1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «**Математические примитивы криптографии**»
2. по дисциплине «Основы информационной безопасности»
3. Выполнили
4. студенты гр. 5131001/40003 Веденеев А. С.
5. <*подпись*>  
     
    Поляков Д. Т.
   1. <*подпись*>
6. Преподаватель
7. ст. преподаватель Вагисаров В. Б.

<*подпись*>

1. Санкт-Петербург
2. 2025

# Цель работы

Приобретение расчетных навыков в модульной арифметике, используемой в криптографических алгоритмах и протоколах, ознакомление с математическим вычислениями, используемыми для сокрытия сообщений на примерах алгоритма шифрования RSA и ранцевой криптосистемы Меркля-Хеллмана.

# ход работы

## Определение параметра K.

**Номер учебной группы** (Nгр): 513

**Порядковый номер в списке группы** (Nсп): 6

**Порядковый номер в алфавите третьей буквы фамилии**: Ф3: 5

Формула: ((Nгр + Nсп) ^11 + Ф3) (mod 11)

Подставляем наши значения: ((513 + 6) ^ 11 + 5) mod 11 = 7

**K** = 7.

## Шифрование строки с использованием шифра Цезаря.

**Исходная строка**: ВеденеевАртёмСергеевич

**K** = 7 – сдвиг из первого пункта.

**Алфавит**: русский, 33 буквы.

Формула шифрования: C = (P + K) mod N, где:

* C – шифросимвол,
* P – позиция исходной буквы в алфавите (0–32),
* K = 7 – сдвиг,
* N = 33 – размер алфавита.

Пример расчета для первой буквы:

* “В” – 2-я буквы алфавита (позиция 1, если считать от 0).
* C = (1 + 7) mod 33 = 8.
* 8-я буква алфавита – ‘й’

Результат:

Зашифрованная строка: ЙмлмфммйЗчщёуШмчкммйпю

\**Приложение 2 – для английских букв*

## Вычисление A и НОД (A, B).

Исходные данные:

* **Номер группы** (Nгр​): 513
* **Порядковый номер в списке** (Nсп): 6
* **B**: 3042006 (дата рождения в формате ДДММГГГГ — 03.04.2006).

1. Вычисление A: A = (Nгр \* (8 + Nсп (mod 7))) ^ 2

Подставляем значения: A = (513 \* (8 + (6 mod 7))) ^ 2 = 51581124

Результат: A = 51581124

1. Поиск НОД (A, B)

B = 3042006 (03.04.2006)

Последовательность вычислений для НОД1:

**НОД (A, (B % 95) + 900)**

(B % 95) + 900 = **911 => GCD (51581124**, **911)**

51581124 ÷ 911 = 56620 (304)

911 ÷ 304 = 2 (303)

304 ÷ 303 = 1 (1)

303 ÷ 1 = 303 (0)

НОД (51581124, (3042006 % 95) + 900) = 1

Последовательность вычислений для НОД2:

**НОД (A, ((B + 50) % 97) + 700)**

((B + 50) % 97) + 700 = **739 => GCD (51581124**, **739)**

51581124 ÷ 739 = 69798 (402)

739 ÷ 402 = 1 (337)

402 ÷ 337 = 1 (65)

337 ÷ 65 = 5 (12)

65 ÷ 12 = 5 (5)

12 ÷ 5 = 2 (2)

5 ÷ 2 = 2 (1)

2 ÷ 1 = 2 (0)

НОД (51581124, 739) = 1

*\*Приложение 1*

## Проверка составного и простого числа

1. Выбор чисел.

Пусть:

N = 10002 (четное число, составное)

N1 = 193 (простое)

1. Метод Миллера для составного числа N = 10002

* Шаг 1: Разложение N - 1

N – 1 = 10002 – 1 = 10001

s = 0, d = 10001

2\*0 \* 10001 = 10001

* Шаг 2: Выбор случайного основания a

Пусть a = 2

* Шаг 3: Вычисление a^d (mod N)

Вычислим модуль: 2^(10001) = ((2^10000) \* 2) mod 10002

Но так как 10002 четное, 2^10000 делится на 10002, значит:

2^(10001) = = 0 mod 10002 => (2^(10001) / 10002) с остатком 0.

