МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации



**ОТЧЁТ**

**по практической работе №2**

**«**Изучение математических основ криптографии**»**

**по дисциплине: «***Программирование***»**

Выполнил:Проверил:

Студент гр. «АБс-324», «АВТФ» *Доцент ЗИ*

*Петров М. И. Архипова А. Б.*

«\_\_» \_\_\_\_\_\_ 2024 г.«\_\_\_» \_\_\_\_\_\_ 2024 г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск 2024

**Цели и задачи работы**: изучение циклических алгоритмов, операторов цикла, программирование циклического вычислительного процесса на примере математических методов основ криптографии.

**Задание к работе**: Реализовать циклический вычислительный процесс. Самостоятельно решить задачи в соответствии с индивидуальным вариантом.

**Задания:**

**Задание 1.** Реализовать a x mod p Сравнения по модулю простого числа через теорему Ферма и свойства сравнений. Программа должна проверять условия выполнения теоремы Ферма и простоту вводимого пользователем модуля. Реализовать алгоритм через разложение степени в двоичный вид (логарифм).

**Задание 2.** Реализовать обобщенный алгоритм Евклида для вычисления с\*d mod m=1.

**Задание 3.** Реализовать расширенный алгоритм Евклида для вычисления взаимообратного числа с -1 mod m=d.

**Задание 4.** Написать программу, использующую алгоритм шифрования данных для преобразования исходного текста. **Вариант 5 - Хьюза (Hughes)**.

**Задание 5.** Найти последнюю цифру «трехэтажного числа».

**Задание 6.** Написать сообщение на тему «Атака посередине». Можно продемонстрировать эмуляцию атаки на базе программы задания 4 данной практической работы.

**Задание 7.** Написать сообщение на тему «Стандарты современной криптографии в РФ».

**Методика выполнения работы**:

1. Разработать алгоритм решения задачи по индивидуальному заданию.
2. Написать и отладить программу решения задачи.
3. Протестировать работу программы на различных исходных данных.
4. Ответить на вопросы по выполненным заданиям, по запросу преподавателя модифицировать код.
5. Ответить на вопросы согласно списку понятий к защите практики (с численными примерами).
6. По запросу преподавателя решить практическое задание на тему «Изучение математических основ криптографии».

**Задания**

# Задание 1. Реализовать a x mod p Сравнения по модулю простого числа через теорему Ферма и свойства сравнений. Программа должна проверять условия выполнения теоремы Ферма и простоту вводимого пользователем модуля. Реализовать алгоритм через разложение степени в двоичный вид (логарифм).

## C++:

#include <iostream>

#include <math.h>

using namespace std;

bool CheckPrime(int p) { // проверка, является ли модуль простым числом

if (p <= 1) return false;

for (int i = 2; i \* i <= p; ++i) {

if (p % i == 0) return false;

}

return true;

}

int algorithm1(int a, int x, int p) { // Алгоритм1 через уменьшение степени

int result = 1;

a %= p;

while (x > 0) {

if (x % 2 == 1)

result = (result \* a) % p;

x /= 2;

a = (a \* a) % p;

}

return result;

}

int algorithm2(int a, int x, int p) { // Алгоритм2 через логарифм

int nums[100]; // инициализируем массив чисел, в котором хранится числовой ряд a

int stepen = 1; // для проверки, чтобы степень не превышала себя

nums[0] = a % p;

int n=1;

while (stepen <= x) { // вычисляем числовой ряд для числа a

stepen \*= 2;

nums[n] = (nums[n-1] \* nums[n-1]) % p;

n += 1;

}

int binary[100]; // инициализируем массив чисел, в котором хранится число в двоичной степени

n = 0;

while (x != 1) { // переводчим степень в двоичное число

binary[n] = x % 2;

x /= 2;

n += 1;

}

binary[n] = x;

n += 1;

int result = 1;

for (int i = 0; i < n; ++i) { // используем конечную формулу алгоритма

result \*= pow(nums[i], binary[i]);

}

result %= p;

return result;

}

int main(){

setlocale(LC\_ALL, "Rus");

cout << "Нахождение остатка числа a^x mod p:" << endl;

int a, x, p;

cout << "Введите числа a, x и p: ";

cin >> a >> x >> p;

if (CheckPrime(p)==false) { // проверка, является ли модуль простым числом

cout << "p не простое число." << endl;

return 1;

