Московский авиационный институт

(государственный технический университет)

**Факультет прикладной математики и физики**

Кафедра вычислительной математики и программирования

**Лабораторная работа №4**

по спецкурсу «Криптография».

Тема: «Криптографические алгоритмы».

|  |  |
| --- | --- |
| Выполнила: | Карпова В.А. |
| Группа: | 08-306 |
| № по списку: | 9 |
| Преподаватель: | Рисенберг Д.В. |
| Оценка: |  |
| Дата: |  |

Москва

2012 г.

**Задание**

Необходимо написать программу на языке C++, C# или Python, реализующую

криптографический алгоритм в соответствии с вариантом. Разрешается использование сторонних реализаций длинной арифметики и генераторов случайных чисел (boost::random,System.Random и т.д.).

**Вариант 2**

Реализовать программу шифрования и расшифровки данных фиксированного размера. Алгоритм симметричного шифрования Blowfish.

*Требования*:

Формат вызова программы шифрования:

<программа> -e <исходный файл> <файл с ключом> <зашифрованный файл>

Формат вызова программы расшифровки:

<программа> -d <исходный файл> <файл с ключом> <расшифрованный файл>

**Описание алгоритма.**

Blowfish— криптографический алгоритм, реализующий блочное симметричное шифрование. Является не запатентованным и свободно распространяемым.

1) Параметры:

-- секретный ключ K (от 32 до 448 бит)

-- 32-битные ключи шифрования P-блоки: Р1-Р18

-- 32-битные таблицы замен S-блоки:

S1[0] S1[1] .. S1[255]

S2[0] S2[1] .. S2[255]

S3[0] S3[1] .. S3[255]

S4[0] S4[1] .. S4[255]

2) Blowfish представляет собой сеть Фейстеля.

Сеть Фейстеля — один из методов построения блочных шифров. Сеть представляет собой определённую многократно повторяющуюся структуру, называющуюся ячейкой Фейстеля. При переходе от одной ячейки к другой меняется ключ. Выбор ключа зависит от конкретного алгоритма. Операции шифрования и расшифрования на каждом этапе просты, и при определённой доработке совпадают.

Перед началом шифрования нужно 521 раз произвести зашифровку ключей, что делает его сложным к расшифровке.

Алгоритм Blowfish разделён *на 2 этапа*:

1. Подготовительный — формирование ключей шифрования по секретному ключу.

Для получения ключей шифрования используются P и S - блоки, и Р-блоки циклично XOR'ятся с входным ключом. Первоначальная инициализация P и S - блоков происходит фиксированной строкой, состоящей из шестнадцатеричных цифр мантиссы числа PI. Затем, с помощью сети Фейстеля, получаем ключи шифрования. При этом учитываем: если ключ короче блоков, то он накладывается циклически.

2. Шифрование текста по полученным ключам.

Перед шифрованием с помощью сети Фейстеля необходимо разделить текст на блоки по 64 бита. Чтобы любой текст можно было разделить на блоки по 64 бита, нужно сделать проверку длины теста. Если длина не кратна 64 битам, то дополняем текст терминальными нулями '\0'.

Дешифрование происходит аналогично, только XOR с Р-блоками применяется в обратном порядке.

Рассмотрим подробнее:

1) Выбор начального значения P-массива и таблицы замен (P и S - блоки).

В Р и S - блоки записывается мантисса числа PI. Данный выбор заключается в инициализации последовательности, не связанной с алгоритмом, которая могла бы быть сохранена как часть алгоритма или получена при необходимости. Как указывает Шнайер: «Подойдёт любая строка из случайных битов цифр числа e, RAND-таблицы, или случайные сгенерированные цифры»

2) Сеть Фейстеля в Blowfish (состоит из 16 раундов) :

На вход подаем 64битный блок.

Разделяем его на 2 блока: XOR-им c ключами .

Затем, итерационно вычисляем:

После 16 раундов меняем местами.

3) Функция F(x), используемая в п.1 выглядит следующим образом:

На вход подается 32-битный блок, который делится на четыре 8-битных блока (X1, X2, X3, X4). Каждый из этих блоков является индексом массива таблицы замен S-блоков. Значения S1[X1] и S2[X2] складываются по модулю , после "XOR"ятся с S3[X3] , и затем складываются с S4[X4] по модулю :

**Оценка сложности**

Суммарная требуемая память 4168 байт: P1-P18:18 переменных по 32 бита; S1-S4: 4x256 переменных по 32 бита. Т.е. суммарный расход памяти ~ 4 Кб.