Это противоречит условиям теста Миллера (он не дал 1 или -1), следовательно, N = 10002 составное.

1. Метод Миллера для простого числа N1 = 193

Разложим N1 – 1:

193 – 1 = 192 = 2^6 \* 3, где: s = 6, d = 3

Шаг 1: Выбор основания

Пусть a = 2

Шаг 2: Вычисление a^d (mod N1)

2^3 mod 193 = 8

Шаг 3: Квадратирование

Теперь последовательно возводим в квадрат:

8^2 mod 193 = 64

64^2 mod 193 = 21

21^2 mod 193 = 121

121^2 mod 193 = 76

76^2 mod 193 = 192

192^2 mod 193 = 1

=> 192 = = -1 mod 193 (192 / 193 – с остатком -1)

Появление -1 означает, что тест Миллера не дал свидетельства составности, значит, 193 – простое.

1. Сравнение количества шагов

Для 10002 (составного) тест завершился за 1 шаг (нахождение 0 при возведении в степень)

Для 193 (простого) потребовалось 6 итераций возведения в квадрат, прежде чем появилась -1.

1. Временная сложность.

Метод Миллера имеет сложность O(k log^3 N), где k – число проверок (обычно 1-2 при составных числах, 5-10 при простых)

Для составного N = 10002 тест сработал за O(log N)

Для простого N1 = 193 потребовалось O(log^3 N).

## Генерация открытого и закрытого ключей RSA с использованием алгоритма Евклида

1. Выбор простых чисел p и q.

Пусть p=587 и q=743. Эти числа простые, и их НОД равен 1.

1. Вычисление n.

n = p \*q = 587 \* 743 = 436141

1. Вычисление функции Эйлера φ(n).

φ (n) = (p - 1) \* (q - 1) = 586 \* 742 = 434812

1. Выбор числа e.

Пусть e = 29, так как: gcd(29, 586) = 1, gcd(29, 742) = 1

(Это означает, что 29 не делится на 586 и 742)

1. Вычисление d.

Нам нужно найти такое d, что: 29 \* d ≡ 1 (mod 434812)

Сначала находим gcd(29, 434812) с вычислением коэффициентов:

Расширенный алгоритм Евклида (нахождение коэффициентов x и y в уравнении ax + by = gcd(a, b))

1. Разложение:

434812 : 29 = 14993, 434812 – 14993 \* 29 = 15

29 : 15 = 1, 29 – 1 \* 15 = 14

15 : 14 = 1, 15 – 1 \* 14 = 1

14 : 1 = 14, 14 – 14 \* 1 = 0

Следовательно: gcd(434812, 29) = 1

1. Обратный ход (выражаем НОД в виде линейной комбинации)

* 1 = 15 – 1 \* 14
* 14 = 29 – 1 \* 15, подставляем:
  + 1 = 15 – 1 \* (29 – 1 \* 15)
  + 1 = 2 \* 15 – 1 \* 29
* 15 = 434812 – 14993 \* 29, подставляем:
  + 1 = (434812 - 14993 \* 29) – 1 \* 29
  + 1 = 434812 – 14994 \* 29

— Значит: d = −14994 ≡ 404825 (mod 434812)

Так как d должно быть положительным, прибавляем модуль:

d = 434812 – 14994 = 404825

Итого получаем:

Открытый ключ (e, n) = (29, 436141)

Закрытый ключ (404825, 436141)

## Реализация RSA-шифрования в ASCII

Исходные данные:

p = 587, q = 743, e = 29, text = “RSA test”, n = 436141, d = 404825

ASCII Таблица:

|  |  |
| --- | --- |
| R | 82 |
| S | 83 |
| A | 65 |
| (space) | 32 |
| t | 116 |
| e | 101 |
| s | 115 |
| t | 116 |

Шифрование: (x^e)(mod n)