}

if (a % p == 0 && x != 0) { // проверка, выполняется ли теорема Ферма

cout << "Теорема Ферма не выполняется: a^x ≡ 0 (mod p)" << endl;

return 1;

}

int result1 = algorithm1(a, x, p);

cout << "Алгоритм 1: " << a << "^" << x << " mod " << p << " = " << result1 << endl;

int result2 = algorithm2(a, x, p);

cout << "Алгоритм 2: " << a << "^" << x << " mod " << p << " = " << result2 << endl;

return 0;

}

## Python:

# Теорема Ферма и св-во сравнений(2 алгоритма)

def CheckPrime(p): # проверка, является ли модуль простым числом

if (p <= 1): return False

for i in range(2, int(p \*\* 0.5) + 1):

if (p % i == 0): return False

return True

def algorithm1(a, x, p): # Алгоритм1 через уменьшение степени

result = 1

a %= p

while (x > 0):

if (x % 2 == 1):

result = (result \* a) % p

x //= 2

a = (a \* a) % p

return result

def algorithm2(a, x, p):

nums = [0] \* 100 # инициализируем массив чисел, в котором хранится числовой ряд a

stepen = 1 # для проверки, чтобы степень не превышала себя

nums[0] = a % p

n = 1

while (stepen <= x): # вычисляем числовой ряд для числа a

stepen \*= 2

nums[n] = (nums[n-1] \* nums[n-1]) % p

n += 1

binary = [0] \* 100 # инициализируем массив чисел, в котором хранится число в двоичной степени

n = 0

while (x != 1): # переводчим степень в двоичное число

binary[n] = x % 2

x //= 2

n += 1

binary[n] = x

n += 1

result = 1

for i in range(n): # используем конечную формулу алгоритма

result \*= pow(nums[i], binary[i])

result %= p

return result

print("Нахождение остатка числа a^x mod p:")

a = int(input("Введите число а: "))

x = int(input("Введите степень x: "))

p = int(input("Введите модуль p: "))

if (CheckPrime(p) == False): print("p не простое число.") # проверка, является ли модуль простым числом

elif (a % p == 0) and (x != 0): print("Теорема Ферма не выполняется: a^x ≡ 0 (mod p)") # проверка, выполняется ли теорема Ферма

else:

result1 = algorithm1(a, x, p)

print(f"Алгоритм 1: {a}^{x} mod {p} = {result1}")

result2 = algorithm2(a, x, p)

print(f"Алгоритм 2: {a}^{x} mod {p} = {result2}")

# Задание 2. Реализовать обобщенный алгоритм Евклида для вычисления с\*d mod m=1.

## С++:

#include <iostream>

#include <vector>

using namespace std;

void algorithm(int c, int m, int &gcd, int &d) { // Алгоритм нахождения НОД(c, m) и числа d

int q;

vector<int> one = {m, 0};

vector<int> two = {c, 1};

vector<int> three(2);

while (two[0] != 0) {

q = one[0] / two[0];

three[0] = one[0] % two[0];

three[1] = one[1] - (q \* two[1]);

one[0] = two[0]; one[1] = two[1];

two[0] = three[0]; two[1] = three[1];

}

gcd = one[0], d = one[1];

}

int main(){

system ("chcp 65001");

cout << "Нахождение d, чтобы c\*d mod m = 1:" << endl;

int c, m;

cout << "Введите числа c и m: ";

cin >> c >> m;

int gcd, d;

algorithm(c, m, gcd, d);

if (gcd != 1) {

cout << "Такого d не существует, т.к. c и m не взаимнопростые.";

} else {

if (d < 0) {

d += m;

}

cout << "Число d: " << d << endl << c\*d << " mod " << m << " = " << (c\*d) % m;

}

return 0;

