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:
    a. Функція `generate_aes_key` створює випадковий 256-бітний AES-ключ, який використовується для симетричного шифрування даних.
    b. Функція `generate_rsa_keypair` генерує пару асиметричних ключів RSA (приватний і публічний ключ).
2. Шифрування:
    a. 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:
    a. 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:
    a. 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:

```
D:\Study\Sem_11\mrkm24-25\venv\Scripts\python.exe D:\Study\Sem_11\mrkm24-25\lab1\FI-31mn-Karaban-Shevchenko\Lab_1.py
Original Message: Secure Message
AES Key: b35797855b8dafd86a51469ae8f8ddd08179040a7cc35fe33f41133e9e8f7861
Initialization Vector (IV): 44a20c44df93195649d4e2b0d4c21970
Ciphertext: 011d47f2bb248a81a5f5ab24ff3f
Data to Sign: Important data
Signature:
 15314e5dbddcdce8a12b8517444d3434a33a6987275f36bd8139071745ce0913dc5d2dce82b10a7753d131e60e7ae0b017e5b06e941c0f845e12001cfbb1d7521651d3ca758cd8a05030542daef7445734081af819943b870429a8a4d311fccddf2b78fe6ee1dcf9e8d8fa48d0a1
 88a88a71ce46ac5c49b2cd086572d42dff1ede7ce4dc40130769249f755a00a7fb5daf4fd6830ca3ed9ead9e7ec4185dce6c2c4c55ce9ba3b72878f01252d6ff3fdc8af0fa089ed5f43e4b1ce40e4b639e018e63fe02118ce42f3d4a3c8e1af3fa43dc057f2cc9a23d98f04f6bd
 77073f67684abb419ce433322ba118b9f02a8117d4d2a3d8ea4cdf6c1ccbae0d34a1719e49
Signature Valid: True

Process finished with exit code 0
```

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.