Лабораторна роботи № 3.
Реалізація основних асиметричних криптосистем

Виконали: *Бараніченко Андрій, Гаврилова Анастасія, Дрозд Софія, Зібаров Дмитро, Колесник Андрій*

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

Завдання: Для першого типу лабораторних робіт – дослідити можливість реалізації одного з чотирьох криптографічних протоколів: розділення секрету, сліпого цифрового підпису, несуперечливого цифрового підпису та розподілу ключів для симетричної криптосистеми за допомогою різних асиметричних алгоритмів (не менше як двох) та порівняти їх ефективність за обраним критерієм.
Підгрупа 1A. Розподіл секрету.
Підгрупа 1B. Розподіл ключів.
Підгрупа 1C. Сліпий цифровий підпис.

Для роботи було обрано варіант Підгрупа 1A. Розподіл секрету.

Код

```
import random
from Cryptodome.PublicKey import RSA
from Cryptodome.Cipher import PKCS1_OAEP
from elgamal.elgamal import Elgamal

import time
from secrets import token_bytes
import warnings


def modular_inverse(a, mod):
    b0, x0, x1 = mod, 0, 1
    while a > 1:
        q = a // b0
        a, b0 = b0, a % b0
        x0, x1 = x1 - q * x0, x0
    return x1 + mod if x1 < 0 else x1

def select_prime_larger_than(value):
    mersenne_primes = [2**x - 1 for x in [17, 19, 31, 61, 89, 107, 127, 521]]
    for prime in mersenne_primes:
        if prime > value:
            return prime
    return 2**128 + 51

def split_secret(secret_bytes, required_shares, distributed_shares, prime_mod=None):

    secret = int.from_bytes(secret_bytes, byteorder='big', signed=False)
    prime_mod = prime_mod or select_prime_larger_than(secret)

    coefficients = [int.from_bytes(token_bytes((prime_mod.bit_length() + 7) // 8), byteorder='big', signed=False) % prime_mod for _ in range(required_shares -
1)]
    coefficients.append(secret)

    shares = [(i, (evaluate_polynomial(i, coefficients, prime_mod)).to_bytes((prime_mod.bit_length() + 7) // 8, byteorder='big', signed=False)) for i in range(1,
distributed_shares + 1)]
    return {"required_shares": required_shares, "prime_mod": prime_mod.to_bytes((prime_mod.bit_length() + 7) // 8, byteorder='big', signed=False), "shares":
shares}

def evaluate_polynomial(x, coefficients, prime_mod):
```

```
    y = 0
    for coefficient in coefficients:
        y = (y * x + coefficient) % prime_mod
    return y

def lagrange_interpolation(x, points, prime_mod):
    y = 0
    for i, (xi, yi) in enumerate(points):
        numerator = yi
        denominator = 1
        for j, (xj, yj) in enumerate(points):
            if j != i:
                numerator = (numerator * (x - xj)) % prime_mod
                denominator = (denominator * (xi - xj)) % prime_mod
        y = (y + numerator * modular_inverse(denominator, prime_mod)) % prime_mod
    return y

def recover_secret(data):
    shares = data['shares']
    required_shares = data['required_shares']
    prime_mod = int.from_bytes(data['prime_mod'], byteorder='big', signed=False)

    if len(shares) < required_shares:
        raise ValueError("not enough shares provided")
    shares = shares[:required_shares]

    points = [(x, int.from_bytes(y, byteorder='big', signed=False)) for x, y in shares]
    return (lagrange_interpolation(0, points, prime_mod)).to_bytes((prime_mod.bit_length() + 7) // 8, byteorder='big', signed=False)[1:]

secret = b"This is my secret"
required_shares = 3
total_shares = 5

split_data = split_secret(secret, required_shares, total_shares)
split_data_rsa = split_data['shares'][:3]
split_data_ecc = split_data['shares'][2:]

print("Original Secret (for splitting):", secret)

rsa_private_key = RSA.generate(1024)
rsa_public_key = rsa_private_key.publickey()
rsa_cipher_encrypt = PKCS1_OAEP.new(rsa_public_key)
rsa_cipher_decrypt = PKCS1_OAEP.new(rsa_private_key)


rsa_encrypted_shares = []
start_time = time.time()
for share in split_data_rsa:
    encrypted_share = rsa_cipher_encrypt.encrypt(share[1])
    rsa_encrypted_shares.append((share[0], encrypted_share))
rsa_time = time.time() - start_time


print("\nRSA Encrypted Shares:")
for idx, encrypted_share in rsa_encrypted_shares:
    print(f"Part {idx}: {encrypted_share}")


rsa_decrypted_shares = []
start_time = time.time()
for idx, encrypted_share in rsa_encrypted_shares:
    decrypted_share = rsa_cipher_decrypt.decrypt(encrypted_share)
    rsa_decrypted_shares.append((idx, decrypted_share))
rsa_decrypt_time = time.time() - start_time


print("\nRSA Decrypted Shares:")
for idx, decrypted_share in rsa_decrypted_shares:
    print(f"Part {idx}: {decrypted_share}")


recovery_data_rsa = {
    'required_shares': required_shares,
    'prime_mod': split_data['prime_mod'],
    'shares': rsa_decrypted_shares
}
reconstructed_secret_rsa = recover_secret(recovery_data_rsa)

print(f"\nReconstructed Secret (RSA): {reconstructed_secret_rsa.decode('utf-8')}")
print(f"\nRSA Encryption Time: {rsa_time:.4f} seconds")
print(f"RSA Decryption Time: {rsa_decrypt_time:.4f} seconds")

elgamal_pb, elgamal_pv = Elgamal.newkeys(128)

elgamal_encrypted_shares = []
start_time = time.time()
```

```
for share in split_data_ecc:
    encrypted_share = Elgamal.encrypt(share[1], elgamal_pb)
    elgamal_encrypted_shares.append((share[0], encrypted_share))
elgamal_time = time.time() - start_time

print("\nElGamal Encrypted Shares:")
for idx, encrypted_share in elgamal_encrypted_shares:
    print(f"Part {idx}: {encrypted_share}")


elgamal_decrypted_shares = []
start_time = time.time()
for idx, encrypted_share in elgamal_encrypted_shares:
    decrypted_share = Elgamal.decrypt(encrypted_share, elgamal_pv)
    decrypted_share_bytes = bytes(decrypted_share)
    elgamal_decrypted_shares.append((idx, decrypted_share_bytes))
elgamal_decrypt_time = time.time() - start_time

print("\nElGamal Decrypted Shares:")
for idx, decrypted_share in elgamal_decrypted_shares:
    print(f"Part {idx}: {decrypted_share}")



recovery_data_ecc = {
    'required_shares': required_shares,
    'prime_mod': split_data['prime_mod'],
    'shares': elgamal_decrypted_shares
}
reconstructed_secret_ecc = recover_secret(recovery_data_ecc)

print(f"Reconstructed Secret (ElGamal): {reconstructed_secret_ecc.decode('utf-8')}")
print(f"ElGamal Encryption Time: {elgamal_time:.4f} seconds")
print(f"ElGamal Decryption Time: {elgamal_decrypt_time:.4f} seconds")
```

