Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 2

**Идентификация и аутентификация пользователей. Протокол Kerberos.**

Выполнила студентка гр. 053503

Зырянова М.М.

Проверил

Протько М.И.

Минск, 2023

# 1. Введение

Целью данной лабораторной было изучить теоретические сведения про алгоритм шифрования DES и протокол авторизации Kerberos, создать приложение, реализующее протокол распределения ключей Kerberos, включая процедуру, реализующую Алгоритм DES.

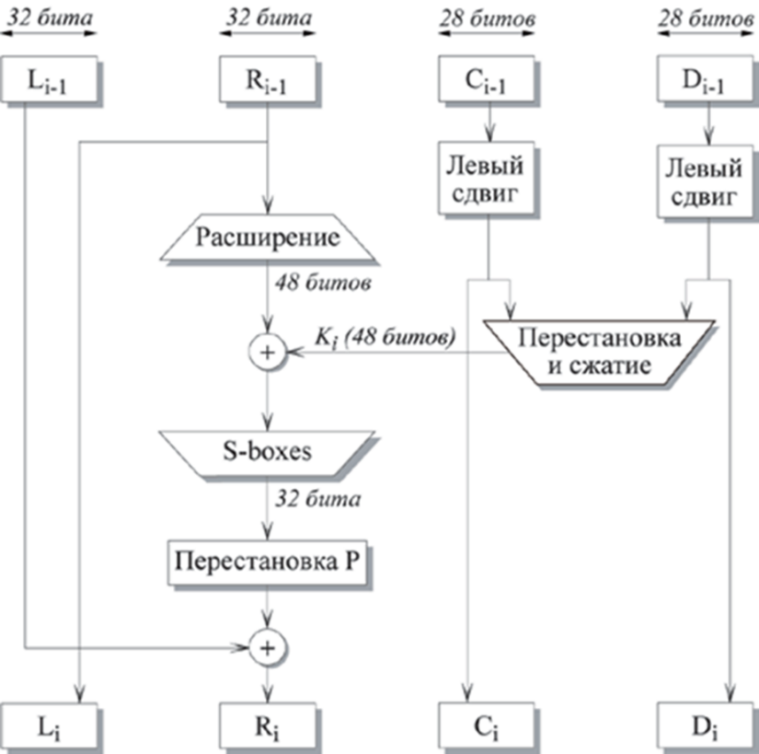
2. Блок-схемы алгоритмов

Протокол Kerberos:



Схема алгоритма DES:





Блок схема основной функции, реализующей шифрование DES:



**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):



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:

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

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

* **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.

* **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**).

* ***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.

* **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*.

* **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).

**Общая схема DES**

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.

Процесс шифрования состоит из четырех этапов.

* Выполняется начальная перестановка (IP) 64-битного исходного текста (забеливание), во время которой биты переупорядочиваются в соответствии со стандартной таблицей.
* Этап состоит из 16 раундов одной и той же функции, которая использует операции сдвига и подстановки.
* Левая и правая половины выхода последней (16-й) итерации меняются местами.
* Выполняется перестановка IP-1 результата, полученного на третьем этапе.

Перестановка IP-1 инверсна начальной перестановке.



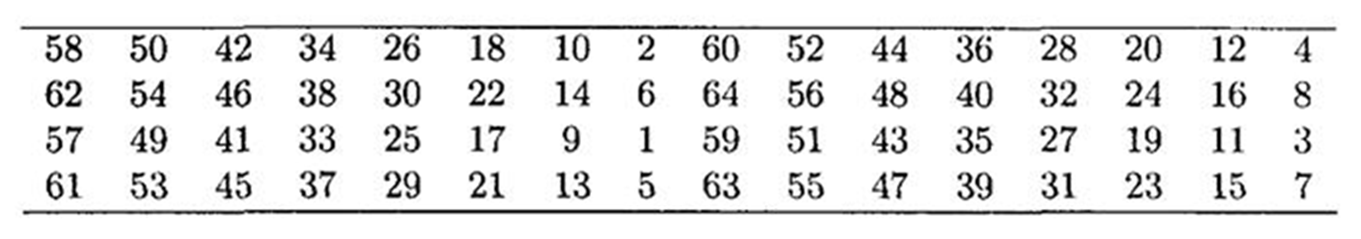
**Шифрование**

**Начальная перестановка**

Начальная перестановка и ее инверсия определяются стандартной таблицей.

