Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Фізико-технічний інститут

ЛАБОРАТОРНА РОБОТА №5

з дисципліни

«Криптографія»

на тему: «Вивчення криптосистеми RSA та алгоритму електронного підпису; ознайомлення з методами генерації параметрів для асиметричних криптосистем»

Виконали:

студенти 3 курсу ФТІ

групи ФБ-74

Заіграєв Костянтин та Новіков Олексій

Перевірили:

Чорний О.

Савчук М. М.

Завадська Л. О.

Мета 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.

Порядок виконання 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.

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 $p\,q \le p_1\,q_1$; p і q – прості числа для побудови ключів абонента А, p_1 і q_1 – абонента В.

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

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.

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 $0 < k < n$.

Кожна з наведених операцій повинна бути реалізована у вигляді окремої процедури, інтерфейс якої повинен приймати лише ті дані, які необхідні для її роботи; наприклад, функція Encrypt(), яка шифрує повідомлення для абонента, повинна приймати на вхід повідомлення та відкритий ключ адресата (і тільки його), повертаючи в якості результату шифротекст. Відповідно, програмний код повинен містити сім високорівневих процедур: GenerateKeyPair(), Encrypt(), Decrypt(), Sign(), Verify(), SendKey(), ReceiveKey().

Кожну операцію рекомендується перевіряти шляхом взаємодії із тестовим середовищем, розташованим за адресою http://asymcryptwebservice.appspot.com/?section=rsa. Наприклад, для перевірки коректності операції шифрування необхідно а) зашифрувати власною реалізацією повідомлення для серверу та розшифрувати його на сервері, б) зашифрувати на сервері повідомлення для вашої реалізації та розшифрувати його локально.

Результати

Згенеровані прості числа (перші 10)

24505235934780533520919600819924642867909269112439058824870925313449164794057534
21143513298995559256415156102842550499174332468405712477669329138433522568 3

16622860636379660845656294090851997027439271107224830842356920549128752063686076
22584847469448009383511714475487181824325880907105371460610748547211491555 9

23013125132577568969405682466947337702288422529286469175410046804326818213288112
96617974856381578766635064334536935575884583870551938915476133207717831574 7

16986473205345729696561981361674820076288595709330198839662649157566353642909182
41695381383917023865399719350331344951266469173413122428289642231496127598 7

23560636353195036968655905847127625049832403923055902622107725493574457270412440
00568462777872102027713633834866892995504762308667367124904685756259061361 9

21319411961035507287818678376900936196034130053837704294843721764075364209392601
86391507110635374286627496972213421370779838841299083202784468370974270250 7

20377015188904890709674836627145283716582847696967675007572903439842600485495835
12972521132929336022013646842395161927760686381669245140238847759521134445 9

17707611776118177314932990695951839795469820815866522079032839768348385028646220
06921501829475342816122624610030913413996854304685241314792694919805682040 7

20388288778080618988570154424219316278546466742202198583345918576336991055170761
46333872870545363609345971629204652373701318838917957129801740032577307867 9

14234788968678112388220113094039973307116874166552839991411012526629613839354341
25495872754839503195930047520905107776458639284073602111460028269774824060 3

Використовувані під час лабораторного практикуму числа p та q

p:
24505235934780533520919600819924642867909269112439058824870925313449164794057534
21143513298995559256415156102842550499174332468405712477669329138433522568 3

q:
16622860636379660845656294090851997027439271107224830842356920549128752063686076
22584847469448009383511714475487181824325880907105371460610748547211491555 9

Використовувані відкриті та закриті ключі

e: 65537

N:
40734712180545967256197869877539230900963986785869734254672514473374951605644859
31536121358014437775666084844898671695754586104315028695749258905479273609540369
27123695408549388469698662949879957374641223011954323266666943252065877329986351
96906713630614584898644644516228231802761780869078774454367875310179 7

d:
14162082747665286234225986315512327022574491339488256938717262649737526471071579
71070548318390511691388860387125077964932301514973494801908034608867437465614939
30209492574502271762768237630239396253906880458387040437661587563707788376030723
44803858635001488978279307602595012119324445077657295335042870703505 3

