Міністерство освіти і науки України

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Фізико-технічний інститут

Комп’ютерний практикум №4

З дисципліни «Криптографія»

Виконали:

ФБ-33 Лозенко Павло,

ФБ-33 Самохвалов Роман

Київ – 2025

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

Мета 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.

Порядок виконання 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.

```
1. ГЕНЕРАЦІЯ ПРОСТИХ ЧИСЕЛ (256 біт)
----------------------------------------------------------------------
Абонент A:
  p  = 624222973020456963889910439207540254899736077558796366338821703252367308176l7
  q  = 994047613135436003209463702914423090550151911891052818779890656802717162518O9

Абонент B:
  p1 = 75778724152460640623685637486350020048076854733901108338929024718117953872853
  q1 = 113914866555445833494915337759158215469528132677072251219822978469096888481371
```

```python
def generate_prime(bits: int) -> int:
    """Генерує випадкове просте число заданої довжини в бітах"""
    while True:
        # Генеруємо випадкове непарне число
        n = random.getrandbits(bits)
        n |= (1 << bits - 1) | 1  # Встановлюємо старший та молодший біти

        if is_prime(n):
            return n

def generate_prime_pair(bits: int) -> Tuple[int, int]:
    """Генерує пару простих чисел p, q таких що p != q"""
    p = generate_prime(bits)
    q = generate_prime(bits)

    while q == p:
        q = generate_prime(bits)

    return p, q
```

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 $pq \leq p_1q_1$; p і q – прості числа для побудови ключів абонента A, p_1 і q_1 – абонента B.

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 .

```
2. ГЕНЕРАЦІЯ КЛЮЧОВИХ ПАР
--------------------------------------------------------------------------
Абонент A:
  Відкритий ключ:  n = 620507356395290909701412250673916741468569255043134947394713145002668240428653284504258111452885422202811693217340653444694889968753976506293557122531915 3
                   e = 65537
  Секретний ключ:  d = 509911106489842184306258421537367594313613162024552172465777669671539752453189492263216881039569041292958868201684128663837067357066643651590926440003993 7

Абонент B:
  Відкритий ключ:  n = 863232324956949404496607821019198613826261958809078653873248805061369119417290558903186839190427222515351320373649131863021284806299275250204755519312146 3
                   e = 65537
  Секретний ключ:  d = 836586018391377153015816243407744949548517661793927424540420290899305708999017161882676758276409932307375900820129765602772758440482584001169044311436753
```

```python
def GenerateKeyPair(bits: int = 256) -> Tuple[Tuple[int, int], Tuple[int, int, int]]:
    """
    Генерує пару ключів RSA
    Повертає: ((n, e), (d, p, q))
    """
    # Генеруємо два простих числа
    p, q = generate_prime_pair(bits)

    # Обчислюємо модуль
    n = p * q

    # Обчислюємо функцію Ойлера
    phi = (p - 1) * (q - 1)

    # Вибираємо e (зазвичай 65537)
    e = 65537
    if gcd(e, phi) != 1:
        # Якщо 65537 не підходить, шукаємо інше
        e = 3
        while gcd(e, phi) != 1:
            e += 2

    # Знаходимо d - обернений до e за модулем phi
    d = mod_inverse(e, phi)

    public_key = (n, e)
    private_key = (d, p, q)

    return public_key, private_key
```

4. Написати програму шифрування, розшифрування і створення повідомлення з цифровим підписом для абонентів А і В. 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.

```
3. ШИФРУВАННЯ ТА РОЗШИФРУВАННЯ
--------------------------------------------------------------------------------
Відкрите повідомлення M = 589807275331948372135589543287261307562241886238691604139632960493566763478540040443076704051847944539394044845490192453505091294732489078613495690568659 0

A шифрує для B:
  Криптограма C = 307464826943216464208071227597093837761537481000356456208072668516871140523857676717724339209794311985763649085497184837174100858302942820819635827493238
  B розшифровує: M' = 589807275331948372135589543287261307562241886238691604139632960493566763478540040443076704051847944539394044845490192453505091294732489078613495690568659 0
  Перевірка: M == M' -> True
```

```
4. ЦИФРОВИЙ ПІДПИС
--------------------------------------------------------------------------------
A підписує повідомлення M = 589807275331948372135589543287261307562241886238691604139632960493566763478540040443076704051847944539394044845490192453505091294732489078613495690568659 0
  Підпис S = 532160135872647902540339545805243719332086262938565426767696680744753725450700800454814888923838890963332533057106791388616679883755801157548939727295171 9
  Перевірка підпису: True
  Перевірка з M' = 589807275331948372135589543287261307562241886238691604139632960493566763478540040443076704051847944539394044845490192453505091294732489078613495690568659 1: False
```

```python
def Encrypt(message: int, public_key: Tuple[int, int]) -> int:
    """Шифрує повідомлення відкритим ключем"""
    n, e = public_key
    if message >= n:
        raise ValueError("Повідомлення занадто велике для даного модуля")
    return power_mod(message, e, n)

def Decrypt(ciphertext: int, private_key: Tuple[int, int, int]) -> int:
    """Розшифровує криптограму секретним ключем"""
    d, p, q = private_key
    n = p * q
    return power_mod(ciphertext, d, n)

# ---------------------------------------------------------------------------
```

```python
def Sign(message: int, private_key: Tuple[int, int, int]) -> int:
    """Створює цифровий підпис повідомлення"""
    d, p, q = private_key
    n = p * q
    if message >= n:
        raise ValueError("Повідомлення занадто велике для підпису")
    return power_mod(message, d, n)

def Verify(message: int, signature: int, public_key: Tuple[int, int]) -> bool:
    """Перевіряє цифровий підпис"""
    n, e = public_key
    verified_message = power_mod(signature, e, n)
    return verified_message == message

#
```

