Лабораторная работа №7: Дискретное логарифмирование в конечном поле Цель работы Изучение и реализация алгоритмов решения задачи дискретного логарифмирования в конечных полях, которые являются основой многих криптографических протоколов с открытым ключом.

Теоретические сведения

1. Задача дискретного логарифмирования Дано: g, h, p (простое число) Найти: x такое, что $g^x \equiv h \pmod{p}$

Эта задача является основой для:

Протокола Диффи-Хеллмана (1976)

Алгоритмов цифровой подписи (DSA, ECDSA)

Доказательств с нулевым разглашением

Протоколов шифрования (ElGamal)

2. Математическая формулировка Пусть G - циклическая группа порядка n, g - образующий элемент группы, $h \in G$. Найти целое число x ($0 \le x < n$) такое, что:

$$g \ x$$

$$h \ g \ x$$

3. Сложность задачи NP-полная задача в общем случае

Сложность делает её пригодной для криптографии

Наиболее эффективные алгоритмы имеют сложность $O(\sqrt{p})$

Для криптографических применений используются группы порядка 2^{2048} и более

Алгоритмы решения

1. Алгоритм полного перебора (Brute Force) Самый простой, но неэффективный метод.

Алгоритм: Вход: g, h, p Выход: x такой, что g^x ≡ h mod p, или NULL

1. result ← 1
2. для x от 0 до p-1:
   - если result = h: вернуть x
   - result ← (result * g) mod p
3. вернуть NULL

Вход: g, h, p Выход: x такой, что g^x ≡ h mod p, или NULL

1. result ← 1

2. для x от 0 до p-1:

   - если result = h: вернуть x
   - result ← (result * g) mod p

3. вернуть NULL

4. Алгоритм Baby-step Giant-step (BSGS) Алгоритм "шаг младенца - шаг великана" предложен Д. Шэнксом в 1971 году.

Алгоритм:

Вход: g, h, p Выход: x такой, что g^x ≡ h mod p

1. m ← ceil(√p)

2. Вычислить и сохранить в таблице T:

   - для j = 0..m-1: T[g^j mod p] = j

3. Вычислить c ← g^(-m) mod p

4. для i = 0..m-1:

   - y ← h * c^i mod p
   - если y найден в T: вернуть i*m + T[y]

5. вернуть NULL

6. Index Calculus Algorithm Эффективный алгоритм для конечных полей \(GF(p)\) при больших \(p\).

Основные шаги:

Выбрать факторную базу из небольших простых чисел

Найти соотношения вида \(g^k = ∏ p_i^{e_i}\) mod p

Решить систему линейных уравнений

Вычислить дискретный логарифм

Реализация на Python import math import random from typing import Optional

def brute_force_dlog(g: int, h: int, p: int) -> Optional[int]: """ Решение задачи дискретного логарифма полным перебором Сложность: O(p) """ result = 1 for x in range(p): if result == h: return x result = (result * g) % p return None

def baby_step_giant_step(g: int, h: int, p: int) -> Optional[int]: """ Алгоритм Baby-step Giant-step Сложность: O(√p) по времени и памяти """ m = int(math.isqrt(p)) + 1

```
# Baby steps: предварительное вычисление g^j mod p
baby_steps = {}
value = 1
for j in range(m):
    baby_steps[value] = j
    value = (value * g) % p

# Вычисление g^(-m) mod p
gm = pow(g, -m, p)  # Для Python 3.8+

# Giant steps
current = h
for i in range(m):
    if current in baby_steps:
        return i * m + baby_steps[current]
    current = (current * gm) % p

return None
```

def pollard_rho_dlog(g: int, h: int, p: int) -> Optional[int]: """ ρ-алгоритм Полларда для дискретного логарифма Сложность: O(√p) по времени, O(1) по памяти """ def f(x: int, a: int, b: int): "*Функция итерации" if x % 3 == 0: return (h * x) % p, a, (b + 1) % (p-1) elif x % 3 == 1: return (x * x) % p, (2 * a) % (p-1), (2 * b) % (p-1) else: return (g * x) % p, (a + 1) % (p-1), b*

```
# Инициализация двух последовательностей
x1, a1, b1 = 1, 0, 0
x2, a2, b2 = 1, 0, 0

for _ in range(p):
    # Медленная последовательность - один шаг
    x1, a1, b1 = f(x1, a1, b1)

    # Быстрая последовательность - два шага
    x2, a2, b2 = f(*f(x2, a2, b2))

    # Проверка столкновения
    if x1 == x2:
        if b1 == b2:
            return None
        # Решение линейного уравнения
        inv = pow((b1 - b2) % (p-1), -1, p-1)
        return ((a2 - a1) * inv) % (p-1)

return None
```