Если М - это произвольные 64 бита, то

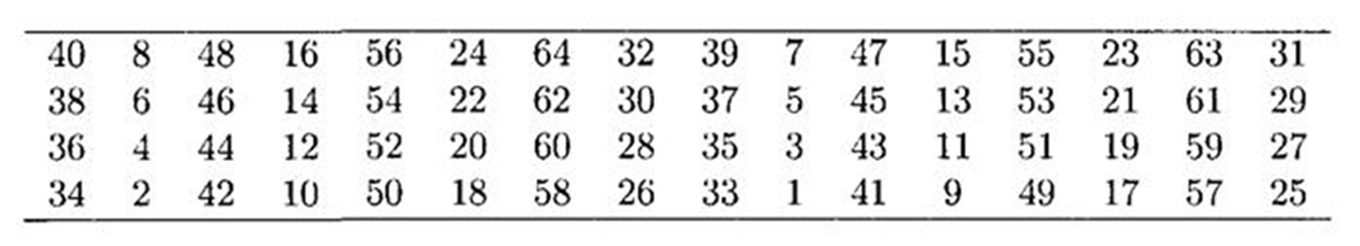
X = IP (M) - переставленные 64 бита.



Если применить обратную функцию перестановки

Y = IP-1 (X) = IP-1 (IP(M)),

то получится первоначальная последовательность бит.



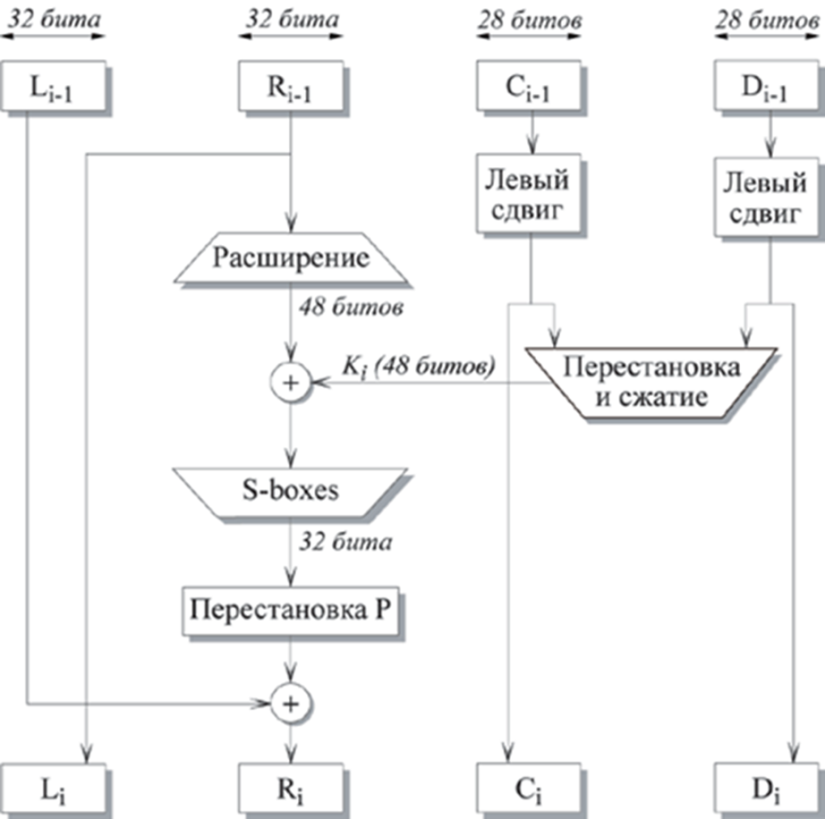
Начальная перестановка и соответствующая заключительная перестановка не влияют на криптостойкость DES (перестановка в первую очередь служит для облегчения побайтной загрузки данных открытого текста и шифротекста в микросхему DES).

Последовательность преобразований отдельного раунда

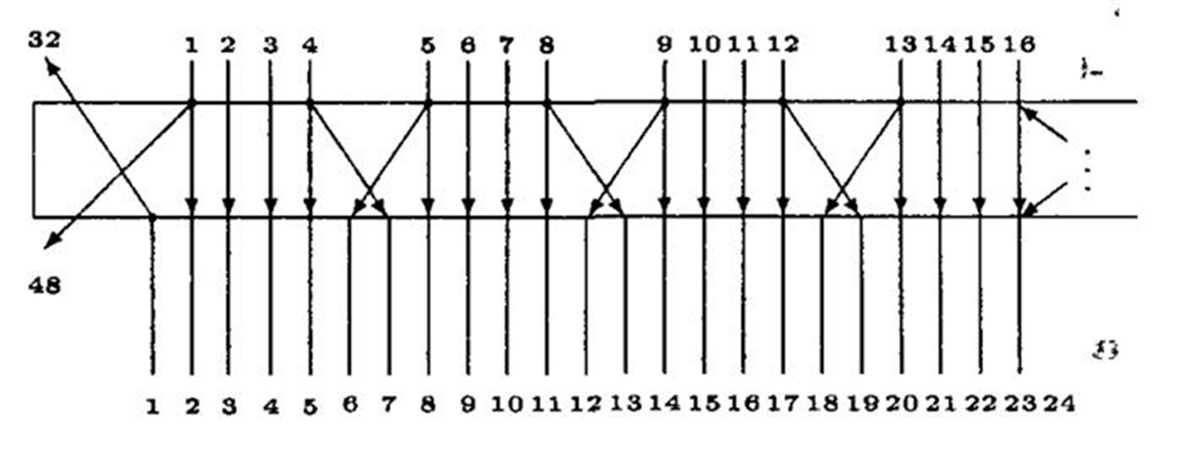
Каждую итерацию можно описать следующим образом:

