НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ» ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

КРИПТОГРАФІЯ

КОМП'ЮТЕРНИЙ ПРАКТИКУМ №4

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

Виконали:

студентки групи ФБ-23

Гуз Вікторія

Шукалович Марія

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

Постановка задачі:

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 p1 , q1 довжини щонайменше 256 біт. При цьому пари чисел беруться так, щоб $pq \leq p1q1$; p і q – прості числа для побудови ключів абонента A, p1 і q1 – абонента B.
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 .
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.
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 $0 < k < n$.

ХІД РОБОТИ:

Першим кроком був пошук випадкового простого числа заданої довжини. В якості тесту перевірки на простоту ми використовували тест Міллера-Рабіна із попередніми пробними діленнями:

```
# Функція для перевірки числа на простоту за допомогою тесту Міллера-Рабіна
def is_prime_miller_rabin(p, k=10):
    if p < 2 or p % 2 == 0:
        return p == 2

    small_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
    for prime in small_primes:
        if p % prime == 0 and p != prime:
            return False

    s = 0
    d = p - 1
    while d % 2 == 0:
        d //= 2
        s += 1
```

```
    def check_composite(a):
        x = pow(a, 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

    for _ in range(k):
        a = random.randint(2, p - 2)
        if math.gcd(a, p) != 1 or check_composite(a):
            return False

    return True
```

```
♥Меню♥
1. Вибір випадкового простого числа
2. Генерація двох пар простих чисел
3. Генерація ключових пар RSA для абонентів A та B
4. Перехід до меню процедур
5. Надсилання ключа
6. Вийти
Виберіть опцію: 1
Випадкове просте число: 769
```

Перша функція is_prime_miller_rabin(p, k=10) перевіряє, чи є число p простим за допомогою тесту Міллера-Рабіна, який є методом для статистичної перевірки простоти числа. Спочатку функція перевіряє, чи є число парним або меншим за 2 (окрім числа 2). Якщо число парне, то воно не є простим, за винятком 2, яке є простим. Далі функція перевіряє, чи ділиться число на малі прості числа (від 2 до 31). Якщо число ділиться на одне з цих чисел, воно також не є простим. Потім функція розкладає число p-1 на вигляд $2^s \times d$, де d непарне, що необхідно для виконання тесту Міллера-Рабіна. Далі для випадкових чисел a функція виконує піднесення до степеня і перевіряє умови, щоб визначити, чи може це число бути свідченням складності числа p. Якщо для жодного з чисел не було знайдено свідчень, що p складене, то функція вважає число простим.

```
# Функція для генерації випадкового простого числа довжини щонайменше n біт
def generate_prime(bits, k=10):
    with open("lab4.txt", "w") as file:
        while True:
            candidate = random.getrandbits(bits) | (1 << (bits - 1)) | 1
            if is_prime_miller_rabin(candidate, k):
                return candidate
            else:
                file.write(f"{candidate}\n")
```

Функція generate_prime(bits, k=10) генерує випадкове просте число з кількістю біт не менше ніж вказано в параметрі bits. Спочатку генерується випадкове число, яке має правильну кількість біт і є непарним. Потім це число перевіряється на простоту за допомогою тесту Міллера-Рабіна. Якщо число не є простим, функція генерує нове число і перевіряє його, поки

