Python RSA: A Simple Implementation Demo

https://github.com/Kaustub26Pvgda/rsa-example

REPORT

Kaustub Pavagada
CS22B1042

Implementation

The RSA implementation follows the standard key generation, encryption, and decryption procedures defined by the RSA algorithm. The modular structure separates key components into individual functions for clarity and maintainability.

The key steps for the implementation include:

- Prime number generation using the SymPy library's randprime function to select large random primes
- Key generation involving computation of modulus (n), totient (ϕ), public exponent (e), and private exponent (d)
- Modular inverse calculation for deriving the private key exponent
- Encryption by converting the plaintext message into an integer and applying modular exponentiation
- Decryption by performing modular exponentiation with the private key and recovering the original plaintext
- Color-coded output for better readability when displaying key values, ciphertext, and decrypted messages

Code

```
import sympy

# We need to generate keys: private and public

# RSA is based on the difficulty in factoring very large prime numbers, so
first we generate those numbers
def generate_prime(bits=512):
    # generate a random prime number in the range [2^511, 2^512]
    return sympy.randprime(2**(bits - 1), 2**bits)

# we need to calculate the modular inverse to get the private key
def mod_inverse(e, phi):
    return pow(e, -1, phi)
```

```
# our keys will be generated in this function
def generate_keys(bits=512):
    p = generate_prime(bits)
    q = generate_prime(bits)
    # calculate modulus (n) and totient (phi)
    n = p * q
    phi = (p - 1) * (q - 1)
    # public exponent e (used in public key)
    e = 65537
    # private exponent d (used in private key)
    d = mod_inverse(e, phi)

    return ((e, n), (d, n))

# encryption part:
def encrypt(message, public_key):
    e, n = public_key
    # convert message to integer
    message_in_int = int.from_bytes(message.encode(), 'big')
    # ciphertext c = [message ^ (public key exponent)] mod (modulus)
    ciphertext = pow(message_in_int, e, n)
    return ciphertext

def decrypt(ciphertext, private_key):
    d, n = private_key
    # decrypted text = [ciphertext ^ (private key exponent)] mod (modulus)
    message_in_int = pow(ciphertext, d, n)
    decrypted_text = message_in_int.to_bytes((message_in_int.bit_length() + 7)
// 8, 'big').decode()
    return decrypted_text

def main():
    YELLOW = "\033[33m"
    GREEN = "\033[32m"
    RESET = "\033[0m"

    # generate the keys
    public_key, private_key = generate_keys(512)

    # input message
    print(f"\n{YELLOW}**************RSA Example**************")
    message = input(f"{YELLOW}Enter your message:{RESET} ")
    ciphertext = encrypt(message, public_key)
    print(f"\n{GREEN}Public exponent e:{RESET} {public_key[0]}")
    print(f"\n{GREEN}Private exponent d:{RESET} {private_key[0]}")
    print(f"\n{GREEN}Modulus (product of the two primes){RESET} =
{public_key[1]}")
    print(f"\n{GREEN}Ciphertext:{RESET} {ciphertext}")

    decrypted_message = decrypt(ciphertext, private_key)
    print(f"\n{YELLOW}Decrypted message:{RESET} {decrypted_message}")

if __name__ == "__main__":
    main()
```

Results

The implementation was tested using sample plaintext messages. The encryption and decryption results matched the original input, verifying the correctness of the RSA implementation.