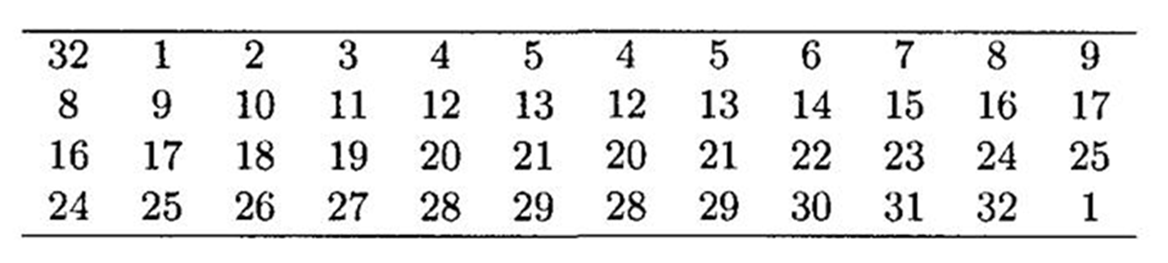
Li = Ri-1

Ri = Li-1F(Ri-1, Ki)



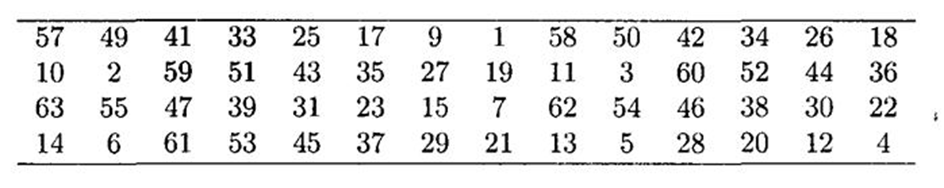
DES — схема перестановки с расширением





**Процедура преобразования ключа**

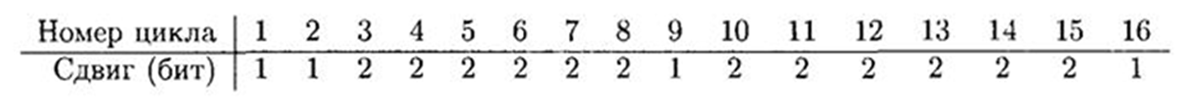
64-битовый ключ DES уменьшается до 56-битового ключа отбрасыванием каждого восьмого бита. Эти биты используются только для контроля четности, позволяя проверять правильность ключа.



После извлечения 56-битного ключа для каждого из 16 циклов генерируется новый 48-битный подключ.

Эти подключи, К2, определяются следующим образом:

* 56-битный ключ делится на две 28-битные половины.
* Половины циклически сдвигаются влево на один или два бита в зависимости от номера цикла.



**DES — подстановка при помощи S-блоков**

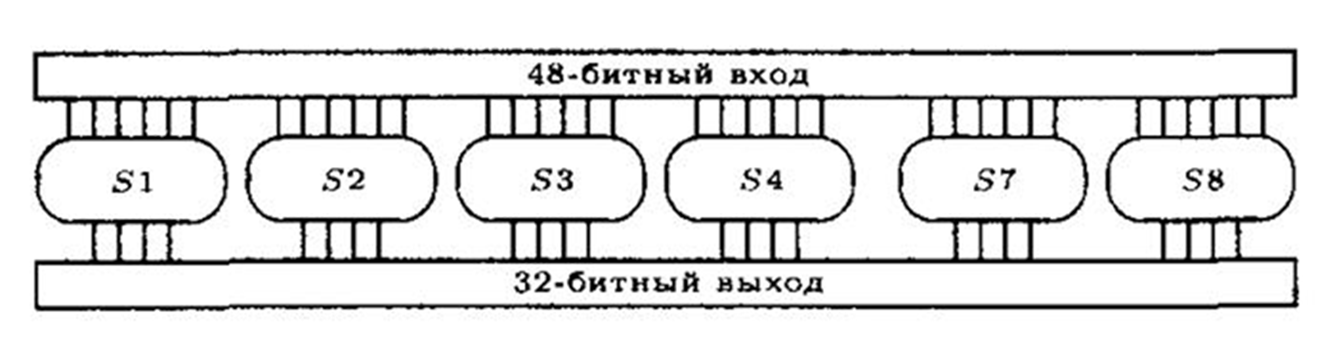
После объединения сжатого блока с расширенным блоком с помощью XOR над 48-битным результатом выполняется операция подстановки.

