Міністерство освіти і науки України Національний технічний
університет України "Київський політехнічний інститут імені Ігоря Сікорського"

Фізико-технічний інститут

КОМП’ЮТЕРНИЙ ПРАКТИКУМ №4

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

Виконали:

ФБ-21 Ємець Валерія

Тютюннікова Віолета

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.

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.

Будемо реалізовувати функцію пошуку випадкового простого числа з заданого інтервалу, у якості датчика випадкових чисел будемо використовувати метод random.randint(). Для тесту перевірки на простоту використовуємо тест Міллера-Рабіна із попереднім пробним діленням. Для цього спочатку треба реалізувати тест пробних ділень, який використовує ознаку Паскаля. Знаємо, що натуральне число N (число яке ми тестуємо на простоту) записане у системі числення з основою B (у нашому випадку десятична система численна => основа 10):

$$N = \overline{a_t a_{t-1} \ldots a_1 a_0} = a_t B^t + a_{t-1} B^{t-1} + \ldots + a_1 B + a_0,$$

Це реалізовано у частині кода, яка виділена блакитним. Ми записуємо залишки від ділення у num_parts =[]. Після чого обчислюємо послідовність $r_i = B^i \bmod m$. - виділено зеленим та наприкінці розраховуємо залишок за формулою

$$N \equiv \sum_{i=0}^{t} a_i r_i \pmod m,$$

```python
#Подільність Паскаля
def pascal(N, m, B=10):
    num_parts = []
    while N > 0:
        num_parts.append(N % B)
        N //= B

    r = [1]
    for i in range(1, len(num_parts)):
        r.append((r[i-1] * B) % m)

    lishok = sum(d * r[i] for i, d in enumerate(num_parts)) % m
    return lishok == 0
```

Далі виконуються пробні ділення на малих числах з використанням функції pascal():

```python
#Пробні ділення
def trial(N):
    small = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    for prime in small:
        if pascal(N, prime):
            return False
    return True
```

Тест Міллера-Рабіна реалізовано відповідно до трьох кроків з методички:

```python
#Міллер-Рабін
def miller_rabin(p, k=10):
    # Крок 0
    if p < 2 or p % 2 == 0:
        return p == 2
    s = 0
    d = p - 1
    while d % 2 == 0:
        d //= 2
        s += 1
    # Крок 1
    def check(x):
        x = pow(x, d, p)
        if x == 1 or x == p - 1:
            return False
        for _ in range(s - 1):
            x = pow(x, 2, p)
            if x == p - 1:
                return False
        return True
    # Крок 2
    counter = 0
    for _ in range(k):
        x = random.randint(2, p - 2)

        if gcd(x, p) != 1:
            return False
        if check(x):
            return False
        counter += 1
    # Крок 3
    if counter < k:
        return False
    return True
```

Крок 0. Знаходимо розклад $p-1=d\cdot 2^s$ та встановлюємо лічильник у 0.

Крок 1. Вибираємо випадкове число $x\in N$ з інтервалу $1<x<p$ (незалежне від раніше обраних x). За допомогою алгоритму Евкліда знаходимо $\gcd(x,p)$. Якщо $\gcd(x,p)=1$, то переходимо до кроку 2. Якщо $\gcd(x,p)>1$, то p – складене число, алгоритм завершує свою роботу.

Крок 2. Перевіряємо, чи є p сильно псевдопростим за основою x:

2.1 Якщо $x^d \bmod p=\pm 1$, то p є сильно псевдопростим за основою x, інакше

2.2 Для всіх послідовних r від 1 до $s-1$ виконати такі дії:

Обчислити $x_r=x^{d\cdot 2^r} \bmod p$ (тобто $x_r=x_{r-1}^2 \bmod p$).

Якщо $x_r=-1$, то p є сильно псевдопростим за основою x,

інакше якщо $x_r=1$, то p не є сильно псевдопростим за основою x,

інакше перейти до наступного значення r.

2.3 Якщо за кроки 2.1 та 2.2 не було встановлено, що p є сильно псевдопростим за основою x, то p не є сильно псевдопростим за основою x.

Якщо p виявилось не псевдопростим за основою x, то p – складене число, алгоритм завершує свою роботу; інакше лічильник збільшується на 1.

