МІНІСТЕРСТВО ОСВІТИ І НАУКИ КРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ

ЛАБОРАТОРНА РОБОТА №3

На тему:

“ Реалізація основних асиметричних криптосистем ”

Виконала:
студентка 6-го курсу
групи: ФІ-31мн
Каюк Ксенія
Коробан Ольга
Кухар Богдан

Київ - 2024

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

Розділення секрету було реалізовано за допомогою алгоритму Шаміра. Секрет розбивався на частки, які представляли собою значення полінома в різних точках. Для захисту часток під час передачі використовували RSA і ECC. А потім отримані зафровані частки розшифровувались до початквого виду за допомогою інтерполяції Лагранджа.

Спочатку генеруємо формулу, в якій секрет буде вільним числом, та обираємо декілька точок, які будуть потім шифруватись в ролі часток.

```python
def split_secret(secret, threshold, num_shares):
    coefficients = generate_polynomial(secret, threshold)
    shares = [(i, evaluate_polynomial(coefficients, i)) for i in range(1, num_shares + 1)]
    print("Частки секрету (Шамір):", shares)
    return shares
```

```
Частки секрету (Шамір): [(1, 13902), (2, 17195), (3, 22224), (4, 28989), (5, 37490)]
```

Далі використовуємо шифрування за допомогою алготиму RSA.

```python
# Шифрування часток за допомогою RSA
def rsa_encrypt_shares(shares):
    rsa_key = RSA.generate(2048)
    public_key = rsa_key.publickey()
    cipher = PKCS1_OAEP.new(public_key)

    encrypted_shares = []
    encryption_time_start = time.time()
    for share in shares:
        encrypted_shares.append((share[0], cipher.encrypt(str(share[1]).encode())))
    encryption_time = time.time() - encryption_time_start

    print(f"Зашифровані частки (RSA): {encrypted_shares}")
    print(f"Час шифрування (RSA): {encryption_time:.5f} с")
    return rsa_key, encrypted_shares, encryption_time
```

```