Подстановки производятся в восьми блоках подстановки, или S-блоках (от substitution).

У каждого S-блока есть 6-битный вход и 4-битный выход, всего используется восемь различных S-блоков.

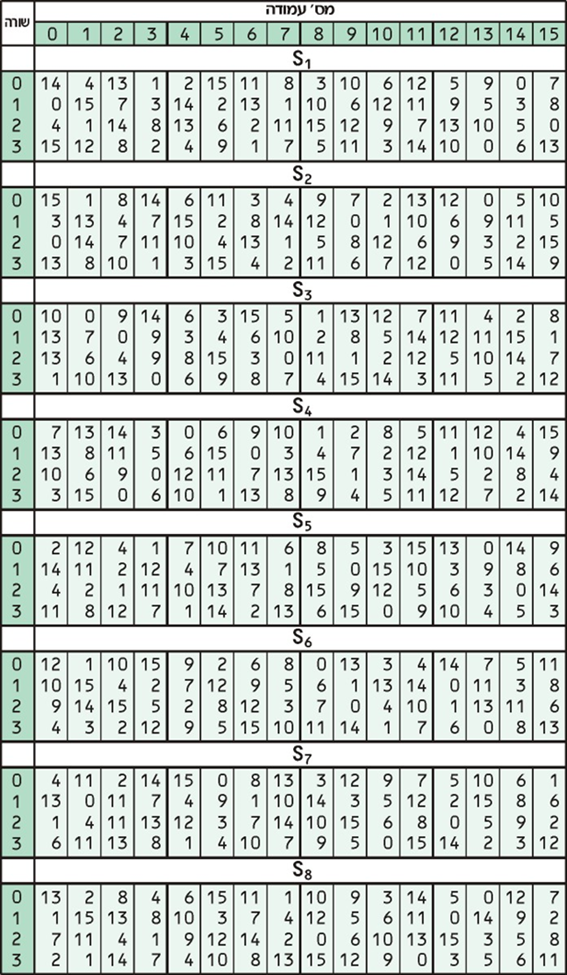
48 битов делятся на восемь 6-битных подблоков.

Каждый отдельный подблок обрабатывается отдельным S-блоком: первый подблок - первым S-блоком, второй - вторым S-блоком и так далее.



Каждый S-блок представляет собой таблицу из четырех строк и шестнадцати столбцов. Каждый элемент в блоке является 4-битным числом. По шести входным битам S-блока определяется, под какими номерами столбцов и строк следует искать выходное значение.

Например, пусть на вход шестого S-блока (то есть биты функции XOR с 31 по 36) попадает 110011. Первый и последний биты, объединяясь, образуют 11, что соответствует строке три шестого S-блока. Средние четыре бита образуют 1001, что соответствует столбцу девять того же S-блока. Элемент шестого S-блока, находящийся на пересечении строки три и столбца девять, - это 14 (строки и столбцы нумеруются с нуля, а не с единицы). Вместо 110011 подставляется 1110.



**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.

Проблемой DES является малая длина ключа.

Также без ответа пока остается вопрос, возможен ли криптоанализ с использованием существующих характеристик алгоритма DES.

Основой алгоритма являются восемь таблиц подстановки, или S-boxes, которые применяются в каждой итерации.

Существует опасность, что эти S-boxes конструировались таким образом, что криптоанализ возможен для взломщика, который знает слабые места S-boxes.

В течение многих лет обсуждалось как стандартное, так и неожиданное поведение S-boxes, но все-таки никому не удалось обнаружить их фатально слабые места.

подстановке, может быть записано целое число без знака в диапазоне от 0 до 15.

Значение блока S1 (четыре младших бита 32-разрядного числа S) заменится на число, стоящее на позиции, номер которой равен значению заменяемого блока.