}

## Python:

def algorithm(c, m): # Алгоритм нахождения НОД(c, m) и числа d

one = [m, 0]

two = [c, 1]

three = [0, 0]

q = 0

while (two[0] != 0):

q = one[0] // two[0]

three[0] = one[0] % two[0]

three[1] = one[1] - (q \* two[1])

one[0], one[1] = two[0], two[1]

two[0], two[1] = three[0], three[1]

return one[0], one[1]

print("Нахождение d, чтобы c\*d mod m = 1:")

c = int(input("Введите число c: "))

m = int(input("Введите число m: "))

gcd, d = algorithm(c, m)

if (gcd != 1):

print("Такого d не существует, т.к. c и m не взаимнопростые.")

else:

if (d < 0):

d += m

print("Число d:", d)

print(c\*d, "mod", m, "=", (c\*d) % m)

# **Задание 3.** Реализовать расширенный алгоритм Евклида для вычисления взаимообратного числа с -1 mod m=d.

## C++:

#include <iostream>

#include <vector>

using namespace std;

int gcd(int c, int m) { // нахождение НОД(c, m)

if (m == 0) return c;

return gcd(m, c % m);

}

int main(){

setlocale(LC\_ALL, "Rus");

cout << "Реализация расширенного алгоритм Евклида для вычисления взаимообратного числа с^-1 mod m = d." << endl;

int c, m;

cout << "Введите числа c и m: ";

cin >> c >> m;

if (gcd(c, m) != 1) {

cout << "Числа не взаимообратные.";

return 1;

}

// реализация расширенного алгоритма и вывод его значений

cout << "Табличное представление данных расширенного алгоритма Евклида: " << endl;

cout << "A | " << c << " " << "1 " << "0" << endl;

cout << "B | " << m << " " << "0 " << "1" << endl;

int q;

vector<int> one = {c, 1, 0};

vector<int> two = {m, 0, 1};

vector<int> three(3);

while (two[0] != 0) {

q = one[0] / two[0];

three[0] = one[0] % two[0];

three[1] = one[1] - (q \* two[1]);

three[2] = one[2] - (q \* two[2]);

one = two;

two = three;

cout << "T | " << three[0] << " " << three[1] << " " << three[2] << " " << q << endl;

}

if (one[1] < 0) {

one[1] += m;

}

cout << "Взаимно обратное: " << one[1];

return 0;

}

## Python:

def gcd(c, m): # нахождение НОД(c, m)

if (m == 0): return c

return gcd(m, c % m)

print("Реализовать расширенный алгоритм Евклида для вычисления взаимообратного числа s^-1 mod m = d.")

c = int(input("Введите число c: "))

m = int(input("Введите число m: "))

if (gcd(c, m) != 1):

print("Числа не взаимообратные.")

exit(1)

# реализация расширенного алгоритма и вывод его значений

print("Табличное представление данных расширенного алгоритма Евклида:")

print(f"A | {c} 1 0")

print(f"B | {m} 0 1")

q = 0

one = [c, 1, 0]

two = [m, 0, 1]

three = [0, 0, 0]

while (two[0] != 0):

q = one[0] // two[0]

three[0] = one[0] % two[0]

three[1] = one[1] - (q \* two[1])

three[2] = one[2] - (q \* two[2])

one = two.copy()

two = three.copy()

print(f"T | {three[0]} {three[1]} {three[2]} {q}")

if (one[1] < 0):

one[1] += m

print(f"Взаимно обратное: {one[1]}")

# **Задание 4.** Написать программу, использующую алгоритм шифрования данных для преобразования исходного текста. **Вариант 5 - Хьюза (Hughes)**.

## C++:

#include <iostream>

#include <time.h>

#include <cmath>

#include <vector>

using namespace std;

// нахождение НОД(a, b) для проверки, являются ли числа взаимопростыми

int VzaimProst(int a, int b) {

if (b == 0) return a;

return VzaimProst(b, a % b);

}

// проверка, является ли число простым

bool CheckPrime(int p) {

if (p <= 1) return false;

for (int i = 2; i \* i <= p; ++i)

{

if (p % i == 0) return false;

}

return true;

}

// функция Эйлера

int fi(int num) {

int result = 1;

int stepen = 0;

for (int i = 2; i <= num; ++i)

{

while (num % i == 0)

{

stepen++;

num /= i;

}

if (stepen != 0) {

result \*= pow(i, stepen - 1) \* (i - 1);

}

stepen = 0;

}

return result;

}

// нахождение остатка от числа в степени

int Pow(int a, int x, int p)

