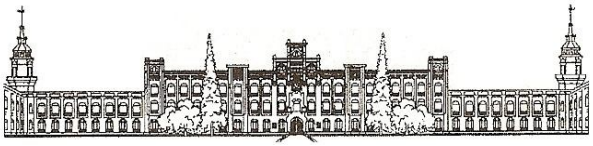

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

КРИПТОГРАФІЯ
КОМП'ЮТЕРНИЙ ПРАКТИКУМ №4

Вивчення криптосистеми RSA та алгоритму електронного підпису; ознайомлення з методами генерації параметрів для асиметричних криптосистем

Виконав студент групи: ФБ-24

ПІБ: Мартинюк Іван Олексійович

Київ 2024

Мета та основні завдання роботи

Ознайомлення з тестами перевірки чисел на простоту і методами генерації ключів для асиметричної криптосистеми типу RSA; практичне ознайомлення з системою захисту інформації на основі криптосхеми RSA, організація з використанням цієї системи засекреченого зв'язку й електронного підпису, вивчення протоколу розсилання ключів.

Хід роботи

1. Написати функцію пошуку випадкового простого числа з заданого інтервалу або заданої довжини, використовуючи датчик випадкових чисел та тести перевірки на простоту. В якості датчика випадкових чисел використовуйте вбудований генератор псевдовипадкових чисел вашої мови програмування. В якості тесту перевірки на простоту рекомендовано використовувати тест Міллера-Рабіна із попередніми пробними діленнями. Тести необхідно реалізовувати власноруч, використання готових реалізацій тестів не дозволяється.

Тест Міллера-Рабіна:

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def trial_division(n):
    if n < 2:
        return False
    for i in range(2, 100):
        if n % i == 0:
            return False
    return True

def miller_rabin_test(p, k=10):
    if p < 2:
        return False
    if p != 2 and p % 2 == 0:
        return False

    factorization_base = p - 1
    while factorization_base % 2 == 0:
        factorization_base //= 2

    for _ in range(k):
        witness = random.randint(1, p - 1)
        if gcd(witness, p) > 1:
            continue

        exponent = factorization_base
        residue = pow(witness, exponent, p)

        while exponent != p - 1 and residue != 1 and residue != p - 1:
            residue = (residue * residue) % p
            exponent *= 2

        if residue != p - 1 and exponent % 2 == 0:
            return False

    return True
```

Тест працює корректно

```
#Debug
p = [659, 600, 359, 14]
for i in p:
    result = miller_rabin_test(i)
    if result: print(f"\n{i} сильно псевдопросте\n")
    else: print(f"\n{i} складене число\n")
```

```
659 сильно псевдопросте

600 складене число

359 сильно псевдопросте

14 складене число
```

Функція генерації випадкового простого числа з заданого інтервалу з використанням тесту Міллера-Рабіна:

```
def generate_random_prime(start, end, k=10):
    while True:
        p = random.randint(start, end)
        if trial_division(p) and miller_rabin_test(p, k):
            return p

#Debug
for _ in range(4):
    print("\n",generate_random_prime(1337, 8120),"\n")
```

```
4597

5843

2111

2357
```

2. За допомогою цієї функції згенерувати дві пари простих чисел p, q і p_1, q_1 довжини щонайменше 256 біт. При цьому пари чисел беруться так, щоб $pq \le p_1 q_1$; p і q – прості числа для побудови ключів абонента A, p_1 і q_1 – абонента B.

Генерується 4 рандомних числа, сортуються в порядку зростання, двом найменшии призначимо p та q, найбільшим - p1 і q1.

```
def GenerateKeyPairs():
    bit_length = 256

    primes = [generate_random_prime(2**(bit_length-1), 2**bit_length - 1) for _ in range(4)]

    primes.sort()

    p, q = primes[:2]
    p1, q1 = primes[2:]

    return p, q, p1, q1


#Debug

p, q, p1, q1 = GenerateKeyPairs()
print(f"\nAlice's public key: ({p}, {q})")
print(f"Bob's public key: ({p1}, {q1})\n")
if p*q <= p1*q1: print("pq <= p1q1\n")
```

```
Alice's public key: (67284149490242537804650152602092680028436386104129912005227638691296870646057, 773982545980634148997544349659252773345518759598020081929456206644604858620 89)
Bob's public key: (941420841790363052193787424863469411024489090169190535967427682205419438316 17, 952022851312306964245104048033348412990792092371418298255506084723719714787 23)

pq <= p1q1
```

3. Написати функцію генерації ключових пар для RSA. Після генерування функція повинна повертати та/або зберігати секретний ключ (d, p, q) та відкритий ключ (n, e). За допомогою цієї функції побудувати схеми RSA для абонентів A і B – тобто, створити та зберегти для подальшого використання відкриті ключі (e, n), (e_1, n_1) та секретні d і d_1.

Подрібно додати розширений алгоритм Евкліда

