НТУУ «Київський політехнічний інститут ім. Ігоря Сікорського»

Навчально-науковий Фізико-технічний інститут

Криптографія

Комп'ютерний практикум №4

Вивчення криптосистеми RSA та алгоритму електронного підпису; ознайомлення з методами генерації параметрів для асиметричних криптосистем

Варіант №6

Виконали:

Студенти 3 курсу НН ФТІ

групи ФБ-31

Гаврилюк Володимир

Гек Роман

Мета роботи:

Ознайомлення з тестами перевірки чисел на простоту і методами генерації ключів для асиметричної криптосистеми типу RSA; практичне ознайомлення з системою захисту інформації на основі криптосхеми RSA, організація з використанням цієї системи засекреченого зв'язку й електронного підпису, вивчення протоколу розсилання ключів.

Порядок виконання:

1. Написати функцію пошуку випадкового простого числа з заданого інтервалу або заданої довжини, використовуючи датчик випадкових чисел та тести перевірки на простоту. В якості датчика випадкових чисел використовуйте вбудований генератор псевдовипадкових чисел вашої мови програмування. В якості тесту перевірки на простоту рекомендовано використовувати тест Міллера-Рабіна із попередніми пробними діленнями. Тести необхідно реалізовувати власноруч, використання готових реалізацій тестів не дозволяється.

2. За допомогою цієї функції згенерувати дві пари простих чисел p, q і p1, q1 довжини щонайменше 256 біт. При цьому пари чисел беруться так, щоб pq <= p1q1; p і q – прості числа для побудови ключів абонента A, p1 і q1 – абонента B.

3. Написати функцію генерації ключових пар для RSA. Після генерування функція повинна повертати та/або зберігати секретний ключ (d, p, q) та відкритий ключ (n, e). За допомогою цієї функції побудувати схеми RSA для абонентів A і B – тобто, створити та зберегти для подальшого використання відкриті ключі (e, n) , (e1, n1) та секретні d і d1.

4. Написати програму шифрування, розшифрування і створення повідомлення з цифровим підписом для абонентів A і B. Кожна з операцій (шифрування, розшифрування, створення цифрового підпису, перевірка цифрового підпису) повинна бути реалізована окремою процедурою, на вхід до якої повинні подаватись лише ті ключові дані, які необхідні для її виконання. За допомогою датчика випадкових чисел вибрати відкрите повідомлення M і знайти криптограму для абонентів A и B, перевірити правильність розшифрування. Скласти для A і B повідомлення з цифровим підписом і перевірити його.

5. За допомогою раніше написаних на попередніх етапах програм організувати роботу протоколу конфіденційного розсилання ключів з підтвердженням справжності по відкритому каналу за допомогою алгоритму RSA. Протоколи роботи кожного учасника (відправника та приймаючого) повинні бути реалізовані у вигляді окремих процедур, на вхід до яких повинні подаватись лише ті ключові дані, які необхідні для виконання. Перевірити роботу програм для випадково обраного ключа 0 < k < n.

Кожна з наведених операцій повинна бути реалізована у вигляді окремої процедури, інтерфейс якої повинен приймати лише ті дані, які необхідні для її роботи; наприклад, функція Encrypt(), яка шифрує повідомлення для абонента, повинна приймати на вхід повідомлення та відкритий ключ адресата (і тільки його), повертаючи в якості результату шифротекст. Відповідно, програмний код повинен містити сім

високорівневих процедур: GenerateKeyPair(), Encrypt(), Decrypt(), Sign(), Verify(), SendKey(), ReceiveKey().

Кожну операцію рекомендується перевіряти шляхом взаємодії із тестовим середовищем, розташованим за адресою:

http://asymcryptwebservice.appspot.com/?section=rsa

Наприклад, для перевірки коректності операції шифрування необхідно а) зашифрувати власною реалізацією повідомлення для серверу та розшифрувати його на сервері, б) зашифрувати на сервері повідомлення для вашої реалізації та розшифрувати його локально.