{

int result = 1;

if (CheckPrime(p) == true)

{ // простое

x %= p - 1;

}

else { // не простое

x %= fi(p);

}

for (int i = 1; i <= x; ++i) {

result = (result \* a) % p;

}

return result;

}

// Алгоритм нахождения числа d(c^-1 mod m)

int inversia(int c, int m)

{

int q;

vector<int> one = { m, 0 };

vector<int> two = { c, 1 };

vector<int> three(2);

while (two[0] != 0)

{

q = one[0] / two[0];

three[0] = one[0] % two[0];

three[1] = one[1] - (q \* two[1]);

one[0] = two[0]; one[1] = two[1];

two[0] = three[0]; two[1] = three[1];

}

if (one[1] < 0)

{

one[1] += m;

}

return one[1];

}

int main() {

setlocale(LC\_ALL, "Rus");

srand(time(0));

string message;

cout << "Введите сообщение: ";

cin >> message;

int g, n;

cout << "Введите числа g и n: ";

cin >> g >> n;

int x = 100 + rand() % 100000; // абонент A выбирает случайное больше число и генерирует закрытый ключ

cout << "x = " << x << endl;

int k = Pow(g, x, n);

cout << "k = " << k << endl;

int y = 100 + rand() % 100000; // абонент B выбирает случайное больше число

while (VzaimProst(y, n - 1) != 1)

{

y = 100 + rand() % 100000;

}

cout << "y = " << y << endl;

int Y = Pow(g, y, n); // абонент B посылает абоненту A

cout << "Y = " << Y << endl;

int X = Pow(Y, x, n); // абонент А посылает абоненту B

cout << "X = " << X << endl;

int z = inversia(y, n - 1); // абонент B вычисляет

cout << "z = " << z << endl;

int ksh = Pow(X, z, n);

cout << "k' = " << ksh << endl;

if (k == ksh)

{

int key = k;

for (int i = 0; i < message.length(); ++i)

{

message[i] = message[i] ^ key;

}

cout << "Зашифрованное сообщение: " << message << endl;

for (int i = 0; i < message.length(); ++i)

{

message[i] = message[i] ^ key;

}

cout << "Расшифрованное сообщение: " << message;

}

else

{

cout << "Ключи не равны!";

}

return 0;

}

## Python:

import random

def VzaimProst(a, b): # нахождение НОД(a, b) для проверки, являются ли числа взаимопростыми

if (b == 0): return a

return VzaimProst(b, a % b)

def CheckPrime(p): # проверка, является ли число простым

if (p <= 1): return False

for i in range(2, int(p \*\* 0.5) + 1):

if (p % i == 0): return False

return True

def fi(num): # функция Эйлера

result = 1

stepen = 0

for i in range(2, num+1):

while (num % i == 0):

stepen += 1

num //= i

if (stepen != 0):

result \*= pow(i, stepen - 1) \* (i - 1)

stepen = 0

return result

def Pow(a, x, p): # нахождение остатка от числа в степени

result = 1

if (CheckPrime(p) == True): x %= p - 1 # простое

else: x %= fi(p) # не простое

for i in range(1, x+1):

result = (result \* a) % p

return result

def inversia(c, m): # алгоритм нахождения числа d(c^-1 mod m)

q = 0

one = [m, 0]

two = [c, 1]

three = [0, 0]

while (two[0] != 0):

q = one[0] // two[0]

three[0] = one[0] % two[0]

three[1] = one[1] - (q \* two[1])

one[0], one[1] = two[0], two[1]

two[0], two[1] = three[0], three[1]

if (one[1] < 0):

one[1] += m

return one[1]

message = input("Введите сообщение: ")

g = int(input("Введите число g: "))

n = int(input("Введите число n: "))

x = random.randrange(10000, 100000) # абонент A выбирает случайное больше число и генерирует закрытый ключ

print("x =", x)

k = Pow(g, x, n)

print("k =", k)

y = random.randrange(10000, 100000) # абонент B выбирает случайное больше число

while VzaimProst(y, n-1) != 1:

y = random.randrange(10000, 100000)

print("y =", y)

Y = Pow(g, y, n) # абонент B посылает абоненту A

print("Y =", Y)

