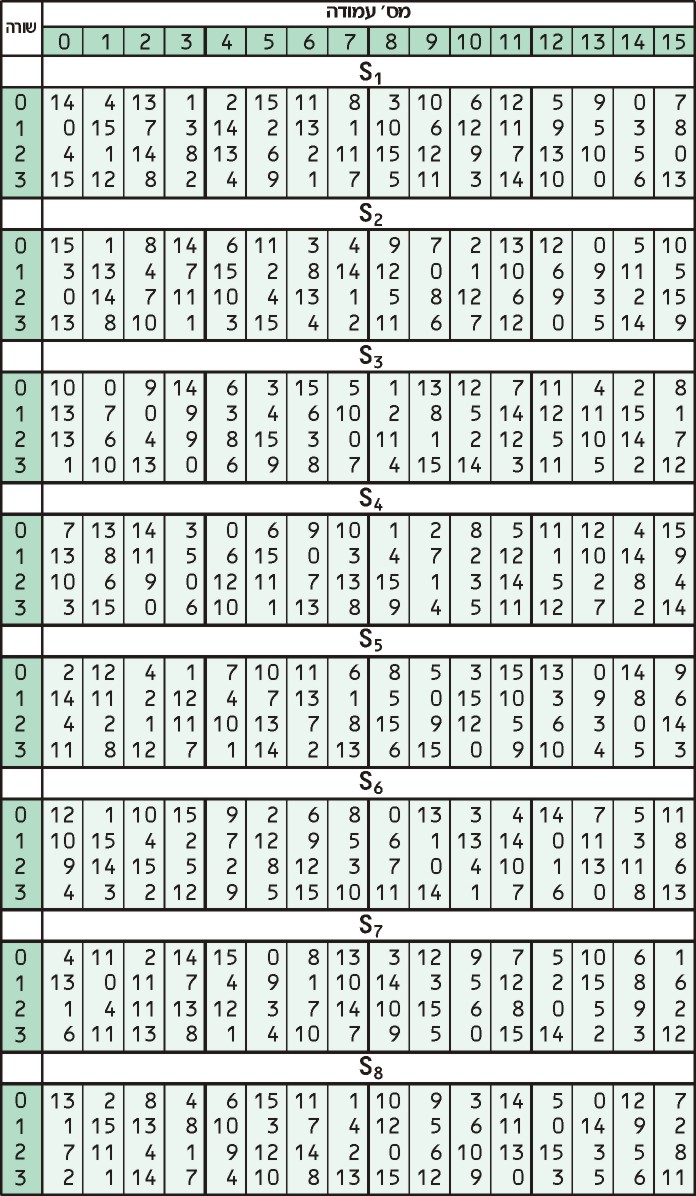


Рисунок 4. S-boxes

Далее полученное 32-битное значение обрабатывается с помощью перестановки *Р*, целью которой является максимальное переупорядочивание бит, чтобы в следующем *раунде* шифрования с большой вероятностью каждый бит обрабатывался другим *S-box*.

**Создание подключей**

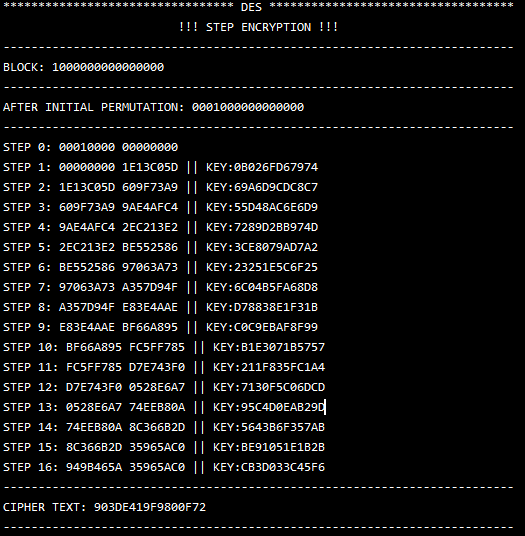
Ключ для отдельного *раунда Ki*состоит из 48 бит. Ключи *Ki* получаются по следующему алгоритму. Для 56-битного ключа, используемого на входе алгоритма, вначале выполняется перестановка в соответствии с таблицей Permuted Choice 1 (РС-1). Полученный 56-битный ключ разделяется на две 28-битные части, обозначаемые как C0 и D0 соответственно. На каждом *раунде Ci* и *Di*независимо циклически сдвигаются влево на 1 или 2 бита, в зависимости от номера *раунда*. Полученные значения являются входом следующего *раунда*. Они также представляют собой вход в Permuted Choice 2 (РС-2), который создает 48-битное выходное значение, являющееся входом функции *F*(*Ri-1*, *Ki*).