```
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x
```

```
def GenerateRsaKeys(p, q, p1, q1):
    n = p * q
    phi_n = (p - 1) * (q - 1)

    while True:
        public_exponent = random.randint(2, phi_n - 1)
        gcd_result, _, private_exponent = extended_gcd(phi_n, public_exponent)
        if gcd_result == 1:
            break

    n1 = p1 * q1
    phi_n1 = (p1 - 1) * (q1 - 1)

    while True:
        public_exponent_1 = random.randint(2, phi_n1 - 1)
        gcd_result_1, _, private_exponent_1 = extended_gcd(phi_n1, public_exponent_1)
        if gcd_result_1 == 1:
            break

    public_key = (n, public_exponent)
    private_key = (private_exponent, p, q)

    public_key_1 = (n1, public_exponent_1)
    private_key_1 = (private_exponent_1, p1, q1)

    return public_key, private_key, public_key_1, private_key_1


#Debug
p, q, p1, q1 = GenerateKeyPairs()
alice_public_key, alice_private_key, bob_public_key, bob_private_key = GenerateRsaKeys(p, q, p1, q1)
print("\nAlice's RSA public pey:", alice_public_key)
print("Alice's RSA private key:", alice_private_key)
print("Bob's RSA public key:", bob_public_key)
print("Bob's RSA private key:", bob_private_key,"\n")
```

```
Alice's RSA public pey: (54603499732885495511277678124910643550503084792271647442985104149100736725910201042860773322775582180307459
99962443827469373487973488172613292324785575577, 53616831124031530701751111699498995127723386122684007848408419199699019260908639029 9
10565734540656259785875109055306338580933790414325905209602392315838737)
Alice's RSA private key: (26035558984239132351379392794953927976817640474281818083912157932716357321846781779756148632381854186662997
74118264707529292904793805342182238985978283571 3, 613702352853611987270933161835508837950688076467687202630430210984659541368 61, 889
73912970984176454394325596474744412398551230855837110608307879853269904557)
Bob's RSA public key: (98372500434777280060389668426447557408321272675976187536649073258481689504723763864337730148177587933944034642
78038539543979963386275823489454692565911783, 91579552671842367800885803313207435853869500268488108497821415006503871669573432705204
32311124170123020023466349797294707887798706627230256633348748678117)
Bob's RSA private key: (-21679074885565653210894658085268096883344921827508313030718479235351113286055346782386504255635655718333411
75984669252636390324787406258934895542348320787, 91414334271025097833658299420562420138029017750295912614205718280679705763303, 1076
11679524047749830892405367053299158886632166193805144733545211731305340161)
```

У функції створення ключів RSA для кожної знайденої пари (p, q) та (p1, q1), обчислюється n = p * q та phi_n = (p - 1) * (q - 1). Аналогічно для (p1, q1), це частина алгоритму Ейлера. n - модуль, який використовується у всіх операціях шифрування та розшифрування. Він обчислюється як добуток двох простих чисел, p та q, для кожної пари (p, q) та (p1, q1). Модуль n є великим непарним числом. phi_n (функція Ейлера для n) представляє собою кількість натуральних чисел, менших за n, які взаємно прості з n. Для простого числа p, phi(p) = p – 1. Таким чином, phi_n обчислюється як (p - 1) * (q - 1) для пар (p, q) та (p1, q1).

Генеруємо випадкові числа e та e1 так, щоб вони були взаємно простими з phi_n та phi_n1 відповідно. Це гарантує, що їхній GCD дорівнює 1.

Для кожної пари (e, phi_n) та (e1, phi_n1) розширенм алгоритмом Евкліда знаходимо d та d1, такі що (e * d) % phi_n = 1 та (e1 * d1) % phi_n1 = 1.

Публічний ключ представляється парою (n, e) для кожного абонента.
Приватний ключ представляється трійкою (d, p, q) для відповідного абонента.

4. Написати програму шифрування, розшифрування і створення повідомлення з цифровим підписом для абонентів *A* і *B*. Кожна з операцій (шифрування, розшифрування, створення цифрового підпису, перевірка цифрового підпису) повинна бути реалізована окремою процедурою, на вхід до якої повинні подаватись лише ті ключові дані, які необхідні для її виконання.

За допомогою датчика випадкових чисел вибрати відкрите повідомлення *M* і знайти криптограму для абонентів *A* и *B*, перевірити правильність розшифрування. Скласти для *A* і *B* повідомлення з цифровим підписом і перевірити його.

Для шифрування використовується операція залишку від ділення (pow(message, e, n)), де e - публічний експонент, n – модуль

```
def Encrypt(message, public_key):
    n, e = public_key
    cipher_text = pow(message, e, n)
    return cipher_text
```

Для розшифрування використовується операція залишку від ділення (pow(cipher_text, d, n)), де d - приватний експонент, n - модуль. Результат - відкрите повідомлення.