X = Pow(Y, x, n) # абонент А посылает абоненту B

print("X =", X)

z = inversia(y, n-1) # абонент B вычисляет

print("z =", z)

ksh = Pow(X, z, n)

print("k' =", ksh)

if (k == ksh):

key = k

ArrEncrypted = []

ArrDecrypted = []

Encrypted = ""

Decrypted = ""

for i in range(len(message)): # Шифрование

ArrEncrypted.append(chr(ord(message[i]) ^ key))

for i in range(len(ArrEncrypted)):

Encrypted += ArrEncrypted[i]

print("Зашифрованное сообщение:", Encrypted)

for i in range(len(Encrypted)): # Дешифрование

ArrDecrypted.append(chr(ord(Encrypted[i]) ^ key))

for i in range(len(ArrDecrypted)):

Decrypted += ArrDecrypted[i]

print("Расшифрованное сообщение:", Decrypted)

else: print("Ключи не равны!")

# **Задание 5.** Найти последнюю цифру «трехэтажного числа».

## C++:

**#include <iostream>**

**#include <math.h>**

**using namespace std;**

**int LastNum(int chisla, int chislas[]) { // нахождение уникальной комбинации последних цифр в числе**

**int k = 0, n = 0; // k - условие выхода из цикла / n - счётчик**

**for (int i = 0; k == 0; i++) {**

**chislas[i] = pow(chisla, i+1);**

**chislas[i] = chislas[i] % 10;**

**if (chislas[i] == chislas[i-1] || chislas[i] == chislas[i-2] || chislas[i] == chislas[i-3] || chislas[i] == chislas[i-4]) {**

**k = k + 1;**

**} else {**

**n = n + 1;**

**}**

**}**

**return n;**

**}**

**int main(){**

**setlocale(LC\_ALL, "Rus");**

**int a, b, c;**

**cout << "Введите число, первую и вторую степень: ";**

**cin >> a >> b >> c;**

**a = a % 10;**

**int nums1[10];**

**int n1;**

**n1 = LastNum(a, nums1); // находим уникальные конечные цифры и их кол-во**

**int step1 = b % n1;**

**if (step1 == 0) {**

**b = nums1[n1-1]; // если остаток = 0, то конечная цифра - последняя цифра комбинации**

**} else {**

**b = nums1[step1-1]; // иначе (цифра комбинации) - 1**

**}**

**int nums2[10];**

**int n2;**

**n2 = LastNum(b, nums2);**

**int step2 = c % n2;**

**if (step2 == 0) {**

**c = nums2[n2-1];**

**} else {**

**c = nums2[step2-1];**

**}**

**cout << "Последняя цифра: " << c; // вывод последней цифры числа**

**return 0;**

**}**

## Python:

def LastNum(num, nums): # нахождение уникальной комбинации последних цифр в числе

k = 0 # условие выхода из цикла

n = 0 # счётчик

i = 0

while (k == 0):

nums[i] = pow(num, i+1)

nums[i] = nums[i] % 10

if nums[i] == nums[i-1] or nums[i] == nums[i-2] or nums[i] == nums[i-3] or nums[i] == nums[i-4]: k += 1

else: n += 1

i += 1

return n

a = int(input("Введите число: "))

b = int(input("Введите 1-ю степень: "))

c = int(input("Введите 2-ю степень: "))

a = a % 10

nums1 = [0] \* 10

n1 = LastNum(a, nums1) # находим уникальные конечные цифры и их кол-во

step1 = b % n1

if (step1 == 0): b = nums1[n1-1] # если остаток = 0, то конечная цифра - последняя цифра комбинации

else: b = nums1[step1-1] # иначе (цифра комбинации) - 1

nums2 = [0] \* 10

n2 = LastNum(b, nums2)

step2 = c % n2

if (step2 == 0): c = nums2[n2-1]

else: c = nums2[step2-1]

print("Последняя цифра:", c)

# **Задание 6. Написать сообщение на тему «Атака посередине».**

**Атака "Встреча посередине"** - это тип криптоаналитической атаки, обычно используемой против алгоритмов шифрования, использующих симметричные ключи. Кроме того, атака использует тот факт, что процессы шифрования и дешифрования являются обратными операциями друг друга. Таким образом, злоумышленник использует метод грубой силы, чтобы найти секретный ключ, который мы используем в процессе шифрования.

