**Міністерство освіти і науки України**

**Національний технічний університет України**

**“Київський політехнічний інститут імені Ігоря Сікорського”**

**Фізико-технічний інститут**

**Методи реалізації криптографічних механізмів**

Лабораторна робота №4

**Виконали:**

студенти ФІ-31мн

Карабан А.

Шевченко Ю.

**Перевірила:**

Селюх П. В.

Київ – 2024

**Лабораторна робота №4. Дослідження особливостей реалізації існуючих програмних систем, які використовують криптографічні механізми захисту інформації.**

**Мета:**

Отримання практичних навичок побудови гібридних криптосистем.

**Завдання:**

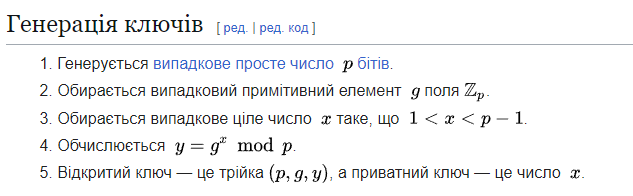
Для другого типу лабораторних робіт ― розробити реалізацію асиметричної криптосистеми у відповідності до стандартних вимог Crypto API або стандартів PKCS та дослідити стійкість стандартних криптопровайдерів до атак, що використовують недосконалість механізмів захисту операційної системи.

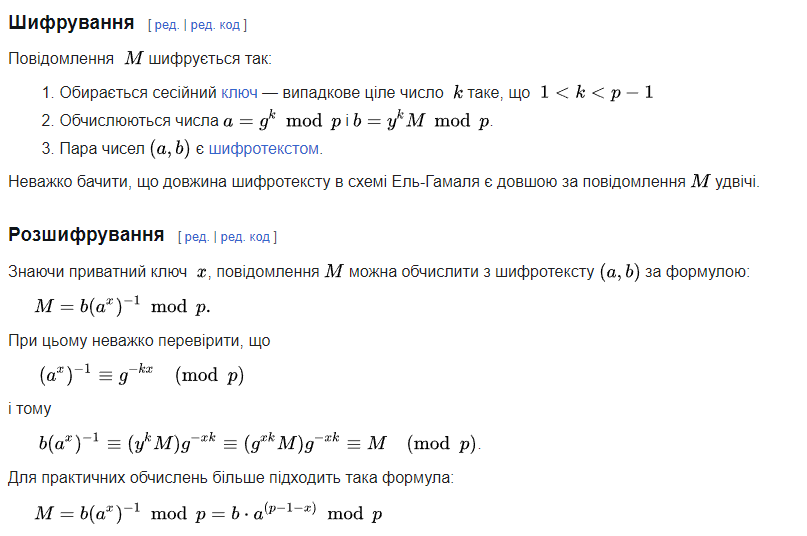
Бібліотека OpenSSL під Windows платформу. Кр\с Ель Гамаля.

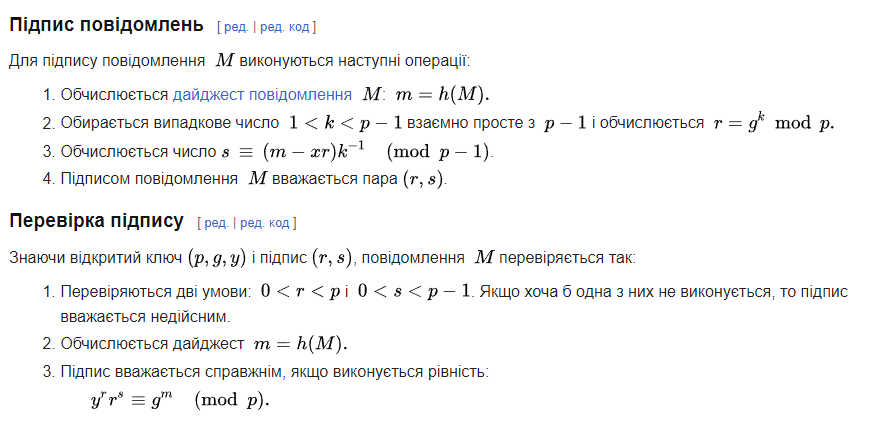
Оформлення результатів: контрольний приклад роботи з асиметричною криптосистемою. Приклад атаки або демонстрація їх неможливості.

