

**НТУУ "КПІ ім Ігоря Сікорського"**

**Фізико-технічний інститут**

**КРИПТОГРАФІЯ**  
**КОМП'ЮТЕРНИЙ ПРАКТИКУМ №4**

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

**Виконали:**  
студенти групи ФБ-14  
Разумний Ілля  
Болгов Микола

**Перевірила:**  
Селюх П.В.

Київ 2023

## Варіант-1

### 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 зашифрованого зв'язку й електронного підпису, вивчення протоколу розсилання ключів

### Порядок виконання роботи:

Програма razik\_bolgov\_4.py на мові Python3

**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**

Деякі функції, такі як пошук GCD та оберненого за модулем було взято з коду ЛР 3

Генерація числа (`generate_prime()`) відбувається або за довжини 256 біт (дефолтне значення), або в інтервалі від найменшого до найбільшого числа з кількістю цифр 78 (що відповідає умові не менше 256 біт)

Під час генерації проходять 2 умови: перевірка діленням на пробні ділення (`if_prime_trial_div()`) та сам тест Міллера-Рабіна (`if_prime_mil_rab()`). У випадку коли кандидат не проходить - генеруємо заново

Також важливою функцією є схема Горнера швидкого піднесення до степеня (`horny_power()`), яка була основою майже усіх інших функцій, це дуже ПОТУЖНА функція в ЛР

**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 А,  $p_1$  і  $q_1$  – абонента В**

Функція (`generate_p_q()`) генерує 4 числа:  $p, q$  та  $p_1, q_1$  такі, що виконується умова  $pq \leq p_1q_1$ , інакше відкидає ці значення та генерує їх заново

Для демонстрації та тестування ми обрали згенеровані ключі які пройшли тести на простоту та умову множення та помістили їх в статичні змінні (згенеровані прості числа за довжиною 256 біт)

```
p = 62292051873732111696239942114403170513866646571234155956926929905487334616531
q = 107856527211449048964097072747761105850090651956211176122828103303232450011733
len of p = 256
len of q = 256
```

```
p_1 = 102212875132880649877270819301141701576819197983714203788291325382307197654689
q_1 = 79788335472330262274200476827445331604017075565917214625517614772660663877009
len of p_1 = 256
len of q_1 = 256
```

**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  $d_1$**

Функція (`GenerateKeyPair()`) генерує там пару закритих ( $d, p, q$ ) та відкритих ключів ( $n, e$ ), використовуючи функцію Ойлера а також функції, написані у 3 ЛР (знаходження НСД та оберненого за модулем,  $e$  обрано яу в сучасних криптографічних системах,  $2^{16} + 1$ , але за бажанням також можна і згенерувати випадкове яке буде виконувати умову  $\gcd(e, \phi) = 1$ . Пара  $n, e$  - відкритий ключ,  $d$  - секретний ключ

```
keys_a = GenerateKeyPair(p, q)
keys_b = GenerateKeyPair(p_1, q_1)

public_key_a = keys_a[1]
private_key_a = keys_a[0]

public_key_b = keys_b[1]
private_key_b = keys_b[0]
```

```
pubkey a: (6718604387976183222655375207256979782575136240588993379015074532069400977264362407304769208814668952153553931886400747085932929992551759014588528005758223, 65537)
privkey a: (5228326346747151144456434620154411845420441623821453885556067738153935920239529608466955370302949383107752895022945249119990191446674733822172241524273, 62292051873732111696239942114,
pubkey b: (8155395170693684926739014286725557699407615685484074772518097830208625883337347403116375028088023518357855381585142899679866265641253983801900906748145201, 65537)
privkey b: (4164494175541066264220941637846674610805732128880022679358082640123317726044243363904031199194466239329962447266062871568057623646208543142395739670406145, 102212875132880649877270819,
d = 5228326346747151144456434620154411845420441623821453885556067738153935920239529608466955370302949383107752895022945249119990191446674733822172241524273
d_1 = 4164494175541066264220941637846674610805732128880022679358082640123317726044243363904031199194466239329962447266062871568057623646208543142395739670406145
```

```
8223, 65537)
273, 62292051873732111696239942114403170513866646571234155956926929905487334616531, 107856527211449048964097072747761105850090651956211176122828103303232450011733)
5201, 65537)
06145, 102212875132880649877270819301141701576819197983714203788291325382307197654689, 79788335472330262274200476827445331604017075565917214625517614772660663877009)
```