Атака работает путем разделения пространства ключей на две части. Во-первых, она шифрует открытый текст, используя все возможные ключи из половины пространства ключей. Кроме того, злоумышленник расшифровывает полученный зашифрованный текст, используя все возможные ключи из другой половины пространства ключей. Наконец, злоумышленник проверяет соответствие промежуточных значений, полученных на этапах шифрования и дешифрования.

Если найдено совпадение, ключ, используемый при шифровании и дешифровании, находится в середине ключевого пространства. Повторяя процесс с меньшими пробелами по обе стороны от соответствующего ключа, злоумышленник может в конечном итоге найти секретный ключ, используемый при шифровании. Атака "Встреча посередине" особенно эффективна против алгоритмов шифрования с относительно небольшим пространством ключей и уязвима для исчерпывающего поиска ключей.

**Порядок работы:**

Злоумышленник обычно выполняет пять шагов, чтобы реализовать или запустить атаку "встреча посередине":

Первым шагом является разделение пространства ключей. Злоумышленник делит пространство ключей на две части, каждая из которых содержит половину возможных ключей. Кроме того, разделение пространства ключей имеет решающее значение для успеха атаки. Вероятность успешной реализации MITM-атаки высока, если пространство ключей невелико.

Затем злоумышленник шифрует открытый текст, используя все возможные ключи из половины пространства ключей. Кроме того, он сохраняет полученные промежуточные значения в таблице.

Продвигаясь вперед, злоумышленник расшифровывает зашифрованный текст, используя все возможные ключи из другой половины пространства ключей, и сохраняет полученные промежуточные значения в отдельной таблице.

Следующим шагом является сопоставление промежуточных значений из двух таблиц. Злоумышленник сравнивает промежуточные значения на этапах шифрования и дешифрования, чтобы увидеть, есть ли совпадение. Если мы находим совпадение, соответствующий ключ находится в середине ключевого пространства.

Наконец, злоумышленник повторяет процесс с меньшими пробелами по обе стороны от соответствующего ключа, пока не будет найден используемый секретный ключ.

Важно отметить, что атака "встреча посередине" требует значительных вычислительных мощностей и памяти. Следовательно, это может быть невозможно для алгоритмов шифрования с большими пространствами ключей или другими строгими мерами безопасности.

MITM-атаки обычно применимы только к блочным шифрам и могут быть неэффективны против потоковых шифров. Таким образом, MITM-атаки могут быть эффективными в определенных ситуациях, но они не являются надежным методом взлома шифрования.

**Влияние:**

Влияние атаки "встреча посередине" может быть значительным. Давайте обсудим некоторые последствия.

Злоумышленник может получить секретный ключ для шифрования, а также расшифровать сообщения, запустив MITM-атаку. Это может позволить злоумышленнику расшифровывать другие сообщения, зашифрованные тем же ключом.

Более того, такая атака может привести к утечке конфиденциальной информации, что может иметь ряд негативных последствий, в зависимости от характера информации и от того, кто имеет к ней доступ.

Используя MITM-атаку, злоумышленник может скомпрометировать систему, используемую для шифрования и дешифрования сообщений. Это может позволить злоумышленнику получить доступ к сети, а также к системам, подключенным к сети. Более того, злоумышленник может установить бэкдоры для выполнения вредоносных действий.

Наконец, MITM-атака может нанести ущерб репутации системы или организации. Это может привести к потере доверия пользователей, заказчиков и партнеров.

**Методы предотвращения:**

Мы можем использовать несколько механизмов предотвращения, чтобы снизить риск атаки "встреча посередине" (MITM). Давайте обсудим некоторые основные методы.

Одним из ключевых недостатков MITM-атаки является то, что она требует больших вычислительных мощностей для поиска по большому количеству возможных ключей. При использовании более длинных ключей количество возможных ключей увеличивается экспоненциально, что затрудняет поиск методом перебора.

Кроме того, некоторые шифры более уязвимы для MITM-атак. Используя шифры, устойчивые к MITM-атакам, мы можем эффективно снизить риск атаки.

При атаке MITM злоумышленник ищет ключ, который расшифровывает зашифрованный текст в открытый. Следовательно, мы можем использовать случайные ключи, чтобы уменьшить вероятность того, что злоумышленник угадает правильный ключ.