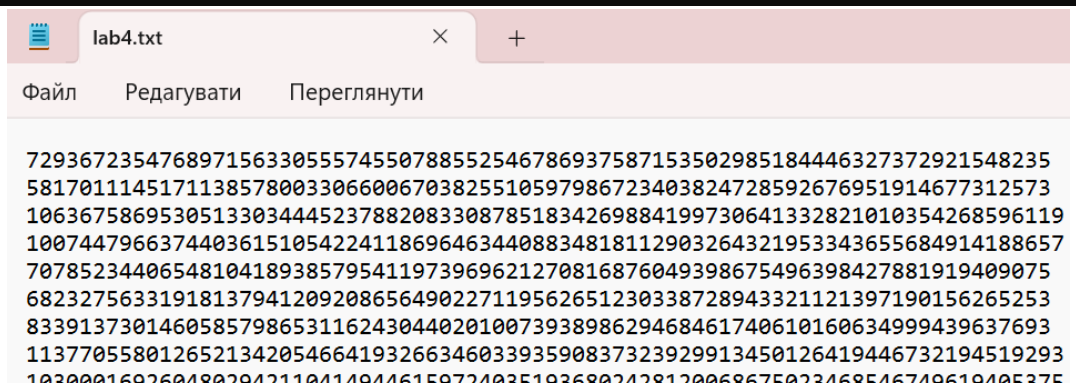
не знайде просте число. Ця функція зазвичай використовується для генерації великих простих чисел у криптографії, де важливим є те, щоб числа були великими і простими для безпеки, наприклад, в алгоритмі RSA. Числа, що не пройшли перевірку на простоту записані у файл lab4.txt. На цьому етапі труднощів не виникало.

Після цього ми реалізували функцію для генерації двох пар простих чисел:

```
def generate_prime_pairs(bits=256):
    while True:
        # Генерація простих чисел для абонента A
        p = generate_prime(bits)
        q = generate_prime(bits)
        pq = p * q

        # Генерація простих чисел для абонента B
        p1 = generate_prime(bits)
        q1 = generate_prime(bits)
        p1q1 = p1 * q1

        if pq <= p1q1:
            return (p, q), (p1, q1)
```

Функція generate_prime_pairs(bits=256) створює дві пари простих чисел для двох абонентів: A і B. Вона починає з генерації простих чисел p і q для абонента A, використовуючи функцію generate_prime(bits), після чого обчислює їх добуток pq=p·q. Аналогічно, для абонента B генеруються прості числа p1 і q1, а також обчислюється їх добуток p1q1=p1·q1. Функція продовжує генерувати числа, доки модуль абонента A не виявиться меншим або рівним модулю абонента B . Це гарантує, що модулі двох абонентів мають визначену залежність, необхідну для подальшого використання в криптографічних операціях.
Під час реалізації функції труднощів не виникало. Весь процес виявився зрозумілим і логічним.

```
♥Меню♥
1. Вибір випадкового простого числа
2. Генерація двох пар простих чисел
3. Генерація ключових пар RSA для абонентів A та B
4. Перехід до меню процедур
5. Надсилання ключа
6. Вийти
Виберіть опцію: 2
Пара простих чисел для абонента A: p = 774240734916022605471708989074949404639165912604702099158545822230163569181 23,
q = 958069005049720133562217913060019754111664405135704592883258522594052770495 17
Пара простих чисел для абонента B: p1 = 101852478149704697569302912223749335489701118117826822583477385282817923570281
, q1 = 112252080626814363507168479805729075622402150983370373510227786641163974356759
```


lab4.txt
Файл
Редагувати
Переглянути
729367235476897156330555745507885525467869375871535029851844463273729215482355817011145171138578003306600670382551059798672340382472859267695191467731257310636758695305133034445237882083308785183426988419973064133282101035426859611910074479663744036151054224118696463440883481811290326432195334365568491418865770785234406548104189385795411973969621270816876049398675496398427881919409075682327563319181379412092086564902271195626512303387289433211213971901562652538339137301460585798653116243044020100739389862946846174061016063499943963769311377055801265213420546641932663460339359083732392991345012641944673219451929310300016926048029421104149446159724035193689242812006867502346854674961940537 5
```

Дані p та q є просто прикладами пар випадкових простих чисел і не використовуватимуться для генерації пар ключів.
Далі написали функцію та процедуру генерації ключових пар для RSA:
```

```
def generate_rsa_key_pair(bits=256):
    (p, q), (p1, q1) = generate_prime_pairs(bits)
    n_a = p * q
    phi_a = (p - 1) * (q - 1)

    n_b = p1 * q1
    phi_b = (p1 - 1) * (q1 - 1)

    e = 2**16 + 1

    if math.gcd(e, phi_a) != 1 or math.gcd(e, phi_b) != 1:
        raise ValueError("Невдалий вибір e, знайдено спільний дільник з phi.")

    d_a = mod_inverse(e, phi_a)
    d_b = mod_inverse(e, phi_b)

    public_key_a = (e, n_a)
    private_key_a = (d_a, p, q)

    public_key_b = (e, n_b)
    private_key_b = (d_b, p1, q1)

    return (public_key_a, private_key_a), (public_key_b, private_key_b)

def GenerateKeyPair(bits=256):
    global rsa_keys
    (public_key_a, private_key_a), (public_key_b, private_key_b) =
generate_rsa_key_pair(bits)

    rsa_keys["A"] = {"public_key": public_key_a, "private_key": private_key_a}
    rsa_keys["B"] = {"public_key": public_key_b, "private_key": private_key_b}
```