**Хід роботи:**

Схема Ель Гамаля ― це криптографічна система, яка забезпечує шифрування, дешифрування, підпис та перевірку цифрового підпису на основі задачі дискретного логарифмування.







Алгоритм Ель Гамаля широко застосовується для забезпечення конфіденційності та аутентифікації у криптографічних протоколах, таких як передача зашифрованих даних, цифрові підписи та безпечний обмін ключами.

Функція *generate\_keys* генерує приватний та відкритий ключі. Використовується Diffie-Hellman для створення необхідних параметрів.

Функція *encrypt* реалізує шифрування. Вона створює випадковий ефемерний ключ, обчислює загальний ключ та зашифровує повідомлення.

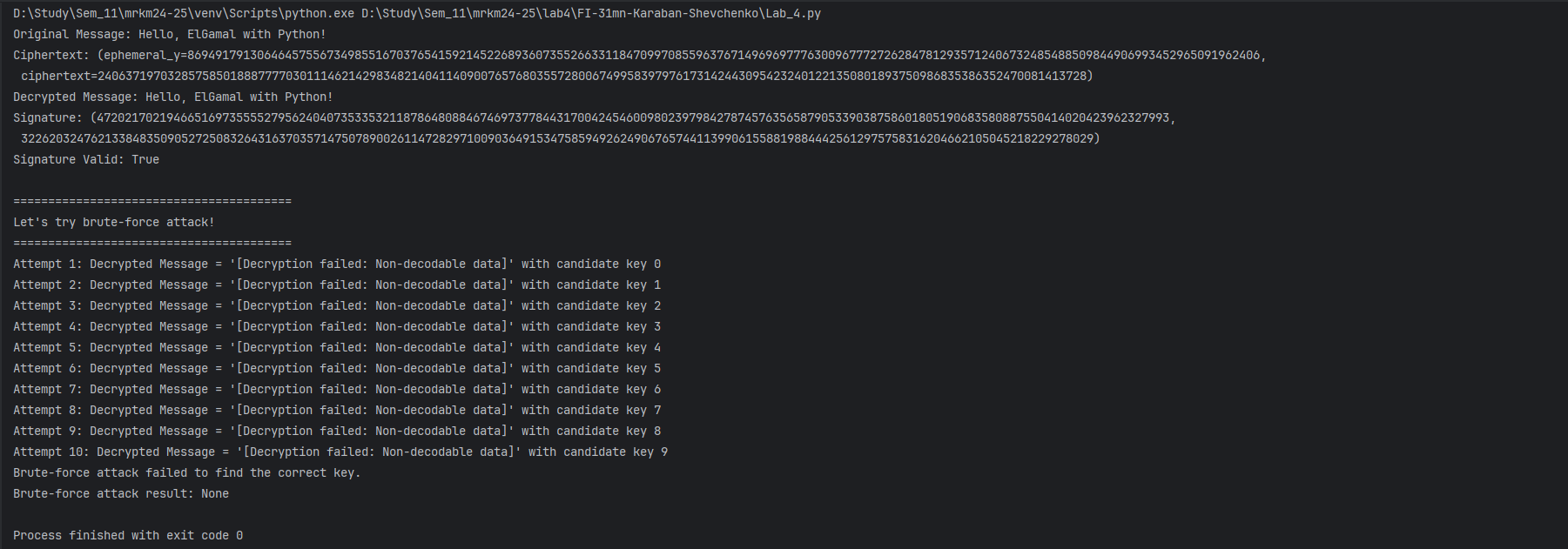
Функція *decrypt* приймає приватний ключ, ефемерний публічний ключ та шифротекст для дешифрування.