Зашифровані частки (RSA): [(1, b'Q\x16\xa5\x94\xa9\xef\x84m6Pg\xb4\xce\xdef\xe1\x03\xb6>\x05)\xf06\xb5\xf2\xb6\x98\x1d)R\xe8\xa3\x86\xec\x14\xde0\xae\xd8
Z=\xc1Z\xfan\x92\xecI\xf8D^\x8fn4\x07\x92\xc4`f\xd9a\x86,]\xa5,r}\xf6AO8\t\xc6s\x11M\xd8\x99Za\xcc($\xa3Tv\x96\xd3e\xf8n\x84\xd8\xac`\xea\xb9\x9cP\xe7\x8
ae=A\x16\x96\xd0w\xd9\xb65J\xd8\x87\xf0\xef\x9b.\x00\xe1\xa7\xbb\x17\x91\x8d\x8b\xf8\xe6\xd4\xc0\xc0\xca\x04-\xb4ZW\x88k\xb3\xa5\x0btxIa89+O\xf9+\ru\xf2#
\x07\xaa~& \x9f\xd5\x05\xb1eBMhE\x91\xf1#\xfe\xb4\x95x\xf4\xcc\xca\xc5\x0bh\x10\xc8)[z%\xfcl\xb9W\x05\xed>!\x9cj\xb3|>\x9a\n\xc3j\x91N\xa1t\xa6\xd0\xd9\x
92\xc6\xdbaW\\>\x05\xe5[c\n\xca\x8e\x89 \xef\x93\x1b\xb2S3Z\xdbX7U\xad<9\x1d1\xd9J\x14\x18@\x89\x04\xdfZ\x19'), (2, b"4\x8e\xeevh\x82\xe7Ye\xebZ\xa0\xb39
H*eF\xb6V\xdc\x13\x80:\x91\xd7\x0b\x9f\x7f\x17\xdf\x9cX\xe0\xbd@\xa7\x16\xac\x19\xf2b\xfdUJ1j\n\xb3\x95T\xac\xdd\x1f\xd3NR\x8fE\xe9\xa6V\xc9r\xea\xf4{\x9
4r\x8a\xa1\x97'\x0c5yF\xe6\xc0\x1b\xb77\x12\x0e2\xc3`G\xf1\x821\xdf\xb1\xf6YkkD\xe0\x101\x0b\x8c\xce\xd0\x9a\x12\x08\xc8\xab\xa3\xc0\xc51\xda{>?\x99\xce
\xf7\x8e\x89\x7fYX\x07A\x838\x06\x04\x9d\xa5\x94g\xa8boM\xe8\xeb\xf4\x9cu`\xf8\x1b\x15\xf6\xb6\x01\xf0\xdeBvk#\x83\xf1\x01\xfe\xa9\xeb\xa6OG^\xd5\xcf\\1
\xc2A\xeb\x18\x1c\x8c\xe1V\x15\xbf\x9b\x90zc$\xaaK\xa1U&\n\xeb\xb2\xf6\xdd\x8f\x1e\xb3\xc4\x15\xc5@\xb5\xe6\xa6\xbf\xb6\xad\x91d\x8f\xf9H-\xe4\xd7\xb6\xf
5[\xba\x17a\x11\x8d\x9bH`t\t\xbaH\x90Yn\xdb\xa2\xafM@\xd4Hw\x01\x1e\xa8\x89V\x94`\xc0.\x0eD\x83"), (3, b'\x92\xd1\x07\xcc\xce\xcfj\xce\x9a\xc8\x9bN\t\xc9
\xef\x13\x01\xa4;\xae\x956\xd2\x89:\xb19iF\xbb\xa4e\xbf6\xbfm#S\xfd\xdb\xabZw\xdc\xd4X\x13\xbb\xd6B\xb1\xbf\x1b<\x86\xb3\xe5\xae)g\xb5\x17\xb9$E:\xd0x\x8
c$\x983\xc2\xb91\x97\xea\xd7r\xdaB\x9f%\x94\xd3\xe0\x04\n\x9b\xe6\xd2\x02Rq\xf2\x94\xa5\xf3N~\x18\x98\xf9\x13\x97\xb7\xf3\xddW\xfe*\x88e\x9e\x11\xc0b\'\x
a9\xf79\x13\xb4\x81v\xbe2,\xc6\xa9\xeb\xb0\xfeQVf?\x8f\x00%0W\xc5\xa4\xfbe\xe4,?}$w\x15\x7fq\xdb\xa6\xc0\xa5Aik+\r\xefB2\x9d\x18\xc4\xe3\xbdn\xff\x9a\xa5