R = (82^29)(mod 27685) = 333773

S = (83^29)(mod 27685) = 295059

A = (65^29)(mod 27685) = 22248

(space) = (32^29)(mod 27685) = 163678

t = (116^29)(mod 27685) = 205879

e = (101^29)(mod 27685) = 310128

s = (115^29)(mod 27685) = 120674

Расшифрование: (y^d)(mod n)

R = (333773^404825)(mod 436141) = 82

S = (295059^404825)(mod 436141) = 83

A = (22248^404825)(mod 436141) = 65

(space) = (163678^404825)(mod 436141) = 32

t = (205879^404825)(mod 436141) = 116

e = (310128^404825)(mod 436141) = 101

s = (120674^404825)(mod 436141) = 115

*\*Приложение 3*

## Цифровая подпись и проверка с помощью RSA

Исходные данные:

p = 587, q = 743, e = 29, text = “RSA test”, n = 436141, d = 404825

Подпись текста: s = (x^d)(mod n)

R = (82^404825)(mod 436141) = 15815

S = (83^404825)(mod 436141) = 368221

A = (65^404825)(mod 436141) = 309125

(space) = (32^404825)(mod 436141) = 216459

t = (116^404825)(mod 436141) = 143509

e = (101^404825)(mod 436141) = 433244

s = (115^404825)(mod 436141) = 75414

Проверка подписи: (s^e)(mod n)

R = (15815^29)(mod 436141) = 82

S = (368221^29)(mod 436141) = 83

A = (309125^29)(mod 436141) = 65

(space) = (216459^29)(mod 436141) = 32

t = (143509^29)(mod 436141) = 116

e = (433244^29)(mod 436141) = 101

s = (75414^29)(mod 436141) = 115

Итог: цифровая подпись смоделирована верно.

## Моделирование установления сеансового ключа по схеме Диффи-Хеллмана.

Начальные данные: a = 5, n = 436141

1. Пользователь A:

X = 39 (x < f(n))

A = a^x (mod n) = 5^39 (mod 436141) = 326937

1. Пользователь B:

y = 298 (y < f(n))

B = a^y (mod n) = 5^298 (mod 436141) = 132472

1. Сеансовые ключи:

* A: B^x mod n = 260987
* B: A^y mod n = 260987
* B^x = A^y = a^xy (mod n) = 260987

Сеансовый ключ вычислен правильно.

## Реализация утилиты шифрования и дешифрования с помощью алгоритма Меркля-Хеллмана.

\**Приложение 4*

## Ответы на контрольные вопросы

1. Что такое вычет? На чем основан алгоритм шифрования Цезаря?

Вычет – это число, которое является остатком от деления одного числа на другое. Записывается как: a ≡ r (mod b) и означает, что число a при делении на b дает остаток r.

Алгоритм шифрования Цезаря - то простой метод подстановки, в котором каждая буква исходного текста заменяется буквой, находящейся на k позиций вперед в алфавите. Например, при k=3:

A -> D, B -> E, C -> F.

1. Каковы особенности чисел Кармайкла?

Число Кармайкла – это составное число N, которое удовлетворяет:

a^(N−1) ≡ 1 (mod N), для всех a, GCD (a, N) = 1.

Это делает их "псевдопростыми" по малой теореме Ферма, что затрудняет проверку их составности. Пример самого малого числа Кармайкла: 561.