Виконання роботи

В ході виконання роботи отримали 5 модулів. Кожен модуль відповідає за окремий етап криптографічних перетворень відповідно до вимог методичних вказівок.

arithmetic_ops.py

Це модуль базової арифметики. Він містить низькорівневі математичні функції, необхідні для роботи RSA.

Швидке піднесення до степеню за модулем (a^b (mod m))

```python
def fast_power_mod(base: int, exponent: int, modulus: int) -> int:  10 usages
    """
    Швидке піднесення до степеня за модулем за схемою Горнера

    Args:
        base: основа
        exponent: показник степеня
        modulus: модуль

    Returns:
        result: base^exponent mod modulus
    """
    result = 1
    base = base % modulus

    while exponent > 0:
        if exponent & 1:
            result = (result * base) % modulus
        exponent >>= 1
        base = (base * base) % modulus

    return result
```

Знаходження оберненого елементу за модулем за допомогою розширеного алгоритму Евкліда

```python
def extended_gcd(num1: int, num2: int) -> tuple[int, int, int]:  2 usages
    """
    Розширений алгоритм Евкліда

    Returns:
        (gcd, x, y) де gcd = num1*x + num2*y
    """
    if num2 == 0:
        return num1, 1, 0

    gcd_val, x1, y1 = extended_gcd(num2, num1 % num2)
    x = y1
    y = x1 - (num1 // num2) * y1

    return gcd_val, x, y


def compute_mod_inverse(element: int, modulus: int) -> int:  2 usages
    """
    Знаходження модульного оберненого елемента

    Args:
        element: елемент для якого шукаємо обернений
        modulus: модуль

    Returns:
        inverse: element^(-1) mod modulus

    Raises:
        ValueError: якщо обернений елемент не існує
    """
    gcd_val, x, _ = extended_gcd(element, modulus)

    if gcd_val != 1:
        raise ValueError(f"Обернений елемент не існує: gcd({element}, {modulus}) = {gcd_val}")

    return (x % modulus + modulus) % modulus
```

Також є реалізація класичного алгоритму Евкліда, вона знадобиться

```python
def find_gcd(num1: int, num2: int) -> int:  6 usages
    """
    Алгоритм Евкліда для знаходження НСД

    Args:
        num1, num2: числа для знаходження НСД

    Returns:
        НСД(num1, num2)
    """
    while num2 != 0:
        num1, num2 = num2, num1 % num2
    return num1
```

primality_test.py

Це модуль тестів числа на простоту

Ця функція перевіряє подільність числа на малі прості числа.

```python
# Малі прості числа для тесту пробних ділень (перші 50 простих)
SMALL_PRIMES = [
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
    157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229
]


def trial_division_test(candidate: int) -> bool:  1 usage
    """
    Тест пробних ділень - перевірка на поділ малими простими

    Args:
        candidate: число для перевірки

    Returns:
        True якщо число пройшло тест, False якщо точно складене
    """
    if candidate < 2:
        return False

    if candidate == 2:
        return True

    if candidate % 2 == 0:
        return False

    for prime in SMALL_PRIMES:
        if candidate == prime:
            return True
        if candidate % prime == 0:
            return False

    return True
```

Імовірнісний тест, який перевіряє умову сильної псевдопростоти. Виконується 8 раундів для точності

```python
def miller_rabin_test(candidate: int, rounds: int = 8) -> bool:  1 usage
    if candidate < 2:
        return False
    if candidate == 2 or candidate == 3:
        return True
    if candidate % 2 == 0:
        return False

    s = 0
    d = candidate - 1
    while d % 2 == 0:
        s += 1
        d //= 2

    for _ in range(rounds):
        witness = random.randint( a: 2, candidate - 2)

        if find_gcd(witness, candidate) != 1:
            return False

        x = fast_power_mod(witness, d, candidate)

        if x == 1 or x == candidate - 1:
            continue

        is_composite = True
        for _ in range(s - 1):
            x = fast_power_mod(x,  exponent: 2, candidate)
            if x == candidate - 1:
                is_composite = False
                break
            if x == 1:
                return False

        if is_composite:
            return False

    return True
```