Функція generate_rsa_key_pair генерує ключі RSA для двох абонентів (A і B). Спочатку обчислюються прості числа p,q для A і p1,q1 для B, потім відповідні модулі n і функції Ейлера. Вибирається спільний відкритий експонент і перевіряється, чи взаємно просте воно з обчисленими phi. Далі знаходяться приватні експоненти, які є оберненими до e. Повертаються пари ключів для обох абонентів: відкриті (e, n) та приватні (d, p, q). GenerateKeyPair викликає generate_rsa_key_pair, щоб згенерувати ключі, і зберігає їх у словник rsa_keys для абонентів A і B.

```
♥Меню♥
1. Вибір випадкового простого числа
2. Генерація двох пар простих чисел
3. Генерація ключових пар RSA для абонентів A та B
4. Перехід до меню процедур
5. Надсилання ключа
6. Вийти
Виберіть опцію: 3
Відкритий ключ абонента A: (65537, 550804694719657684122231778505852896234218061156031732423822883833346229029367732726140148947332024753636596571739126956785670005265785582927158543720254870)
Секретний ключ абонента A: (254823967284129895823520568904549488286334309081143203489483953153647201454199066582768789231611918264903325621748247727989485716497538052443809385604609, 67099219012785465658274687570165324519874173775819632784937544932085770951649, 8208809324810535192735129384369657481666340807385721236250223095991117878846 3)
Відкритий ключ абонента B: (65537, 650282993629004169623164812888965297490851609327280476626749090199867469774775769962378279070548592434899140258170769327683186562191923854887625233077884870)
Секретний ключ абонента B: (272438777424532210588571894769249739432783200898715840620419380596878269924771022739954263162134212434947570018873320817168805963153674950990620219141856 9, 589912378789614310926717884724429489749289341415717975333945493747813855165 43, 1102338274309921485908859567648533175681412873613229010258572164492281363883 09)
```

Після цього написали програму шифрування:

```
def Encrypt(user, message):
    global rsa_keys, messages
    e, n = rsa_keys[user]["public_key"]

    ciphertext = pow(message, e, n)
    messages[user] = {"message": message, "ciphertext": ciphertext}
```

Вона отримує два параметри: ім'я користувача (user) і повідомлення (message). Спочатку з глобального словника rsa_keys витягується публічний ключ користувача, який складається з показника e і модуля n. Потім повідомлення шифрується за допомогою піднесення його до степеня e за модулем n.

Із шифруванням проблем не виникло. Далі ми реалізували програму розшифрування:

```
def Decrypt(user):
    global messages, decrypted_message, rsa_keys
    if "ciphertext" not in messages.get(user, {}):
        print(f"Помилка: Криптограма для {user} відсутня!")
        return
    d, p, q = rsa_keys[user]["private_key"]
    n = p * q

    ciphertext = messages[user]["ciphertext"]
    decrypted_message = pow(ciphertext, d, n)

    if decrypted_message == messages[user]["message"]:
        pass
    else:
        print("Помилка розшифрування!")
```

Спочатку перевіряється, чи є в глобальному словнику messages криптограма для вказаного користувача. Якщо її немає, виводиться повідомлення про помилку. Далі отримуються приватний ключ користувача (d, p, q) і обчислюється модуль n = p * q. Використовуючи метод піднесення криптограми до степеня d за модулем n, отримується розшифроване повідомлення. Якщо розшифроване повідомлення не збігається з оригінальним текстом, виводиться повідомлення про помилку.

