



**Міністерство освіти і науки України  
Національний технічний університет  
України  
«Київський політехнічний інститут імені  
Ігоря Сікорського»**

**Лабораторна робота з криптографії**

**Вивчення крипtosистеми RSA та алгоритму електронного  
підпису; ознайомлення з методами генерації параметрів для  
асиметричних крипtosистем**

**Виконали:**

Студенти групи ФБ-33

Бондар Марина Вікторівна,

Романовська Крістіна Миколаївна

**Перевірив:**

к.ф.-м.н., ст. викл. кафедри  
математичних методів  
захисту інформації Селюх  
П.В

**Київ 2025**

**Мета роботи:** ознайомлення з тестами перевірки чисел на простоту та методами генерації ключів для асиметричної криптосистеми типу RSA; практичне дослідження системи захисту інформації на основі схеми RSA, організація шифрованого зв'язку та електронного підпису, а також вивчення протоколу конфіденційної розсылки ключів.

## Порядок виконання роботи

0. Уважно прочитати методичні вказівки до виконання комп'ютерного практикуму.
1. Реалізувати підпрограми із необхідними математичними операціями: обчисленням оберненого елементу за модулем із використанням розширеного алгоритму Евкліда, розв'язуванням лінійних порівнянь. При розв'язуванні порівнянь потрібно коректно обробляти випадок із декількома розв'язками, повертаючи їх усі.
2. За допомогою програми обчислення частот біграм, яка написана в ході виконання комп'ютерного практикуму №1, знайти 5 найчастіших біграм запропонованого шифртексту (за варіантом).
3. Переbrати можливі варіанти співставлення частих біграм мови та частих біграм шифртексту (розглядаючи пари біграм із п'яти найчастіших). Для кожного співставлення знайти можливі кандидати на ключ  $(a, b)$  шляхом розв'язання системи (1).
4. Для кожного кандидата на ключ дешифрувати шифртекст. Якщо шифртекст не є змістовним текстом російською мовою, відкинути цього кандидата.
5. Повторювати дії 3-4 доти, доки дешифрований текст не буде змістовним.

Для виконання даного завдання, у першу чергу була виконання генерація простих чисел. Необхідно знайти два великих, випадкових, простих числа  $p$  і  $q$  (довжиною  $\geq 256$  біт є високим рівнем криптографічної стійкості).

### Розширений алгоритм Евкліда (функція egcd)

Ця функція обчислює найбільший спільний дільник чисел  $a$  і  $b$ , а також знаходить такі коефіцієнти  $x$  і  $y$ , що виконується рівність  $ax + by = \text{gcd}(a, b)$ .

```
def egcd(a: int, b: int) -> Tuple[int, int, int]:  
    if b == 0:  
        return a, 1, 0  
    g, x1, y1 = egcd(b, a % b)  
    return g, y1, x1 - (a // b) * y1
```

### Обернений елемент за модулем (функція mod\_inverse)

Ця функція використовує розширений алгоритм Евкліда, щоб знайти число  $x$ , що  $ax$  дорівнює 1 за модулем  $m$ .

Обернений елемент існує за умови, коли числа  $a$  і  $m$  є взаємно простими.

```
def mod_inverse(a: int, m: int) -> Optional[int]:  
    g, x, _ = egcd(a, m)  
    return None if g != 1 else x % m
```

## Тест Міллера-Рабіна та Генерація Простих Чисел

Для генерації ключів RSA необхідні великі прості числа. Використовується імовірнісний тест Міллера-Рабіна.

Перевірка, чи є число  $n$  сильно псевдопростим за випадковими основами (базами)  $a$ .

Якщо число не проходить перевірку хоча б один раз — воно точно складене.

Якщо проходить усі перевірки — воно вважається імовірно простим.

```
def miller_rabin(n: int, rounds: int = 8) -> bool:
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0:
        return False

    d = n - 1
    r = 0
    while d % 2 == 0:
        d //= 2
        r += 1

    for _ in range(rounds):
        a = secrets.randrange(n - 3) + 2
        x = pow(a, d, n)

        if x in (1, n - 1):
            continue

        composite = True
        for _ in range(r - 1):
            x = pow(x, 2, n)

            if x == n - 1:
                composite = False
                break
            if x == 1:
                return False

        if composite:
            return False

    return True
```

## Генерація випадкового простого числа

Функція генерує випадкове число заданої бітової довжини, гарантує що воно непарне та має необхідну довжину.