**Выводы**

До появления Blowfish существовавшие алгоритмы были либо запатентованными, либо ненадёжными.

Алгоритм был разработан Брюсом Шнайером в качестве альтернативы устаревшему DES и запатентованному IDEA. По заявлению автора, критерии проектирования Blowfish были:

-- скорость (шифрование на 32-битных процессорах происходит за 26 тактов);

-- простота (за счёт использования простых операций, уменьшающих вероятность ошибки реализации алгоритма);

-- настраиваемая стойкость.

Криптостойкость зависит от F(x) и ее можно настраивать за счёт изменения количества раундов шифрования (увеличивая длину массива P) и количества используемых S-box. При уменьшении используемых S-box возрастает вероятность появления слабых ключей, но уменьшается используемая память. Адаптируя Blowfish на 64-битной архитектуру, можно увеличить количество и размер S-box (а следовательно и память для массивов P и S), а также усложнить F(x).

Blowfish зарекомендовал себя как надёжный алгоритм, поэтому реализован во многих программах, где не требуется частая смена ключа и необходима высокая скорость шифрования/расшифрования.

* защита электронной почты и файлов
* GnuPG (безопасное хранение и передача)
* в линиях связи: связка ElGamal (не запатентован) или RSA (действие патента закончилось в 2000 году) и Blowfish вместо IDEA
* PuTTY (сетевой уровень)
* SSH (транспортный уровень)
* OpenVPN (создание зашифрованных каналов)
* 1Password (Mac, iPhone)

**Реализация**

#! /usr/bin/python3

# -\*- coding: utf-8 -\*-

#~ Blowfish implementation

#~ look info http://www.fish-journal.com/2011/10/blowfish.html

#~ Импортируются блоки

from blowfish\_blocks import blocks

from copy import deepcopy

P, S = deepcopy(blocks.P), deepcopy(blocks.S)

#~ Получение kго байта из числа n

def get\_byte(n,k):

return (n>>(k\*8))&0xFF

#~ Получение kго 32х битного блока из числа n

def get\_block32(n,k):

return (n>>(k\*32))&0xFFFFFFFF

#~ Разделить 32х битный блок по байтам

def split\_block32(block):

return [get\_byte(block,i) for i in range(3,-1,-1)]

def F(x):

global S

x3, x2, x1, x0 = [get\_byte(x,i) for i in range(4)]

return (((S[0][x0]+S[1][x1]) % 2\*\*32 ^ S[2][x2]) +S[3][x3]) % 2\*\*32

#~ Сеть Фейстеля для шифрования

def feistel\_encrypt(block):

global P

l, r = get\_block32(block,0), get\_block32(block,1)

n = 16

for i in range(n):

l = l ^ P[i]

r = F(l) ^ r

l, r = r, l

l, r = r, l

r = r ^ P[n]

l = l ^ P[n+1]

return l, r

#~ Сеть Фейстеля для расшифровки

def feistel\_decrypt(block):

global P

l, r = get\_block32(block,1), get\_block32(block,0)

n = 16

for i in range(n+1,1,-1):

l = l ^ P[i]

r = F(l) ^ r

l, r = r, l

l,r = r, l

r = r ^ P[1]

l = l ^ P[0]

return l, r

#~ Создание секретных ключей

def create\_cryptkey(key):

key\_len = len(key)

if not key or key\_len < 8 or key\_len > 56:

raise ValueError("Incorrect length of the key")

global P, S

P, S = deepcopy(blocks.P), deepcopy(blocks.S)

#~ Цикличное применение XOR: ключ XORится с Р-блоком

index = 0

for i in range (len (P)):

key\_32bit = 0

for j in range(4):

key\_32bit <<= 8

key\_32bit |= key[(index+j)%key\_len]

P[i] = P[i] ^ key\_32bit

index = index+4

#~ Начальное изменение P и S блоков с помощью шифрования Фейстеля

block = 0

for i in range(0,len(P),2):

l,r = feistel\_encrypt(block)

P[i] = l

P[i+1] = r

block = r; block <<= 32; block |= l

for i in range(0,len(S)):

for j in range(0,len(S[i]),2):

l,r = feistel\_encrypt(block)

S[i][j] = l

S[i][j+1] = r

block = r

block <<= 32; block |= l

def Blowfish(text,key,mode):

