



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 коректно

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

Генерується 4 випадкових числа, сортується в порядку зростання, двома найменшими призначимо  $p$  та  $q$ , найбільшим -  $p_1$  і  $q_1$ .

```
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, 77398254598063414899754434965925
277334551875959802008192945620664460485862089)
Bob's public key: (94142084179036305219378742486346941102448909016919053596742768220541943831617, 9520228513123069642451040480333484
1299079209237141829825550608472371971478723)

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 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, "\n")
```

```

Alice's RSA public key: (54603499732885495511277678124910643550503084792271647442985104149100736725910201042860773322775582180307459
99962443827469373487973488172613292324785575577, 53616831124031530701751111699489951277233861226840078484084191996990192609086390299
10565734540656259785875109055306338580933790414325905209602392315838737)
Alice's RSA private key: (2603555898423913235137939279495392797681764047428181808391215793271635732184678177975614863238185418666299
741182647075292929047938053421822389859782835713, 61370235285361198727093316183550883795068807646768720263043021098465954136861, 889
73912970984176454394325596474744412398551230855837110608307879853269904557)
Bob's RSA public key: (9837250043477728006038966842644755740832127267597618753664907325848168950472376386433773014817758793394403464
278038539543979963386275823489454692565911783, 9157955267184236780088580331320743585386950026848810849782141500650387166957343270520
432311124170123020023466349797294707887798706627230256633348748678117)
Bob's RSA private key: (-21679074885565653210894658085268096883344921827508313030718479235351113286055346782386504255635655718333411
75984669252636390324787406258934895542348320787, 91414334271025097833658299420562420138029017750295912614205718280679705763303, 1076
11679524047749830892405367053299158886632166193805144733545211731305340161)

```

У функції створення ключів RSA для кожної знайденої пари  $(p, q)$  та  $(p_1, q_1)$ , обчислюється  $n = p * q$  та  $\phi_n = (p - 1) * (q - 1)$ . Аналогічно для  $(p_1, q_1)$ , 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  $(p_1, q_1)$ . 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  $(p_1, q_1)$ .

Генеруємо випадкові числа  $e$  та  $e_1$  так, щоб вони були взаємно простими з  $\phi_n$  та  $\phi_{n_1}$  відповідно. Це гарантує, що їхній GCD дорівнює 1.

Для кожної пари  $(e, \phi_n)$  та  $(e_1, \phi_{n_1})$  розширеним алгоритмом Евкліда знаходимо  $d$  та  $d_1$ , такі що  $(e * d) \% \phi_n = 1$  та  $(e_1 * d_1) \% \phi_{n_1} = 1$ .

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  $A$  і  $B$ , перевірити правильність розшифрування. Скласти для  $A$  і  $B$  повідомлення з цифровим підписом і перевірити його.

Для шифрування використовується операція залишку від ділення  $(\text{pow}(\text{message}, e, n))$ , 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  $(\text{pow}(\text{cipher\_text}, d, n))$ , 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: 34771247410169902544791404769321088466954135882051320968090038166185046
05455047600622612249893888032161486818052190565214587592924404965601675784349702157
Decrypted Message for Alice: 1211

Message to Bob: Hello, Bob!
Signed Message for Bob: ('Hello, Bob!', 4891128992772138807445019542391009478639751681421940194734226314
757050445443702503814155489605661713429838698100198378775341485238302490573294911015885980)
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, private_key_B[0], 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])

```

```
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: 34307859548798037099134598343925005359686492551819824289075146300609149820079724984818966717035
64289878440219949644365216652408639234018629444743529129043

Message to Alice: 1728
Encrypted Cipher Text for Alice: 31676062496814223212584328408832359565080201981158301084489897116772039
41958339032158973221684221868191690749511438936468631585035573163657794537740456699
Decrypted Message for Alice: 1728

Message to Bob: Hello, Bob!
Signed Message for Bob: ('Hello, Bob!', 2600258187927207127998731042191663548861380154475035356683078682
390384633107764580516165618748671866210617738425548006330026448898297319950927313361232179)
Signature Verification Result for Bob: True

Key exchange successful!
```

## Тестове середовище

<http://asymcryptwebservice.appspot.com/>

Для успішної перевірки функція RSA потрібно було дещо підлаштувати їх під формат який очікує сервер:

```
def Encrypt(M_, pubkey):
    if not isinstance(M_, int):
        M_ = int.from_bytes(M_.encode('utf-8'), byteorder='big')

    if 0 <= M_ <= pubkey[0] - 1:
        C_ = pow(M_, pubkey[1], pubkey[0])
        return C_
    else:
        return None

def Decrypt(C_, privkey, text=False):
    n_ = privkey[1] * privkey[2]

    if 0 <= C_ <= n_ - 1:
        M_ = pow(C_, privkey[0], n_)

        if text:
            return M_.to_bytes((M_.bit_length() + 7) // 8, byteorder='big').decode('utf-8')
        else:
            return M_
    else:
        return None
```

```
def Sign(message, private_key):
    d, p, q = private_key
    n = p * q

    message_int = int.from_bytes(message.encode('utf-8'), byteorder='big')

    signature = pow(message_int, d, n)
    return (message.encode('utf-8').hex(), signature)

def Verify(signed_message, public_key):
    message, signature = signed_message
    n, e = public_key
    if not isinstance(signature, int):
        signature = int(signature, 16)

    message_bytes = bytes.fromhex(message)
    message_int = int.from_bytes(message_bytes, byteorder='big')

    decrypted_signature = pow(signature, e, n)

    return decrypted_signature == message_int
```

## Тестування шифрування/дешифрування.

