КОМП’ЮТЕРНИЙ ПРАКТИКУМ №4

Гоголєва Поліна ФБ-12

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

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.

Як же боляче це було по цим вашим математичним поясненням у методичці… і хоч вже існуюча реалізація тесту Міллера-Рабіна, котру я знайшла в інтернеті чомусь була набагато коротша, я написала свою, котра косо-криво, але працює!

Намагалась робити покроково, як вказано у методичці.

От її код

```
import random

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

def mod_exp(x, d, p):
    result = 1
    x = x % p

    while d > 0:
        if d % 2 == 1:
            result = (result * x) % p
        d = d // 2
        x = (x * x) % p

    return result

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 not trial_division(p):  # Перевірка на простоту за допомогою решти
        return False

    # Крок 0: Знаходимо розклад p - 1 = d * 2^S
    d, s = p - 1, 0
    while d % 2 == 0:
        d //= 2
        s += 1

    # Лічильник
    count = 0

    is_pseudo_prime = False

    # Крок 3. Повторюємо k разів
    while count < k:
        # Крок 1: Вибираємо випадкове число x
        x = random.randint(2, p - 1)

        # Знаходимо gcd(x, p)
        g = gcd(x, p)

        # Якщо gcd(x, p) > 1, то p - складене число
        if g > 1:
            return False
        else:
        # Крок 2: Перевіряємо, чи є p сильно псевдопростим за основою x
            if mod_exp(x, d, p) == 1 or mod_exp(x, d, p) == p - 1:
                is_pseudo_prime = True
            else:
                for r in range(1, s):
                    xr = mod_exp(x, d * (2 ** r), p)
                    if xr == p - 1:
                        is_pseudo_prime = True
                        break
                    elif xr == 1:
                        continue
                    else:
                        return False

        if is_pseudo_prime:
            count += 1

    return True
# Тестування
p = 2659  # Приклад простого числа
result = miller_rabin_test(p)
if result:
```

```
    print(f"{p} сильно псевдопросте (може бути простим).")
else:
    print(f"{p} складене число.")
```

```
317 сильно псевдопросте (може бути простим).
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogolev
Data\Local\Programs\Python\Python311\python.exe' 'c:\Users\Po
0168' '--' 'c:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp
599 сильно псевдопросте (може бути простим).
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogolev
Data\Local\Programs\Python\Python311\python.exe' 'c:\Users\Po
0173' '--' 'c:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp
600 складене число.
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogolev
Data\Local\Programs\Python\Python311\python.exe' 'c:\Users\Po
0195' '--' 'c:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp
2659 сильно псевдопросте (може бути простим).
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogolev
Data\Local\Programs\Python\Python311\python.exe' 'c:\Users\Po
0201' '--' 'c:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp
2660 складене число.
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogolev
Data\Local\Programs\Python\Python311\python.exe' 'c:\Users\Po
0206' '--' 'c:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp
2659 сильно псевдопросте (може бути простим).
```

Тепер додамо сюди ф-ю генерації випадкового простого числа з заданого інтервалу.

```
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

print (generate_random_prime(80,580))
```

```
PS C:\
Data\L
0246'
317
PS C:\
Data\L
0251'
139
PS C:\
Data\L
0256'
373
PS C:\
```

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.

Згенеруємо 4 рандомних числа, порівняємо їх, двом більшим призначимо p1 та q1, меншим p і q.

```
def generate_key_pairs():
    bit_length = 256

    # Generate 4 random prime numbers of 256 bits
    primes = [generate_random_prime(2**(bit_length-1), 2**bit_length - 1) for
_ in range(4)]

    # Sort the primes
    primes.sort()

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

    return p, q, p1, q1


p, q, p1, q1 = generate_key_pairs()

print(f"Alice's public key (p, q): ({p}, {q})")
print(f"Bob's public key (p1, q1): ({p1}, {q1})")
```