#~ chiperbyte - зашифрованный текст

chiperbyte = bytearray()

l,r = 0,0

for i in range(len(text)-8,-8,-8):

block64 = 0

#~ Разделяем текст на блоки по 8 байт

for j in range(8):

block64 <<= 8

block64 |= text[i+j]

if mode == "ENCRYPT":

l, r = get\_block32(block64,1), get\_block32(block64,0)

block64 = r; block64 <<= 32; block64 |= l

l, r = feistel\_encrypt(block64)

elif mode == "DECRYPT":

l, r = get\_block32(block64,0), get\_block32(block64,1)

block64 = r; block64 <<= 32; block64 |= l

l, r = feistel\_decrypt(block64)

#~ Добавляем в зашифрованный текст побайтовый список

chiperbyte.extend(split\_block32(l))

chiperbyte.extend(split\_block32(r))

return chiperbyte

def exec\_blowfish(key, plaintext, answer=None, mode=None, outfile=None):

from binascii import a2b\_hex, b2a\_hex

#~ Текст должен быть кратным 64 битам

if mode == "ENCRYPT":

#expand text to mod 64 bit

mod = len(plaintext)%8

if mod != 0:

for i in range(8-mod):

plaintext += b'\0'

#~ Если аргументов командной строки нет, то тестирование на стандартных примерах

if (mode, outfile) == (None, None):

key = a2b\_hex(key)

plaintext = a2b\_hex(plaintext)

create\_cryptkey(key)

encrypttext = Blowfish(plaintext, key, "ENCRYPT")

decrypttext = Blowfish(encrypttext, key, "DECRYPT")

b2a\_txt,b2a\_key,b2a\_enc,b2a\_dec = [

b2a\_hex(bytes(t)).upper() for t in

(plaintext,key,encrypttext,decrypttext)]

print("key = ", b2a\_key, "plaintext = ",b2a\_txt)

print("enc = ", b2a\_enc, "dec = ", b2a\_dec)

if b2a\_enc != answer or b2a\_dec != b2a\_txt:

print("FAIL")

exit(-1)

else:

print("OK")

else:

#~ При заданных режимах шифрования и расшифровки

create\_cryptkey(key)

encrypttext = Blowfish(plaintext, key, mode)

#~ b2a\_enc = b2a\_hex(bytes(encrypttext)).upper()

b2a\_enc = bytes(encrypttext)

print(b2a\_enc)

open(outfile,'wb').write(b2a\_enc)

def standart\_tests():

vectors = (

(b'0000000000000000', b'0000000000000000', b'4EF997456198DD78'),

(b'FFFFFFFFFFFFFFFF', b'FFFFFFFFFFFFFFFF', b'51866FD5B85ECB8A'),

(b'3000000000000000', b'1000000000000001', b'7D856F9A613063F2'),

(b'1111111111111111', b'1111111111111111', b'2466DD878B963C9D'),

(b'49E95D6D4CA229BF', b'02FE55778117F12A', b'CF9C5D7A4986ADB5'),

(b'E0FEE0FEF1FEF1FE', b'0123456789ABCDEF', b'C39E072D9FAC631D'),

(b'07A7137045DA2A16', b'3BDD119049372802', b'2EEDDA93FFD39C79'),

(b'0000000000000000', b'0000000000000000', b'4EF997456198DD78'),

(b'FFFFFFFFFFFFFFFF', b'FFFFFFFFFFFFFFFF', b'51866FD5B85ECB8A'),

)

for k,t,a in vectors:

exec\_blowfish(k,t,a)

def main():

#~ Работа с аргументами командной строки

import optparse

cl = optparse.OptionParser()

cl.add\_option("-e",action = "store", dest = "enc", nargs = 3)

cl.add\_option("-d",action = "store", dest = "dec", nargs = 3)

opts, args = cl.parse\_args()

enc, dec = opts.enc, opts.dec

fin, fkey, fout, mode = 0,0,0,0

if enc != None:

fin, fkey, fout = enc

mode = "ENCRYPT"

elif dec != None:

fin, fkey, fout = dec

mode = "DECRYPT"

else:

print("No args: standart test...")

standart\_tests()

return None

key = open(fkey,'rb').read()

plaintext = open(fin,'rb').read()

exec\_blowfish(key=key, plaintext=plaintext, mode=mode, outfile=fout)

if \_\_name\_\_ == "\_\_main\_\_":

main()