НТУУ «Київський політехнічний інститут імені Ігоря Сікорського»

НН ФТІ

КРИПТОГРАФІЯ

КОМП’ЮТЕРНИЙ ПРАКТИКУМ №4

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

Виконали:
Соколовська Дарія, Дудник Нікіта
ФБ-13

Київ, 2023

I. Написати функцію пошуку випадкового простого числа з заданого інтервалу або заданої довжини, використовуючи датчик випадкових чисел та тести перевірки на простоту. В якості тесту перевірки на простоту рекомендовано використовувати тест Міллера-Рабіна із попередніми пробними діленнями.

```python
def test_easy_deviders(p:int):
    if p % 2 == 0 or p % 3 == 0 or p % 5 == 0 or 0 % 7== 0 or p % 11 == 0:
        return False
    return True

def miller_rabin(p:int, k:int = 50):

    if test_easy_deviders(p):
        d, s = p-1, 0

        while d % 2 == 1:
            s += 1
            d //= 2

        for _ in range(k):
            x = random.randint(2, p-1)
            if gcd(x, p) != 1:  return False

            x_1 = horner_pow(x, d, p)
            if x_1 == 1 or x_1 == -1 : return True

            for r in range(1, s+1):
                x_r = horner_pow(x, d*(2**r), p)
                if x_r == -1 :  return True

    return False
```

Тут ми інтегрували попередню перевірку на прості дільники (2, 3, 5, 7, 11) одразу в тест Міллера-Рабіна таким чином, щоб безпосередньо сам тест починався якщо на найпростіші дільники наше число не ділиться.

Варто зауважити, що всі ці перевірки спрямовані на велиикі, мінімум двоцифрові числа, якщо число буде 2, 3, …., 11, отримаємо False, хоча вони прості.

```python
print(miller_rabin(324))        # ділиться на 2
print(miller_rabin(321))        # ділиться на 3
print(miller_rabin(657))        # ділиться на 3
print(miller_rabin(179))        # посте
```

```
False
False
False
True
```

Ось такі маємо результати, отже працює справно.

Генерація простих чисел:

```python
def get_random_prime(start:int = None, end:int = None, bits = None, toprint:bool=False):
    if bits is not None:
        while True:
            num = random.getrandbits(bits)
            if miller_rabin(num) :
                if toprint: print(num)
                return num

    elif start is not None and end is not None:
        while True:
            num = random.randint(start, end)
            if miller_rabin(num):
                if toprint: print(num)
                return num
    else:
        raise ValueError("Некорректно задано початкове та кінцеве значення інтервалу або кіль
```

```
get_random_prime(bits=64, toprint=True)
get_random_prime(bits=128, toprint=True)
get_random_prime(start=690, end=20578, toprint=True)
```

```
2997207483450952543
183122338390583110035304629070480049219
20101
```

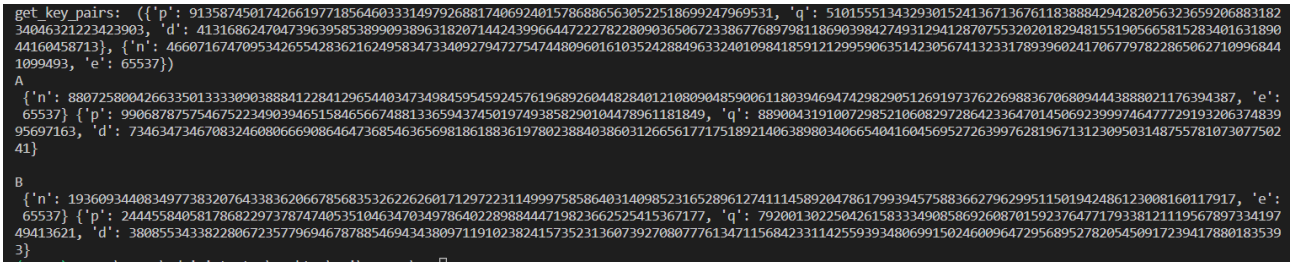
Як видно, йде одразу перевірка на простоту числа. Генерація можлива як при заданні певної кількості бітів, так і при заданому інтервалі, при чому обидві межі включені.

II. **За допомогою цієї функції згенерувати дві пари простих чисел p, q і p1 , q1 довжини щонайменше 256 біт. При цьому пари чисел беруться так, щоб pq ≤ p1q1 ; p і q – прості числа для побудови ключів абонента A, p1 і q1 – абонента B.**