Також, головна функція для цього модулю

```python
def is_prime(candidate: int, test_type: str = "miller_rabin", rounds: int = 8) -> bool:  4 usages
    """
    Перевірка числа на простоту з використанням обраного тесту

    Args:
        candidate: число для перевірки
        test_type: тип тесту ("miller_rabin", "fermat", "solovay_strassen")
        rounds: кількість раундів перевірки

    Returns:
        True якщо число ймовірно просте, False якщо складене
    """
    # Спочатку тест пробних ділень
    if not trial_division_test(candidate):
        return False

    # Потім обраний імовірнісний тест
    if test_type == "miller_rabin":
        return miller_rabin_test(candidate, rounds)
    elif test_type == "fermat":
        return fermat_test(candidate, rounds)
    elif test_type == "solovay_strassen":
        return solovay_strassen_test(candidate, rounds)
    else:
        raise ValueError(f"Невідомий тип тесту: {test_type}")

```

В цьому модулі були реалізовані і інші методи імовірнісних тестів, але далі буде показано як програма відпрацювала саме із тестом Міллера-Рабіна

prime_generation.py

Це модуль для генерації чисел

Функція, яка генерує випадкове непарне число потрібної довжини біт і потім викликає для нього тест

```
10  def generate_random_prime(bit_length: int, test_type: str = "miller_rabin") -> int:
14      Args:
15          bit_length: довжина числа в бітах (мінімум 8)
16          test_type: тип тесту простоти
17
18      Returns:
19          Просте число заданої довжини
20      """
21      if bit_length < 8:
22          raise ValueError("Довжина має бути мінімум 8 біт")
23
24      while True:
25          # Генеруємо випадкове число потрібної довжини
26          candidate = random.randrange(2**(bit_length - 1), 2**bit_length)
27
28          # Робимо непарним
29          if candidate % 2 == 0:
30              candidate += 1
31
32          # Перевіряємо на простоту
33          if is_prime(candidate, test_type):
34              return candidate
```

Функція для генерації просто числа p = 2 * I * p’ + 1

Вона використовує число p’ з попередньої функції, і якщо їй вдається, то повертає саме те стійке число, яке не розкладається на багато маленьких множників, якщо від нього відняти одиницю. Якщо не вдається – генерує звичайне просте число

Це потрібно для того, щоби наше число не було вразливе до таких атак як метод Полларда, де використовується мала теорема ферма, яка каже, що будь-яке число, піднесене до степеня p-1, дає 1 при діленні за модулем p. З цих міркувань за деяким алгоритмом можна дізнатися p

```python
def generate_strong_prime(bit_length: int, test_type: str = "miller_rabin") -> int:  2 usages

    Args:
        bit_length: довжина результуючого числа в бітах (мінімум 16)
        test_type: тип тесту простоти

    Returns:
        "Гарне" просте число
    """
    if bit_length < 16:
        raise ValueError("Довжина має бути мінімум 16 біт")

    max_attempts = 10000
    attempts = 0

    while attempts < max_attempts:
        # Генеруємо просте p' на 1 біт менше
        p_prime = generate_random_prime(bit_length - 1, test_type)

        # Перевіряємо числа виду p = 2*i*p' + 1 для малих i
        for i in range(1, 100):
            p = 2 * i * p_prime + 1

            # Перевіряємо чи не перевищили потрібну довжину
            if p.bit_length() > bit_length:
                break

            # Перевіряємо чи p просте
            if is_prime(p, test_type):
                return p

        attempts += 1

    # Якщо не вдалося згенерувати "гарне" число, повертаємо звичайне просте
    print("Увага: не вдалося згенерувати strong prime, повертаємо звичайне просте")
    return generate_random_prime(bit_length, test_type)
```

