**Міністерство освіти і науки України**

**Національний технічний університет України**

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

**Фізико-технічний інститут**

**Методи реалізації криптографічних механізмів**

Лабораторна робота №1

**Виконали:**

студенти ФІ-31мн

Карабан А.

Шевченко Ю.

**Перевірила:**

Селюх П. В.

Київ – 2024

**Лабораторна робота №1. Вибір та реалізація базових фреймворків та бібліотек**

**Мета:**

Вибір базових бібліотек/сервісів для подальшої реалізації криптосистеми.

**Завдання:**

Для другого типу лабораторних робіт ― вибір бібліотеки реалізації основних криптографічних примітивів з точки зору їх ефективності за часом та пам’яттю для різних програмних платформ.

Порівняння бібліотек OpenSSL, Crypto++, CryptoLib, PyCrypto для розробки гібридної криптосистеми під Linux платформу.

Оформлення результатів роботи. Опис функції бібліотеки реалізації основних криптографічних примітивів обраної бібліотеки, з описом алгоритму, вхідних та вихідних даних, кодів повернення. Контрольний приклад роботи з функціями. Обґрунтування вибору бібліотеки.

**Хід роботи:**

1. Проведемо порівняльний аналіз бібліотек.

**OpenSSL** ― одна з найпопулярніших бібліотек для роботи з криптографічними алгоритмами. Вона написана на мові C і забезпечує високу продуктивність, підтримку широкого спектру алгоритмів шифрування, підпису та хешування. Проте, через складність API, вона може бути важкою у використанні для початківців.

**Crypto++** ― це сучасна бібліотека, написана на C++, яка надає широкий набір криптографічних алгоритмів та примітивів. Вона орієнтована на легкість використання та кросплатформенність. Основним недоліком є те, що вона не завжди настільки оптимізована, як OpenSSL.

**CryptoLib** ― це компактна бібліотека, яка орієнтована на простоту і використовується переважно для навчальних цілей або у вбудованих системах. Вона не має такого широкого спектру можливостей, як OpenSSL чи Crypto++.

**PyCrypto** ― це бібліотека для Python, яка забезпечує базові криптографічні примітиви. Вона ідеально підходить для Python-розробників, але має обмежену продуктивність у порівнянні з OpenSSL чи Crypto++.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Бібліотека | OpenSSL | Crypto++ | CryptoLib | PyCrypto |
| Платформа | C, Linux | C++, Linux | C, Embedded | Python, Linux |
| Продуктивність | Висока | Середня | Низька | Середня |
| Простота використання | Середня | Висока | Висока | Висока |
| Підтримка алгоритмів | Велика | Велика | Обмежена | Середня |
| Документація | Складна, але повна | Достатня | Обмежена | Добра |
| Розмір і залежності | Велика | Середня | Низька | Низька |
| Актуальність підтримки | Висока | Висока | Середня | Низька |

**Обґрунтування вибору:** Для реалізації гібридної криптосистеми на Linux платформі було обрано бібліотеку OpenSSL. Вона забезпечує високу продуктивність, підтримує великий набір криптографічних алгоритмів, є надійною і активно підтримується спільнотою. Незважаючи на складність її API, OpenSSL є оптимальним вибором для задачі, оскільки забезпечує високу ефективність та можливість інтеграції в системи виробничого рівня.

2. Реалізуємо основні криптографічні примітиви за допомогою OpenSSL.

Код демонструє реалізацію основних криптографічних операцій, необхідних для побудови гібридних криптосистем. Основна увага зосереджена на чотирьох ключових аспектах криптографії:

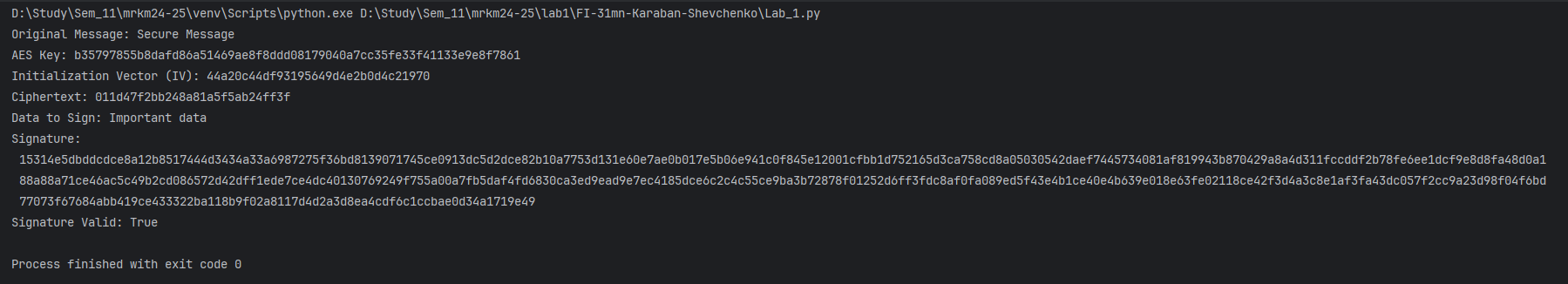
1. Генерація ключів:
   1. Функція generate\_aes\_key створює випадковий 256-бітний AES-ключ, який використовується для симетричного шифрування даних.
   2. Функція generate\_rsa\_keypair генерує пару асиметричних ключів RSA (приватний і публічний ключ).
2. Шифрування:
   1. Функція encrypt\_with\_aes використовує AES-шифрування в режимі CFB (Cipher Feedback Mode). Цей режим забезпечує стійкість до перехоплення і зручний для роботи зі змінними обсягами даних.
3. Цифрові підписи:
   1. Функція sign\_data генерує цифровий підпис для переданих даних за допомогою приватного ключа RSA. Підписи забезпечують аутентифікацію даних і гарантують, що вони не були змінені.
4. Перевірка підписів:
   1. Функція verify\_signature перевіряє автентичність цифрового підпису з використанням публічного ключа RSA. Це дозволяє одержувачу переконатися, що дані були підписані дійсним відправником.