III. **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) , (e1 ,n1) та секретні d і d1 .**

```python
def get_keys_pairs(bits:int=256):
    secret_keys = {}
    open_keys = {}

    # for A, n and e - keys
    secret_keys['p'] = get_random_prime(bits=bits)
    secret_keys['q'] = get_random_prime(bits=bits)

    open_keys['n'] = secret_keys['p'] * secret_keys['q']
    euler = (secret_keys['p'] - 1) * (secret_keys['q'] - 1)

    open_keys['e'] = E   = 2 ^ 16 + 1
    secret_keys['d'] = get_inversed(open_keys['e'], euler)

    return secret_keys, open_keys

# print(get_keys_pairs())

def generate_pairs_AB():
    A_secretKeys, A_openKeys = get_keys_pairs()
    # перевірка q1*p1 <= q2*p2
    # міняємо
    B_secretKeys, B_openKeys = get_keys_pairs()
    if B_secretKeys['q'] * B_secretKeys['p'] > A_secretKeys['q'] * A_secretKeys['p']:
        A_secretKeys, B_secretKeys = B_secretKeys, A_secretKeys

    return A_openKeys, A_secretKeys, B_openKeys, B_secretKeys
```

Генеруємо пари ключів, для виконання нашого завдання взяли дефолтну довжину у 256 бітів, проте алгоритм дає можливість використати довільну.

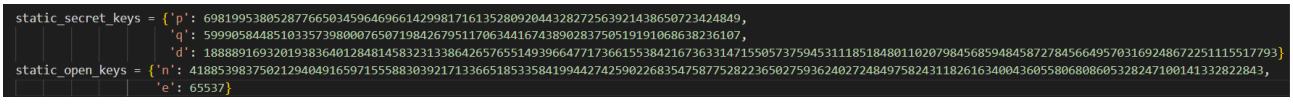


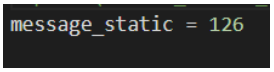
Shall the great magic begin now!

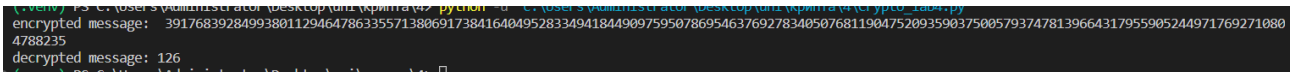
IV. Написати програму шифрування, розшифрування і створення повідомлення з цифровим підписом для абонентів А і В. Кожна з операцій (шифрування, розшифрування, створення цифрового підпису, перевірка цифрового підпису) повинна бути реалізована окремою процедурою, на вхід до якої повинні подаватись лише ті ключові дані, які необхідні для її виконання. За допомогою датчика випадкових чисел вибрати відкрите повідомлення M і знайти криптограму для абонентів А и В, перевірити правильність розшифрування. Скласти для А і В повідомлення з цифровим підписом і перевірити його.

```python
def cipher_message(M: int, openKeys: dict):
    n, e = openKeys['n'], openKeys['e']
    if M >= n:
        raise ValueError(f"Ваше повідомлення занадто велике для шифрування!\nПовідомлення: {M}\nn: {n}")
    elif M < 0:
        raise ValueError("Ваше повідомлення менше 0")
    res = horner_pow(M, e, n)
    return res

def decipher_message(C: int, privateKeys: dict):
    p, q, d = privateKeys['p'], privateKeys['q'], privateKeys['d']
    n = p * q

    if C >= n:
        raise ValueError(f"Криптограма завелика для розшифрування!\nКриптограма: {C}\nn: {n}")
    elif C < 0:
        raise ValueError("Криптограма менше нуля!")
    res = horner_pow(C, d, n)
    return res
```

Для демонстрації використаємо статичні пари ключів, щоб спростити собі задачу. Почнемо з перевірки функцій шифрування:







Отже, все працює справно. Тепер перейдемо до перевірки цифрового підпису:

```
csign = crypto_sign(message_static, static_secret_keys)

print("crypto sogn: ", csign)
print("verification: ", verification(message_static, csign, static_open_keys))
```

```
crypto sign:  269520331842109789663815095513132817156137078816174304307255664755860492206008567704352643021423361726375709241849049827143508576856427419353921879771665 4
verification:
True
```

Тепер спробуємо зробити повноцінну сесію. Визначимо ключі A, B, створимо для них повідомлення (в даному випадку генеруємо обов’язково різні щоб побачити що все передається справно), і запустимо все.