Для генерації q ми просто намагаємось згенерувати q відмінне від p таким же ж методом

```python
def generate_safe_prime_pair(bit_length: int, test_type: str = "miller_rabin") -> tuple[int, int]:  2 usages
    Returns:
        (p, q): пара різних простих чисел
    """
    p = generate_strong_prime(bit_length, test_type)

    # Генеруємо q відмінне від p
    while True:
        q = generate_strong_prime(bit_length, test_type)
        if q != p:
            break

    return p, q
```

rsa_system.py

Цей модуль реалізує саму логіку RSA за допомогою попередніх модулів

Клас – контейнер для зберігання ключів

```python
@dataclass  2 usages
class RSAKeyPair:
    """Структура для зберігання пари ключів RSA"""
    public_key: tuple[int, int]  # (e, n)
    private_key: tuple[int, int, int]  # (d, p, q)

    def __repr__(self):
        e, n = self.public_key
        d, p, q = self.private_key
        return (f"RSAKeyPair(\n"
                f"  public=(e={e}, n={n} ({n.bit_length()} bits)),\n"
                f"  private=(d={d}, p={p}, q={q})\n"
                f")")
```

Функція, яка генерує ключі

```python
def GenerateKeyPair(bits: int = 512, public_exponent: int = 65537) -> RSAKeyPair:  3 usages
    """
    Генерація пари ключів RSA

    Args:
        bits: бітова довжина модуля n (буде згенеровано два простих по bits//2 біт)
              Рекомендується мінімум 512 біт, для реальних систем 2048+ біт
        public_exponent: відкрита експонента e (за замовчуванням 65537 = 2^16 + 1)

    Returns:
        RSAKeyPair з відкритим та секретним ключами
    """
    if bits < 32:
        raise ValueError("Бітова довжина має бути мінімум 32")

    # Генеруємо два простих числа p та q
    half_bits = bits // 2
    p, q = generate_safe_prime_pair(half_bits)

    # Обчислюємо модуль n та функцію Ойлера φ(n)
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Перевіряємо чи e взаємно просте з φ(n)
    e = public_exponent
    if find_gcd(e, phi_n) != 1:
        raise ValueError(f"Публічна експонента e={e} не взаємно проста з φ(n)")

    # Обчислюємо секретну експоненту d
    d = compute_mod_inverse(e, phi_n)

    return RSAKeyPair(
        public_key=(e, n),
        private_key=(d, p, q)
    )
```

Шифрування, дешифрування

```python
def Encrypt(plaintext: int, public_exponent: int, modulus: int) -> int:  5 usages

    if plaintext < 0 or plaintext >= modulus:
        raise ValueError(f"Повідомлення має бути в діапазоні [0, {modulus-1}]")

    return fast_power_mod(plaintext, public_exponent, modulus)


def Decrypt(ciphertext: int, private_exponent: int, modulus: int) -> int:  5 usages

    if ciphertext < 0 or ciphertext >= modulus:
        raise ValueError(f"Шифротекст має бути в діапазоні [0, {modulus-1}]")

    return fast_power_mod(ciphertext, private_exponent, modulus)
```

Функції для цифрового підпису

```python
def Sign(message: int, private_exponent: int, modulus: int) -> tuple[int, int]:  4 usages

    if message < 0 or message >= modulus:
        raise ValueError(f"Повідомлення має бути в діапазоні [0, {modulus-1}]")

    signature = fast_power_mod(message, private_exponent, modulus)
    return (message, signature)


def Verify(message: int, signature: int, public_exponent: int, modulus: int) -> bool:  4 usages

    if message < 0 or message >= modulus:
        return False

    recovered = fast_power_mod(signature, public_exponent, modulus)
    return message == recovered
```