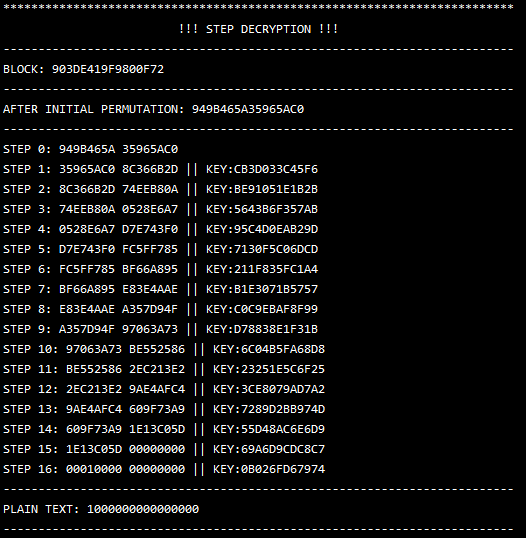
**Дешифрование**

Процесс дешифрования аналогичен процессу шифрования. На входе алгоритма используется зашифрованный текст, но ключи *Ki* используются в обратной последовательности. *K16* используется на первом *раунде*, *K1* используется на последнем *раунде*.

# 4. Результаты выполнения программы

**1) DES**





**2) Kerberos**

# 

# 5. Выводы

DES был национальным стандартом США в 1977—1980 гг., но в настоящее время DES используется (с ключом длины 56 бит) только для устаревших систем, чаще всего используют его более криптоустойчивый вид (3DES, DESX). 3DES является простой эффективной заменой DES, и сейчас он рассмотрен как стандарт. В ближайшее время DES и Triple DES будут заменены алгоритмом AES (Advanced Encryption Standard — Расширенный Стандарт Шифрования).

Kerberos является одним из самых распространенных протоколов аунтефикации. В настоящее время множество ОС поддерживают данный протокол, в число которых входят: Windows 2000 и более поздние версии, которые используют Kerberos как метод аутентификации в домене между участниками, различные UNIX и UNIX подобные ОС (Apple Mac OS X, Red Hat Enterprise Linux 4, FreeBSD, Solaris, AIX, OpenVMS).

# 6. КОД ПРОГРАММЫ

**Класс Buffer:**

package com.september**;**public class Buffer {  
 public static String *message***;**}

**Класс Client:**

package com.september**;**import java.math.BigInteger**;**import java.sql.Timestamp**;**public class Client {  
 String id = "Client\_C"**;** String key = "1234567890ABCDEF"**;** String ss\_id = "Server1"**;** void startProtocol(){  
 System.*out*.println("Client(C) sends his ID to Authentication Server(AS): " + id)**;** Buffer.*message* = id**;** if (!KerberosServer.AS.*giveTGT*()){  
 return**;** }  
  