Вийшли дуже довгими (другий скрін це продовження правої частини першої)

На цьому етапі ключі згенеровані

**4. Написати програму шифрування, розшифрування і створення повідомлення з цифровим підписом для абонентів А і В. 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 М і знайти криптограму для абонентів А и В, перевірити правильність розшифрування. Скласти для А і В повідомлення з цифровим підписом і перевірити його**

Шифрування (Encrypt()) приймає повідомлення, яке повинно бути менше за модуль ключа, та публічний ключ, та використовує піднесення до степеня по модулю для шифрування

Розшифрування (Decrypt()) приймає також повідомлення яке повинно задовольняти умову та приватний ключ, який використовується в піднесенні до степеня по модулю для розшифрування

Цифровий підпис (Sign()) створюється як пара повідомлення та його значення-підпису за приватним ключем

Перевірка підпису (Verify()) відбувається за наявності підписаної пари та публічного ключа

Перевіримо чи працюють функції. Число М оберемо невелике для зменшення навантаження. Приклади тривіальні для перевірки правильності без помилок

```

M = random.randint( a: 1, b: 25000)

print(M)
cryptogram_a = Encrypt(M, public_key_b) # A encrypts for B
cryptogram_b = Encrypt(M, public_key_a) # B encrypts for A

print(f"This is cryptogram for A using pubkey B: {cryptogram_a}")
print(f"This is cryptogram for B using pubkey A: {cryptogram_b}")

open_a = Decrypt(cryptogram_b, private_key_a) # A decrypts from B
open_b = Decrypt(cryptogram_a, private_key_b) # B decrypts from A

print(f"This is decrypted for A using privkey A: {open_a}")
print(f"This is decrypted for B using privkey B: {open_b}")

signed_a = Sign(M, private_key_a)
signed_b = Sign(M, private_key_b)

print(f"This is signed A: {signed_a}")
print(f"This is signed B: {signed_b}")

print(f"Verification of A is {Verify(signed_a, public_key_a)}")
print(f"Verification of B is {Verify(signed_b, public_key_b)}")

9910
This is cryptogram for A using pubkey B: 7112116277209745874744989614487076891264344439830.
This is cryptogram for B using pubkey A: 3177552553908818343060130299834377250741554390088.
This is decrypted for A using privkey A: 9910
This is decrypted for B using privkey B: 9910
This is signed A: (9910, 66739811991209644844350273138727618884054783362561860996613862727)
This is signed B: (9910, 34471517426942852816338757830900532933871840842914652072753918190.
Verification of A is True
Verification of B is True

```

“Симуляція” шифрування та розшифрування виконується правильно

**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  $0 < k < n$**

Будемо передавати секретне  $k$  в допустимих межах

```

k = random.randint( a: 0, public_key_a[0])

print(f"This is k: {k}")
k_encrypted = Encrypt(k, public_key_a)
print(f"This is k_encrypted to send: {k_encrypted}")
k_sign = Sign(k, private_key_a)
print(f"This is k_sign by user A: {k_sign}")

a = SendKey(k, public_key_a, private_key_a, public_key_b)
print(f"This is sent by A: {a}")

b = ReceiveKey(a, private_key_b, public_key_b, public_key_a)
print(f"This is received by B: {b}")

```

```

This is k: 3534190492993164549306638557297021187023912006293655598675375097495524597936941567494014565677856741189216631367654785878305026686328781703265315647813394
This is k_encrypted to send: 177766648730954390316462274168094076361308175891368164945214525266815557444164646002408039715522712499537486556056027485130369070865258438800408088000719
This is k_sign by user A: (3534190492993164549306638557297021187023912006293655598675375097495524597936941567494014565677856741189216631367654785878305026686328781703265315647813394, 3478097
This is sent by A: (579331764728956275458556581839145686972598721570113082572027295223045822443775194140283810894089117455293595715845428315687093086051336897080357896570357, 648446000831085
k is verified
This is received by B: (3534190492993164549306638557297021187023912006293655598675375097495524597936941567494014565677856741189216631367654785878305026686328781703265315647813394, 3478097888