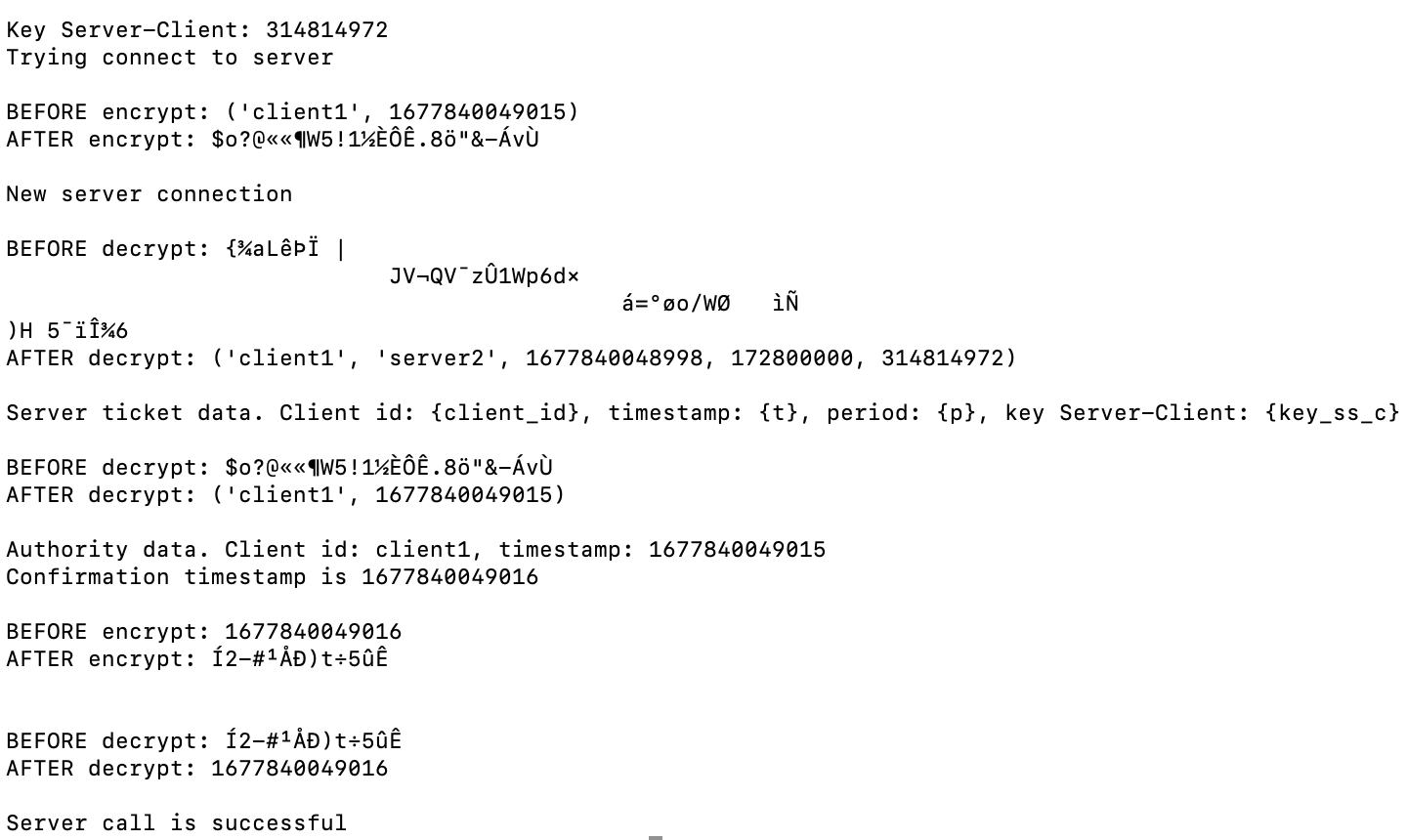
Например, в этом случае S1=0 заменится на 4, если S1=1, то оно заменится на 10 и т.д.

# 4. Результаты выполнения программы

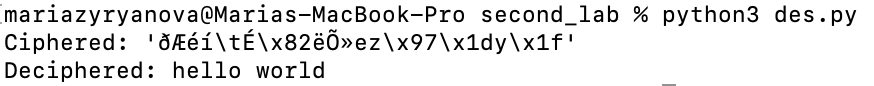
Kerberos:





****

Des:



# 5. Выводы

В рамках данной лабораторной работы был изучен и реализован протокол распределения ключей Kerberos, а также реализован алгоритм шифрования DES. В результате проделанной работы были выявлены следующие преимущества и недостатки данных алгоритмов:

Kerberos:

+ Эффективный доступ к ресурсам и проверка подлинности.

+ Взаимная проверка подлинности.

+ Делегированная проверка подлинности.

- KDC должен быть способен одновременно обрабатывать большое количество запросов. Поэтому он должен быть масштабируемым.

- Секретный ключ временно хранится на рабочей станции пользователя, что может привести к его компрометации.

- Kerberos уязвим к подбору паролей, он не сможет узнать, что атакующим производится подбор пароля по словарю.

- Kerberos не защищает сетевой трафик, если не включено шифрование.

- Kerberos требует, чтобы системные часы на всех клиентах и серверах были синхронизированы.

DES

+ Используется только один ключ длиной 56 бит  
+ Зашифровав сообщение с помощью одного пакета программ, для расшифровки можно

использовать любой другой пакет программ, соответствующий стандарту DES

+ Относительная простота алгоритма обеспечивает высокую скорость обработки  
+ Достаточно высокая стойкость алгоритма

- Допускает успешное применение лобовых атак.