Более того, протоколы обмена ключами, такие как Диффи-Хеллман, позволяют двум сторонам безопасно согласовать общий секретный ключ без его передачи. Это может затруднить злоумышленнику перехват ключа и запуск MITM-атаки.

Наконец, мы можем реализовать аутентификацию сообщения. Аутентификация сообщения гарантирует, что сообщение не было подделано во время передачи. Это может затруднить злоумышленнику изменение сообщения и запуск MITM-атаки.

# Задание 7. Написать сообщение на тему «Стандарты современной криптографии в РФ».

На сегодняшний день существует огромное множество различных алгоритмов шифрования. Надежность некоторых из них подтверждена исследованиями экспертов в области криптографии, другие не могут этим похвастать. А еще есть целая группа алгоритмов, принятых в качестве стандартов в тех или иных странах. Естественно, это говорит об их высоком качестве: надежности, скорости обработки данных, универсальности и т. п. Еще одним преимуществом этой группы является то, что по мере устаревания алгоритма в качестве национального стандарта принимается новая, более современная разработка. Так было, например, с американским стандартом шифрования. Первоначально им был алгоритм DES, а в конце прошлого века ему на смену пришла более современная разработка AES.

В нашей стране в качестве стандарта используется технология, описанная в ГОСТе 28147-89 «Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования». Как видно из названия, этот ГОСТ был принят в 1989 году и с тех пор не изменялся. Но мало кто знает, что этот алгоритм шифрования разработали в КГБ еще в конце 70-х годов. Но стоит ли доверять такому старому алгоритму? Не подведет ли он? И не лучше ли обратить свое внимание на более современные разработки? Что ж, давайте попробуем разобраться в этом вопросе.

Во-первых, нужно учитывать, что алгоритм, описанный в ГОСТе 28147-89, создавался с достаточно большим «запасом прочности». По этому параметру он на порядок превосходил американский DES, который сначала заменили на тройной DES (то есть одну и ту же информацию «прогоняли» три раза), а потом на AES. Таким образом, и на сегодняшний день криптостойкость российского стандарта вполне удовлетворяет всем современным требованиям. Вторая причина большого распространения ГОСТа 28147-89 - наше законодательство. Так, например, государственные организации и многие коммерческие структуры обязаны использовать для защиты данных сертифицированные средства защиты. Однако получение сертификата возможно только в том случае, если «в указанных криптосредствах реализованы криптографические алгоритмы, объявленные государственными или отраслевыми стандартами Российской Федерации». Таким образом, у многих компаний просто-напросто нет выбора. Однако, как мы уже говорили, ГОСТ 28147-89 удовлетворяет всем современным требованиям, а поэтому вряд ли кто-то будет расстраиваться из-за необходимости его использования.

Алгоритм, описанный в ГОСТе 28147-89, является типичным представителем класса симметричных. В его основе лежит так называемая сеть Фейстеля. Принцип ее работы заключается в следующем. Первоначальный блок данных (в ГОСТе 28147-89 это 64 бита) разбивается на несколько подблоков. Часть этих подблоков преобразовывается по определенным законам, а потом накладывается на необработанные подблоки. Затем они меняются местами и снова обрабатываются. И все эти действия повторяются определенное число раз.

На основе сети Фейстеля построен целый ряд различных алгоритмов. Вот только, несмотря на внешнее сходство, их криптостойкость и скорость работы очень сильно различаются. Все зависит от действий, которые выполняются над подблоками. Именно поэтому они и называются «основным криптографическим преобразованием». В ГОСТе 28147-89 используются относительно простые для реализации, быстрые для исполнения и устойчивые к взлому операции. Таким образом, этот алгоритм отличается в лучшую сторону от большинства своих «собратьев».

Алгоритм, описанный в ГОСТе 28147-89, может работать в трех различных режимах. Первый из них - простая замена. В этом случае блоки шифруются независимо друг от друга и от положения в массиве исходной информации. То есть две одинаковые последовательности размером в 64 бита и после обработки останутся одинаковыми. Нужно ли говорить, что это недопустимо для качественного и надежного алгоритма шифрования. Кроме того, в режиме простой замены очень желательно, чтобы объем исходного массива данных был кратен 64 битам. В противном случае наблюдается снижение криптоустойчивости алгоритма к некоторым видам атак.