Код програми

```
import math, random
import concurrent.futures


def miller_rabinTest(p, k):
    d = p - 1;
    s = 0;

    while (d & 1 == 0):
        d >>= 1;
        s += 1;

    for i in range(k):
        x = random.randint(2, p-2)

        if (math.gcd(x, p) != 1):
            return False

        x=pow(x, d, p)

        if (x==1 or x==p-1):
            continue

        for r in range(1, s):
            x = pow(x, 2, p);

            if (x == 1):
                return False;

            if (x == p - 1):
                break;

        if (x != p - 1):
            return False;

    return True;


def find_prime(n0, n1):
    i=0
    n0 = (n0/2)-1
    n1 = (n1/2)-1
    while True:
        i+=1
        x = random.randint(n0, n1)
        #print("New num!", countBits(x))
        x |= 1
        while (x < n1 and miller_rabinTest(x, 50) == False):
            x += 2
        x*=2
        x|=1
        if (miller_rabinTest(x, 50) == True):
            print(i)
            return x


def find_prime2(bit):
    n0 = pow(2, bit-2) + 1
    n1 = pow(2, bit-1) - 1
    x = random.randint(n0, n1)
    x |= 1
    while (x < n1 and miller_rabinTest(x, 50) == False):
        x += 2
    x *= 2
    x |= 1
    if (pow(2, x-1, x) == 1 and x % 3 != 0):
        #print(x)
        return x
    return False



def countBits(n):
    count = 0
    while (n):
        count += 1
        n >>= 1
    return count
```

```
def generate_safe_primes(bits, count):
    random_nums = list()
    while(len(random_nums) < count):
        with concurrent.futures.ProcessPoolExecutor() as executor:
            for data in concurrent.futures.as_completed({executor.submit(find_prime2, bits) for i in range(72)}):
                if data.result() != False:
                    #print((hex(data.result())[2:]).upper())
                    random_nums.append(data.result())
    return random_nums[:count]

def inverse(a, m):
    a%=m;
    if a == 1:
        return 1;
    try:
        return ((1 - m * inverse(m % a, a)) // a)%m;
    except:
        return;

def dec2hex(n):
    return (hex(n)[2:]).upper()

def hex2dec(h):
    return int(h, 16)

class RSA_Interface:
    def __init__(self, min_bit):
        temp_arr = generate_safe_primes((min_bit//2)+1, 10)
        print('safe_primes:\n')
        for i in range(10):
            print(temp_arr[i])
        self.p = temp_arr[0]
        self.q = temp_arr[1]
        temp_arr = []
        while (self.p == self.q):
            self.p = generate_safe_primes((min_bit//2)+1, 1)[0]
        print("\n\nKeys Generated Successfuly")
        print('p: {}\n q: {}'.format(self.p, self.q))
        self.N = self.p * self.q
        self.bits = countBits(self.N)
        self.e = pow(2, 16) | 1
        print('\n e:{} \n N:{}'.format(self.e, self.N))
        self.openKey = (self.e, self.N)
        self.d = inverse(self.e, (self.p - 1)*(self.q - 1))
        self.privateKey = (self.d, self.N)
        print('\n d: {}'.format(self.d))

    def Encrypt(self, M, e, N):
        return pow(M, e, N)

    def Encrypt_K(self, M, openKey):
        return pow(M, openKey[0], openKey[1])

    def Encrypt_Self(self, M):
        return pow(M, self.e, self.N)

    def Decrypt(self, C, d, N):
        return pow(C, d, N)

    def Decrypt_K(self, C, privateKey):
        return pow(C, privateKey[0], privateKey[1])

    def Decrypt_Self(self, C):
        return pow(C, self.d, self.N)

    def Sign(self, M, d, N):
        return (M, pow(M, d, N))

    def Sign_K(self, M, privateKey):
        return (M, pow(M, privateKey[0], privateKey[1]))

    def Sign_Self(self, M):
        return (M, pow(M, self.d, self.N))

    def Verify(self, M, S, e, N):
        return M == pow(S, e, N)

    def Verify_K(self, M, S, openKey):
        return M == pow(S, openKey[0], openKey[1])

    def Verify_Self(self, M, S):
        return M == pow(S, self.e, self.N)
```

```
    def SendKey(self, M, e, N1, d, N2):
        temp = self.Sign(M, d, N2)
        return (self.Encrypt(temp[0], e, N1), self.Encrypt(temp[1], e, N1))

    def SendKey_K(self, M, openKey, privateKey):
        temp = self.Sign_K(M, privateKey)
        return (self.Encrypt_K(temp[0], openKey), self.Encrypt_K(temp[1], openKey))

    def SendKey_Self(self, M, openKey):
        temp = self.Sign_Self(M)
        return (self.Encrypt_K(temp[0], openKey), self.Encrypt_K(temp[1], openKey))

    def ReceiveKey(self, M, e, N1, d, N2):
        verification = self.Verify(M, e, N1)
        return (verification, self.Decrypt(M, d, N2))

    def ReceiveKey_K(self, M, S, openKey, privateKey):
        verification = self.Verify_K(M, S, openKey)
        return (verification, self.Decrypt_K(M, privateKey))

    def ReceiveKey_Self(self, M, S, openKey):
        M1 = self.Decrypt_Self(M)
        S1 = self.Decrypt_Self(S)
        verification = self.Verify_K(M1, S1, openKey)
        return (verification, M1)



def get_KeyFromHex(e, N):
    return (hex2dec(e), hex2dec(N))

A=0
def main():
    global A
    A = RSA_Interface(1024)
    print("A open Keys:")
    for elem in A.openKey:
        print(dec2hex(elem))

if __name__ == '__main__':
    main()
```

Висновки:

Під час данного комп’ютерного практикуму, ми ознайомились з тестами перевірки чисел на простоту і методами генерації ключів для асиметричної криптосистеми типу RSA. Також, використовуючи криптосистему типу RSA, організували канал засекреченого зв’язку й електронний підпис, ознайомились із протоколом розсилання ключів.