Надсилання та отримання ключів

```python
def SendKey(secret_key: int,  2 usages
            sender_private_exp: int, sender_modulus: int,
            receiver_public_exp: int, receiver_modulus: int) -> tuple[int, int]:

    if sender_modulus > receiver_modulus:
        raise ValueError("Модуль відправника має бути <= модуля отримувача")

    # Підписуємо ключ секретним ключем відправника
    _, signature = Sign(secret_key, sender_private_exp, sender_modulus)

    # Шифруємо ключ та підпис відкритим ключем отримувача
    encrypted_key = Encrypt(secret_key, receiver_public_exp, receiver_modulus)
    encrypted_signature = Encrypt(signature, receiver_public_exp, receiver_modulus)

    return (encrypted_key, encrypted_signature)


def ReceiveKey(encrypted_key: int, encrypted_signature: int,  2 usages
               receiver_private_exp: int, receiver_modulus: int,
               sender_public_exp: int, sender_modulus: int) -> tuple[int, int, bool]:

    # Розшифровуємо ключ та підпис секретним ключем отримувача
    key = Decrypt(encrypted_key, receiver_private_exp, receiver_modulus)
    signature = Decrypt(encrypted_signature, receiver_private_exp, receiver_modulus)

    # Перевіряємо підпис відкритим ключем відправника
    verified = Verify(key, signature, sender_public_exp, sender_modulus)

    return (key, signature, verified)
```

Протокол цифрового підпису забезпечує автентифікацію джерела і цілісність повідомлення. Схема – підпис і саме повідомлення (ключ) шифруються окремо.

Алгоритм для протоколу:
- Генерація підпису (на стороні A) – в результаті (M, S)

- Перевірка підпису (на стороні B) – якщо m == recovered, то True, інакше False

Протокол конфіденційного розсилання ключів з автентифікацією

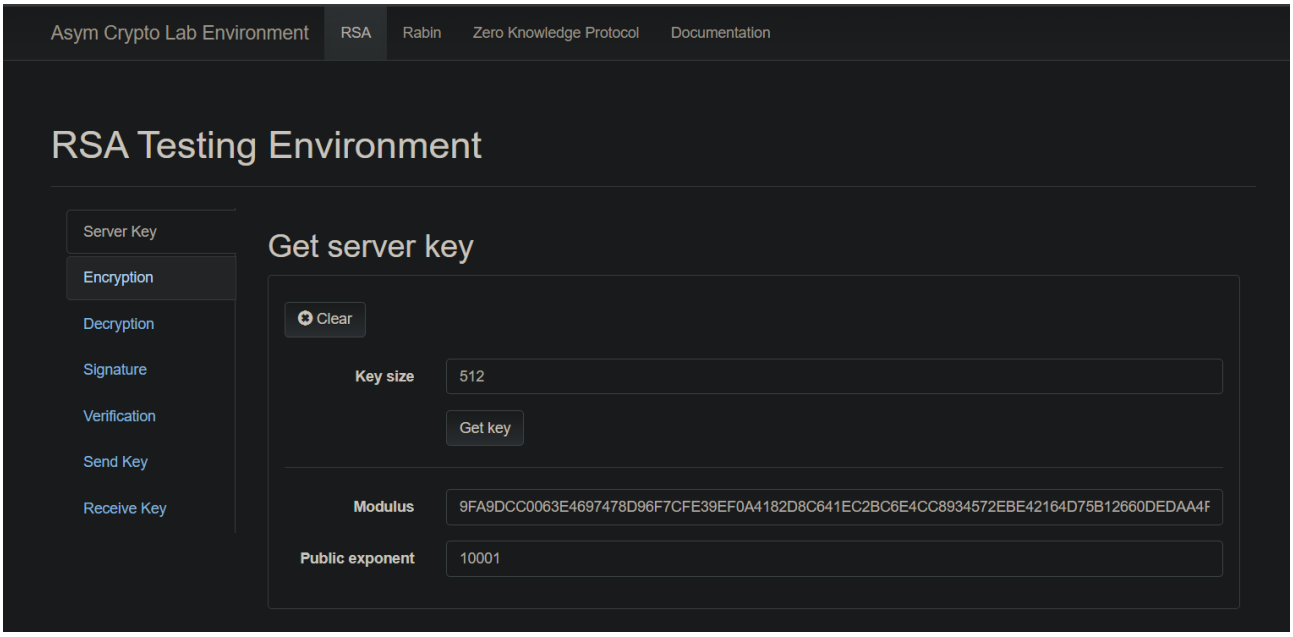
Алгоритм:
- Формування захищеного пакету в SendKey (підпис ключа, перевірка модулів (модуль A не може бути більшим за модуль B), шифрування підпису і відправка)