 DESDerivedKey c\_key = new DESDerivedKey(key)**;** String TGT\_encrypted = Buffer.*message***;** String TGT = DESCrypto.*decrypt*(TGT\_encrypted**,** c\_key)**;** TGT = Utils.*hex2text*(TGT)**;** System.*out*.println("Client decrypts crypted TGT: " + TGT)**;** String c\_tgs = TGT.split(" ")[**1**]**;** c\_tgs = c\_tgs.substring(**0,16**)**;** TGT = TGT.split(" ")[**0**]**;** System.*out*.println("Client splits key (" + c\_tgs + ") and encrypted TGT: " + TGT)**;** Timestamp timestamp = new Timestamp(System.*currentTimeMillis*())**;** String Aut = id + " " + timestamp.getTime()**;** BigInteger t = new BigInteger(Aut.split(" ")[**1**])**;** System.*out*.println("Client creates Aut: " + Aut)**;** DESDerivedKey k\_c\_tgs = new DESDerivedKey(c\_tgs)**;** Aut = DESCrypto.*encrypt*(Utils.*text2hex*(Aut)**,** k\_c\_tgs)**;** System.*out*.println("and encrypt it:" + Aut)**;** Buffer.*message* = TGT + " " + Aut + " " + ss\_id**;** KerberosServer.TGS.*giveTGT*()**;** String message = Buffer.*message***;** message = Utils.*hex2text*(DESCrypto.*decrypt*(message**,** k\_c\_tgs))**;** System.*out*.println("Client encrypts message: " + message)**;** String c\_ss = message.split(" ")[**1**]**;** DESDerivedKey k\_c\_ss = new DESDerivedKey(c\_ss)**;** String TGS = message.split(" ")[**0**]**;** String Aut2 = id + " " + timestamp.getTime()**;** System.*out*.println("Client Aut2: " + Aut2)**;** Aut = DESCrypto.*encrypt*(Utils.*text2hex*(Aut2)**,** k\_c\_ss)**;** message = TGS + " " + Aut**;** System.*out*.println("Message for SS: " + message)**;** Buffer.*message* = message**;** Server.*checkClient*()**;** String time = Buffer.*message***;** time = Utils.*hex2text*(DESCrypto.*decrypt*(time**,**k\_c\_ss))**;** time = time.substring(**0,13**)**;** System.*out*.println("-------------------------------------------------------------------------")**;** System.*out*.println("Client decrypt t4 + 1:" + time + "(Client's t4: " + t.toString() + ")")**;** t = t.add(new BigInteger("1"))**;** BigInteger t2 = new BigInteger(time)**;** if (t.compareTo(t2) != **0**){  
 System.*out*.println("-------------------------------------------------------------------------")**;** System.*out*.println("Client t4 + 1 doesn't match server's. Connection failed.")**;** return**;** }  
 System.*out*.println("Client's key: " + c\_ss)**;** System.*out*.println("-------------------------------------------------------------------------")**;** System.*out*.println("Success!")**;** }  
  
}

**Класс DESCrypto:**

package com.september**;**import static com.september.Utils.\***;**class DESCrypto {  
 static boolean *DEBUG* = true**;** private static String xor\_(String a**,** String b) {  
 StringBuilder ans = new StringBuilder()**;** for (int i = **0;** i < a.length()**;** i++) {  
 if (a.charAt(i) == b.charAt(i)) {  
 ans.append("0")**;** }  
 else {  
 ans.append("1")**;** }  
 }  
 return ans.toString()**;** }  
  
 //takes hex message and derived key  
 static String encrypt(String text**,** DESDerivedKey key) {  
 StringBuilder message = new StringBuilder(*hex2bin*(text))**;** if (message.length() % **64** != **0**) {  
 int add = **64** - (message.length() % **64**)**;** message.append("0".repeat(Math.*max*(**0,** add)))**;** }  
  