Крок 3. Якщо лічильник менший за k, то переходимо до кроку 1, інакше алгоритм завершує роботу.

Далі вже спробуємо знайти просте число на заданому інтервалі. Почнемо з малих значень, нехай це буде інтервалі 22-143:

```
(base) violetta@MacBook-Pro-Violetta ~ % /opt/anacond
Початок діапазона: 22
Кінець діапазона: 143
Випадкове просте число у заданому діапазоні:  31
```

Тепер перевіримо на більшому інтервалі:

```
(base) violetta@MacBook-Pro-Violetta ~ % /opt/anaconda
Початок діапазона: 2
Кінець діапазона: 4572
Випадкове просте число у заданому діапазоні:  3643
```

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.

```
#-----------------------------------------ДРУГЕ ЗАВДАННЯ----------
def gen_pairs(bit_length=256):
    start = 2**(bit_length - 1)
    end = 2**bit_length - 1

    p = find(start, end)
    q = find(start, end)
    p1 = find(start, end)
    q1 = find(start, end)

    while p * q > p1 * q1:
        p1 = find(start, end)
        q1 = find(start, end)

    print("---"*50)
    print("Перша пара:")
    print(f"p = {p}, q = {q}")
    print("\nДруга пара:")
    print(f"p1 = {p1}, q1 = {q1}")
    print("---"*50)
    return (p, q), (p1, q1)
```

Генеруємо пари на основі попередніх функцій та встановлюємо зазначені початок та кінець інтервалу на якому будуть обиратись прості числа (для того щоб вони були не менше 256 біт).

Перевіримо виконання:

```
Перша пара:
p = 103965443793626110778257901212625546964280399407308501294733345948392063947187, q = 113521834142449903569741415775275495911202071038929747543531844037713058613099

Друга пара:
p1 = 114575932199470110652388100590412887342401241819286464076302518894010292103991, q1 = 106373031785253483363227885780774183656113969003766973516757203970432528301897
```

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) , (e1, n1) та секретні d і d1.

Для реалізації генерації ключових пар, згідно до теоретичних вимог, нам необхідні деякі математичні функції – пошук оберненого елемента за модулем, який в свою чергу потребує розширеного алгоритму Евкліда та функція Ойлера.

Реалізацію пошуку оберненого елемента та розширеного алгоритму Евкліда візьмемо з комп'ютерного практикуму 4. Функцію Ойлера реалізовано відповідно до формули $\varphi(n) = (p-1)(q-1)$

```python
#Розширений Евкліда
def gcd_evc(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = gcd_evc(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

#Обернений за модулем
def obratn(a, m):
    gcd, x, _ = gcd_evc(a, m)
    if gcd != 1:
        return None
    return x % m

#Функція Ойлера
def oiler(p, q):
    return (p - 1) * (q - 1)
```

Далі реалізовано саму функцію генерації RSA відповідно до логіки, яка розписана у теоретичних відомостях:

```python
#RSA
def rsa_pairs(bit_length=256):
    (p, q), _ = gen_pairs(bit_length=256)
    n = p * q
    phi = oiler(p, q)
    e = 2 ** 16 + 1
    if gcd(e, phi) != 1:
        raise ValueError("e повинно бути взаємнопростим з функцією Ойлера")
    d = obratn(e, phi)
    return (d, p, q), (n, e)
```

Перевіримо як воно працює:

```
Абонент A:
Секретний ключ: d = 269351657496681678009101135332669675497307862968078095479351220620309065497489386896414158516828604097990006124110714723079309417371598141261347664621 4601
Відкритий ключ: n = 577879974379154323883931682531743625333651927041573121531745865250047311536629923682810592743998359669676727909641269125459067285756343833239613627730743, e = 65537
p = 913952840581767086943371117217916455009563392487229773007952207514745144519 49, q = 6322864252069186547715798096946317492917989964525256586754177586099343737907


Абонент B:
Секретний ключ: d1 = 722670883787204945713532103736505551248138167253226664097250657749222552451699318742476413562667739726020417074795722007951633584314531177479671410635507 3
Відкритий ключ: n1 = 924076282573938121226616070914470651711104346424106215927682601153313921527091239266165745247089370582286427052025473940381679114228534455048633771625406 1, e1 = 65537
p1 = 881982297106346584599861771674605863164269570927461900714676657732414722740 81, q1 = 104772656504069942737183165061515642699315073862708012869559522122569789173581
```