- Возможен криптоанализ с использованием существующих характеристик алгоритма DES.

# 6. КОД ПРОГРАММЫ

**KERBEROS:**

import random  
import time  
  
from second\_lab.des import DES  
  
  
def current\_time\_in\_milliseconds():  
 return int(round(time.time() \* 1000))  
  
  
def convert\_hour\_to\_milliseconds(hour):  
 return hour \* 3\_600\_000  
  
  
def create\_key():  
 return random.randint(100\_000\_000, 999\_999\_999)  
  
  
class KDC:  
 available\_clients = ["client1", "client2"]  
 clients\_keys = [create\_key(), create\_key()]  
 available\_servers = ["server1", "server2"]  
 server\_keys = [create\_key(), create\_key()]  
  
 def \_\_init\_\_(self):  
 self.cipher\_algorithm = DES()  
 self.tgs\_id = 1  
 self.key\_tgs = create\_key()  
  
 def get\_permission\_ticket(self, client\_id):  
 print("Call for getting new permission ticket")  
 if client\_id in self.available\_clients:  
 t = current\_time\_in\_milliseconds()  
 p = convert\_hour\_to\_milliseconds(48)  
 key\_tgs\_c = create\_key()  
 ticket = (client\_id, self.tgs\_id, t, p, key\_tgs\_c)  
 print("New permission ticket:", ticket)  
  
 encrypted\_ticket = self.cipher\_algorithm.encrypt(ticket, self.key\_tgs)  
 bundle = (encrypted\_ticket, key\_tgs\_c)  
  
 index = self.available\_clients.index(client\_id)  
 client\_key = self.clients\_keys[index]  
 encrypted\_bundle = self.cipher\_algorithm.encrypt(bundle, client\_key)  
  
 return encrypted\_bundle  
  
 print("Unknown client\_id:", client\_id)  
  
 def get\_server\_ticket(self, permission\_ticket, authority, server\_id):  
 print("Call for getting new server ticket")  
 permission\_ticket = self.cipher\_algorithm.decrypt(  
 permission\_ticket, self.key\_tgs  
 )  
 client\_id, key\_tgs\_c = permission\_ticket[0], permission\_ticket[4]  
 t, p = permission\_ticket[2], permission\_ticket[3]  
  
 print(  
 f"Permission ticket data. Client id: {client\_id}, timestamp: {t},"  
 f" period: {p}, key TGS-Client: {key\_tgs\_c}"  
 )  
  
 authority = self.cipher\_algorithm.decrypt(authority, key\_tgs\_c)  
 auth\_client\_id = authority[0]  
 auth\_t = authority[1]  
  
 print(f"Authority data. Client id: {auth\_client\_id}, timestamp: {auth\_t}")  
  
 if client\_id != auth\_client\_id:  
 print("Invalid client")  
 return None  
  
 if auth\_t < t or auth\_t > t + p:  
 print("Ticket is expired")  
 return None  
  
 t = current\_time\_in\_milliseconds()  
 p = convert\_hour\_to\_milliseconds(48)  
 key\_ss\_c = create\_key()  
 server\_ticket = (client\_id, server\_id, t, p, key\_ss\_c)  
 print("New server ticket:", server\_ticket)  
  
 index = self.available\_servers.index(server\_id)  
 server\_key = self.server\_keys[index]  
 encrypted\_server\_ticket = self.cipher\_algorithm.encrypt(  
 server\_ticket, server\_key  
 )  
 bundle = (encrypted\_server\_ticket, key\_ss\_c)  
 encrypted\_bundle = self.cipher\_algorithm.encrypt(bundle, key\_tgs\_c)  
  
 return encrypted\_bundle  
  
  
class Client:  
 def \_\_init\_\_(self, client\_id, client\_key, kdc, server):  
 self.client\_id = client\_id  
 self.client\_key = client\_key  
 self.kdc = kdc  
 self.servers = server  
 self.cipher\_algorithm = DES()  
 self.permission\_ticket = None  
 self.key\_tgs\_c = None  
  
 def make\_server\_call(self, server\_number):  
 print("\nCall server", server\_number)  
 server = self.servers[server\_number]  
  
 if self.permission\_ticket is None or self.key\_tgs\_c is None:  
 print("Trying to get permission ticket")  
 permission\_ticket\_bundle = self.kdc.get\_permission\_ticket(self.client\_id)  
 if permission\_ticket\_bundle is None:  
 return None  
  
 permission\_ticket\_bundle = self.cipher\_algorithm.decrypt(  
 permission\_ticket\_bundle, self.client\_key  
 )  
 permission\_ticket, key\_tgs\_c = (  
 permission\_ticket\_bundle[0],  
 permission\_ticket\_bundle[1],  
 )  
 print("Key TGS-Client:", key\_tgs\_c)  
  