1. Основные свойства мультипликативной группы кольца вычетов по модулю pqpq:
   1. Коммутативность: Умножение элементов в мультипликативной группе кольца вычетов по модулю pq является коммутативным, то есть для любых элементов a, b из группы выполняется свойство ab = ba.
   2. Ассоциативность: Умножение элементов в группе ассоциативно, то есть для любых элементов a, b, c из группы выполняется свойство a(bc) = (ab)c.
   3. Существование единицы: В мультипликативной группе кольца вычетов по модулю pq существует единичный элемент, обозначаемый как 1, который удовлетворяет условию 1a = a1 = a для любого элемента a.
   4. Существование обратного элемента: Для каждого элемента a из мультипликативной группы существует обратный элемент a^(-1), такой что a a^(-1) = a^(-1) a = 1.
   5. Замкнутость: Результат умножения двух элементов из мультипликативной группы также является элементом этой же группы.
   6. Порядок группы: Порядок мультипликативной группы кольца вычетов по модулю pq равен φ(pq) = (p-1)(q-1), где φ - функция Эйлера. Порядок мультипликативной группы кольца вычетов по модулю n обозначает количество элементов в этой группе
2. Почему порядок группы (Zn​/Z)∗ должен иметь большой простой делитель?
   1. Безопасность RSA: В криптографическом алгоритме RSA безопасность основана на сложности задачи факторизации больших составных чисел n, состоящих из двух больших простых множителей p и q. Если порядок группы мультипликативной группы (Z/nZ) имеет большой простой делитель, то это осложняет атаки, основанные на разложении n на простые множители.Большой простой делитель затрудняет разложение числа n на p и q*.*
   2. Усиление модулярной арифметики: Большой простой делитель порядка группы (Z/nZ) способствует увеличению сложности вычислений в модулярной арифметике, что делает криптографические методы более стойкими к различным типам атак.
   3. Уменьшение вероятности коллизий и атак: Большой простой делитель порядка группы (Z/nZ) повышает вероятность уникальности получаемых ключей шифрования и уменьшает возможность успешных атак, основанных на алгоритмах разделения и анализа группы элементов.
3. Алгоритм расчета кодов символов при декодировании шифрограмм по алгоритму Меркля-Хеллмана:
4. Генерация ключей:

* Закрытый ключ создается как сверхвозрастающая последовательность ww длиной 8.
* Вычисляется модуль nn, который больше суммы всех элементов ww.
* Выбирается число ee, взаимно простое с nn.
* Открытый ключ формируется по формуле:

Βi = (wi \* e) mod  n

1. Шифрование:

* Сообщение представляется в виде двоичной строки длиной 8.
* Для каждого i, если бит сообщения message[i] = 1, то суммируется соответствующее значение из открытого ключа:

c = ∑ message[i] \* β[i]

* Полученное зашифрованное число cc передается для расшифрования.

1. Дешифрование:

* Вычисляется обратный элемент d, такой что:

(e \* d) mod n = 1

* Дешифрованное значение s получается, как:

s = (c \* d) mod n

* Далее, используя жадный алгоритм, из s поэтапно вычитаются элементы ww, начиная с наибольшего.
* Если s больше или равен w[i], то устанавливается decrypted[i] = 1, иначе decrypted[i] = 0
* В результате получается исходное двоичное сообщение.

Этот алгоритм основан на том, что разложение числа ss по сверхвозрастающей последовательности однозначно определяет исходное сообщение.

# Выводы

В ходе лабораторной работы были изучены основы криптографических методов, включая принципы модульной арифметики и ее применение в криптографических алгоритмах. Были рассмотрены простейшие способы шифрования, включая шифр Цезаря, а также реализованы и протестированы более сложные криптосистемы, такие как алгоритм RSA и ранцевая криптосистема Меркля-Хеллмана. Практическая реализация позволила закрепить знания о шифровании и расшифровании сообщений, генерации ключей и свойствах криптографических систем.

# Приложения

1. Шифрование строки с помощью шифра Цезаря (английские буквы)

#include <stdio.h>

#include <string.h>

#include <locale.h>

void ceaserCipher(char \*text, int shift)

{

    for (int i = 0; text[i] != '\0'; i++) {

        char ch = text[i];

        if (ch >= 'a' && ch <= 'z') {

            text[i] = 'a' + (ch - 'a' + shift) % 26;

        } else if (ch >= 'A' && ch <= 'Z') {

            text[i] = 'A' + (ch - 'A' + shift) % 26;

        }

    }

}

int main(void)

{

    setlocale(LC\_ALL, "ru\_RU.UTF-8");

    char text[100];

    int shift = 0;

    printf("введите строку: ");

    fgets(text, sizeof(text) / sizeof(text[0]), stdin);

    text[strcspn(text, "\n")] = 0; // удаляем символ новой строки

    printf("введите сдвиг: ");

    scanf("%d", &shift);

    ceaserCipher(text, shift);

    printf("результат шифрования: %s\n", text);

    return 0;

}

1. Утилита для вычисления НОД, а также нахождения коэффициентов.