 StringBuilder result = new StringBuilder()**;** for (int q = **0;** q < message.length() / **64;** q++) {  
 String pt = message.substring(q \* **64,** q \* **64** + **64**)**;** // Initial Permutation Table  
 int[] initial\_perm = {**58, 50, 42, 34, 26, 18, 10, 2,  
 60, 52, 44, 36, 28, 20, 12, 4,  
 62, 54, 46, 38, 30, 22, 14, 6,  
 64, 56, 48, 40, 32, 24, 16, 8,  
 57, 49, 41, 33, 25, 17, 9, 1,  
 59, 51, 43, 35, 27, 19, 11, 3,  
 61, 53, 45, 37, 29, 21, 13, 5,  
 63, 55, 47, 39, 31, 23, 15, 7**}**;** // Initial Permutation  
 //String left = pt.substring(0, 32);  
 //String right = pt.substring(32);  
 if (*DEBUG*) {  
 System.*out*.println()**;** System.*out*.println("BLOCK: " + *bin2hex*(pt))**;** System.*out*.println("-------------------------------------------------------------------------")**;** }  
 pt = *permute*(pt**,** initial\_perm**, 64**)**;** if (*DEBUG*){  
 System.*out*.println("AFTER INITIAL PERMUTATION: " + *bin2hex*(pt))**;** System.*out*.println("-------------------------------------------------------------------------")**;** }  
 // Splitting  
 String left = pt.substring(**0, 32**)**;** String right = pt.substring(**32**)**;** if (*DEBUG*) {  
 System.*out*.println("STEP 0: " + *bin2hex*(left) + " " + *bin2hex*(right))**;** }  
 // Expansion D-box Table  
 int[] exp\_d = {**32, 1, 2, 3, 4, 5, 4, 5,  
 6, 7, 8, 9, 8, 9, 10, 11,  
 12, 13, 12, 13, 14, 15, 16, 17,  
 16, 17, 18, 19, 20, 21, 20, 21,  
 22, 23, 24, 25, 24, 25, 26, 27,  
 28, 29, 28, 29, 30, 31, 32, 1**}**;** // S-box Table  
 int[][][] s = {{{**14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7**}**,** {**0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8**}**,** {**4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0**}**,** {**15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13**}}**,** {{**15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10**}**,** {**3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5**}**,** {**0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15**}**,** {**13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9**}}**,** {{**10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8**}**,** {**13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1**}**,** {**13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7**}**,** {**1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12**}}**,** {{**7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15**}**,** {**13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9**}**,** {**10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4**}**,** {**3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14**}}**,** {{**2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9**}**,** {**14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6**}**,** {**4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14**}**,** {**11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3**}}**,** {{**12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11**}**,** {**10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8**}**,** {**9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6**}**,** {**4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13**}}**,** {{**4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1**}**,** {**13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6**}**,** {**1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2**}**,** {**6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12**}}**,** {{**13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7**}**,** {**1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2**}**,** {**7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8**}**,** {**2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11**}}}**;** // Straight Permutation Table  
 int[] per = {**16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10, 2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25**}**;** //System.out.println();  
 for (int i = **0;** i < **16;** i++) {  
 // Expansion D-box  
 String right\_expanded = *permute*(right**,** exp\_d**, 48**)**;** // XOR RoundKey[i] and right\_expanded  
 String x = *xor\_*(key.rkb.get(i)**,** right\_expanded)**;** // S-boxes  
 StringBuilder op = new StringBuilder()**;** for (int j = **0;** j < **8;** j++) {  
 int row = **2** \* (x.charAt(j \* **6**) - '0') + (x.charAt(j \* **6** + **5**) - '0')**;** int col = **8** \* (x.charAt(j \* **6** + **1**) - '0') + **4** \* (x.charAt(j \* **6** + **2**) - '0') + **2** \* (x.charAt(j \* **6** + **3**) - '0') + (x.charAt(j \* **6** + **4**) - '0')**;** int val = s[j][row][col]**;** op.append(Character.*forDigit*(val / **8, 10**))**;** val = val % **8;** op.append(Character.*forDigit*(val / **4, 10**))**;** val = val % **4;** op.append(Character.*forDigit*(val / **2, 10**))**;** val = val % **2;** op.append(Character.*forDigit*(val**, 10**))**;** }  
 // Straight D-box  
 String op2 = *permute*(op.toString()**,** per**, 32**)**;** // XOR left and op  
 x = *xor\_*(op2**,** left)**;** left = x**;** // Swapper  
 if (i != **15**) {  
 String mid = left**;** left = right**;** right = mid**;** }  
 if (*DEBUG*)  
 System.*out*.println("STEP " + (i + **1**) + ": " + *bin2hex*(left) + " " + *bin2hex*(right) + " || KEY:" + *bin2hex*(key.rkb.get(i)))**;** }  
  