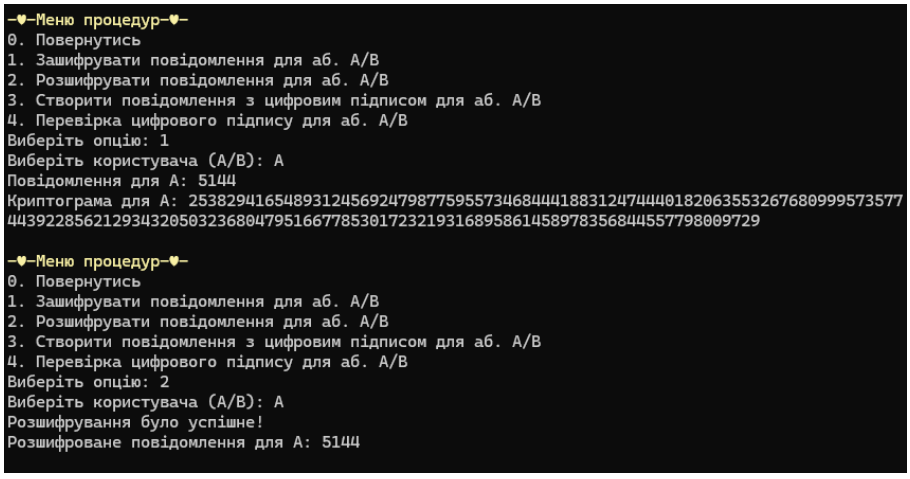
Також програма для створення повідомлення з цифровим підписом для абонентів A і B:

```
def Sign(user):
    global messages, signature, rsa_keys
    if "message" not in messages.get(user, {}):
        print(f"Помилка: Повідомлення для {user} відсутнє!")
        return
    d, p, q = rsa_keys[user]["private_key"]
    n = p * q

    message = messages[user]["message"]
    hash_value = hash(message)
    signature = pow(hash_value, d, n)

    messages[user]["signature"] = signature
    print(f"Цифровий підпис для {user}: {signature}")
```

Вона перевіряє, чи існує повідомлення для вказаного користувача в глобальному словнику messages. Якщо повідомлення немає, виводиться помилка. Потім із приватного ключа користувача (d, p, q) обчислюється модуль n = p * q.
Хеш повідомлення обчислюється за допомогою вбудованої функції hash. Далі цифровий підпис генерується шляхом піднесення цього хешу до степеня d за модулем n
Та його перевірка:

```
def Verify(user, signature):
    global messages, rsa_keys
    if user not in messages:
        print(f"Помилка: Повідомлення для {user} відсутнє!")
        return

    if "signature" not in messages[user]:
        print(f"Помилка: Підпис для {user} відсутній!")
        return

    e, n = rsa_keys[user]["public_key"]

    verified_hash = pow(signature, e, n)
    # print(f"Перевірений хеш для {user}: {verified_hash}")

    if "message" not in messages[user]:
        print(f"Помилка: Повідомлення для перевірки підпису відсутнє!")
        return

    message = messages[user]["message"]
    if verified_hash == hash(message):
        print("Підпис правильний!")
    else:
        print("Підпис неправильний!")
```

Спочатку вона перевіряє, чи існує повідомлення для заданого користувача у глобальному словнику messages. Якщо повідомлення або підпис відсутні, виводиться відповідна помилка. Далі з публічного ключа користувача (e, n) обчислюється перевірений хеш шляхом піднесення підпису до степеня e за модулем n. Якщо у словнику немає оригінального повідомлення для перевірки, виводиться помилка.
Функція обчислює хеш від повідомлення за допомогою вбудованої функції hash. Якщо перевірений хеш збігається з обчисленим хешем повідомлення, підпис вважається правильним
Для A:



```
-♥-Меню процедур-♥-
0. Повернутись
1. Зашифрувати повідомлення для аб. A/B
2. Розшифрувати повідомлення для аб. A/B
3. Створити повідомлення з цифровим підписом для аб. A/B
4. Перевірка цифрового підпису для аб. A/B
Виберіть опцію: 3
Виберіть користувача для підпису (A/B): A
Цифровий підпис для A: 259920398129505190903767819585572741241773566143093877879383932386880668164231080362695703049400223249554977391775209730287881165739471010728800286324280 5


-♥-Меню процедур-♥-
0. Повернутись
1. Зашифрувати повідомлення для аб. A/B
2. Розшифрувати повідомлення для аб. A/B
3. Створити повідомлення з цифровим підписом для аб. A/B
4. Перевірка цифрового підпису для аб. A/B
Виберіть опцію: 4
Виберіть користувача для перевірки підпису (A/B): A
Підпис правильний!
```