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

Операцію шифрування реалізовано за логікою відповідною до формули:

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

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

Операція розшифрування також реалізована відповідно до формули:

```
def decrypt(cipher, private, public):
    d, p, q = private
    n, _ = public
    return pow(cipher, d, n)
```

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

Створення цифрового підпису

```
def sign(message, private, public):
    d, p, q = private
    n, _ = public
    return pow(message, d, n)
```

$$S = m^d \bmod n$$

Перевірка цифрового підпису також відбувається за формулою:

```
def verify(message, sign, public):
    n, e = public
    return pow(sign, e, n) == message
```

$$M = S^e \bmod n$$

Тобто, якщо ця рівність виконується, то повідомлення не було змінено і відповідає початковому стану.

Переглянемо, чи працює:

```
Повідомлення від A до B:  6443
Зашифроване:  924198164039045557301661464197992424151637638043112909075443478616091863718356046515626409074370040528981645324494291731714457643038209051278237765285505


Повідомлення від B до A:  1631
Зашифроване:  8239615858130228445691439579601287975684487670976154879975849470242203524039250847352833360446273117552331662386901608160346791785345591114142443313969 95
Розшифрування повідомлення від B:  1631


Розшифрування повідомлення від A:  6443
Підпис повідомлення від A:  6201934235023526691812793452544467545534469395833232106924221782200589024513829986681500154712811109437329787022625854189148193134460200247595988522491 61


Підпис повідомлення від B:  2816822944025175999267249263079240042000377279429994317973844764805904191791066381225406611914611955136809572132464178438751095570261081002026295293089749
Перевірка цифрового підпису для A:  True


Перевірка цифрового підпису для B:  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 .

```
196  #----------------------------------------П'ЯТЕ ЗАВДАННЯ----------------------------------------
197  def send_key(k, sender_se, sender_pub, receiver_pub):
198      en_k = encrypt(k, receiver_pub)
199      signa = sign(k, sender_se)
200      en_s = encrypt(signa, receiver_pub)
201      return en_k, en_s
202
203  def receive_key(en_key, en_signa, receiver_se, sender_pub):
204      de_k = decrypt(en_key, receiver_se)
205      de_s = decrypt(en_signa, receiver_se)
206      valid_s = verify(de_k, de_s, sender_pub)
207      return de_k, valid_s
208
209  k = random.randint(1, public_b[1] - 1)
210  en_k, en_s = send_key(k, secret_a, public_a, public_b)
211  de_k, valid_s = receive_key(en_k, en_s, secret_b, public_a)
```

Тут ми генеруємо значення k відповідно до вимог – 0 < k < n (тобто менше за відкритий ключ, тому у коді граничний розмір генерації – public_b[1]-1). Також створюємо функції send_key та receive_key, які будуть відповідати за отримання та надсилання ключа.

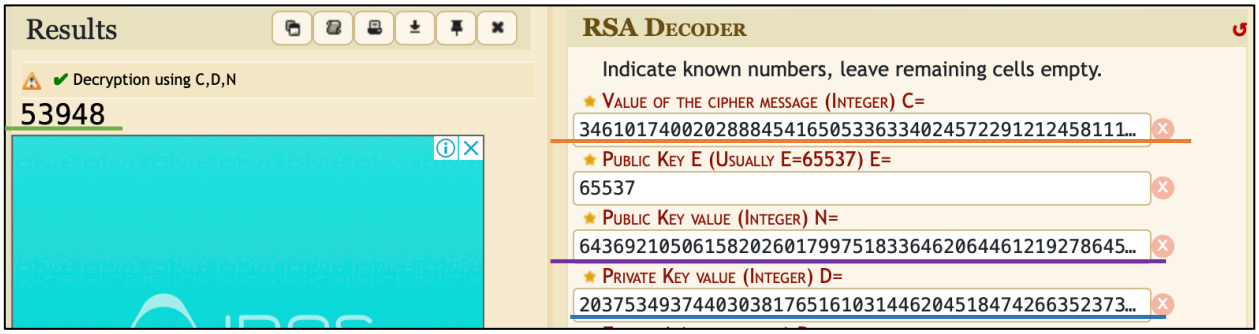
Тут для відправки ключа необхідно спочатку зашифрувати k, створити підпис та зашифрувати підпис.

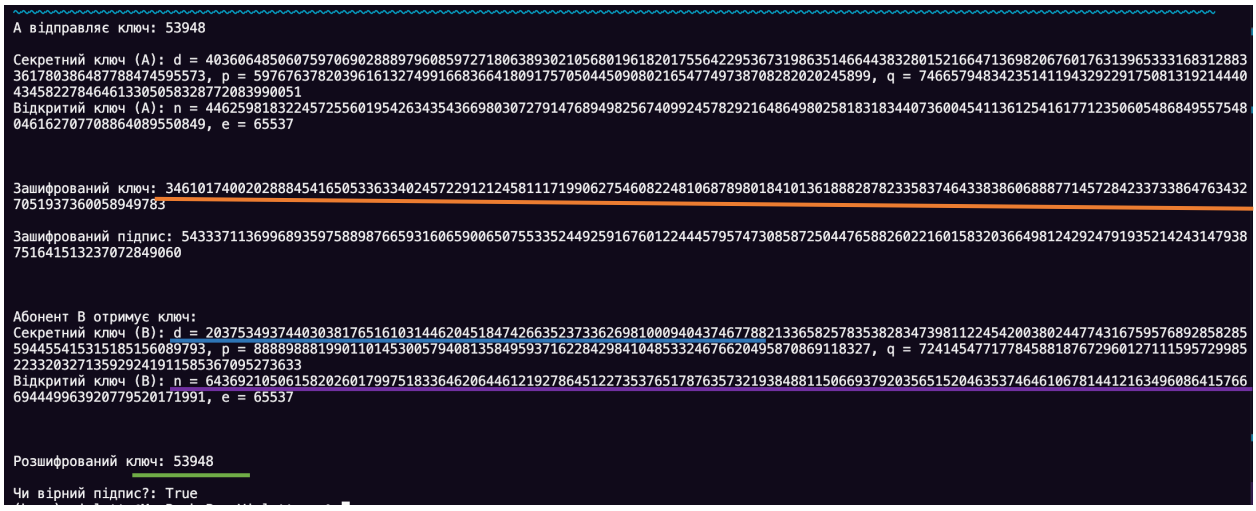
А для отримання ключа необхідно спочатку розшифрувати ключ, потім розшифрувати підпис та перевірити його.

Переглянемо як працює:

```
А відправляє ключ: 53948

Секретний ключ (A): d = 40360648506075970690288897960859727180638930210568019618201755642295367319863514664438328015216647136982067601763139653331683
36178038648778847459557З, p = 59767637820396161327499166836641809175705044509080216547749738708282020245899, q = 74665794834235141194329229175081319
43458227846461330505832877208З990051
Відкритий ключ (A): n = 44625981832245725560195426343543669803072791476894982567409924578292164864980258183183440736004541136125416177123506054868495
04616270770886408955084З, e = 65537



Зашифрований ключ: 34610174002028884541650533633402457229121245811171990627546082248106878980184101361888287823358374643383860688877145728423373386470
7051937360058949783

Зашифрований підпис: 54333711369968935975889876659316065900650755335244925916760122444579574730858725044765882602216015832036649812429247919352142431
751641513237072849060



Абонент В отримує ключ:
Секретний ключ (B): d = 20375349374403038176516103144620451847426635237336269810009404374677882133658257835382834739811224542003802447743167595768928
59445541531518515608979З, p = 88889888199011014530057940813584959371622842984104853324676620495870869118327, q = 72414547717784588187672960127111595
22332032713592924191158536709527З633
Відкритий ключ (B): n = 64369210506158202601799751833646206446121927864512273537651787635732193848811506693792035651520463537464610678144121634960864
69444996392077952017199З, e = 65537



Розшифрований ключ: 53948

Чи вірний підпис?: True
```

Тепер перевіримо це з використання сервісу https://www.dcode.fr/rsa-cipher





Бачимо, що все коректно працює.

Висновок: під час комп’ютерного практикуму ознайомились з механікою шифрування RSA. На основі цього ми реалізували відповідні функції (шифрування, дешифрування, створення цифрового підпису, його перевірку, надсилання та отримання ключа). Також набуті навички були протестовані на сторонньому ресурсі