 String combine = left + right**;** // Final Permutation Table  
 int[] final\_perm = {**40, 8, 48, 16, 56, 24, 64, 32,  
 39, 7, 47, 15, 55, 23, 63, 31,  
 38, 6, 46, 14, 54, 22, 62, 30,  
 37, 5, 45, 13, 53, 21, 61, 29,  
 36, 4, 44, 12, 52, 20, 60, 28,  
 35, 3, 43, 11, 51, 19, 59, 27,  
 34, 2, 42, 10, 50, 18, 58, 26,  
 33, 1, 41, 9, 49, 17, 57, 25**}**;** // Final Permutation  
 result.append(*bin2hex*(*permute*(combine**,** final\_perm**, 64**)))**;** }  
 return result.toString()**;** }  
  
 static String decrypt(String message**,** DESDerivedKey key){  
 DESDerivedKey reversal = new DESDerivedKey(key)**;** reversal.rev()**;** return *encrypt*(message**,** reversal)**;** }  
}

**Класс DESDerivedKey:**

package com.september**;**import java.util.ArrayList**;**import static com.september.Utils.\***;**import static java.util.Collections.*reverse***;**class DESDerivedKey {  
 ArrayList<String> rkb**;** DESDerivedKey(DESDerivedKey toCopy){  
 this.rkb = new ArrayList<>(toCopy.rkb)**;** }  
  
 DESDerivedKey(String key){  
 key = *hex2bin*(key)**;** int[] keyp = { **57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36, 63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4** }**;** key = *permute*(key**,** keyp**, 56**)**;** // key without parity  
 int[] shift\_table = { **1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1** }**;** int[] key\_comp = { **14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10, 23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13, 2, 41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32** }**;** String left = key.substring(**0, 28**)**;** String right = key.substring(**28**)**;** ArrayList<String> rkb = new ArrayList<>()**;** // rkb for RoundKeys in binary  
 for (int i = **0;** i < **16;** i++) {  
 left = shift\_left(left**,** shift\_table[i])**;** right = shift\_left(right**,** shift\_table[i])**;** String combine = left.concat(right)**;** // Key Compression  
 String RoundKey = *permute*(combine**,** key\_comp**, 48**)**;** rkb.add(RoundKey)**;** }  
 this.rkb = rkb**;** }  
  
 private String shift\_left(String kS**,** int shifts) {  
 StringBuilder k = new StringBuilder(kS)**;** StringBuilder s = new StringBuilder()**;** for (int i = **0;** i < shifts**;** i++) {  
 for (int j = **1;** j < **28;** j++) {  
 s.append(k.charAt(j))**;** }  
 s.append(k.charAt(**0**))**;** k = new StringBuilder(s)**;** s.delete(**0,**s.length())**;** }  
 return k.toString()**;** }  
  
 void rev(){  
 *reverse*(this.rkb)**;** }  
}

**Класс KerberosServer:**

package com.september**;**import java.math.BigInteger**;**import java.nio.BufferUnderflowException**;**import java.sql.Timestamp**;**import java.util.Arrays**;**import static com.september.DESCrypto.*encrypt***;**public class KerberosServer {  
 private static String[] *keys\_as\_tgs* = {"FEDCBA0987654321"}**;** private static String[] *keys\_c\_tgs* = {"1122334455667788"}**;** public static class AS{  
 static String[] *clients* = {"Client\_C"}**;** static String[] *keys\_c* = {"1234567890ABCDEF"}**;** //static String[] keys\_as\_tgs = {"FEDCBA0987654321"};  
 //static String[] keys\_c\_tgs = {"1122334455667788"};  
  