Для B:

```
-♥-Меню процедур-♥-
0. Повернутись
1. Зашифрувати повідомлення для аб. A/B
2. Розшифрувати повідомлення для аб. A/B
3. Створити повідомлення з цифровим підписом для аб. A/B
4. Перевірка цифрового підпису для аб. A/B
Виберіть опцію: 1
Виберіть користувача (A/B): B
Повідомлення для B: 5144
Криптограма для B: 5409858202964587711398548176347641646625168755621170967076345746713271487291070082291298297244810878285107873302208773197636885078362510253362668490035768

-♥-Меню процедур-♥-
0. Повернутись
1. Зашифрувати повідомлення для аб. A/B
2. Розшифрувати повідомлення для аб. A/B
3. Створити повідомлення з цифровим підписом для аб. A/B
4. Перевірка цифрового підпису для аб. A/B
Виберіть опцію: 2
Виберіть користувача (A/B): B
Розшифрування було успішне!
Розшифроване повідомлення для B: 5144

-♥-Меню процедур-♥-
0. Повернутись
1. Зашифрувати повідомлення для аб. A/B
2. Розшифрувати повідомлення для аб. A/B
3. Створити повідомлення з цифровим підписом для аб. A/B
4. Перевірка цифрового підпису для аб. A/B
Виберіть опцію: 3
Виберіть користувача для підпису (A/B): B
Цифровий підпис для B: 4573907105283196422000674737146004822579575580964613403955798784080325433727247181743995592511621199743386243259725090764849106418783670177424684507483974


-♥-Меню процедур-♥-
0. Повернутись
1. Зашифрувати повідомлення для аб. A/B
2. Розшифрувати повідомлення для аб. A/B
3. Створити повідомлення з цифровим підписом для аб. A/B
4. Перевірка цифрового підпису для аб. A/B
Виберіть опцію: 4
Виберіть користувача для перевірки підпису (A/B): B
Підпис правильний!
```

Після цього за допомогою раніше написаних на попередніх етапах програм ми організували роботу протоколу конфіденційного розсилання ключів з підтвердженням справжності по відкритому каналу за допомогою алгоритму RSA:

```python
def SendKey(sender, receiver):
    global rsa_keys, messages, storage

    if sender not in rsa_keys or receiver not in rsa_keys:
        print(f"Ключі для {sender} або {receiver} не згенеровані.")
        return

    n_receiver = rsa_keys[receiver]["public_key"][1]
    e_receiver = rsa_keys[receiver]["public_key"][0]

    k = random.randint(1, n_receiver - 1)
    print(f"\n{sender} відправляє ключ {k} абоненту {receiver}\n")
```

```
    messages[sender] = {"message": k}
    messages[receiver] = {"message": k}

    Encrypt(receiver, k)
    print(f"Зашифрований ключ: {messages[receiver]['ciphertext']}\n")
    Sign(sender)
    s1 = pow(messages[sender]["signature"],e_receiver, n_receiver)

    storage[receiver] = {
        "encrypted_key": messages[receiver]["ciphertext"],
        "s1": s1,
        "hash": hash(k),
        "sender": sender
    }
```