Мій код ранився нескінченно довго і ламався, тож, признаюсь, я закинула його в аішку для дебагінгу та оптимізації, звідки отримала більш оптимізований варіант перевірки теста міллера-рабіна, бо проблема була саме у ньому, ще й працюємо ми з дуже великими числами. Тому тепер код виглядає так:

```
import random

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

''' Тестування
p = 2659  # Приклад простого числа
result = miller_rabin_test(p)
if result:
    print(f"{p} сильно псевдопросте (може бути простим).")
else:
    print(f"{p} складене число.") '''

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

#print (generate_random_prime(80,580))

def generate_key_pairs():
    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


p, q, p1, q1 = generate_key_pairs()

print(f"Alice's public key (p, q): ({p}, {q})")
print(f"Bob's public key (p1, q1): ({p1}, {q1})")
```

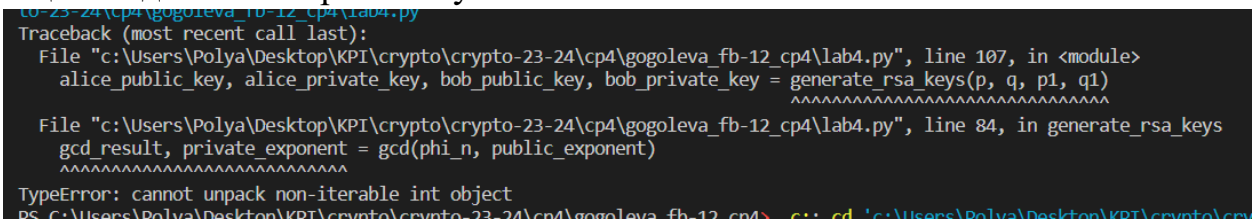
```
to-23-24\cp4\gogoleva_fb-12_cp4\test_lab4.py'
Alice's public key (p, q): (587527600045660397094321855658928251094624512123646770927860605105652965764 79, 956332241920710388753033424649425930927562614835948301327055929173650246940 79)
Bob's public key (p1, q1): (100928219363706940952025950141883807759743823214975786110684053333404358879641, 108440301414608609557444842303220279298239850577111097040465552884234219763629)
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogoleva_fb-12_cp4>  c:; cd 'c:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogoleva_fb-12_cp4'; & 'C:\Users\Polya\AppData\Local\Progra
n\Python311\python.exe' 'c:\Users\Polya\.vscode\extensions\ms-python.python-2023.22.1\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '54665' '--' 'c:\Users\Polya\Desktop\KPI\cry
to-23-24\cp4\gogoleva_fb-12_cp4\test_lab4.py'
Alice's public key (p, q): (807816827116143588924801288820772808195767397392961156488837121869473800514 67, 855592648018784353288615445271904303879473370238812844534549756724566896568 11)
Bob's public key (p1, q1): (926564226642018531324722018645489901103975217694446673392115367660765248917 53, 103946353241894589627835633425402512798816961615102723893700326455340182695307)
PS C:\Users\Polya\Desktop\KPI\crypto\crypto-23-24\cp4\gogoleva_fb-12_cp4>
```

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.

```
def generate_rsa_keys(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 = 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 = 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
```

```
# Generate RSA key pairs for Alice and Bob
p, q, p1, q1 = generate_key_pairs()
alice_public_key, alice_private_key, bob_public_key, bob_private_key =
generate_rsa_keys(p, q, p1, q1)

# Output the results
print("Alice's RSA Public Key:", 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)
```

З цим кодом мали проблемку



Довелось пофіксити gcd, замінила його на extended версію, бо функція НСД мала повертати три числа для правильного генерування РСА ключів.

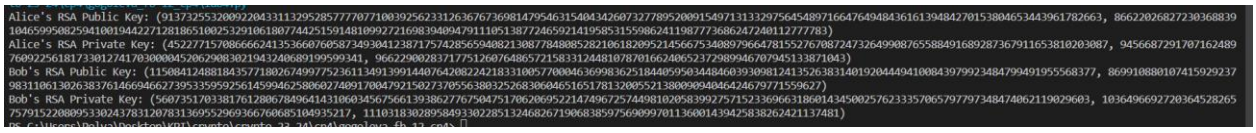
Новий код виглядає так

```
import random

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

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 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

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

def generate_key_pairs():
    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

def generate_rsa_keys(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

# Generate RSA key pairs for Alice and Bob
p, q, p1, q1 = generate_key_pairs()
alice_public_key, alice_private_key, bob_public_key, bob_private_key =
generate_rsa_keys(p, q, p1, q1)

# Output the results
print("Alice's RSA Public Key:", 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)
```