 static boolean giveTGT(){  
 String id = Buffer.*message***;** System.*out*.println("Authentication Server(AS) checked that this client exist in DB.")**;** if (!Arrays.*asList*(*clients*).contains(id)){  
 System.*out*.println("ERROR. CLIENT DOES NOT EXIST!")**;** return false**;** }  
 System.*out*.println("OK.")**;** System.*out*.println("-------------------------------------------------------------------------")**;** Timestamp timestamp = new Timestamp(System.*currentTimeMillis*())**;** String TGT = id + " " + TGS.*id* + " " + timestamp.getTime() + " " + "10000000"**;** System.*out*.println("Authentication Server(AS) creates TGT: " + TGT)**;** DESDerivedKey as\_tgs = new DESDerivedKey(*keys\_as\_tgs*[**0**])**;** TGT = Utils.*text2hex*(TGT)**;** String TGT\_encrypted = DESCrypto.*encrypt*(TGT**,** as\_tgs)**;** System.*out*.println("Encrypted TGT: " + TGT\_encrypted)**;** DESDerivedKey c = new DESDerivedKey(*keys\_c*[**0**])**;** String messageForC = TGT\_encrypted + " " + *keys\_c\_tgs*[**0**]**;** System.*out*.println("TGT + key for C: " + messageForC)**;** messageForC = DESCrypto.*encrypt*(Utils.*text2hex*(messageForC)**,** c)**;** System.*out*.println("Encrypted TGT + key for C: " + messageForC)**;** Buffer.*message* = messageForC**;** return true**;** }  
 }  
  
 public static class TGS{  
 static String *id* = "tgs1"**;** private static String[] *keys\_tgs\_ss* = {"1234567890ABCDEF"}**;** static boolean giveTGT(){  
 System.*out*.println(Buffer.*message*)**;** String ID = Buffer.*message*.split(" ")[**2**]**;** String Aut = Buffer.*message*.split(" ")[**1**]**;** String TGT = Buffer.*message*.split(" ")[**0**]**;** DESDerivedKey k\_as\_tgs = new DESDerivedKey(*keys\_as\_tgs*[**0**])**;** TGT = Utils.*hex2text*(DESCrypto.*decrypt*(TGT**,** k\_as\_tgs))**;** System.*out*.println("Encrypted TGT: " + TGT)**;** DESDerivedKey k\_c\_tgs = new DESDerivedKey(*keys\_c\_tgs*[**0**])**;** Aut = Utils.*hex2text*(DESCrypto.*decrypt*(Aut**,** k\_c\_tgs))**;** System.*out*.println("Encrypted Aut: " + Aut)**;** if (!Aut.split(" ")[**0**].equals(TGT.split(" ")[**0**])){  
 System.*out*.println("Authentication error.")**;** return false**;** }  
  
 if (!TGT.split(" ")[**1**].equals(*id*)){  
 System.*out*.println("Wrong TGS server.")**;** return false**;** }  
  
 BigInteger n1 = new BigInteger(Aut.split(" ")[**1**].substring(**0,13**))**;** BigInteger n2 = new BigInteger(TGT.split(" ")[**2**])**;** BigInteger n3 = new BigInteger(TGT.split(" ")[**3**].substring(**0,7**))**;** n1 = n1.subtract(n2)**;** if (n1.compareTo(n3) > **0**){  
 //if (Integer.parseInt(Aut.split(" ")[1].substring(0,13)) - Integer.parseInt(TGT.split("")[2]) > Integer.parseInt(TGT.split(" ")[3].substring(0,7)) ){  
 System.*out*.println("Time of ticket expired!")**;** return false**;** }  
  