Потім число перевіряється тестом Міллера-Рабіна доти, доки не буде знайдено просте.

```

def random_prime(bits: int, rounds: int = 8, trackfails: int = 30) -> int:
    if bits < 2:
        raise ValueError("bits must be >= 2")

    fails: List[int] = []

    while True:
        candidate = secrets.randbits(bits) | (1 << (bits - 1)) | 1

        if miller_rabin(candidate, rounds):
            if fails:
                print(
                    f"Failed candidates (last {len(fails)-trackfails}:)\n"
                    f"{fails[-trackfails:]} (total {len(fails)})"
                )
            return candidate
        else:
            fails.append(candidate)

```

## Генерація ключових пар RSA

Функція виконує:

1. Генерацію двох великих простих чисел  $p$  і  $q$ .
2. Обчислення модуля  $n = p \cdot q$ .
3. Обчислення значення функції Ейлера  $\phi = (p - 1)(q - 1)$ .
4. Вибір відкритої експоненти  $e$  (звичайно число 65537).
5. Обчислення закритої експоненти  $d$  як оберненого до  $e$  за модулем  $\phi$ .

У підсумку функція повертає відкритий ключ ( $n, e$ ) та закритий ключ ( $d, p, q$ ).

```

def make_keypair(bits: int = 256) -> Tuple[Tuple[int, int], Tuple[int, int, int]]:
    p = random_prime(bits)
    q = random_prime(bits)
    while q == p:
        q = random_prime(bits)

    n = p * q
    phi = (p - 1) * (q - 1)

    e = 65537
    if math.gcd(e, phi) != 1:
        while True:
            e = secrets.randrange(phi - 2) + 2
            if math.gcd(e, phi) == 1:
                break

    d = mod_inverse(e, phi)
    if d is None:

```

```
    raise RuntimeError("Failed to compute modular inverse")  
  
    return (n, e), (d, p, q)
```

#### 4. Операції RSA

Операції RSA базуються на швидкому піднесенні до степеня за модулем.

##### Шифрування

Піднесення повідомлення до степеня  $e$  за модулем  $n$ .

$$C \equiv M^e \pmod{n}$$

```
def rsa_encrypt(m: int, e: int, n: int) -> int:  
    return pow(m, e, n)
```

##### Розшифрування

Піднесення шифротексту до степеня  $d$  за модулем  $n$ .

$$M \equiv C^d \pmod{n}$$

```
def rsa_decrypt(c: int, d: int, n: int) -> int:  
    return pow(c, d, n)
```

##### Цифровий підпис

Створюється за допомогою секретного ключа (операція розшифрування, застосована до повідомлення або його гешу)

$$S \equiv M^d \pmod{n}$$

```
def rsa_sign(m: int, d: int, n: int) -> int:  
    return pow(m, d, n)
```

**Перевірка підпису** Перевірка відбувається шляхом піднесення підпису до степеня  $e$  за модулем  $n$  і порівняння результату з оригінальним повідомленням.

$$M \equiv S^e \pmod{n}$$

```
def rsa_verify(m: int, s: int, e: int, n: int) -> bool:  
    return m == pow(s, e, n)
```

##### Протокол конфіденційного розсилання ключів

Цей протокол забезпечує одночасно:

конфіденційність переданого ключа (через шифрування відкритим ключем отримувача)  
автентифікацію відправника (через цифровий підпис)

##### Створення Цифрового Підпису (Автентифікація):

**Конфіденційність.** А шифрує ключ k відкритим ключем B ( $e_1$ ,  $n_1$ )

$$k_1 = k^{e_1} \pmod{n_1}$$

**Автентифікація.** А підписує ключ k своїм секретним ключем d за своїм модулем n

$$S = k^d \pmod{n}$$

**Конфіденційність Підпису.** А шифрує підпис S відкритим ключем B ( $e_1$ ,  $n_1$ )

$$S_1 = S^{e_1} \pmod{n_1}$$

```
def send_key(msg: int, pub_sender, sec_sender, pub_receiver):  
    n_s, e_s = pub_sender  
    d_s = sec_sender[0]  
    n_r, e_r = pub_receiver  
  
    if not (0 < msg < n_r):  
        raise ValueError("Message must be in range (0, n_receiver)")  
  
    signature = rsa_sign(msg, d_s, n_s)  
    encrypted_msg = rsa_encrypt(msg, e_r, n_r)  
    encrypted_sig = rsa_encrypt(signature, e_r, n_r)  
  
    return encrypted_msg, encrypted_sig
```

### Розшифрування (Конфіденційність)

**Відновлення Ключа.** В розшифровує  $k_1$  і отримує оригінальний секретний ключ k.

$$k = k_1^{d_1} \pmod{n_1}$$

**Відновлення Підпису.** В розшифровує  $S_1$  і отримує оригінальний цифровий підпис S.

$$S = S_1^{d_1} \pmod{n_1}$$

### Перевірка Підпису (Автентифікація)

Якщо рівність виконується, це означає: 1) підпис S був створений за допомогою секретного ключа d Абонента A; 2) повідомлення k не було змінено після підписання.