#include <stdio.h>

// расширенный алгоритм Евклида

// ax + by = НОД(a, b)

long long extended\_gcd(long long a, long long b, long long \*x, long long \*y) {

    // базовый случай: если b = 0, то НОД = a

    // x = 1, y = 0

    if (b == 0) {

        \*x = 1;

        \*y = 0;

        return a;

    }

    long long x1, y1;

    // рекурсивно вычисляем НОД(b, a % b) и коэффициенты x1, y1

    long long gcd = extended\_gcd(b, a % b, &x1, &y1);

    // x = y1 (коэффициент от b на следующем шаге становится коэффициентом a)

    \*x = y1;

    // y = x1 - (a / b) \* y1 (коэффициент b корректируется с учетом деления)

    \*y = x1 - (a / b) \* y1;

    // возвращаем НОД, который остается неизменным на всех шагах

    return gcd;

}

// алгоритм Евклида (поиск НОД)

long long gcd(long long a, long long b)

{

    // большее число записываем на первое место

    if (a < b)

    {

        int temp = a;

        a = b;

        b = temp;

    }

    while (b != 0)

    {

        int temp = b;

        b = a % b;

        a = temp;

    }

    return a;

}

int main(void)

{

    long long a = 0, b = 0, x = 0, y = 0;

    printf("enter a, b: ");

    scanf("%lld %lld", &a, &b);

    printf("[standart] GCD(%lld, %lld) = %d\n", a, b, gcd(a, b));

    printf("[extended] GCD(%lld, %lld) = %d\n", a, b, extended\_gcd(a, b, &x, &y));

    // если y отрицательный, то d = y + a

    printf("coefficients (ax + by = GCD(a, b)): %lld\*%lld + %lld\*%lld = %lld", a, x, b, y, gcd(a, b));

    return 0;

}

1. Шифрование с помощью метода RSA

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

// функция возведения в степень по модулю

long long simple\_pow(long long base, long long exp, long long mod) {

    long long result = 1;

    for (long long i = 0; i < exp; i++) {

        result = (result \* base) % mod;

    }

    return result;

}

// функция для шифрования строки

void encrypt(char \*text, long long e, long long n) {

    printf("encrypted: ");

    for (int i = 0; i < strlen(text); i++) {

        long long encrypted\_char = simple\_pow((long long)text[i], e, n);

        printf("%lld ", encrypted\_char);

    }

    printf("\n");

}

// функция для расшифрования массива чисел

void decrypt(long long \*encrypted\_text, int length, long long d, long long n) {

    printf("decrypted: ");

    for (int i = 0; i < length; i++) {

        char decrypted\_char = (char)simple\_pow(encrypted\_text[i], d, n);

        printf("%c", decrypted\_char);

    }

    printf("\n");

}

int main() {

    long long p = 587, q = 743;

    long long n = p \* q; // 436141

    long long e = 29;

    long long d = 404825;

    char text[] = "RSA test";

    printf("original text: %s\n", text);

    // шифрование

    long long encrypted\_text[strlen(text)];

    printf("encrypted: ");

    for (int i = 0; i < strlen(text); i++) {

        encrypted\_text[i] = simple\_pow((long long)text[i], e, n);

        printf("%lld ", encrypted\_text[i]);

    }

    printf("\n");

    // расшифрование

    decrypt(encrypted\_text, strlen(text), d, n);

    return 0;

}

1. Утилита, реализующая шифрование методом Меркля-Хеллмана.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define KEY\_SIZE 8 // размер ключа

// функция для вычисления наибольшего общего делителя (НОД)

int gcd(int a, int b) {

    while (b != 0) {

        int temp = b;

        b = a % b;

        a = temp;

    }

    return a;

}

// функция для вычисления обратного элемента по модулю (расширенный алгоритм Евклида)

int mod\_inverse(int a, int m) {

    for (int x = 1; x < m; x++) {

        if ((a \* x) % m == 1)

            return x;