 Timestamp timestamp = new Timestamp(System.*currentTimeMillis*())**;** String c\_ss = Utils.*bin2hex*(Utils.*randomKey*())**;** System.*out*.println("TGS server generate password C\_SS: " + c\_ss)**;** String TGS = TGT.split(" ")[**0**] + " " + ID + " " + timestamp.getTime() + " 1000000 " + c\_ss**;** System.*out*.println("TGS creates TGS ticket: " + TGS)**;** TGS = Utils.*text2hex*(TGS)**;** DESDerivedKey k\_tgs\_ss = new DESDerivedKey(*keys\_tgs\_ss*[**0**])**;** TGS = DESCrypto.*encrypt*(TGS**,** k\_tgs\_ss)**;** System.*out*.println("Encrypted TGS: " + TGS)**;** TGS = Utils.*text2hex*(TGS + " " + c\_ss)**;** TGS = DESCrypto.*encrypt*(TGS**,** k\_c\_tgs)**;** System.*out*.println("Encrypted TGS + K\_c\_ss: " + TGS)**;** Buffer.*message* = TGS**;** return true**;** }  
 }  
  
}

**Класс Main:**

package com.september**;**import javax.swing.\***;**import java.math.BigInteger**;**import java.util.ArrayList**;**import static java.util.Collections.*reverse***;**public class Main {  
  
 public static void main(String[] args) {  
 //---------------------------------------------------------------------------------------------  
 // DES  
 String pt = "1000000000000000"**;** String key = "1234567890ABCDEF"**;** DESCrypto.*DEBUG* = true**;** DESDerivedKey dKey = new DESDerivedKey(key)**;** System.*out*.println("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**;** System.*out*.println(" !!! STEP ENCRYPTION !!!")**;** System.*out*.print("-------------------------------------------------------------------------")**;** String cipher = DESCrypto.*encrypt*(pt**,** dKey)**;** System.*out*.println("-------------------------------------------------------------------------")**;** System.*out*.println("CIPHER TEXT: " + cipher )**;** System.*out*.println("-------------------------------------------------------------------------\n\n")**;** System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**;** System.*out*.println(" !!! STEP DECRYPTION !!!")**;** System.*out*.print("-------------------------------------------------------------------------")**;** dKey.rev()**;** String text = DESCrypto.*encrypt*(cipher**,** dKey)**;** System.*out*.print("-------------------------------------------------------------------------")**;** System.*out*.println("\nPLAIN TEXT: " + text)**;** System.*out*.println("-------------------------------------------------------------------------\n")**;** //---------------------------------------------------------------------------------------------  
 // Kerberos-Server  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* KERBEROS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**;** System.*out*.println("Starting...")**;** DESCrypto.*DEBUG* = false**;** System.*out*.println("-------------------------------------------------------------------------")**;** Client client\_C = new Client()**;** client\_C.startProtocol()**;** System.*out*.println("-------------------------------------------------------------------------")**;** }  
}

**Класс Server:**

package com.september**;**import java.math.BigInteger**;**public class Server {  
 private static String[] *keys\_tgs\_ss* = {"1234567890ABCDEF"}**;** static boolean checkClient(){  
 String message = Buffer.*message***;** DESDerivedKey k\_tgs\_ss = new DESDerivedKey(*keys\_tgs\_ss*[**0**])**;** String TGS = message.split(" ")[**0**]**;** TGS = DESCrypto.*decrypt*(TGS**,** k\_tgs\_ss)**;** TGS = Utils.*hex2text*(TGS)**;** System.*out*.println("Server got TGS: " + TGS)**;** String c\_ss = TGS.split(" ")[**4**].substring(**0,16**)**;** DESDerivedKey k\_c\_ss = new DESDerivedKey(c\_ss)**;** String Aut2 = message.split(" ")[**1**]**;** Aut2 = Utils.*hex2text*(DESCrypto.*decrypt*(Aut2**,** k\_c\_ss))**;** System.*out*.println("Server got Aut2: " + Aut2)**;** if (!Aut2.split(" ")[**0**].equals(TGS.split(" ")[**0**])){  
 System.*out*.println("Authentification failed. Server don't recognize client.")**;** return false**;** }  
  