Кожна функція реалізовує окремий примітив, що дозволяє легко використовувати їх як окремо, так і в комбінації. Нижче подано короткий огляд:

* AES-шифрування:
  + Створюється AES-ключ
  + Шифрується текст за допомогою цього ключа і випадково згенерованого вектора ініціалізації (IV)
  + Результати (IV і шифротекст) повертаються для подальшого використання
* RSA-ключі
  + Генеруються приватний і публічний ключі RSA з розміром 2048 біт
  + Приватний ключ використовується для підпису, а публічний ― для перевірки
* Цифрові підписи
  + Дані підписуються приватним ключем RSA з використанням алгоритму SHA-256 для хешування
  + Підпис передається для перевірки
* Перевірка підпису
  + Підпис перевіряється з використанням публічного ключа RSA
  + Повертається булевий результат: True, якщо підпис дійсний, або False, якщо ні

import os  
  
from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import hashes  
from cryptography.hazmat.primitives.asymmetric import padding, rsa  
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  
from cryptography.hazmat.primitives.hashes import SHA256  
  
  
def generate\_aes\_key() -> bytes:  
 *"""Generates a random AES key.  
  
 Returns:  
 bytes: A randomly generated 256-bit AES key.  
 """* return os.urandom(32)  
  
  
def encrypt\_with\_aes(key: bytes, plaintext: str) -> tuple[bytes, bytes]:  
 *"""Encrypts plaintext using AES encryption with CFB mode.  
  
 Args:  
 key (bytes): The AES encryption key.  
 plaintext (str): The plaintext message to encrypt.  
  
 Returns:  
 tuple[bytes, bytes]: A tuple containing the initialization vector (IV) and the ciphertext.  
 """* iv = os.urandom(16)  
 cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend())  
 encryptor = cipher.encryptor()  
 ciphertext = encryptor.update(plaintext.encode()) + encryptor.finalize()  
  
 return iv, ciphertext  
  
  
def generate\_rsa\_keypair() -> tuple[rsa.RSAPrivateKey, rsa.RSAPublicKey]:  
 *"""Generates an RSA private and public key pair.  
  
 Returns:  
 tuple[rsa.RSAPrivateKey, rsa.RSAPublicKey]: A tuple containing the RSA private key and public key.  
 """* private\_key = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048, backend=default\_backend())  
 public\_key = private\_key.public\_key()  
  
 return private\_key, public\_key  
  
  
def sign\_data(private\_key: rsa.RSAPrivateKey, data: bytes) -> bytes:  
 *"""Signs data using an RSA private key.  
  
 Args:  
 private\_key (rsa.RSAPrivateKey): The RSA private key used for signing.  
 data (bytes): The data to sign.  
  
 Returns:  
 bytes: The generated signature.  
 """* return private\_key.sign(  
 data, padding.PSS(mgf=padding.MGF1(SHA256()), salt\_length=padding.PSS.MAX\_LENGTH), hashes.SHA256()  
 )  
  
  
def verify\_signature(public\_key: rsa.RSAPublicKey, signature: bytes, data: bytes) -> bool:  
 *"""Verifies an RSA signature.  
  
 Args:  
 public\_key (rsa.RSAPublicKey): The RSA public key used for verification.  
 signature (bytes): The signature to verify.  
 data (bytes): The original data.  
  
 Returns:  
 bool: True if the signature is valid, False otherwise.  
 """* try:  
 public\_key.verify(  
 signature,  
 data,  
 padding.PSS(mgf=padding.MGF1(SHA256()), salt\_length=padding.PSS.MAX\_LENGTH),  
 hashes.SHA256(),  
 )  
 return True  
 except Exception:  
 return False  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # AES encryption example  
 original\_message = 'Secure Message'  
 aes\_key = generate\_aes\_key()  
 iv, ciphertext = encrypt\_with\_aes(aes\_key, original\_message)  
  
 print(f'Original Message: {original\_message}')  
 print(f'AES Key: {aes\_key.hex()}')  
 print(f'Initialization Vector (IV): {iv.hex()}')  
 print(f'Ciphertext: {ciphertext.hex()}')  
  
 # RSA key generation and signing example  
 private\_key, public\_key = generate\_rsa\_keypair()  
 data = b'Important data'  
 signature = sign\_data(private\_key, data)  
  
 print(f'Data to Sign: {data.decode()}')  
 print(f'Signature: {signature.hex()}')  
  
 # Verify the signature  
 is\_valid = verify\_signature(public\_key, signature, data)  
 print(f'Signature Valid: {is\_valid}')

Приклад виконання програми:



**Висновки:**

У ході виконання лабораторної роботи ми здобули практичні навички роботи з криптографічними примітивами за допомогою бібліотеки OpenSSL у Python. Ми успішно реалізували симетричне шифрування з використанням AES та режиму CFB, а також асиметричне шифрування на основі RSA. Опанували створення цифрових підписів із використанням алгоритму SHA-256 для забезпечення автентичності та цілісності даних. Робота дозволила нам не лише поглибити розуміння криптографічних алгоритмів, але й освоїти основи побудови безпечних систем, що відповідають сучасним стандартам захисту інформації.