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 $0 < k < n$.

```
5. ПРОТОКОЛ КОНФІДЕНЦІЙНОГО РОЗСИЛАННЯ КЛЮЧІВ
--------------------------------------------------------------------------
A передає ключ k = 19448498704801410020686483134387456100300993878992076015308750193092411792459247456639453303769112458978628895111511941372711591516772034284886158333287163 абоненту B

A формує повідомлення:
  k1 = 428355108018426106921520258179990531079434998505289722250826135250042477314475540796549606591654381580487120247318669415585181531375556025776427657165793 7
  S1 = 693623065953777056722938004785439546002776526190714623039922476675959077350524328250166614249636164588203710873908619428543754761861874258671707594947286 6

B отримує та перевіряє:
  k' = 19448498704801410020686483134387456100300993878992076015308750193092411792459247456639453303769112458978628895111511941372711591516772034284886158333287163
  Перевірка: k == k' -> True
```

```python
def SendKey(k: int, sender_private_key: Tuple[int, int, int],
            receiver_public_key: Tuple[int, int]) -> Tuple[int, int]:
    """
    Відправник формує повідомлення (k1, S1) для передачі ключа
    k - ключ для передачі
    """
    d, p, q = sender_private_key
    n = p * q
    n1, e1 = receiver_public_key

    if n1 <= n:
        raise ValueError("n1 має бути більше за n")

    # Шифруємо ключ відкритим ключем отримувача
    k1 = power_mod(k, e1, n1)

    # Створюємо підпис
    S = power_mod(k, d, n)

    # Шифруємо підпис відкритим ключем отримувача
    S1 = power_mod(S, e1, n1)

    return k1, S1
```

```python
def ReceiveKey(k1: int, S1: int, receiver_private_key: Tuple[int, int, int],
               sender_public_key: Tuple[int, int]) -> Optional[int]:
    """
    Отримувач розшифровує та перевіряє ключ
    Повертає ключ k якщо підпис вірний, інакше None
    """
    d1, p1, q1 = receiver_private_key
    n1 = p1 * q1
    n, e = sender_public_key

    # Розшифровуємо ключ
    k = power_mod(k1, d1, n1)

    # Розшифровуємо підпис
    S = power_mod(S1, d1, n1)

    # Перевіряємо підпис
    k_verified = power_mod(S, e, n)

    if k_verified == k:
        return k
    else:
        return None
```

Висновки:

Дослідження алгоритму RSA та методів його практичного застосування дозволяє зробити наступні висновки:

Щодо криптосистеми RSA: Асиметрична природа алгоритму забезпечує надійний механізм шифрування та розшифрування даних. Схема Горнера оптимізує процес піднесення до степеня за модулем, що значно підвищує ефективність обчислень при роботі з великими числами.

Щодо цифрового підпису: Використання хеш-функції SHA-256 гарантує цілісність даних та дозволяє створювати компактні цифрові відбитки повідомлень будь-якого розміру. Механізм цифрового підпису забезпечує одночасно автентифікацію відправника та підтвердження незмінності переданої інформації.

Щодо протоколу розсилання ключів: Розроблений протокол успішно реалізує конфіденційну передачу даних через відкриті канали зв'язку, поєднуючи шифрування для захисту інформації та цифровий підпис для автентифікації. Умова $n\_B \geq n\_A$ є критичною для коректної роботи системи, оскільки запобігає втраті даних при вкладеному шифруванні. Шифрування підпису S_1 забезпечує повну конфіденційність всіх компонентів повідомлення.

Практична реалізація підтвердила ефективність RSA як надійного засобу захисту інформації в сучасних комунікаційних системах. Математичні основи алгоритму гарантують високий рівень криптографічної стійкості, що робить його придатним для застосування у критичних системах безпеки.