 self.permission\_ticket = permission\_ticket  
 self.key\_tgs\_c = key\_tgs\_c  
 else:  
 print("Permission ticket and key TGS-Client already defined")  
 permission\_ticket = self.permission\_ticket  
 key\_tgs\_c = self.key\_tgs\_c  
  
 print("Trying to get server ticket")  
 bundle = self.call\_tgs(permission\_ticket, key\_tgs\_c, server.server\_id)  
  
 if bundle is None:  
 return None  
  
 bundle = self.cipher\_algorithm.decrypt(bundle, key\_tgs\_c)  
 server\_ticket, key\_ss\_c = bundle[0], bundle[1]  
 print("Key Server-Client:", key\_ss\_c)  
  
 print("Trying connect to server")  
 t = current\_time\_in\_milliseconds()  
 authority = (self.client\_id, t)  
 authority\_encrypted = self.cipher\_algorithm.encrypt(authority, key\_ss\_c)  
 confirm\_t = server.connect(server\_ticket, authority\_encrypted)  
  
 if confirm\_t is None:  
 return  
  
 confirm\_t = self.cipher\_algorithm.decrypt(confirm\_t, key\_ss\_c)  
 if confirm\_t != t + 1:  
 print("Server returns wrong confirmation timestamp")  
 return  
  
 print("Server call is successful")  
  
 def call\_tgs(self, permission\_ticket, key\_tgs\_c, server\_id):  
 t = current\_time\_in\_milliseconds()  
 print(f"Call TGS. Server id: {server\_id}, timestamp: {t}")  
 authority = (self.client\_id, t)  
 authority\_encrypted = self.cipher\_algorithm.encrypt(authority, key\_tgs\_c)  
 bundle = self.kdc.get\_server\_ticket(  
 permission\_ticket, authority\_encrypted, server\_id  
 )  
  
 return bundle  
  
  
class Server:  
 def \_\_init\_\_(self, server\_id, server\_key):  
 self.server\_id = server\_id  
 self.server\_key = server\_key  
 self.cipher\_algorithm = DES()  
  
 def connect(self, server\_ticket, authority):  
 print("New server connection")  
 server\_ticket = self.cipher\_algorithm.decrypt(server\_ticket, self.server\_key)  
 client\_id, server\_id = server\_ticket[0], server\_ticket[1]  
 t, p, key\_ss\_c = server\_ticket[2], server\_ticket[3], server\_ticket[4]  
  
 print(  
 "Server ticket data. Client id: {client\_id}, timestamp: {t}, "  
 "period: {p}, key Server-Client: {key\_ss\_c}"  
 )  
  
 if server\_id != self.server\_id:  
 print("Wrong server")  
 return None  
  
 authority = self.cipher\_algorithm.decrypt(authority, key\_ss\_c)  
 auth\_client\_id, auth\_t = authority[0], authority[1]  
  
 print(f"Authority data. Client id: {auth\_client\_id}, timestamp: {auth\_t}")  
  
 if client\_id != auth\_client\_id:  
 print("Invalid client")  
 return None  
  
 if auth\_t < t or auth\_t > t + p:  
 print("Ticket is expired")  
 return None  
  
 confirm\_t = auth\_t + 1  
 print("Confirmation timestamp is", confirm\_t)  
 encrypted\_confirm\_t = self.cipher\_algorithm.encrypt(confirm\_t, key\_ss\_c)  
  
 return encrypted\_confirm\_t  
  
  
def processing():  
 kdc = KDC()  
 server\_1 = Server(kdc.available\_servers[0], kdc.server\_keys[0])  
 server\_2 = Server(kdc.available\_servers[1], kdc.server\_keys[1])  
 client = Client(  
 kdc.available\_clients[0], kdc.clients\_keys[0], kdc, [server\_1, server\_2]  
 )  
  
 print(f"server0 id: {server\_1.server\_id}, server0 key: {server\_1.server\_key}")  
 print(f'server1 id: {server\_2.server\_id}, server1" key: {server\_2.server\_key}')  
 print(f"Client id: {client.client\_id}, Client key: {client.client\_key}")  
  
 client.make\_server\_call(0)  
 client.make\_server\_call(1)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 processing()