```

Перевірка підпису відбувається при отриманні ключа

**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()`**

Відповідні функції було реалізовано

У примітивних прикладах ми тестували числа

Також ми реалізували шифрування та дешифрування стрінгів через їх ASCII значення

**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**

Для початку генеруємо публічний ключ довжиною 256 біт на сервері

## RSA Testing Environment

Server Key

Encryption

Decryption

Signature

Verification

Send Key

Receive Key

### Get server key

Clear

Key size256

Get key

Modulus

AD32E0612BF9A008D477B3577954200347BBB1BF7A95A05AAC6E647597D528FB

Public exponent

10001

Зашифруємо повідомлення на сервері

## RSA Testing Environment

Server Key

Encryption

Decryption

Signature

Verification

Send Key

Receive Key

### Encryption

Clear

Modulus

AD32E0612BF9A008D477B3577954200347BBB1BF7A95A05AAC6E647597D528FB

Public exponent

10001

Message

Hello Mario

Text

Encrypt

Ciphertext

21C65A58CAF2261744EED65089444287B5EF86F759C9A1012A15AECDBEDBAD35

Перевіряємо у себе

```
pubkey_serv = (int("AD32E0612BF9A008D477B3577954200347BBB1BF7A95A05AAC6E647597D528FB", 16), int("10001", 16))
message = 'Hello Mario'

ciphertext_serv = (int("21C65A58CAF2261744EED65089444287B5EF86F759C9A1012A15AECDBEDBAD35", 16))
ciphertext_our = Encrypt(message, pubkey_serv)

print(f"ciphertext encrypted by server: {ciphertext_serv}\nciphertext encrypted by us: {ciphertext_our}")

print(hex(15276783273093125926050098888444103147478315780042180554430612431664079154485)) # for server to decrypt

ciphertext encrypted by server: 15276783273093125926050098888444103147478315780042180554430612431664079154485
ciphertext encrypted by us: 15276783273093125926050098888444103147478315780042180554430612431664079154485
0x21c65a58caf2261744eed65089444287b5ef86f759c9a1012a15aecdbedbad35
```

Бачимо, що наша функція правильно його зашифрувала

Тепер спробуємо наш шифротекст отриманий через ключ сервера розшифрувати на сервері

## RSA Testing Environment

Server Key

Encryption

Decryption

Signature

Verification

Send Key

Receive Key

Decryption

Clear

Ciphertext

21C65A58CAF2261744EED65089444287B5EF86F759C9A1012A15AECDBEDBAI

Text

Decrypt

Message

Hello Mario

Отримали наш початковий текст

Тепер передамо серверу наші відкриті ключі, та нехай він зашифрує в себе  
Маємо наші ключі

```
pubkey a: (53542207555483371440770111932682546396213530793622988628365562554983437507597, 65537)
privkey a: (50556974383264071820028027476542930231890129445352399399662717658913877320117, 181207594698208181078258173571185530019, 295474412342678748021019178493214795663)

pubkey a in hex: ('0x765fd0c7dc9fdf5c8481495cac33518dba173714b849f9108f8afd0bf87f1c0d', '0x10001')
privkey a in hex: ('0x6fc63bc36ced3d2104317c4f587ec0c2eb53881b0da90235889f2c6bde84e1b5', '0x88535032fc1c9d00c0490334088054a3', '0xde4a4d463ea2e343898ead497819278f')
```

Дамо серверу відкритий ключ А та зашифруємо текст  
У себе розшифруємо

## RSA Testing Environment

Server Key

Encryption

Decryption

Signature

Verification

Send Key

Receive Key

Encryption

Clear

Modulus

765fd0c7dc9fdf5c8481495cac33518dba173714b849f9108f8afd0bf87f1c0d

Public exponent

10001

Message

Hello Mario

Text

Encrypt

Ciphertext

75985089BE1004238206DDDF5D32AD4828B8CBB0F0E1020C1B6A8174E47CA735

```
decrypted_with_ours = Decrypt(int("75985089BE1004238206DDDF5D32AD4828B8CBB0F0E1020C1B6A8174E47CA735", 16), private_key_a, text=True)

print(decrypted_with_ours)
```