$$k = S^e \pmod{n}$$

```
def receive_key(enc_msg, enc_sig, pub_sender, sec_receiver, pub_receiver):  
    n_r, e_r = pub_receiver  
    d_r = sec_receiver[0]  
    n_s, e_s = pub_sender
```

```

msg = rsa_decrypt(enc_msg, d_r, n_r)
sig = rsa_decrypt(enc_sig, d_r, n_r)
ok = rsa_verify(msg, sig, e_s, n_s)

return msg, ok

```

### Демонстрація протоколу (функція main\_demo)

Функція демонструє повну роботу системи:

1. Генерує ключові пари для А та В.
2. Створює випадковий ключ k для передавання.
3. Виконує протокол send\_key.
4. Виконує протокол receive\_key.
5. Перевіряє, чи ключ отримано правильно та чи пройшла перевірка підпису.

```

def main_demo():
    print("Generating RSA keys...\n")

    pubA, secA = make_keypair(256)
    pubB, secB = make_keypair(256)

    nA, eA = pubA
    dA = secA[0]
    nB, eB = pubB
    dB = secB[0]

    k = secrets.randrange(min(nA, nB) - 2) + 1

    print("Subscriber A")
    print(f'n_A = {nA}')
    print(f'e_A = {eA}')
    print(f'd_A = {dA}\n')

    print("Subscriber B")
    print(f'n_B = {nB}')
    print(f'e_B = {eB}')
    print(f'd_B = {dB}\n')

    print("Original transmitted key")
    print(f'k = {k}\n')

    enc_msg, enc_sig = send_key(k, pubA, secA, pubB)
    sig = rsa_sign(k, dA, nA)

    print("Sender A sends:")
    print(f"signature = {sig}")

```

```

print(f"encrypted_message = {enc_msg}")
print(f"encrypted_signature = {enc_sig}\n")

received_k, ok = receive_key(enc_msg, enc_sig, pubA, secB, pubB)

print("Receiver B gets:")
print(f"received key = {received_k}")
print(f"signature valid = {ok}")

print("\nCompleted" if received_k == k and ok else "\nProtocol failed")

if __name__ == "__main__":
    main_demo()

```

**Перевірка:** При виконанні перевірки, всі НЕХ були згенеровані через незалежний ресурс <https://www.rapidtables.com/>

From To

Decimal Hexadecimal

Enter decimal number

2190047884677918167192234269777468810

= Convert × Reset ↕ Swap

Hex result

(2190047884677918167192234269777468810)<sub>10</sub> =  
688697498856538535287722985016934  
003919448920805342591015839167672  
728499419809127187671532882873002  
74216534016236700194723)<sub>10</sub> =  
(29D0BB279BDC177E654B06D49CFAD E4D60C39F4914E619669AEA1875E26 5DCFFE6C0613B28A7657C89AAB75F 1EEE829CA564AC08093F996097FA41 80646E57A3)<sub>16</sub>

Hex number (128 digits)

29D0BB279BDC177E654B06D49CFAD E4D60C39F4914E619669AEA1875E26 5DCFFE6C0613B28A7657C89AAB75F 1EEE829CA564AC08093F996097FA41 80646E57A3	16	↙
--	----	---

Для кожного отриманих значень була здійснена перевірка, яка завершилась успішно, що і слугує підтвердженням коректності передачі.

## RSA Testing Environment

Verify

Message: 29D0BB279BDC177E654B06D49CFADE4D60C Bytes

Signature: D888454B7589B08BFBBB5018ABFB42FF17649C522E686B818F

Modulus: A384E37E5C8E802086D33ACED4A16425F5FF11FD560DFF32BC

Public exponent: 10001

Verify

Verification: true ✓

## Verify

Message: 244E6941F0B6E7CE29094F6127841B641DD1 Bytes

Signature: 23ACAD4902D4E238A08D9C15F7E135844662C8776A51CED327

Modulus: 5B01F2036248BD0DF282CAC904256A5B51C77A21A3DB676EF

Public exponent: 10001

Verify

Verification: true ✓

### Висновки:

У ході виконання лабораторної роботи було реалізовано повний цикл криптографічних операцій на основі алгоритму RSA: генерацію простих чисел, побудову відкритих та закритих ключів, шифрування і розшифрування даних, формування та перевірку цифрового підпису, а також протокол автентифікованої передачі секретного ключа між двома сторонами.

Усі етапи були успішно протестовані, а отримані результати підтвердили коректність роботи реалізованих алгоритмів. Тестування була здійснене ресурсом із вказівок до виконання даної роботи.