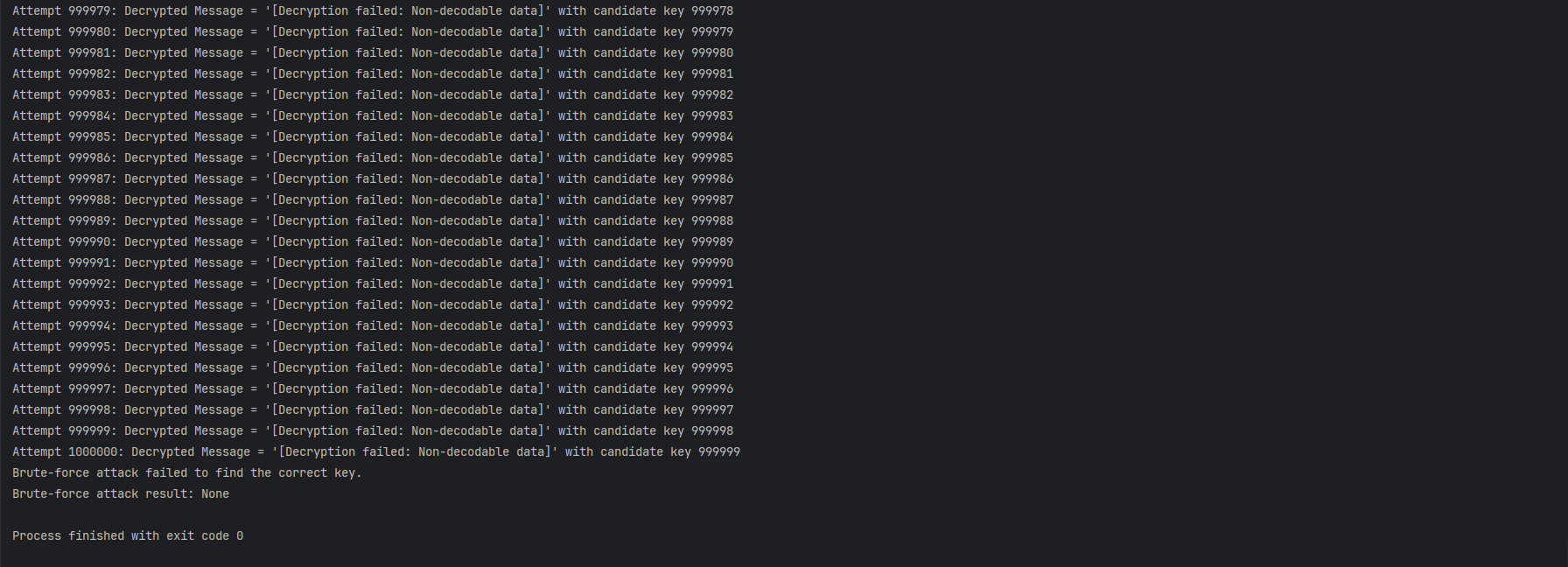
Функція *sign* виконує підпис повідомлення.

Функція *verify* перевіряє валідність підпису.

import os  
  
from cryptography.hazmat.primitives import hashes  
from cryptography.hazmat.primitives.asymmetric import dh  
from cryptography.hazmat.primitives.asymmetric.dh import DHPrivateKey, DHPublicKey  
from cryptography.hazmat.primitives.kdf.hkdf import HKDF  
  
  
def string\_to\_int(data: str) -> int:  
 *"""Convert a string to an integer using UTF-8 encoding.  
  
 Args:  
 data (str): The input string to convert.  
  
 Returns:  
 int: The integer representation of the input string.  
 """* return int.from\_bytes(data.encode('utf-8'), 'big')  
  
  
def int\_to\_string(data: int) -> str:  
 *"""Convert an integer back to a string using UTF-8 decoding.  
  
 Args:  
 data (int): The integer to convert back to a string.  
  
 Returns:  
 str: The decoded string, or an error message if decoding fails.  
 """* try:  
 return data.to\_bytes((data.bit\_length() + 7) // 8, 'big').decode('utf-8')  
 except UnicodeDecodeError:  
 return '[Decryption failed: Non-decodable data]'  
  
  
def generate\_keys() -> tuple[DHPrivateKey, DHPublicKey]:  
 *"""Generate a private-public key pair using Diffie-Hellman.  
  
 Returns:  
 tuple[DHPrivateKey, DHPublicKey]: A tuple containing the private and public keys.  
 """* parameters = dh.generate\_parameters(generator=2, key\_size=512)  
 private\_key = parameters.generate\_private\_key()  
 public\_key = private\_key.public\_key()  
  
 return private\_key, public\_key  
  
  
def encrypt(public\_key: DHPublicKey, message: str) -> tuple[int, int]:  
 *"""Encrypt a message using the recipient's public key.  
  