```
#Testing environment
print("\n\n\n==Testing environment==\n")
s = requests.Session()
url = "http://asymcryptwebservice.appspot.com/rsa/"

#Encryption

#Getting public key from server
payload = {'keySize': '256'}
r_serverKey= s.get(url+"serverKey", params=payload)
server_pub_key = (int(r_serverKey.json()['modulus'], 16), int(r_serverKey.json()['publicExponent'], 16))
print("Server's public key:")
print(r_serverKey.json(), '\n')

message_to_server = 'hello!488'
print('Message:', message_to_server+'\n')

#Encrypting message with my implementation
cipher_text_for_server = Encrypt(message_to_server, server_pub_key)
print("My cipher text for server: ", hex(cipher_text_for_server)[2:])

#Encrypting message with server's implementation
payload = {'modulus': r_serverKey.json()['modulus'], 'publicExponent': r_serverKey.json()['publicExponent'], 'message': message_to_server.encode('utf-8').hex()}
r_encrypt = s.get(url+"encrypt", params=payload)
print("Cipher text generated by server: ", r_encrypt.json()['cipherText'])

#Decrypting my cipher text using server
print('\nSending cipher text generated by server...')
payload = {'cipherText': r_encrypt.json()['cipherText'], 'expectedType': 'BYTES'}
r = s.get(url+"decrypt", params=payload)
print("Decrypted bytes:")
hexdump.hexdump(bytes.fromhex(r.json()['message']))

#Decrypting server's cipher text using server
print('\nSending my cipher text...')
payload = {'cipherText': hex(cipher_text_for_server)[2:], 'expectedType': 'BYTES'}
r = s.get(url+"decrypt", params=payload)
print("Decrypted bytes:")
hexdump.hexdump(bytes.fromhex(r.json()['message']))
```

## Тестування цифрового підпису.

```
#Signing
message_server = "kitty"
print("\n\nMessage: ", message_server)

print("\nServer signing message...")
payload = {'message': message_server.encode('utf-8').hex(), 'expectedType': 'BYTES'}
r = s.get(url+"sign", params=payload)
signature = r.json()['signature']
print("Server's signature:", r.json()['signature'])

print("Verifying with server public key...")
verified = Verify((message_server.encode('utf-8').hex(), signature), server_pub_key)
print(verified)

message_bob = "Disintegration"
print("\nMessage: ", message_bob)
print("\nBob signing the message...")
message_bob_bytes, signature_bob = Sign(message_bob, bob_private_key)
print(f"Signature: {hex(signature_bob)[2:]}")

print("Sending message with public key and signature to server...")
payload = {'modulus': hex(bob_public_key[0])[2:], 'publicExponent': hex(bob_public_key[1])[2:], 'message': message_bob.encode('utf-8').hex(), 'signature': hex(signature_bob)[2:]}
r = s.get(url+"verify", params=payload)
print("Server response:", r.json())
```



Тестування пройшло успішно.

```
==Testing environment==

Server's public key:
{'modulus': '85CBE917F6AC3610F0BEC485C85584EECD31379ADCEC7B1BE739650774FF3EA7', 'publicExponent': '10001'}

Message: hello1488

My cipher text for server:      2a5a08b5effc8de043b667ed09b8ca1de7559cf9e19c746976d2dcdbf87ef338
Cipher text generated by server: 2A5A08B5EFFC8DE043B667ED09B8CA1DE7559CF9E19C746976D2DCDBF87EF338

Sending cipher text generated by server...
Decrypted bytes:
00000000: 68 65 6C 6C 6F 31 34 38 38          hello1488

Sending my cipher text...
Decrypted bytes:
00000000: 68 65 6C 6C 6F 31 34 38 38          hello1488

Message: kitty

Server signing message...
Server's signature: 19b42f114557b9050cd600ce893dd74b7b6aa0372e17530b6d89132985914874
Verifying with server public key...
True

Message: Disintegration

Bob signing the message...
Signature: 8aff86126c77e6048ebd2acf2a4680976637e83275a7618a5542c58cfddd8de9984275c2dd1784a5e428c16ebca5ca369d5c1850ff8b4cca4755a40825b01205
Sending message with public key and signature to server...
Server response: {'verified': True}
```

## Висновок

Лабораторна 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.