- Отримання і перевірка в ReceiveKey (дешифрування ключа приватним ключем отримувача, дешифрування підпису, верифікація справжності ключа)

Приклад виводу програми:

```
================================================================================
 ЛОКАЛЬНЕ ТЕСТУВАННЯ RSA
================================================================================

[1] Генерація ключової пари для абонента A...

Абонент A:
  p = 623549934864357100446656105350960705437286440946505362423680227838858356068l9
  q = 942406892409661735683927417052982103633055129525261424063397844097232563713O7
  n = p*q = 587637756377765765212197407820504847035157162358032775957474139340938118319782628548874297422235307679428459168971781607021818927317372001891444432514243 3
  n (6іт): 511
  e = 65537
  d = 22967699360081282444894390346403148842341808007386681654867784731095609961946428598254587314868978679584503733253992123335806565391879119856451957595690133

[2] Генерація ключової пари для абонента B...

Абонент B:
  p = 92724403501883055390101091375273614774365015735743406798318005639114561017903
  q = 75950079804272542090968370077825944864947646349247873190186093507579884388807
  n = p*q = 7042425845771586423967728306266613880124095769133573286277421587100238503843654664797721446614437186970518589118421958846755474170818185684341788939811721
  n (6іт): 512
  e = 65537
  d = 44144513201724604709250995094334309597536960540118761031716855550248698899307909725514822858883924235682121454531692004198002366241691586310236916081702 9

================================================================================
 ТЕСТУВАННЯ ШИФРУВАННЯ ТА РОЗШИФРУВАННЯ
================================================================================

Випадкове повідомлення M = 151091565031074553560996051841024489678292233618057456528421253724195560312882113283231797838966405434330283074736398079681195268105274422763328963905767 5

[1] Шифрування для абонента A:
  Шифротекст C = 20614702303997496137799396955137866019369665169330868149436581082634289152422838561627529502127048987933663791693488264679900955623742956214177582426242 60
  Розшифроване M' = 151091565031074553560996051841024489678292233618057456528421253724195560312882113283231797838966405434330283074736398079681195268105274422763328963905767 5
  Перевірка: M == M' ? True

[2] Шифрування для абонента B:
  Повідомлення M = 37487801325438084667577466718119299452671834082652671837533454783966220056572401197052563656027078221663519956118021851444492752460817911556314559188690 08
  Шифротекст C = 59322505091896465151651300897107495938940965124809135691025094573416425181152124208926931377024131029886603803874125296139402796624970331614421502237297 8
  Розшифроване M' = 37487801325438084667577466718119299452671834082652671837533454783966220056572401197052563656027078221663519956118021851444492752460817911556314559188690 08
  Перевірка: M == M' ? True

================================================================================
 ТЕСТУВАННЯ ЦИФРОВОГО ПІДПИСУ
================================================================================

[1] Підпис повідомлення абонентом A:
  Повідомлення M = 1070467288254901375616445036664901756831861172068470811419197120499244077324349615754897148003694329879705696054975351754970953006432543325512940641863947
  Підпис S = 3133657849069114331169524138932422901362207885768604000190799495059461295424200114262602125955961466105514751333998486729136836482321351367582029684620469
  Перевірка підпису: True

[2] Підпис повідомлення абонентом B:
  Повідомлення M = 16464338445500473089341933244757173016029250828158126095713190746635827986447946224234560235225513046769865609641349947269421271919920226698150429157159
  Підпис S = 2257254036113444439069034953885845530296216082540858701643279551903996611561819688775827466597002372984155124056863281755718044619665609507389503801056424
  Перевірка підпису: True

================================================================================
 ПРОТОКОЛ КОНФІДЕНЦІЙНОГО РОЗСИЛАННЯ КЛЮЧІВ
================================================================================

[1] Генерація секретного ключа та підготовка до відправки:
  Секретний ключ k = 2985339730219424130

[2] Абонент A підписує та шифрує ключ для абонента B:
  k1 (зашифрований ключ) = 5964676533638674866417234366987155757169495397943331099773231366008644747278987350596699641853494196387896553837454763492935271400146036552304675188013 87
  S1 (зашифрований підпис) = 8137625185679322752978725973234749382319578209557989904459400607951672976539594595142775287162412165426705976731033234597322788827058553085349712031968 86

[3] Абонент B отримує, розшифровує та перевіряє:
  Розшифрований ключ k = 2985339730219424130
  Розшифрований підпис S = 2223016237089095039464681300492894059210985315578840827763116930368510733457226595795301114876030377571221334353162355524072164821191014208983140078362689
  Підпис перевірено: True
  Ключ коректний: True

================================================================================
 ЗАВЕРШЕННЯ ЛОКАЛЬНОГО ТЕСТУВАННЯ
================================================================================

Всі локальні операції виконано успішно!

================================================================================
Програма завершена успішно!
================================================================================
```