    }

    return -1; // eсли обратного элемента не существует

}

// генерация закрытого ключа (сверхвозрастающая последовательность, модуль и множитель)

void generate\_private\_key(int w[KEY\_SIZE], int \*n, int \*e) {

    srand(time(NULL)); // инициализация генератора случайных чисел

    int sum = 0;

    printf("Генерация закрытого ключа по формуле:\n");

    printf("w[i] = sum(w[0]...w[i-1]) + случайное\_значение\n");

    for (int i = 0; i < KEY\_SIZE; i++) {

        w[i] = sum + (rand() % 10 + 1); // создаём сверхвозрастающую последовательность

        sum += w[i];

        printf("w[%d] = %d\n", i, w[i]);

    }

    \*n = sum + (rand() % 10 + 1); // Выбираем модуль n

    printf("n = sum(w) + случайное\_значение = %d\n", \*n);

    do {

        \*e = rand() % (\*n - 1) + 1; // выбираем экспоненту e

    } while (gcd(\*e, \*n) != 1); // проверяем, чтобы e и n были взаимно простыми

    printf("e = %d (НОД с n = %d)\n", \*e, gcd(\*e, \*n));

}

// генерация открытого ключа (преобразование закрытого ключа)

void generate\_public\_key(int w[KEY\_SIZE], int n, int e, int beta[KEY\_SIZE]) {

    printf("Генерация открытого ключа по формуле:\n");

    printf("beta[i] = (w[i] \* e) mod n\n");

    for (int i = 0; i < KEY\_SIZE; i++) {

        beta[i] = (w[i] \* e) % n; // преобразуем элементы закрытого ключа

        printf("beta[%d] = %d\n", i, beta[i]);

    }

}

// шифрование сообщения

int encrypt(int message[KEY\_SIZE], int beta[KEY\_SIZE]) {

    int c = 0;

    printf("Шифрование по формуле:\n");

    printf("c = sum(message[i] \* beta[i])\n");

    for (int i = 0; i < KEY\_SIZE; i++) {

        printf("message[%d] = %d, beta[%d] = %d, произведение = %d\n", i, message[i], i, beta[i], message[i] \* beta[i]);

        c += message[i] \* beta[i];

    }

    return c;

}

// дешифрование сообщения

void decrypt(int c, int w[KEY\_SIZE], int n, int e) {

    int d = mod\_inverse(e, n); // вычисляем d — обратный элемент к e по модулю n

    printf("дешифрование по формуле:\n");

    printf("s = (c \* d) mod n\n");

    int s = (c \* d) % n;

    printf("s = %d\n", s);

    int decrypted[KEY\_SIZE]; // массив для расшифрованного сообщения

    printf("поиск исходного сообщения, вычитая w[i] из s\n");

    for (int i = KEY\_SIZE - 1; i >= 0; i--) {

        if (s >= w[i]) {

            decrypted[i] = 1;

            s -= w[i];

        } else {

            decrypted[i] = 0;

        }

        printf("s = %d, w[%d] = %d, decrypted[%d] = %d\n", s, i, w[i], i, decrypted[i]);

    }

    // вывод расшифрованного сообщения

    printf("расшифрованное сообщение: ");

    for (int i = 0; i < KEY\_SIZE; i++) {

        printf("%d", decrypted[i]);

    }

    printf("\n");

}

int main(void) {

    int w[KEY\_SIZE], n, e;

    generate\_private\_key(w, &n, &e);

    int beta[KEY\_SIZE];

    generate\_public\_key(w, n, e, beta);

    int message[KEY\_SIZE] = {1, 0, 1, 1, 0, 0, 1, 0}; // исходное сообщение

    printf("исходное сообщение: ");

    for (int i = 0; i < KEY\_SIZE; i++) {

        printf("%d", message[i]);

    }

    printf("\n");

    int c = encrypt(message, beta);

    printf("зашифрованное сообщение: %d\n", c);

    decrypt(c, w, n, e);

    return 0;

}