Спочатку перевіряється наявність RSA-ключів у обох користувачів, і якщо хоча б один з них їх не має, робота функції припиняється з відповідним повідомленням. Потім із публічного ключа отримувача беруться значення e та n. Генерується випадковий ключ k, який відправник хоче передати отримувачу. Цей ключ зберігається як повідомлення в словниках messages для обох користувачів. Для забезпечення конфіденційності ключ шифрується публічним ключем отримувача за допомогою функції Encrypt. Далі відправник підписує ключ цифровим підписом через функцію Sign, а підпис перетворюється для передачі з використанням публічного ключа отримувача. Усі необхідні дані, включаючи зашифрований ключ, підпис та хеш ключа, зберігаються у сховищі storage для отримувача.
Труднощів з реалізацією не виникло, оскільки кожен етап функції чітко відповідає стандартним операціям алгоритму RSA.

```
def ReceiveKey(receiver, sender):
    global rsa_keys, storage

    if receiver not in rsa_keys or sender not in rsa_keys:
        print(f"Ключі для {receiver} або {sender} не згенеровані.")
        return

    if receiver not in storage:
        print(f"Немає повідомлень для {receiver}.")
        return

    data = storage[receiver]

    if data["sender"] != sender:
        print("Дані не відповідають вказаному відправнику.")
        return

    Decrypt(receiver)
    d1, p1, q1 = rsa_keys[receiver]["private_key"]
    n1 = p1 * q1
    signature = pow(data["s1"], d1, n1)
    Verify(sender, signature)

    if data["hash"] == hash(decrypted_message):
        print(f"\n{receiver} успішно прийняв та розшифрував ключ:
{decrypted_message}")
```

```
        print("Підтвердження автентичності пройдено успішно :)")
    else:
        print(f"Помилка перевірки підпису або даних. Ключ відхилено.")
```

ReceiveKey відповідає за прийом та перевірку безпечного ключа, отриманого від іншого користувача (sender), за допомогою алгоритму RSA. Спочатку перевіряється, чи обидва користувачі мають згенеровані ключі. Якщо хоча б один із них їх не має, функція завершується повідомленням про помилку. Потім перевіряється, чи є в сховищі storage дані для отримувача (receiver), і чи співпадає відправник ключа з очікуваним (sender).
Дані, отримані зі сховища, розшифровуються за допомогою функції Decrypt. Підпис, переданий від відправника, перевіряється шляхом піднесення s1 до приватного ключа отримувача, після чого функція Verify підтверджує коректність підпису за оригінальним повідомленням. Якщо хеш розшифрованого повідомлення збігається з хешем ключа, отриманим під час передачі, ключ приймається, і користувач успішно завершує процес отримання та перевірки даних. У разі невідповідності повідомляється про помилку.
Труднощів із реалізацією цієї функції не виникло, оскільки її логіка заснована на вже відпрацьованих криптографічних операціях

```
♥Меню♥
1. Вибір випадкового простого числа
2. Генерація двох пар простих чисел
3. Генерація ключових пар RSA для абонентів A та B
4. Перехід до меню процедур
5. Надсилання ключа
6. Вийти
Виберіть опцію: 5
Виберіть відправника (A/B): A
Виберіть отримувача (A/B): B

A відправляє ключ 28756276560716693273062469020375533117848593389044415421607189923934870652717850352706405
0945799289041299461905532887430677282787394791033080702089107283 абоненту B

Зашифрований ключ: 57610603252546033857523910323948409095345985332305296838733774798227118235617193790878746
6989625707149068543021539501793479260755495610274925070177911587 2

Цифровий підпис для A: 54530291790740126206274204313081353729114540269369335632870889216873964176231372405985
8701248901334555153484506784625482041562357962054912217480417922794 93

Розшифрування було успішне!
Підпис правильний!

B успішно прийняв та розшифрував ключ: 28756276560716693273062469020375533117848593389044415421607189923934870
6527178503527064050094579928904129946190553288743067728278739479103308070208910728 3
Підтвердження автентичності пройдено успішно :)
```

Висновки: у ході роботи було вивчено принципи криптосистеми RSA, зокрема генерацію ключів, перевірку чисел на простоту та операції шифрування і розшифрування. Розглянуто алгоритм електронного підпису, його використання для автентифікації та цілісності даних. Досліджено протокол розсилання ключів, включаючи шифрування ключа публічним ключем отримувача, створення цифрового підпису відправником і перевірку його отримувачем. Це дозволило зрозуміти основи роботи сучасних криптосистем та методи забезпечення безпеки передачі інформації.