 Args:  
 public\_key (DHPublicKey): The recipient's public key.  
 message (str): The plaintext message to encrypt.  
  
 Returns:  
 tuple[int, int]: A tuple containing the ephemeral public key component and the ciphertext.  
 """* shared\_parameters = public\_key.parameters()  
 ephemeral\_private = shared\_parameters.generate\_private\_key()  
 ephemeral\_public = ephemeral\_private.public\_key()  
 shared\_key = ephemeral\_private.exchange(public\_key)  
 derived\_key = HKDF(  
 algorithm=hashes.SHA256(),  
 length=32,  
 salt=None,  
 info=b'elgamal-encryption',  
 ).derive(shared\_key)  
 message\_int = string\_to\_int(message)  
 ciphertext = (message\_int \* int.from\_bytes(derived\_key, 'big')) % shared\_parameters.parameter\_numbers().p  
  
 return ephemeral\_public.public\_numbers().y, ciphertext  
  
  
def decrypt(private\_key: DHPrivateKey, ephemeral\_key\_y: int, ciphertext: int) -> str:  
 *"""Decrypt a ciphertext using the recipient's private key.  
  
 Args:  
 private\_key (DHPrivateKey): The recipient's private key.  
 ephemeral\_key\_y (int): The ephemeral public key component from the encryption.  
 ciphertext (int): The encrypted message as an integer.  
  
 Returns:  
 str: The decrypted plaintext message.  
 """* parameters = private\_key.parameters()  
 ephemeral\_public\_numbers = dh.DHPublicNumbers(ephemeral\_key\_y, parameters.parameter\_numbers())  
 ephemeral\_public\_key = ephemeral\_public\_numbers.public\_key()  
 shared\_key = private\_key.exchange(ephemeral\_public\_key)  
 derived\_key = HKDF(  
 algorithm=hashes.SHA256(),  
 length=32,  
 salt=None,  
 info=b'elgamal-encryption',  
 ).derive(shared\_key)  
 shared\_key\_int = int.from\_bytes(derived\_key, 'big')  
 p = parameters.parameter\_numbers().p  
 plaintext\_int = (ciphertext \* pow(shared\_key\_int, -1, p)) % p  
  
 return int\_to\_string(plaintext\_int)  
  
  
def sign(private\_key: DHPrivateKey, message: str) -> tuple[int, int]:  
 *"""Sign a message using the sender's private key.  
  
 Args:  
 private\_key (DHPrivateKey): The sender's private key.  
 message (str): The plaintext message to sign.  
  
 Returns:  
 tuple[int, int]: The signature as a tuple of (r, s).  
 """* parameters = private\_key.parameters()  
 p = parameters.parameter\_numbers().p  
 q = (p - 1) // 2  
 k = os.urandom(32)  
 k = int.from\_bytes(k, 'big') % q  
  
 while k in (0, (p - 1) % k):  
 k = os.urandom(32)  
 k = int.from\_bytes(k, 'big') % q  
  
 r = pow(2, k, p)  
 k\_inv = pow(k, -1, q)  
 message\_int = string\_to\_int(message)  
 s = (k\_inv \* (message\_int - private\_key.private\_numbers().x \* r)) % q  
  
 return r, s  
  
  
def verify(public\_key: DHPublicKey, message: str, signature: tuple[int, int]) -> bool:  
 *"""Verify a signature using the sender's public key.  
  
 Args:  
 public\_key (DHPublicKey): The sender's public key.  
 message (str): The plaintext message that was signed.  
 signature (tuple[int, int]): The signature to verify as (r, s).  
  
 Returns:  
 bool: True if the signature is valid, False otherwise.  
 """* r, s = signature  
 parameters = public\_key.parameters()  
 p = parameters.parameter\_numbers().p  
 q = (p - 1) // 2  
  
 if not (0 < r < p) or not (0 < s < q):  
 return False  
  
 message\_int = string\_to\_int(message)  
 v1 = (pow(public\_key.public\_numbers().y, r, p) \* pow(r, s, p)) % p  
 v2 = pow(2, message\_int, p)  
  
 return v1 == v2  
  
  
def brute\_force\_attack(public\_key: DHPublicKey, ephemeral\_y: int, ciphertext: int, max\_attempts: int) -> str | None:  
 *"""Attempt to brute-force the private key of the receiver.  
  