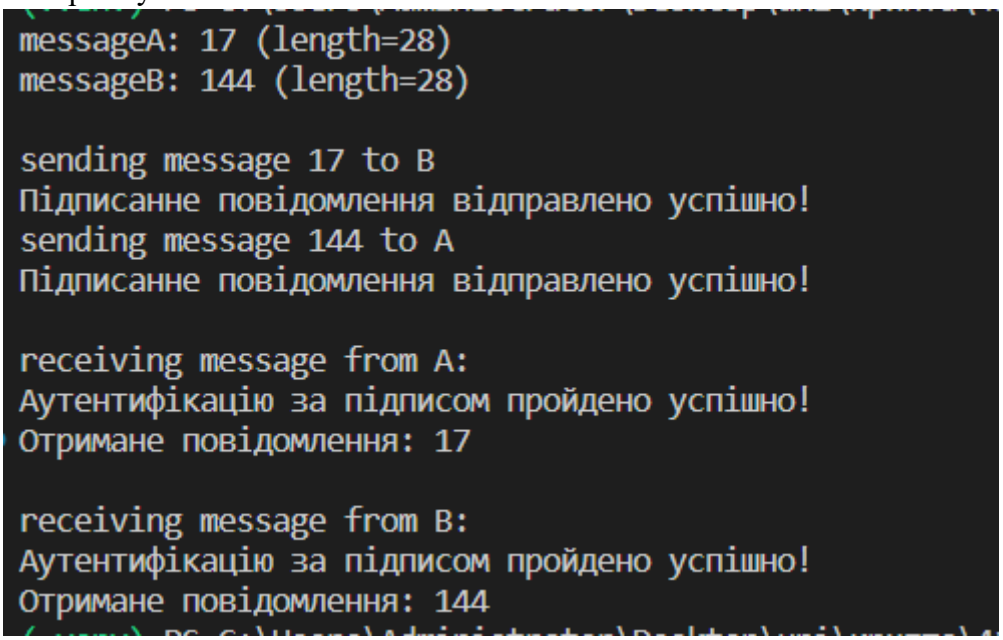
Наші функції передачі та отримання повідомлень:

```
def send_message(M:int, A_privateKeys:dict, B_openKeys:dict):
    encrypted_M = cipher_message(M, B_openKeys)
    S = crypto_sign(M, A_privateKeys)
    S1 = horner_pow(S, B_openKeys['e'], B_openKeys['n'])
    if encrypted_M and S and S1:
        print("Підписанне повідомлення відправлено успішно!")

    return encrypted_M, S1

def receive_message(encrypted_M:int, S1:int, B_privateKeys:dict, A_openKeys:dict):
    M = decipher_message(encrypted_M, B_privateKeys)
    d = B_privateKeys['d']
    n = B_privateKeys['p'] * B_privateKeys['q']
    S = horner_pow(S1, d, n)
    verified = verification(M, S, A_openKeys)

    if not verified:
        print("Аутентифікацію провалено! Підпис та/або ключі невірні!")
    else:
        print("Аутентифікацію за підписом пройдено успішно!\nОтримане повідомлення:", M)
```

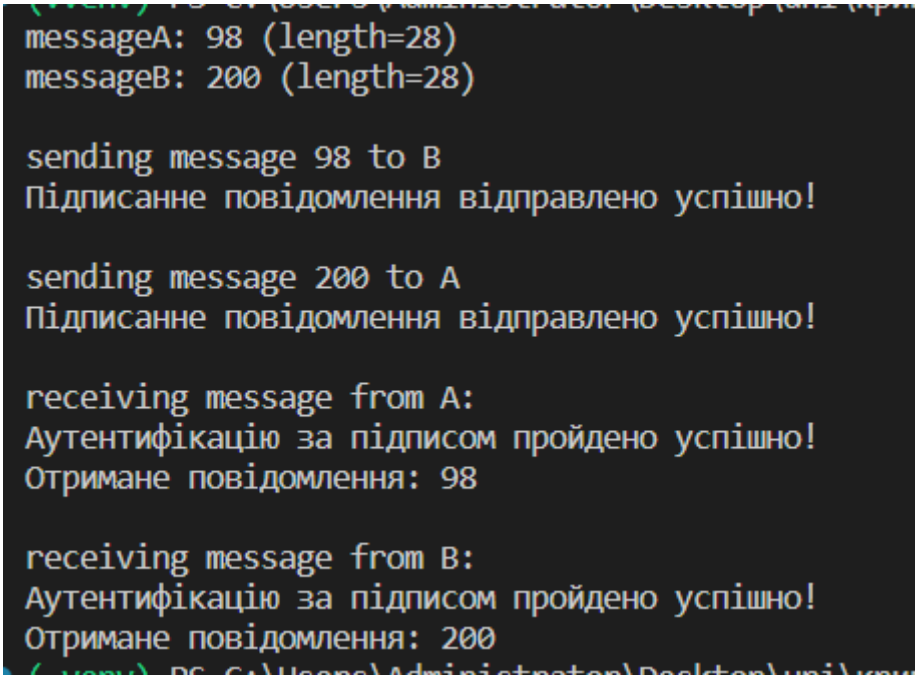
У них, як можемо побачити, відбуваються усі необхідні дії, нам вистачає просто їх запустити, отримати необхідні значення, і перенаправити у функцію отримання повідомлення.

Передбачається, що перед цим, ми отримаємо необхідні відкриті ключі, що не передбачено у цих функціях. Тож їх ми генеруємо самі, і передаємо у send_message() та receive_message().

Тепер запускаємо:



І ще раз:



Висновки:

Ми отримали розуміння того, як будується з’єднання між двома абонентами в криптосистемі RSA, змогли самостійно реалізувати (спрощену, в певній мірі) мережу, яка включає шифрування, створення підпису, його перевірку, та розшифрування самого повідомлення. Для цього ми створили самостійно функції генерації пар ключів, для яких необхідно було написати функції генерації простих чисел із перевіркою на простоту (тест Міллера-Рабіна).

Складнощі виникали при реалізації безпосередньо з’єднання, адже інколи функції не розпізнавали цифровий підпис або ж сповіщали про неправильні ключі.