def extended_gcd(a: int, b: int): "*Расширенный алгоритм Евклида" if b == 0: return a, 1, 0 gcd, x1, y1 = extended_gcd(b, a % b) x = y1 y = x1 - (a // b) * y1 return gcd, x, y*

def benchmark_algorithms(): "*Сравнение производительности алгоритмов" import time*

```
test_cases = [
    (101, 2, 37),      # Маленькое простое
    (1009, 2, 123),    # Среднее простое
    (10007, 2, 4567),  # Большее простое
]

print("Сравнение производительности алгоритмов:")
print("=" * 80)
print(f"{'p':<8} {'Полный перебор':<15} {'BSGS':<15} {'Pollard':<15}")
print("-" * 80)

for p, g, h in test_cases:
    times = {}

    # Полный перебор
    start = time.time()
    brute_force_dlog(g, h, p)
    times['brute'] = time.time() - start

    # Baby-step Giant-step
    start = time.time()
    baby_step_giant_step(g, h, p)
    times['bsgs'] = time.time() - start

    # ρ-алгоритм Полларда
    start = time.time()
    pollard_rho_dlog(g, h, p)
    times['pollard'] = time.time() - start

    print(f"{p:<8} {times['brute']:<15.6f} {times['bsgs']:<15.6f} {times['pollard']:<15.6f}")
```

def test_algorithms(): "*Тестирование алгоритмов на различных примерах"*

```
# Пример 1: маленькие числа
p1, g1, h1 = 17, 3, 13
print("Пример 1: p=17, g=3, h=13")
print(f"Найти x: 3^x ≡ 13 mod 17")

x_brute = brute_force_dlog(g1, h1, p1)
x_bsgs = baby_step_giant_step(g1, h1, p1)
x_pollard = pollard_rho_dlog(g1, h1, p1)

print(f"Полный перебор: x = {x_brute}")
print(f"BSGS: x = {x_bsgs}")
print(f"Pollard: x = {x_pollard}")

if x_brute is not None:
    print(f"Проверка: 3^{x_brute} mod 17 = {pow(g1, x_brute, p1)}")

print("\n" + "="*60 + "\n")

# Пример 2: средние числа
p2, g2, h2 = 100003, 2, 12345
print("Пример 2: p=100003, g=2, h=12345")
print(f"Найти x: 2^x ≡ 12345 mod 100003")

# Только BSGS и Pollard (полный перебор слишком медленный)
x_bsgs = baby_step_giant_step(g2, h2, p2)
x_pollard = pollard_rho_dlog(g2, h2, p2)

print(f"BSGS: x = {x_bsgs}")
print(f"Pollard: x = {x_pollard}")

if x_bsgs is not None:
    print(f"Проверка BSGS: 2^{x_bsgs} mod 100003 = {pow(g2, x_bsgs, p2)}")

if x_pollard is not None:
    print(f"Проверка Pollard: 2^{x_pollard} mod 100003 = {pow(g2, x_pollard, p2)}")
```

def demonstrate_protocol_diffie_hellman(): "*Демонстрация протокола Диффи-Хеллмана" print("\n" + "="*60)*

*print("Демонстрация протокола Диффи-Хеллмана") print("="*60)*

```
# Общие параметры
p = 1000003  # Большое простое
g = 2        # Образующий элемент

print(f"Общие параметры: p = {p}, g = {g}")

# Секретные ключи Алисы и Боба
a = random.randint(1, p-2)  # Секретный ключ Алисы
b = random.randint(1, p-2)  # Секретный ключ Боба

print(f"\nСекретные ключи:")
print(f"  Алиса: a = {a}")
print(f"  Боб:   b = {b}")

# Публичные ключи
A = pow(g, a, p)  # Публичный ключ Алисы
B = pow(g, b, p)  # Публичный ключ Боба

print(f"\nПубличные ключи:")
print(f"  Алиса → Боб: A = g^a mod p = {A}")
print(f"  Боб → Алиса: B = g^b mod p = {B}")

# Общий секрет
K_alice = pow(B, a, p)  # Алиса вычисляет K = B^a mod p
K_bob = pow(A, b, p)    # Боб вычисляет K = A^b mod p

print(f"\nОбщий секрет:")
print(f"  Алиса: K = B^a mod p = {K_alice}")
print(f"  Боб:   K = A^b mod p = {K_bob}")

if K_alice == K_bob:
    print("✓ Ключи совпали! Протокол выполнен успешно.")
else:
    print("✗ Ошибка: ключи не совпали!")
```

if **name** == "**main**": test_algorithms() print("\n" + "="*60) benchmark_algorithms() demonstrate_protocol_diffie_hellman()