 Args:  
 public\_key (DHPublicKey): The recipient's public key.  
 ephemeral\_y (int): The ephemeral public key component from encryption.  
 ciphertext (int): The encrypted message as an integer.  
 max\_attempts (int): The maximum number of attempts for brute-forcing.  
  
 Returns:  
 str | None: The decrypted plaintext if successful, otherwise None.  
 """* parameters = public\_key.parameters()  
 p = parameters.parameter\_numbers().p  
  
 for i in range(max\_attempts):  
 private\_key\_candidate = i  
 ephemeral\_public\_numbers = dh.DHPublicNumbers(ephemeral\_y, parameters.parameter\_numbers())  
 ephemeral\_public\_key = ephemeral\_public\_numbers.public\_key()  
 shared\_key = private\_key\_candidate \* ephemeral\_public\_key.public\_numbers().y % p  
 derived\_key = HKDF(  
 algorithm=hashes.SHA256(),  
 length=32,  
 salt=None,  
 info=b'elgamal-encryption',  
 ).derive(shared\_key.to\_bytes((shared\_key.bit\_length() + 7) // 8, 'big'))  
 shared\_key\_int = int.from\_bytes(derived\_key, 'big')  
 plaintext\_int = (ciphertext \* pow(shared\_key\_int, -1, p)) % p  
 plaintext = int\_to\_string(plaintext\_int)  
 print(f"Attempt {i + 1}: Decrypted Message = '{plaintext}' with candidate key {private\_key\_candidate}")  
  
 if plaintext == 'Hello, ElGamal with Python!':  
 print(f'Success! Decrypted message matches with candidate private key {private\_key\_candidate}')  
 return plaintext  
  
 print('Brute-force attack failed to find the correct key.')  
 return None  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 message = 'Hello, ElGamal with Python!'  
  
 # Generate key pairs  
 private\_key, public\_key = generate\_keys()  
  
 # Encryption  
 ephemeral\_y, ciphertext = encrypt(public\_key, message)  
 print(f'Original Message: {message}')  
 print(f'Ciphertext: (ephemeral\_y={ephemeral\_y}, ciphertext={ciphertext})')  
  
 # Decryption  
 decrypted\_message = decrypt(private\_key, ephemeral\_y, ciphertext)  
 print(f'Decrypted Message: {decrypted\_message}')  
  
 # Signing  
 signature = sign(private\_key, message)  
 print(f'Signature: {signature}')  
  
 # Verification  
 is\_valid = verify(public\_key, message, signature)  
 print(f'Signature Valid: {is\_valid}')  
  
 # Brute-force attack  
 print(f"\n{'=' \* 40}\nLet's try brute-force attack!\n{'=' \* 40}")  
 max\_attempts = 1000000  
 result = brute\_force\_attack(public\_key, ephemeral\_y, ciphertext, max\_attempts)  
 print(f'Brute-force attack result: {result}')

Приклад виконання програми:





**Висновки:**

У ході виконання роботи була розроблена асиметрична криптосистема на основі алгоритму Ель Гамаля з використанням бібліотеки OpenSSL на платформі Windows. Основною метою було дослідити стійкість стандартних криптопровайдерів до атак, що можуть виникати через недоліки механізмів захисту операційної системи, а також перевірити безпеку реалізованого алгоритму. Асиметричний алгоритм Ель Гамаля, який лежить в основі розробленої системи, забезпечує високий рівень безпеки завдяки використанню пари ключів: приватного та публічного. Для оцінки стійкості системи було здійснено брутфорс-атаку, що полягала у спробах підібрати приватний ключ шляхом перебору можливих значень. Результати атаки показали, що дешифрувати зашифроване повідомлення даним способом без правильного ключа майже неможливо, навіть за умов систематичного перебору діапазону можливих ключів.

Таким чином, лабораторна робота дозволила не лише отримати практичні навички побудови криптосистеми, а й наочно переконатися у важливості дотримання криптографічних стандартів і правильного налаштування захисних механізмів.