\xbd\xaa\xdd(z\x95\xd48\x19m\xe1\xf2\x88\xb4\x089\x1a\xcc\x95\xe6\x93\x90\x85\xf8":\xa1fvO\x98y\xe1FJ\xca\xca\xc1\xa0EFc3\xec\xff\x9bIx\xaa\x99\x83\xa5\x
da\x12\xe1\x08\x1f$\x0b]+\xfa\x0eA\x0fx\xf1\x19Pw\'\xdbi\xdcq\xbe\xb2\xf4Zd'), (4, b';\x10i\xb7\n\xc9\x9f\n\x93\x96\xbehm\xd13\xa8\x83\xc4\x9a\xbc\xc7c\x
e8\xe0J\x17\xd2@\xf7\x02w\xba\x80\xbe\x81LX\xfbIO<(\xef{\x0e\xf4\xe0c\xff\x87\xd4-\x8d\xef\xcai8\xd1\xbe\xd59h\xbc\x89\xc7b\xbe!\xf4#5\xb2\xb2\xfcG[&\xe4
\x0f&\xc2\xb8\x1cY\x11\xb0\xa2\xeeD\x02\xfb\xa0\x8a\xf4\xady\xfd\xe1A\xc1\xd5\xb2\xd0\x8e\xe8\xfe\xaf\xc1\xc7\xd2\xc0\xe4\x94\x1b\x1f\xf8\x9a\xb2L\x04g\x
04dOb\x83L\x80\xefJ\x91\xd1ze\xa9\x04\x95\xa3\x89BP\xc5+\x98\xed\x08\x920-;m\xdeE\\\xce\x8c\xc9o\xad\xde\xfa\xa95\xba\xadl\xaf\xec}\xa0m\xe4\xb2\xf0\x10D
\x9a\x89\xcb:\x175Y\xb0\xa0\xef\xbb\xb6\xd2\x18\xce\n\xf7Sc\x1a\xf2\xfc\xe1\xe0Cn\xf0\t]#)\x84\x7f\x87\xea\xa9R\xa9\x1826\xdc\xe4\x1b\x9d\x1b\xc1]B\xd7\x
05\x05b\x84=\xc7\x16\xc3t\x19\xd9T#,vE\x8f&M\x19I=K&\xbf|$\x10\xc1}'), (5, b"{\xe5\xca9\xf8\xed\xcf\xbb\x8f\xc3@w\x88.7\x9d\xba\x1a[b\xf4UZ\xaf\x84\xe7\x
f74\xc3\xbe:\x12\x16\x99|z\x18\xcc\x8eD\xd3p\x81W\xd6\xbb_\xe5\x8bG\x0069\x90\xb0Tc\xbc\x12Q\x11\xdb \xf4\x0c\xea\xf5\xa2\x98\xa8)A\xab3\xb4e.\xb7,\x98y
(\xd1\xa2_s\x1c\x9e\xd4\x86\xb7M\xa0\xc7\x0e\xac\x82M- O\xc0^\x98\xa5:\xdb4W\xa6\xd4\x85\xe7\x89\xc6\x9f\xd4\xeb0\xdc\xec\xa2\xfe\x93\xe9\xb4\xcdX\xb3{\x
96E{\x93\xe9\xbe\xe4\x1f\xaf\xf2\xd4\xbb\xa3\xcf\x9a\x1b\xbdWR\x0b\xee@'I\xc1o2\x98\xf3\x9e\x93qS\xb4\x7f\x9b\xa2\xdd]\x1bl\xe8\xd2y\x9c\xf0\xd26\xa3\x01
\xffa\x8c\xb1\xd4B\t!\x11s\x93\x90\xfb\x8dw\xe6_8\x0c\xefe\xb6\xe0\xd6\xf6\xb3\x96\xf7\xc3\xd9yl\x00\xab+\xd44\x10\x9c\x97\x87O\xdb1\x82\xde\xba\xe40jm\x
04\x97%\x84\xfai9\xf0\xae\xbe?\x17\xb6a\xe3\xe9-\x87qi\x84\x01\x1f#\xc1")]
Час шифрування (RSA): 0.00399 с
```