Результат роботи

Original Secret (for splitting): b'This is my secret'

Reconstructed Secret (RSA): This is my secret

RSA Encryption Time: 0.0091 seconds
RSA Decryption Time: 0.0053 seconds

Reconstructed Secret (ElGamal): This is my secret
ElGamal Encryption Time: 0.3820 seconds
ElGamal Decryption Time: 0.3194 seconds

Висновки

У цій лабораторній роботі ми дослідили метод розподілу секрету за допомогою алгоритмів RSA та ElGamal, реалізуючи підхід, що ґрунтується на схемі розподілу секрету Шаміра. Ця схема дозволяє поділити секрет на кілька частин (часток), і для його відновлення потрібно лише частину з них. У нашому випадку секрет було розподілено на 5 часток, з яких 3 достатньо для відновлення.

Результати тестування показали, що RSA виконує шифрування й дешифрування часток значно швидше — 0.0091 і 0.0053 секунд відповідно, тоді як ElGamal потребує більше часу: 0.3820 та 0.3194 секунд. Попри різницю у швидкості, обидва алгоритми успішно впоралися з

відновленням секрету, що підтвердило працездатність схеми Шаміра в обраних реалізаціях.

Таким чином, RSA виявився оптимальним для випадків, де важлива швидкість, а ElGamal — для сценаріїв, де потрібна додаткова стійкість. Вибір алгоритму залежить від конкретних вимог: коли пріоритет — швидкість і ефективність, краще підходить RSA; коли потрібна стійкість до криптоаналізу — ElGamal.