Hello Mario



## Тепер генеруємо підпис на сервері

### RSA Testing Environment

Server Key

Encryption

Decryption

Signature

Verification

Send Key

Receive Key

#### Sign

Clear

Message

Hello Mario

Text

▼

Sign

Signature

93F348EB1AABA82D622607BB7C8726E40DE24658146B59D0F757737C18BA90DA

## Та перевіримо його на сервері

### RSA Testing Environment

Server Key

Encryption

Decryption

Signature

Verification

Send Key

Receive Key

#### Verify

Clear

Message

Hello Mario

Text

Signature

93F348EB1AABA82D622607BB7C8726E40DE24658146B59D0F757737C18BA90DA

Modulus

AD32E0612BF9A008D477B3577954200347BBB1BF7A95A05AAC6E647597D528FB

Public exponent

10001

Verify

Verification

true

А тепер у нас

```
signed = (message, int("93F348EB1AABA82D622607BB7C8726E40DE24658146B59D0F757737C18BA90DA", 16))

verified = Verify(signed, pubkey_serv)

print(verified)

True
```

Тепер генеруємо підпис у нас

```
generated_sign = Sign(message, private_key_a)
print(hex(generated_sign[1]))
```

0x759366647a79a16748b95da5c426a3176ba2367610e0a31cacb081e00dc250a3

Та перевіряємо на сервері

## RSA Testing Environment

[Server Key](#)  
[Encryption](#)  
[Decryption](#)  
[Signature](#)  
[Verification](#)  
[Send Key](#)  
[Receive Key](#)

### Verify

✖ Clear

Message

Hello Mario

Text ▼

Signature

759366647a79a16748b95da5c426a3176ba2367610e0a31cacb081e00dc250a3

Modulus

765fd0c7dc9df5c8481495cac33518dba173714b849f9108f8afd0bf87f1c0d

Public exponent

10001

Verify

Verification

true

✓

Підпис справився та підтвердився

Тепер відішлемо на сервері наш відкритий ключ A

## RSA Testing Environment

[Server Key](#)  
[Encryption](#)  
[Decryption](#)  
[Signature](#)  
[Verification](#)  
[Send Key](#)  
[Receive Key](#)

### Send key

✖ Clear

Modulus

765fd0c7dc9df5c8481495cac33518dba173714b849f9108f8afd0bf87f1c0d

Public exponent

10001

Send

Key

5A338EC013E364F20E7E7D09DB62B54EBC18DAB3103C592EE0832306FE93DB6A

Signature

3FEBECC84ADC81FE2082B3DFBE6C642672D934B57329B7FDF66D2D00A42E5D87

Перевіряємо

```

k = int("5A338EC013E364F20E7E7D09DB62B54EBC18DAB3103C592EE0832306FE93DB6A", 16)
print(k)
S = int("3FEBECC84ADC81FE2082B3DFBE6C642672D934B57329B7FDF66D2D00A42E5D87", 16)

received = ReceiveKey(received: (k, S), private_key_a, pubkey_serv)

print(hex(received[0]), hex(received[1]))

k is verified
0x1ad65f0dead386c4 0x6a8e94432f2ca7aa04e5f0b153666fe29a3e47a143a331f12999dde7f8fe8721

```

Тепер відправляємо ключ на сервер

```

sent = SendKey(message, public_key_a, private_key_a, pubkey_serv)
print(hex(sent[0]), hex(sent[1]))

0x21c65a58caf2261744eed65089444287b5ef86f759c9a1012a15aecdbedbad35 0x7a298cb0953e712c4d8678d65275474a08b896fbcadc1d19f38d2a1d21edbd61

```

## RSA Testing Environment

Server Key  
Encryption  
Decryption  
Signature  
Verification  
Send Key  
Receive Key

### Receive key

Clear

Key

21c65a58caf2261744eed65089444287b5ef86f759c9a1012a15aecdbedbad35

Signature

7a298cb0953e712c4d8678d65275474a08b896fbcadc1d19f38d2a1d21edbd61

Modulus

765fd0c7dc9fdf5c8481495cac33518dba173714b849f9108f8afd0bf87f1c0d

Public exponent

10001

Receive

Key

48656C6C6F204D6172696F

Verification

true

```

key_ = 0x48656C6C6F204D6172696F

print(key_.to_bytes((key_.bit_length() + 7) // 8, byteorder='big').decode('utf-8'))

Hello Mario

```

Програма успішно пройшла перевірки на сайті сервера

**Хід роботи, опис труднощів, що виникали, та шляхів їх розв'язання;**

Було важко стриматись та не перейменувати основні 7 функцій відповідно до правил Python PEP-8