На первый взгляд кажется, что режим простой замены абсолютно непригоден для практического использования. Однако на самом деле это не совсем так. Дело в том, что есть информация, которая практически всегда кратна 64 битам и не имеет в себе повторяющихся блоков этого размера. Речь идет, конечно же, о ключах шифрования. И действительно, задача их кодирования достаточно часто встает перед пользователями. И именно для ее решения подходит режим простой замены. Тем более что работает он быстрее других способов шифрования.

Вторым режимом работы алгоритма, описанного в ГОСТе 28147-89, является гаммирование. Этим словом обозначают «наложение (снятие) на открытые (зашифрованные) данные криптографической гаммы, то есть последовательности элементов данных, вырабатываемых с помощью некоторого криптографического алгоритма, для получения зашифрованных (открытых) данных». В рассматриваемом алгоритме для этой цели используется операция побитового сложения по модулю 2, поскольку она является обратной самой себе и к тому же наиболее просто реализуется аппаратно.

Таким образом, использование гаммирования решает обе проблемы простой замены. Во-первых, применение различных гамм для шифрования одинаковых последовательностей приводит к тому, что в закодированном тексте они отличаются друг от друга. Ну и, во-вторых, никто не может помешать «обрезать» гамму до нужного размера для шифрования блоков, отличных от стандартных. Таким образом, режим гаммирования избавлен от всех недостатков простой замены и может использоваться для надежного шифрования любой информации.

Третий и последний режим работы алгоритма, описанный в ГОСТе 28147-89, - это гаммирование с обратной связью. В принципе, он очень похож на предыдущий. Единственное отличие заключается в том, что очередной элемент гаммы вырабатывается с помощью определенных преобразований предыдущей последовательности информации. То есть фактически получается, что результат кодирования блока данных зависит от ключа шифрования и предыдущего уже зашифрованного блока. Таким образом, мы имеем связь всего закодированного текста. И если кто-то поменяет в нем какой-либо бит, то при расшифровке искаженным окажется не один, а два блока данных. В принципе, сильного влияния на криптостойкость обратная связь не оказывает. Однако она может послужить дополнительной защитой от некоторых типов атак.

В ГОСТе 28147-89 описана еще одна очень полезная функция алгоритма шифрования. Речь идет о возможности создания имитовставки. Что это? Имитовставка - контрольная комбинация, зависящая от открытых данных и секретной ключевой информации. Она нужна для того, чтобы обнаружить все случайные или преднамеренные изменения в зашифрованных данных. На первый взгляд кажется, что имитовставка не нужна. Однако на самом деле это не так. Конечно, обнаружить внесенные в текст изменения несложно. Тем более что полноценно заменить одни слова другими без знания ключа нельзя. Но если злоумышленник внесет изменения в файл, полученный путем шифрования бинарной, звуковой, графической или видеоинформации, то при декодировании искажения перейдут и на исходные данные. И заметить их зачастую просто-напросто нереально. Именно тогда имитовставка и оказывается незаменимой. Да и в случае с текстом ее использование позволяет точно узнать, вносились изменения в зашифрованный файл или нет.

Итак, как мы видим, алгоритм, принятый в качестве государственного стандарта РФ и описанный в ГОСТе 28147-89, сохраняет свою актуальность и сегодня. Он обеспечивает надежную защиту любых данных и не может быть взломан с помощью известных типов криптографических атак. Хотя, конечно, очень многое зависит от конкретной реализации. Часто бывает так, что на базе надежного алгоритма из-за ошибок при разработке создаются системы шифрования, не удовлетворяющие современным требованиям. А поэтому лучше всегда отдавать предпочтение хорошо зарекомендовавшим себя сертифицированным продуктам.

**Вывод:**

В ходе выполнения практической работы №2 я познакомился с различными алгоритмами, которые пригодятся в дальнейшем изучении криптографии. Научился пользоваться алгоритмом распределения ключей Хьюга. Познакомился с базовой информацией по теме “Криптография”.

**Ссылка на репозиторий:** https://github.com/Mixassss/Praktika2.git