**DES:**

from second\_lab.des\_resources import (  
 D\_BOX,  
 FP,  
 IP,  
 KP\_1,  
 KP\_2,  
 S\_BOX,  
 SHIFTS,  
 STRAIGHT\_PERMUTATION,  
)  
  
  
def split\_bites(bites: list, nth: int) -> list:  
 return [bites[nth \* i : nth \* (i + 1)] for i in range(len(bites) // nth)]  
  
  
def convert\_bits\_to\_string(bites: list) -> str:  
 return "".join(  
 [  
 chr(int(string, 2))  
 for string in [  
 "".join(list(map(str, byte))) for byte in split\_bites(bites, 8)  
 ]  
 ]  
 )  
  
  
def remove\_two\_bites(character: str) -> list:  
 return (  
 list(bin(character)[2:])  
 if isinstance(character, int)  
 else list(bin(ord(character))[2:])  
 )  
  
  
def pull\_bite\_number(bites: list, nth: int) -> None:  
 [bites.insert(0, "0") for i in range(nth - len(bites))]  
  
  
def convert\_string\_to\_bits(string: str) -> list:  
 bites = []  
 for character in string:  
 byte = remove\_two\_bites(character)  
 pull\_bite\_number(byte, 8)  
  
 bites.extend(byte)  
  
 return list(map(int, bites))  
  
  
def permute(key: list or str, table: list) -> list:  
 return [key[i - 1] for i in table]  
  
  
def xor(a: list, b: list) -> list:  
 return [x ^ y for x, y in zip(a, b)]  
  
  
def generate\_keys(password: str) -> list:  
 keys = []  
 key = convert\_string\_to\_bits(password)  
 key = permute(key, KP\_1)  
 left, right = split\_bites(key, 28)  
 for i in range(16):  
 left, right = left\_shift(left, SHIFTS[i]), left\_shift(right, SHIFTS[i])  
 compose = left + right  
 keys.append(permute(compose, KP\_2))  
  
 return keys  
  
  
def left\_shift(key: list, nth\_shifts: int) -> list:  
 return key[nth\_shifts:] + key[:nth\_shifts]  
  
  
def right\_shift(key: list, nth\_shifts: int) -> list:  
 return key[len(key) - nth\_shifts :] + key[: len(key) - nth\_shifts]  
  
  
def add\_padding(text: list or str) -> list:  
 pad\_len = 8 - (len(text) % 8)  
 text += pad\_len \* chr(pad\_len)  
  
 return text  
  
  
def remove\_padding(data: list or str) -> list:  
 pad\_len = ord(data[-1])  
  
 return data[:-pad\_len]  
  
  
def substitute(expanded\_value: list) -> list:  
 blocks = split\_bites(expanded\_value, 6)  
 result = list()  
  
 for i in range(len(blocks)):  
 block = blocks[i]  
 row = int(str(block[0]) + str(block[5]), 2)  
 column = int("".join([str(x) for x in block[1:][:-1]]), 2)  
 table\_value = S\_BOX[i][row][column]  
  
 bites = remove\_two\_bites(table\_value)  
 pull\_bite\_number(bites, 4)  
  
 result += [int(x) for x in bites]  
  
 return result  
  
  
def des(key: str, text: str, \*, encrypt: bool = True, padding: bool = False):  
 if len(key) < 8:  
 raise Exception("Length of key is lower than 8 symbols!")  
 elif len(key) > 8:  
 key = key[:8]  
  
 if padding and encrypt:  
 text = add\_padding(text)  
 elif len(text) % 8 != 0:  
 raise Exception("Data size should be multiple of 8!")  
  
 keys = generate\_keys(key)  
 text\_blocks = split\_bites(text, 8)  
 result = []  
  
 for block in text\_blocks:  
 block = convert\_string\_to\_bits(block)  
 block = permute(block, IP)  
 left, right = split\_bites(block, 32)  
  
 for i in range(16):  
 right\_expanded = permute(right, D\_BOX)  
 if encrypt:  
 temp\_result = xor(keys[i], right\_expanded)  
 else:  
 temp\_result = xor(keys[15 - i], right\_expanded)  
  
 temp\_result = substitute(temp\_result)  
 temp\_result = permute(temp\_result, STRAIGHT\_PERMUTATION)  
 temp\_result = xor(left, temp\_result)  
 left, right = right, temp\_result  
 result += permute(right + left, FP)  
  
 return (  
 remove\_padding(convert\_bits\_to\_string(result))  
 if padding and not encrypt  
 else convert\_bits\_to\_string(result)  
 )  
  
  
class DES:  
 @staticmethod  
 def encrypt(data: list or tuple, key: int) -> str:  
 print("\nBEFORE encrypt:", data)  
 cipher = des(str(key), str(data), encrypt=True, padding=True)  
 print("AFTER encrypt:", cipher, end="\n\n")  
  
 return cipher  
  
 @staticmethod  
 def decrypt(data: str, key: int) -> tuple:  
 print("\nBEFORE decrypt:", data)  
 plain\_text = eval(des(str(key), data, encrypt=False, padding=True))  
 print("AFTER decrypt:", plain\_text, end="\n\n")  
  
 return plain\_text  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 key = "ilya\_lox"  
 text = "Shevchen"  
 res = des(key, text, encrypt=True, padding=True)  
 print("Ciphered:", repr(res))  
 sd = des(key, res, encrypt=False, padding=True)  
 print("Deciphered:", sd)