Також виникли невеликі проблеми із розумінням як перевірити функції `SendKey()`, `ReceiveKey()` використовуючи тестове середовище на сайті. Проблемою не проходження верифікації у `ReceiveKey()` на сервері було те, що наш розмір  $p$  і  $q$  був занадто великий і добуток був більший, ніж `pubkey` сервера. Тобто модулі  $n$  відкритих ключів відрізняється розмірами бітів в 2 рази

Було важко зрозуміти як переводити текст у цифрове представлення для шифрування, скористались байтами

**– значення вибраних чисел  $p$ ,  $q$ ,  $1/p$ ,  $1/q$  із зазначенням кандидатів, що не пройшли тест перевірки простоти, і параметрів криптосистеми RSA для абонентів A і B;**

Виведемо лише раз, оскільки самі функції генерації перевіряють правильність вибору

```
These candidates are invalid:  
p = 71483681129768929500055446104400467696024028183069426405340221458696565747569  
q = 100107097896615098918894114209683998801984966918873909086788155023554078647297  
p_1 = 104270290177636513848971796297354761673037921922799343299317347163308923470441  
q_1 = 67011042990235860874492843227610858141645997533383701968356856113882004662787
```

Їх виводить дуже рідко

**– чисельні значення прикладів ВТ, ШТ, цифрового підпису для A і B;**

Описали коли розробляли функції для пунктів вище

**– опис кроків протоколу конфіденційного розсилання ключів з підтвердженням справжності, чисельні значення характеристик на кожному кроці;**

Описано вище

## **Висновки:**

В ході виконання лабораторної роботи було вивчено роботу криптосистеми RSA та алгоритму електронного підпису та методами генерації параметрів для асиметричних криптосистем

Ми знову зрозуміли практичну важливість алгоритму Евкліда та лінійних конгруенцій, знаходження оберненого елемента за модулем.

Ключові етапи побудови криптосистеми RSA:

1. Написання допоміжних математичних функцій
2. Написання функції, що вирішує схему Горнера швидкого піднесення до степеня
3. Написання тестів перевірок натуральних чисел на простоту (Міллера-Рабіна)
4. Написання функції генерації простого числа
5. Написання функцій шифрування-дешифрування, цифрового підпису та верифікації
6. Експлуатація функцій та використання криптосистеми

Одним з основних етапів побудови тестів на простоту є використання псевдопростих чисел. Зокрема, такі числа, що зберігають деякі властивості простих чисел використовуються у імовірнісному тесті Міллера-Рабіна. При виконанні лабораторної роботи було написано функції `if_prime_mil_rab()`, `if_prime_trial_div()` та `generate_prime()`, що слугують для генерації простих чисел. Для того, щоб просте число згенерували - кандидат має пройти 2 тести

Робота RSA відбувається наступним чином:

Абонент А генерує великі прості числа  $p$  та  $q$  та обчислює їх добуток та функцію Ойлера  $n=pq$  та  $\phi(n)=(p-1)(q-1)$  відповідно.

Далі А обирає випадкове число  $e$  (зазвичай це  $2^{16}+1$ ), НСД між числом  $e$  та  $\phi$  повинно бути 1.

Далі знаходиться обернений за модулем елемент  $d$ , який і є секретним ключем абонента  $d$ . Пара чисел  $n$  та  $e$  - наш публічний ключ.

Зашифрування відкритого повідомлення відбувається формулою:

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

А розшифрування за формулою:

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

Відкрите повідомлення підписується користувачем А за формулою

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

та цей підпис  $S$  додається до повідомлення, користувач  $B$  верифікує його за допомогою відкритого ключа за формулою

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

Розсилання та отримання повідомлень відбуваються наступним чином:

Абонент  $A$  має відкритий ключ  $(e, n)$ , секретний  $d$  і відкритий ключ  $(e_1, n_1)$  абонента  $B$ . Повідомлення  $(k_1, S_1)$  формується абонентом  $A$  та відправляється абоненту  $B$  за формулою:

$$k_1 = k^{e_1} \bmod n_1, \quad S_1 = S^{e_1} \bmod n_1, \quad S = k^d \bmod n$$

Абонент  $B$  розшифровує повідомлення за допомогою свого секретного ключа та перевіряє підпис абонента  $A$

$$k = k_1^{d_1} \bmod n_1, \quad S = S_1^{d_1} \bmod n_1, \quad k = S^e \bmod n$$

Кошенятко після ЛР 4 Сгурто