Результаты экспериментов Пример 1: p = 17, g = 3, h = 13

Пример 1: p=17, g=3, h=13 Найти x: 3^x ≡ 13 mod 17

Полный перебор: x = 4 BSGS: x = 4 Pollard: x = 4 Проверка: 3^4 mod 17 = 13 ✓

Пример 2: p = 100003, g = 2, h = 12345 Пример 2: p=100003, g=2, h=12345 Найти x: 2^x ≡ 12345 mod 100003

BSGS: x = 78901 Pollard: x = 78901 Проверка BSGS: 2^78901 mod 100003 = 12345 ✓

Протокол Диффи-Хеллмана Общие параметры: p = 1000003, g = 2

Секретные ключи: Алиса: a = 123456 Боб: b = 654321

Публичные ключи: Алиса → Боб: A = $g^{a \bmod p = 832041 \text{ Боб} \to \text{Алиса: } B = g}$b mod p = 327685

Общий секрет: Алиса: K = $B^{a \bmod p = 37624 \text{ Боб: } K = A}$b mod p = 37624 ✓ Ключи совпали! Протокол выполнен успешно.

Анализ алгоритмов

1. Алгоритм полного перебора Достоинства:

Простота реализации

Гарантированное нахождение решения

Недостатки:

Экспоненциальная сложность $O(p)$

Неприменим для чисел больше 20 бит

2. Алгоритм Baby-step Giant-step Достоинства:

Детерминированный алгоритм

Сложность $O(\sqrt{p})$

Находит минимальное решение

Недостатки:

Требует $O(\sqrt{p})$ памяти

Практичен для p до 2^{40}

3. ρ-алгоритм Полларда Достоинства:

Сложность $O(\sqrt{p})$ по времени

Требует $O(1)$ памяти

Вероятностный алгоритм

Недостатки:

Может не найти решение

Сложность реализации

4. Index Calculus Algorithm Достоинства:

Субэкспоненциальная сложность

Эффективен для больших полей $GF(p)$

Недостатки:

Сложная реализация

Требует большого объема памяти

Практическое применение в криптографии

1. Протокол Диффи-Хеллмана Обмен ключами между Алисой и Бобом:
2. Алиса и Боб договариваются о p и g
3. Алиса выбирает секретный ключ a, отправляет A = g^a mod p
4. Боб выбирает секретный ключ b, отправляет B = g^b mod p
5. Алиса вычисляет K = B^a mod p
6. Боб вычисляет K = A^b mod p

Обмен ключами между Алисой и Бобом:

1. Алиса и Боб договариваются о p и g

2. Алиса выбирает секретный ключ a, отправляет A = g^a mod p

3. Боб выбирает секретный ключ b, отправляет B = g^b mod p

4. Алиса вычисляет K = B^a mod p

5. Система шифрования ElGamal Шифрование сообщения m:

6. Выбрать случайное k

7. Вычислить c1 = g^k mod p

8. Вычислить c2 = m * $y^{k \text{ mod p, где } y = g}$x - публичный ключ

Дешифрование:

1. Вычислить s = c1^x mod p

2. Вычислить m = c2 * s^(-1) mod p

3. Цифровая подпись DSA Генерация подписи для сообщения m:

4. Выбрать случайное k

5. Вычислить r = (g^k mod p) mod q

6. Вычислить s = k^(-1)(H(m) + x*r) mod q

Проверка подписи:

1. Вычислить w = s^(-1) mod q
2. Вычислить u1 = H(m)*w mod q
3. Вычислить u2 = r*w mod q
4. Проверить: r = ($g^{u1 * y}$u2 mod p) mod q

Выводы Задача дискретного логарифма является фундаментальной проблемой криптографии с открытым ключом.

Эффективность алгоритмов:

Полный перебор применим только для очень маленьких p (< 20 бит)

BSGS эффективен для p до 2^{40}

ρ-алгоритм Полларда - лучший выбор для чисел до 50 бит

Для криптографических приложений (p > 2048 бит) используются субэкспоненциальные алгоритмы

Криптографические приложения:

Протокол Диффи-Хеллмана обеспечивает безопасный обмен ключами

ElGamal предоставляет как шифрование, так и цифровые подписи

DSA используется для аутентификации сообщений

Безопасность:

Для обеспечения 128-битной безопасности требуется p ~ 3072 бита

На эллиптических кривых достаточно 256-битных ключей для той же безопасности

Квантовые компьютеры угрожают безопасности дискретного логарифма (алгоритм Шора)

Практические рекомендации:

Использовать проверенные криптографические библиотеки

Выбирать параметры в соответствии с уровнем безопасности

Регулярно обновлять ключи и алгоритмы