Перевірка на сайті

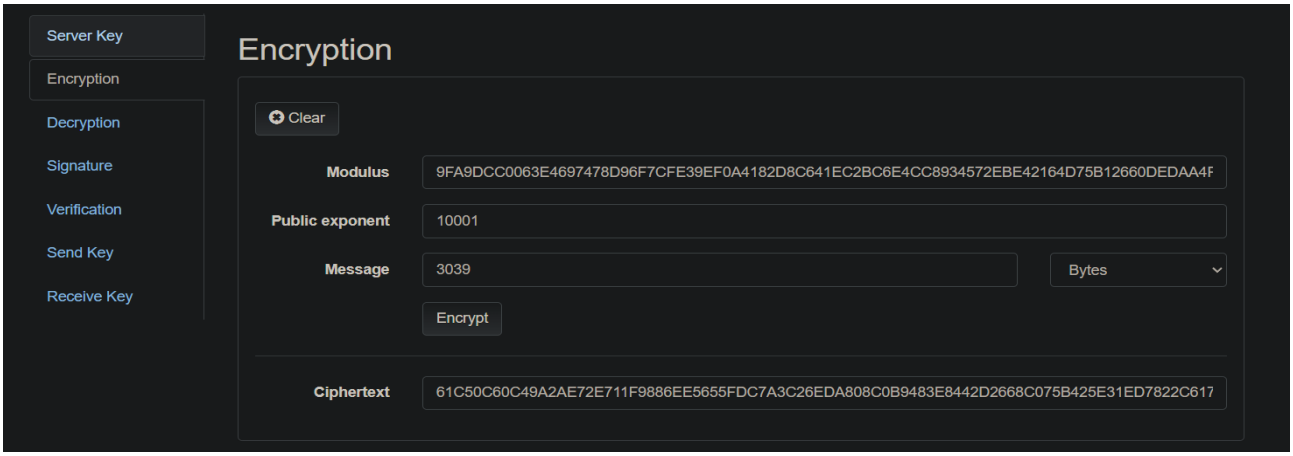


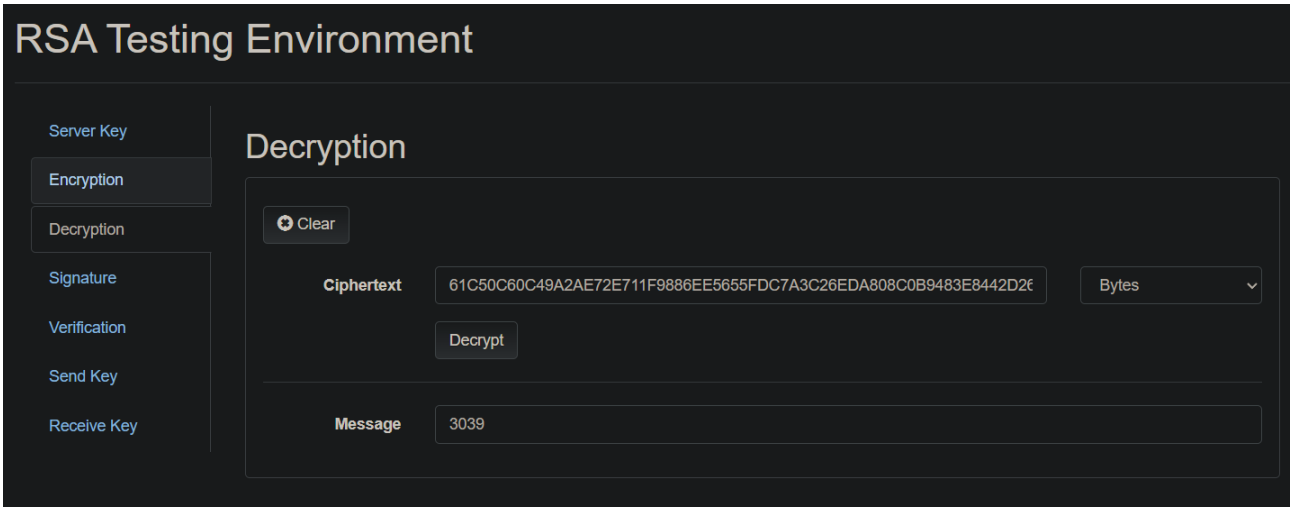
Було написано додатковий скрипт, який перевіряє правильність нашої системи із параметрами із сайту http://asymcryptwebservice.appspot.com/?section=rsa

main.py verify_server.py arithmetic_ops.py primality_tests.py prime_generation.py rsa_system.py

```python
from rsa_system import Encrypt

server_n_hex = "9FA9DCC0063E4697478D96F7CFE39EF0A4182D8C641EC2BC6E4CC8934572EBE42164D75B12660DEDAA4F793E153319E3BF273516E3D5EEDFBDA7B2CA4F428E65"
server_e_hex = "10001"

server_n = int(server_n_hex, 16)
server_e = int(server_e_hex, 16)

print(f"Server N (int): {server_n}")
print(f"Server E (int): {server_e}")

message = 0x3039 # 12345
print(f"\nНаше повідомлення: {message}")

ciphertext = Encrypt(message, server_e, server_n)

ciphertext_hex = hex(ciphertext)[2:].upper()

print(f"\nЗашифроване повідомлення (HEX):")
print(ciphertext_hex)
print("-" * 60)
```

```
C:\Users\vovan\PycharmProjects\crypto25-26\.venv\Scripts\python.exe C:\Users\vovan\PycharmProjects\crypto25-26\lab4\havryliuk_fb-31_gek_fb-31_cp4\verify_server.py
Server N (int): 836225731111389580408571814646778647948948773343275284198417223651291091463137066792287387608650440465953226506522947838381158357552273030941043753134038 9
Server E (int): 65537

Наше повідомлення: 12345

Зашифроване повідомлення (HEX):
61C50C60C49A2AE72E711F9886EE5655FDC7A3C26EDA808C0B9483E8442D2668C075B425E31ED7822C617444ECF8B38BD0176ED70726055A2F257D66703686D7
------------------------------------------------------------
```





Працює коректно

Висновки

Ми ознайомилися із шифруванням RSA. Алгоритм RSA зарекомендував себе як надійний стандарт асиметричного шифрування. Розробка програмних модулів для основних криптографічних операцій дала змогу створити функціональну систему захисту даних. Висока стійкість RSA гарантується складністю математичних задач, що лежать в його основі. Виконання даної роботи сприяло глибокому засвоєнню принципів функціонування систем з відкритим ключем.