```
def Decrypt(cipher_text, private_key):
    d, p, q = private_key
    n = p * q
    plain_text = pow(cipher_text, d, n)
    return plain_text
```

Підписання та перевірка підпису

```
p, q, p1, q1 = GenerateKeyPairs()
alice_public_key, alice_private_key, bob_public_key, bob_private_key = GenerateRsaKeys(p, q, p1, q1)

message_to_alice = random.randint(812, 2014)
cipher_text_for_alice = Encrypt(message_to_alice, alice_public_key)
Decrypted_message_for_alice = Decrypt(cipher_text_for_alice, alice_private_key)

message_to_bob = "Hello, Bob!"
signed_message_for_bob = Sign(message_to_bob, bob_private_key)
signature_verified = Verify(signed_message_for_bob, bob_public_key)


print("\nMessage to Alice:", message_to_alice)
print("Encrypted Cipher Text for Alice:", cipher_text_for_alice)
print("Decrypted Message for Alice:", Decrypted_message_for_alice, "\n")

print("Message to Bob:", message_to_bob)
print("Signed Message for Bob:", signed_message_for_bob)
print("Signature Verification Result for Bob:", signature_verified, "\n")
```

Приклад використання

```
Message to Alice: 1211
Encrypted Cipher Text for Alice: 347712474101699025447914047693210884669541358820513209680900381661850460545504760062261224989388803216148681805219056521458759292440496560167578434970215 7
Decrypted Message for Alice: 1211

Message to Bob: Hello, Bob!
Signed Message for Bob: ('Hello, Bob!', 489112899277213880744501954239100947863975168142194019473422631475705044544370250381415548960566171342983869810019837877534148523830249057329491101588598 0)
Signature Verification Result for Bob: True
```

5. За допомогою раніше написаних на попередніх етапах програм організувати роботу протоколу конфіденційного розсилання ключів з підтвердженням справжності по відкритому каналу за допомогою алгоритму RSA. Протоколи роботи кожного учасника (відправника та приймаючого) повинні бути реалізовані у вигляді окремих процедур, на вхід до яких повинні подаватись лише ті ключові дані, які необхідні для виконання. Перевірити роботу програм для випадково обраного ключа $0 < k < n$.

- Аліса і Боб обмінюються відкритими ключами (e_A, n_A) .
- Аліса обирає випадкове секретне значення k (0 < k < n), яке буде використано для створення спільного ключа.
- Аліса зашифровує секретне значення k за допомогою відкритого ключа Боба (e_B, n_B), отримуючи k_1.
- Аліса обчислює S_1, яке є підписом секретного значення k за допомогою свого секретного ключа (d_A, n_A).
- Аліса відправляє Бобу пару значень (k_1, S_1).
- Боб отримує (k_1, S_1) від Аліси.
- Боб розшифровує k_1 за допомогою свого секретного ключа (d_B, n_B), отримуючи секретне значення k.
- Боб перевіряє підпис S_1, використовуючи відкритий ключ Аліси (e_A, n_A), для перевірки на автентичність.

```
def SendKey(public_key_B, private_key_A, public_key_A):
    k = random.randint(0, public_key_A[0] - 1)
    print("\nk value:", k)
    s = pow(k, private_key_A[0], public_key_A[0])
    k_1 = pow(k, public_key_B[1], public_key_B[0])
    S_1 = pow(s, public_key_B[1], public_key_B[0])
    return k_1, S_1

def RecieveKey(k_1, S_1, public_key_B, private_key_B, public_key_A):
    k = pow(k_1, private_key_B[0], public_key_B[0])
    S = pow(S_1, private_key_B[0], public_key_B[0])
    return k == pow(S, public_key_A[1], public_key_A[0])
```

```python
k_1, S_1 = SendKey(bob_public_key, alice_private_key, alice_public_key)
key_received = RecieveKey(k_1, S_1, bob_public_key, bob_private_key, alice_public_key)

if key_received:
    print("Key exchange successful!")
else:
    print("Key exchange failed!")
```

```
k value: 343078595487980370991345983439250053596864925518198242890751463006091498200797249848189667170356428987844021994964436521665240863923401862944474352912904 3

Message to Alice: 1728
Encrypted Cipher Text for Alice: 3167606249681422321258432840883235956508020198115830108448989711677203941958339032158973221684221868191690749511438936468631585035573163657794537740456699
Decrypted Message for Alice: 1728

Message to Bob: Hello, Bob!
Signed Message for Bob: ('Hello, Bob!', 260025818792720712799873104219166354886138015447503535668307868239038463310776458051616561874867186621061773842554800633002644889829731995092731336123217 9)
Signature Verification Result for Bob: True

Key exchange successful!
```

Висновок

Лабортаторна робота дозволила на практиці розібратися та зрозуміти алгоритм RSA: було реалізовано генерацію ключів RSA, функції шифрування, розшифрування та цифрового підпису, а також протокол конфіденційного розсилання ключів.