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 недосконаліть 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.

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
```

```python
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
```

```python
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:

```
D:\Study\Sem_11\mrkm24-25\venv\Scripts\python.exe D:\Study\Sem_11\mrkm24-25\lab4\FI-31mn-Karaban-Shevchenko\Lab_4.py
Original Message: Hello, ElGamal with Python!
Ciphertext: (ephemeral_y=869491791306464575567349855167037654159214522689360735526633118470997085596376714969697776300967772726284781293571240673248548850984490699345296509196240 6,
 ciphertext=2406371970328575850188877770301114621429834821404114090076576803557280067499583979761731424430954232401221350801893750986835386352470081413728)
Decrypted Message: Hello, ElGamal with Python!
Signature: (4720217021946651697355552795624040735335321187864808846746973778443170042454600980239798427874576356587905339038758601805190683580887550414020423962327993,
 3226203247621338483509052725083264316370357147507890026114728297100903649153475859492626906765744113990615588198844425612975758316204662105045218229278029)
Signature Valid: True

========================================
Let's try brute-force attack!
========================================
Attempt 1: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 0
Attempt 2: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 1
Attempt 3: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 2
Attempt 4: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 3
Attempt 5: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 4
Attempt 6: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 5
Attempt 7: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 6
Attempt 8: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 7
Attempt 9: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 8
Attempt 10: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 9
Brute-force attack failed to find the correct key.
Brute-force attack result: None

Process finished with exit code 0
Attempt 999979: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999978
Attempt 999980: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999979
Attempt 999981: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999980
Attempt 999982: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999981
Attempt 999983: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999982
Attempt 999984: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999983
Attempt 999985: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999984
Attempt 999986: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999985
Attempt 999987: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999986
Attempt 999988: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999987
Attempt 999989: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999988
Attempt 999990: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999989
Attempt 999991: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999990
Attempt 999992: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999991
Attempt 999993: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999992
Attempt 999994: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999993
Attempt 999995: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999994
Attempt 999996: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999995
Attempt 999997: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999996
Attempt 999998: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999997
Attempt 999999: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999998
Attempt 1000000: Decrypted Message = '[Decryption failed: Non-decodable data]' with candidate key 999999
Brute-force attack failed to find the correct key.
Brute-force attack result: None

Process finished with exit code 0
```

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.