І виводить нам


Alice's RSA Public Key: (91373255320092204331132952857777077100392562331263676736981479546315404342607327789520091549713133297564548971664764948436161394842701538046534439617826 63, 866220268272303688391046599508259410019442271281865100253291061807744251591481099272169839409479111051387724659214195853155986241198777368624724011277778 3)
Alice's RSA Private Key: (45227715708666624135366076058734930412387175742856594082130877848085282106182095214566753408979664781552767087247326499087655884916892873679116538102030 87, 945668729170716248976092256181733012741703000045206290830219432406891995993 41, 966229002837177512607648657215833124481078701662406523729899467079451338710 43)
Bob's RSA Public Key: (11508412488184357718026749977523611349139914407642082242183310057700046369983625184405950344846039309812413526383140192044494100843979923484799491955568377, 869108801074159292379831106130263837614669466273953359592561459946258860274091700479215027370556380325268306046516517813200552138009094046424679771559627)
Bob's RSA Private Key: (56073517033817612806784964143106034567566139386277675047517062069522147496725744981020583992757152336966318601434500257623335706579779734847406211902960 3, 1036496692720364528265757915220809533024378312078311369552969366760685104935217, 11103183028958493302285132468267190683859756909970113600143942583826242113748 1)
```

Детальніше про те, що відбувається:

У функції створення ключів ми для кожної знайденої пари (p, q) та (p1, q1), обчислюємо n = p * q та phi_n = (p - 1) * (q - 1). Аналогічно для (p1, q1), це частина алгоритму Ейлера.

n представляє собою 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 кожної пари (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 (найбільший спільний дільник) дорівнює 1.

Для кожної пари (e, phi_n) та (e1, phi_n1), використовуючи розширений алгоритм 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.

Зашифрування відкритого повідомлення M, $0 \le M \le n-1$, здійснюється за формулою

$$C = M^e \bmod n ,$$

Розшифрування криптограми $0 \le C \le n-1$ здійснюється за допомогою особистого ключа d за формулою:

$$M = C^d \bmod n .$$

Цифровий підпис у системі RSA. Щоб підписати відкрите повідомлення M, абонент A зашифровує його на своєму секретному ключі $S = M^d \bmod n$ і додає S до повідомлення M. Підписане повідомлення (M, S) може передаватися як у відкритому, так і в зашифрованому виді. Перевіряється підпис за допомогою відкритого ключа: якщо виконується рівність $M = S^e \bmod n$, то повідомлення M не спотворене і підпис вірний. На практиці повідомлення M, що може бути набагато довше за n, попередньо піддається стисненню за допомогою геш-функції $H(M) = m$, і до повідомлення M додається підпис $S = m^d \bmod n$. Функція H повинна мати властивості *протидії колізіям*: m повинне істотно залежати від кожного біта M і за значенням m практично неможливо підібрати відповідне йому значення M.

Шифрування:

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

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

Дешифрування:

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

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

Створення та перевірка цифрового підпису:

```
def create_digital_signature(message, private_key):
    d, p, q = private_key
    n = p * q
    hashed_message = int.from_bytes(hashlib.sha256(message.encode()).digest(),
byteorder='big')
    signature = pow(hashed_message, d, n)
    return (message, signature)

def verify_digital_signature(signed_message, public_key):
    n, e = public_key
    message, signature = signed_message
    hashed_message = int.from_bytes(hashlib.sha256(message.encode()).digest(),
byteorder='big')
    decrypted_signature = pow(signature, e, n)
    return decrypted_signature == hashed_message
```

message: Повідомлення, для якого ми хочемо створити підпис.

private_key: Секретний ключ, який складається з d, p та q.

Спершу генерується хеш (hashed_message) від відкритого повідомлення.

Далі генерується підпис за допомогою операції залишку від ділення (pow(hashed_message, d, n)), де d - приватний експонент, n - модуль.

signed_message: Підписане повідомлення у форматі (message, signature).

public_key: Відкритий ключ, який складається з n та e.

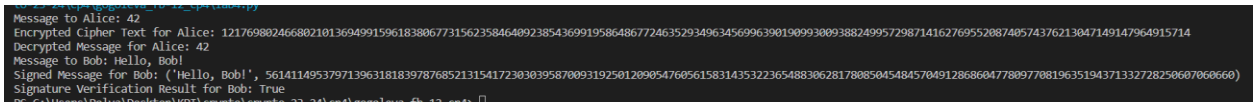
Знову генерується хеш (hashed_message) від відкритого повідомлення.

Перевірка підпису виконується порівнянням з допомогою операції залишку від ділення (pow(signature, e, n)) та порівнянням результату з гешем повідомлення. Якщо вони співпадають, це свідчить про вірність підпису.

Приклад використання:

```
p, q, p1, q1 = generate_key_pairs()
alice_public_key, alice_private_key, bob_public_key, bob_private_key =
generate_rsa_keys(p, q, p1, q1)

message_to_alice = 42
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 = create_digital_signature(message_to_bob,
bob_private_key)
signature_verified = verify_digital_signature(signed_message_for_bob,
bob_public_key)

print("Message to Alice:", message_to_alice)
print("Encrypted Cipher Text for Alice:", cipher_text_for_alice)
print("Decrypted Message for Alice:", decrypted_message_for_alice)

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

Вивід:



Message to Alice: 42

Encrypted Cipher Text for Alice:
12176980246680210136949915961838067731562358464092385436991958648677246352934963456996390190993009388249957298714162769552087405743762130471491479649157 14

Decrypted Message for Alice: 42

Message to Bob: Hello, Bob!

Signed Message for Bob: ('Hello, Bob!',
5614114953797139631818397876852131541723030395870093192501209054760561583143532236548830628178085045484570491286860477809770819635194371332728250607060660)

Signature Verification Result for Bob: True

Підсумовуючи, давайте глянем на основні етапи коду:

Функція generate_key_pairs генерує дві пари ключів для Alice і Bob, використовуючи алгоритм RSA. Публічні ключі (public_key) використовуються для шифрування, приватні (private_key) - для розшифрування та цифрового підпису.

Alice вибирає повідомлення (message_to_alice), шифрує його використовуючи свій публічний ключ (alice_public_key), отримуючи cipher_text_for_alice. Потім розшифровує його, використовуючи свій приватний ключ (alice_private_key), отримуючи decrypted_message_for_alice.

Аналогічно, Bob вибирає повідомлення (message_to_bob), створює цифровий підпис з використанням свого приватного ключа (bob_private_key), отримуючи signed_message_for_bob. Потім він перевіряє цифровий підпис, використовуючи публічний ключ Alice (bob_public_key), отримуючи результат signature_verified.

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 передає свій відкритий ключ (e_A, n_A) Бобу, і на навпаки.

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 автентичності.

```
def send_key(public_key_B, private_key_A, public_key_A):
    k = random.randint(0, public_key_A[0] - 1)
    print("k 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 recv_key(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])
```

```
k_1, S_1 = send_key(bob_public_key, alice_private_key, alice_public_key)
key_received = recv_key(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:
20667600461771710213865951041574063530029126276831399844849128190163009922664650428764217958694957808645135728020548973185662868909356717609658265989501 4

Message to Alice: 42

Encrypted Cipher Text for Alice:
20655913718564340571916404592135320276091363718198323937094424895049649953725513604066770018994972083100766218142690586208597333637997565034693953191507 91

Decrypted Message for Alice: 42

Message to Bob: Hello, Bob!

Signed Message for Bob: ('Hello, Bob!',
53882312238316409055519432552202311948910065503900906506683758286 3

289746554742982577220067022768113013355974230069492554058578229316239798700514927456100 1)

Signature Verification Result for Bob: True

Key exchange successful!

Висновки

Через два з половиною роки навчання я нарешті зрозуміла як працює RSA і тепер маю власний код, що допоможе мені розв’язувати схожі задачі на CTF, котрі раніше здавались мені складними. Все ще відчуваю себе трохи заплутано у цій кількості ключів, але до мене дійшло навіщо ми всі ці роки вчили виш мат і де він знадобиться у нашій сфері, я у це не вірила, чесне слово.