 String t4 = Aut2.split(" ")[**1**].substring(**0,13**)**;** BigInteger num\_t4 = new BigInteger(t4)**;** num\_t4 = num\_t4.add(new BigInteger("1"))**;** String messageForC = num\_t4.toString()**;** System.*out*.println("t4 + 1: " + messageForC)**;** messageForC = DESCrypto.*encrypt*(Utils.*text2hex*(messageForC)**,**k\_c\_ss)**;** System.*out*.println("Server sends encrypted t4 + 1: " + messageForC)**;** System.*out*.println("Server's key: " + c\_ss)**;** Buffer.*message* = messageForC**;** return true**;** }  
  
  
}

**Класс Utils:**

package com.september**;**import java.util.HashMap**;**import java.util.Random**;**public class Utils {  
 static String randomKey(){  
 Random rand = new Random()**;** StringBuilder key = new StringBuilder()**;** for(int i = **0;** i < **64;** i++){  
 key.append((char)('0' + rand.nextInt(**2**)))**;** }  
 return key.toString()**;** }  
  
 static String permute(String k**,** int[] arr**,** int n) {  
 StringBuilder per = new StringBuilder()**;** for (int i = **0;** i < n**;** i++) {  
 per.append(k.charAt(arr[i] - **1**))**;** }  
 return per.toString()**;** }  
  
 static String hex2bin(String s) {  
 HashMap<Character**,** String> mp = new HashMap<>()**;** mp.put('0'**,** "0000")**;** mp.put('1'**,** "0001")**;** mp.put('2'**,** "0010")**;** mp.put('3'**,** "0011")**;** mp.put('4'**,** "0100")**;** mp.put('5'**,** "0101")**;** mp.put('6'**,** "0110")**;** mp.put('7'**,** "0111")**;** mp.put('8'**,** "1000")**;** mp.put('9'**,** "1001")**;** mp.put('A'**,** "1010")**;** mp.put('B'**,** "1011")**;** mp.put('C'**,** "1100")**;** mp.put('D'**,** "1101")**;** mp.put('E'**,** "1110")**;** mp.put('F'**,** "1111")**;** StringBuilder bin = new StringBuilder()**;** for (int i = **0;** i < s.length()**;** i++) {  
 bin.append(mp.get(s.charAt(i)))**;** }  
 return bin.toString()**;** }  
  
 static String text2bin(String s){  
 byte[] bytes = s.getBytes()**;** StringBuilder binary = new StringBuilder()**;** for(byte b : bytes){  
 int val = b**;** for(int i = **0;** i < **8;** i++){  
 binary.append((val & **128**) == **0** ? **0** : **1**)**;** val <<= **1;** }  
 }  
 return binary.toString()**;** }  
  
 static String bin2text(String s){  
 StringBuilder text = new StringBuilder()**;** for(int i = **0;** i <= s.length() - **8;** i += **8**){  
 text.append((char)Integer.*parseInt*(s.substring(i**,**i+**8**)**,2**))**;** }  
 return text.toString()**;** }  
  
 static String bin2hex(String s) {  
 HashMap<String**,** Character> mp = new HashMap<>()**;** mp.put("0000"**,** '0')**;** mp.put("0001"**,** '1')**;** mp.put("0010"**,** '2')**;** mp.put("0011"**,** '3')**;** mp.put("0100"**,** '4')**;** mp.put("0101"**,** '5')**;** mp.put("0110"**,** '6')**;** mp.put("0111"**,** '7')**;** mp.put("1000"**,** '8')**;** mp.put("1001"**,** '9')**;** mp.put("1010"**,** 'A')**;** mp.put("1011"**,** 'B')**;** mp.put("1100"**,** 'C')**;** mp.put("1101"**,** 'D')**;** mp.put("1110"**,** 'E')**;** mp.put("1111"**,** 'F')**;** StringBuilder hex = new StringBuilder()**;** for (int i = **0;** i < s.length()**;** i += **4**) {  
 hex.append(mp.get(s.substring(i**,**i + **4**)))**;** }  
 return hex.toString()**;** }  
  
 static String text2hex(String message){  
 return *bin2hex*(*text2bin*(message))**;** }  
  
 static String hex2text(String message){  
 return *bin2text*(*hex2bin*(message))**;** }  
}