Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Фізико-технічний інститут

Методи реалізації криптографічних механізмів

Лабораторна робота №3

Виконали:
студенти ФІ-31мн
Карабан А.
Шевченко Ю.
Перевірила:
Селюх П. В.

Київ – 2024

Лабораторна робота №3. Реалізація основних асиметричних криптосистем.

Мета:

Дослідження можливостей побудови загальних та спеціальних криптографічних протоколів за допомогою асиметричних криптосистем.

Завдання:

Для другого типу лабораторних робіт — розробити реалізацію асиметричної криптосистеми.
Бібліотека OpenSSL під Windows платформу. Кр\с Ель Гамаля.
Оформлення результатів. Контрольний приклад роботи з асиметричною криптосистемою.

Хід роботи:

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

Генерація ключів [ред. | ред. код]

1. Генерується випадкове просте число p бітів.
2. Обирається випадковий примітивний елемент g поля $\mathbb{Z}_p$.
3. Обирається випадкове ціле число x таке, що $1 < x < p-1$.
4. Обчислюється $y = g^x \mod p$.
5. Відкритий ключ — це трійка (p, g, y), а приватний ключ — це число x.

Шифрування [ред. | ред. код]

Повідомлення M шифрується так:

1. Обирається сесійний ключ — випадкове ціле число k таке, що $1 < k < p-1$
2. Обчислюються числа $a = g^k \mod p$ і $b = y^k M \mod p$.
3. Пара чисел (a, b) є шифротекстом.

Неважко бачити, що довжина шифротексту в схемі Ель-Гамаля є довшою за повідомлення M удвічі.

Розшифрування [ред. | ред. код]

Знаючи приватний ключ x, повідомлення M можна обчислити з шифротексту (a, b) за формулою:

$$M = b(a^x)^{-1} \mod p.$$

При цьому неважко перевірити, що

$$(a^x)^{-1} \equiv g^{-kx} \pmod p$$

і тому

$$b(a^x)^{-1} \equiv (y^k M)g^{-xk} \equiv (g^{xk} M)g^{-xk} \equiv M \pmod p.$$

Для практичних обчислень більше підходить така формула:

$$M = b(a^x)^{-1} \mod p = b \cdot a^{(p-1-x)} \mod p$$

Підпис повідомлень [ред. | ред. код]

Для підпису повідомлення M виконуються наступні операції:

1. Обчислюється дайджест повідомлення M: $m = h(M)$.
2. Обирається випадкове число $1 < k < p - 1$ взаємно просте з $p - 1$ і обчислюється $r = g^k \mod p$.
3. Обчислюється число $s \equiv (m - xr)k^{-1} \pmod{p - 1}$.
4. Підписом повідомлення M вважається пара (r, s).

Перевірка підпису [ред. | ред. код]

Знаючи відкритий ключ (p, g, y) і підпис (r, s), повідомлення M перевіряється так:

1. Перевіряються дві умови: $0 < r < p$ і $0 < s < p - 1$. Якщо хоча б одна з них не виконується, то підпис вважається недійсним.
2. Обчислюється дайджест $m = h(M)$.
3. Підпис вважається справжнім, якщо виконується рівність:
$$y^r r^s \equiv g^m \pmod{p}.$$

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


if __name__ == '__main__':
    message = 'Hello, ElGamal with Python!'
```

```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}')
```

Приклад виконання програми:

```
D:\Study\Sem_11\mrkm24-25\venv\Scripts\python.exe D:\Study\Sem_11\mrkm24-25\lab3\FI-31mn-Karaban-Shevchenko\Lab_3.py
Original Message: Hello, ElGamal with Python!
Ciphertext: (ephemeral_y=669604514215756599236074530316170298965886523158737756119299596670787290235328580997908794425769979331977951950350030157861229079811113120115714273476353 4,
 ciphertext=1451151380539724948159096410980911739307724489381711942832385780025034981109665163452239580946890073000797242433814433376849550956465949994301)
Decrypted Message: Hello, ElGamal with Python!
Signature: (381109192608478074048687291491340571752383140558235900398973143045142582863802179689050403270591598845362357053256133554346519818470330783429592127283206,
 445218799354346472201615710171449222290642743610986698268348783418835571653577735940240963734333811831112357101224688631050692917583253924907677096726620 4)
Signature Valid: True

Process finished with exit code 0
```

Висновки:

У цій лабораторній роботі ми реалізували алгоритм асиметричної криптосистеми Ель Гамаля, який включає етапи генерації ключів, шифрування та розшифрування повідомлень, а також створення та перевірки цифрових підписів. Виконання роботи дало змогу зрозуміти принципи роботи асиметричного шифрування, важливість вибору криптографічно стійких параметрів та складність обчислення, яка забезпечує безпеку алгоритму. Ми здобули практичні навички реалізації криптографічних протоколів, що є важливими для створення захищених систем передачі інформації.