Тепер розшифровуємо за допомогою приватного ключа.

```python
# Розшифрування часток за допомогою RSA
def rsa_decrypt_shares(rsa_key, encrypted_shares):
    decipher = PKCS1_OAEP.new(rsa_key)

    decrypted_shares = []
    decryption_time_start = time.time()
    for share in encrypted_shares:
        decrypted_shares.append((share[0], int(decipher.decrypt(share[1]).decode())))
    decryption_time = time.time() - decryption_time_start

    print(f"Розшифровані частки (RSA): {decrypted_shares}")
    print(f"Час розшифрування (RSA): {decryption_time:.5f} с")
    return decrypted_shares, decryption_time
```

```
Розшифровані частки (ECC): [(1, 13902), (2, 17195), (3, 22224), (4, 28989), (5, 37490)]
Час розшифрування (ECC): 0.00000 с
Відновлений секрет: 12345
```

Наступним кроком буде повернення в початковий вигляд та об’єднання частинок за допомогою інтерполяції Лагранджа, але цей вивід буде показано в кінці.

Паралелльно виконуємо шифрування алгоритмом ECC.

```python
# Шифрування часток за допомогою ECC
def ecc_encrypt_shares(shares):
    private_key = ec.generate_private_key(ec.SECP256R1())
    peer_private_key = ec.generate_private_key(ec.SECP256R1())

    shared_secret = private_key.exchange(ec.ECDH(), peer_private_key.public_key())
    derived_key = HKDF(algorithm=SHA256(), length=32, salt=None, info=b'secret sharing').derive(shared_secret)

    encrypted_shares = []
    encryption_time_start = time.time()
    for share in shares:
        encrypted_shares.append((share[0], int.from_bytes(derived_key[:4], byteorder='big') ^ share[1]))
    encryption_time = time.time() - encryption_time_start

    print(f"Зашифровані частки (ECC): {encrypted_shares}")
    print(f"Час шифрування (ECC): {encryption_time:.5f} с")
    return private_key, peer_private_key, encrypted_shares, encryption_time
```

```
=== Розподіл часток за допомогою ECC ===
Зашифровані частки (ECC): [(1, 1877215477), (2, 1877245328), (3, 1877239915), (4, 1877233542), (5, 1877257417)]
Час шифрування (ECC): 0.00000 с
```

Використовується протокол обміну ключами ECDH (Elliptic Curve Diffie-Hellman**)** для обчислення спільного секрету. Спільний секрет передається в алгоритм HKDF (HMAC-based Key Derivation Function). HKDF створює похідний ключ довжиною 32 байти, використовуючи алгоритм хешування SHA256.

Так само виконуємо розшифрування з використанням приватного ключа.

```python
# Розшифрування часток за допомогою ECC
def ecc_decrypt_shares(private_key, peer_private_key, encrypted_shares):
    shared_secret = private_key.exchange(ec.ECDH(), peer_private_key.public_key())
    derived_key = HKDF(algorithm=SHA256(), length=32, salt=None, info=b'secret sharing').derive(shared_secret)

    decrypted_shares = []
    decryption_time_start = time.time()
    for share in encrypted_shares:
        decrypted_shares.append((share[0], int.from_bytes(derived_key[:4], byteorder='big') ^ share[1]))
    decryption_time = time.time() - decryption_time_start

    print(f"Розшифровані частки (ECC): {decrypted_shares}")
    print(f"Час розшифрування (ECC): {decryption_time:.5f} с")
    return decrypted_shares, decryption_time
```

```
Розшифровані частки (ECC): [(1, 13902), (2, 17195), (3, 22224), (4, 28989), (5, 37490)]
Час розшифрування (ECC): 0.00000 с
Відновлений секрет: 12345
```

Тепер можемо виконати повернення частинок до початкового вигляду за допомогою інтерполяції Лагранджа. Результат цього вже відображений на скрішотах біля дешифрування як «відновлений секрет». В ролі секрету була дана послідовність «12345».

Після цього можемо виконати порівняння двох асиметричних алгоритмів.

```
=== Порівняння ефективності ===
RSA: Час шифрування + розшифрування: 0.02495 с
ECC: Час шифрування + розшифрування: 0.00000 с
ECC працює швидше для цієї задачі.
```

Висновок: у проведеному дослідженні алгоритмів розділення секрету було застосовано комбінацію алгоритму Шаміра для створення часток секрету та асиметричних алгоритмів RSA і ECC для їх шифрування і передачі. Секрет 12345 було розділено на п'ять часток із використанням порогового значення, що дозволяє відновлення секрету за трьома частками. У випадку з RSA зашифровані частки виглядають як довгі випадкові рядки, що свідчить про успішне застосування алгоритму. Однак загальний час шифрування та розшифрування RSA склав 0.02495 секунд, що відображає його більшу обчислювальну складність. Алгоритм ECC продемонстрував значно вищу швидкість. Частки було зашифровано із використанням спільного секрету, згенерованого через протокол ECDH. Зашифровані частки мали вигляд числових значень, отриманих після застосування операції XOR із похідним ключем. Розшифрування було виконане миттєво, а відновлений секрет також дорівнював початковому значенню.

Порівняння алгоритмів показало, що ECC значно перевершує RSA в ефективності для задач розподілу секрету. Це пов'язано з меншою обчислювальною складністю ECC і використанням ключів меншого розміру, що забезпечує швидше виконання операцій шифрування і розшифрування.