

**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.

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

def miller_rabin(p:int, k:int = 50):
    if test_easy_dividers(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 величезні, мінімум двоцифрові числа, якщо число буде 2, 3, ..., 11, отримаємо False, хоча вони прості.

```
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))    # просте
```

```
False
False
False
True
```

Ось такі маємо результати, отже працює справно.

Генерація простих чисел:

```
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("Некоректно задано початкове та кінцеве значення інтервалу або кількість біт")
```

```

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  $p_1, q_1$  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 А,  $p_1$  і  $q_1$  – абонента В.
- 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 А і В – тобто, створити та зберегти для подальшого використання відкриті ключі  $(e, n)$ ,  $(e_1, n_1)$  та секретні  $d$  і  $d_1$ .

```

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 = 216 + 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.

```
get_key_pairs: ({'p': 91358745017426619771856460333149792688174069240157868865630522518699247969531, 'q': 510155513432930152413671367611838884294282056323659206883182
34046321223423903, 'd': 41316862470473963958538990938963182071442439966447222782280903650672338677689798118690398427493129412870755320201829481551905665815283401631890
44160458713}, {'n': 466071674709534265542836216249583473340927947275474480960161035242884963324010984185912129959063514230567413233178939602417067797822865062710996844
1099493, 'e': 65537})
A
{'n': 880725800426633501333090388841228412965440347349845954592457619689260448284012108090485900611803946947429829051269197376226988367068094443888021176394387, 'e':
65537} {'p': 99068787575467522349039465158465667488133659437450197493858290104478961181849, 'q': 889004319100729852106082972864233647014506923999746477729193206374839
95697163, 'd': 73463473467083246080666908646473685463656981861883619780238840386031266561771751892140638980340665404160456952726399762819671312309503148755781073077502
41}
B
{'n': 1936093440834977383207643383620667856835326226260171297223114999758586403140985231652896127411145892847861799394575883662796299511501942486123008160117917, 'e':
65537} {'p': 24445584058178682297278747405351046347034978640228988444719823662525415367177, 'q': 792001302250426158333490858692608701592376477179338121119567897334197
49413621, 'd': 380855343822806723577969467878854694343809711910238241573523136073927080776134711568423311425593934806991502460096472956895278205450917239417880183539
3}
```

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 М і знайти криптограму для абонентів А і В, перевірити правильність розшифрування. Скласти для А і В повідомлення з цифровим підписом і перевірити його.

```
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:

```
static_secret_keys = {'p': 69819953805287766503459646966142998171613528092044328272563921438650723424849,
                      'q': 59990584485103357398000765071984267951170634416743890283750519191068638236107,
                      'd': 188889169320193836401284814583231338642657655149396647717366155384216736314715505737594531118518480110207984568594845872784566495703169248672251115517793}
static_open_keys = {'n': 4188539837502129404916597155588303921713366518533584199442742590226835475877528223650275936240272484975824311826163400436055806808605328247100141332822843,
                    'e': 65537}

message_static = 126

(c:\python\python.exe) C:\Users\pavlenko\Documents\crypt.py
encrypted message: 391768392849938011294647863355713806917384164049528334941844909759507869546376927834050768119047520935903750057937478139664317955905244971769271080
4788235
decrypted message: 126
```

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:  2695203318421097896638150955131328171561370788161743043072556647558604922060085677043526430214233617263757092418490498271435085768564274193539218797716654
verification:
True

```

Тепер спробуємо зробити повноцінну сесію. Визначимо ключі А, В, 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("Підписання 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:

```
messageA: 17 (length=28)
messageB: 144 (length=28)

sending message 17 to B
Підписання повідомлення відправлено успішно!
sending message 144 to A
Підписання повідомлення відправлено успішно!

receiving message from A:
Аутентифікацію за підписом пройдено успішно!
Отримане повідомлення: 17

receiving message from B:
Аутентифікацію за підписом пройдено успішно!
Отримане повідомлення: 144
```

І ще раз:

```
messageA: 98 (length=28)
messageB: 200 (length=28)

sending message 98 to B
Підписання повідомлення відправлено успішно!

sending message 200 to A
Підписання повідомлення відправлено успішно!

receiving message from A:
Аутентифікацію за підписом пройдено успішно!
Отримане повідомлення: 98

receiving message from B:
Аутентифікацію за підписом пройдено успішно!
Отримане повідомлення: 200
```

### 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.