Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 22/10/2024

Assignment 9

PRN: 21510042 **Name:** Omkar Rajesh Auti

Digital Signature System Documentation

Overview

This Python program implements a digital signature system using RSA encryption and SHA-256 hashing. It provides the ability to generate RSA key pairs (private and public), sign a message by hashing it and creating a signature, verify the signature against the hash of the message, and save the RSA keys to files. The program is menu-driven and operates through a command-line interface.

Key Concepts:

- **RSA Encryption**: An asymmetric encryption method that uses a public-private key pair for secure operations.
- **SHA-256 Hashing**: A cryptographic hash function producing a 256-bit hash value to ensure message integrity.
- **Digital Signature**: A method of verifying the authenticity of a message using cryptographic signatures.

Dependencies

The program uses the following libraries:

- cryptography.hazmat.backends.default_backend
- cryptography.hazmat.primitives.asymmetric.rsa
- cryptography.hazmat.primitives.asymmetric.padding
- cryptography.hazmat.primitives.hashes
- cryptography.hazmat.primitives.serialization
- cryptography.exceptions.InvalidSignature
- hashlib

To install the required libraries, use:

bash

Copy code

pip install cryptography

Functions

1. generate_keys()

Generates a new RSA private and public key pair with a key size of 2048 bits.

- **Returns**:
    - private_key: RSA private key object.
    - public_key: RSA public key object.

2. sign_message(private_key, message)

Hashes the message using SHA-256 and signs the hash using the private key.

- **Parameters**:
    - private_key: RSA private key used to sign the message.
    - message: The message to be signed.
- **Returns**:
    - signature: The digital signature.
    - message_hash: The SHA-256 hash of the message in hexadecimal format.

Note: The message hash is printed to the console.

3. verify_signature(public_key, message_hash, signature)

Verifies the signature by checking it against the provided message hash using the public key.

- **Parameters**:
    - public_key: RSA public key used to verify the signature.
    - message_hash: The SHA-256 hash of the message (hexadecimal format).
    - signature: The signature to verify.
- **Returns**:
    - True: If the signature is valid.
    - False: If the signature is invalid.

4. save_keys_to_file(private_key, public_key)

Saves the RSA private and public keys to files in PEM format (private_key.pem and public_key.pem).

- **Parameters**:
  - private_key: The RSA private key.
  - public_key: The RSA public key.

The private key is saved in an unencrypted format for simplicity.

Menu Interface

The program provides a menu with the following options:

1. Generate RSA Keys

Generates a new RSA key pair (private and public).

2. Sign a Message

Prompts the user to input a message, hashes it, and signs the hash using the private key. The message hash is displayed for user reference.

3. Verify Signature

Prompts the user to input the hash of the message and verifies the signature using the public key.

4. Save Keys to Files

Saves the generated private and public keys to files (private_key.pem and public_key.pem).

5. Exit

Exits the program.

Usage

1. **Generating Keys**: After starting the program, select the option to generate RSA keys.
2. **Signing a Message**: Sign a message by selecting the appropriate menu option and entering the message.
3. **Verifying a Signature**: Verify the signature by inputting the hash of the message.
4. **Saving Keys**: Save the generated RSA keys to files for later use.

Security Considerations

- Ensure that private keys are kept secure and not shared.
- Private keys can be stored encrypted for additional security, though this implementation stores them unencrypted for simplicity.

Code:

```
from cryptography.hazmat.backends import default_backend

from cryptography.hazmat.primitives.asymmetric import rsa,
padding

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.exceptions import InvalidSignature

import hashlib
```

```
# Function to generate RSA private and public keys
def generate_keys():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    public_key = private_key.public_key()

    return private_key, public_key

# Function to sign a message and print its hash
def sign_message(private_key, message):
    # Hash the message using SHA-256
    message_hash =
hashlib.sha256(message.encode()).hexdigest()

    # Sign the hashed message using RSA private key
    signature = private_key.sign(
        bytes.fromhex(message_hash),  # Convert the hex hash to
bytes
        padding.PSS(
```

```
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    print(f"Hash of the message: {message_hash}")  # Print the
message hash
    return signature, message_hash

# Function to verify the signature using the provided hash
def verify_signature(public_key, message_hash, signature):
    try:
        # Verify the signature using RSA public key
        public_key.verify(
            signature,
            bytes.fromhex(message_hash),  # Convert the hex hash
back to bytes
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
```

```
        )
        return True
    except InvalidSignature:
        return False


# Function to save keys to files
def save_keys_to_file(private_key, public_key):
    # Save private key
    with open("private_key.pem", "wb") as private_file:
        private_file.write(
            private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption()
            )
        )
    # Save public key
    with open("public_key.pem", "wb") as public_file:
        public_file.write(
            public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
```

```
        )
    )
    print("Keys saved to files: private_key.pem and
public_key.pem")

# Menu for the digital signature system
def menu():
    private_key, public_key = None, None
    signature = None
    message_hash = None

    while True:
        print("\n===== Digital Signature System =====")
        print("1. Generate RSA Keys")
        print("2. Sign a Message")
        print("3. Verify Signature")
        print("4. Save Keys to Files")
        print("5. Exit")

        choice = input("Enter your choice (1/2/3/4/5): ")

        if choice == '1':
```

```
            # Generate RSA private and public keys
            private_key, public_key = generate_keys()
            print("\nRSA Keys Generated!")

        elif choice == '2':
            # Sign a message
            if private_key is None:
                print("You need to generate RSA keys first.")
            else:
                message = input("Enter the message to sign: ")
                signature, message_hash = sign_message(private_key,
message)
                print("\nMessage signed successfully!")

        elif choice == '3':
            # Verify the signature
            if public_key is None or message_hash is None or
signature is None:
                print("You need to sign a message first.")
            else:
                input_hash = input("Enter the hash of the message to
verify: ")
```

```
            verification_result = verify_signature(public_key,
input_hash, signature)
            if verification_result:
                print("\nSignature verified successfully! The message
is authentic.")
            else:
                print("\nSignature verification failed! The message is
not authentic.")

        elif choice == '4':
            # Save RSA keys to files
            if private_key is None or public_key is None:
                print("You need to generate RSA keys first.")
            else:
                save_keys_to_file(private_key, public_key)

        elif choice == '5':
            print("Exiting the program...")
            break

        else:
            print("Invalid choice. Please try again.")
```

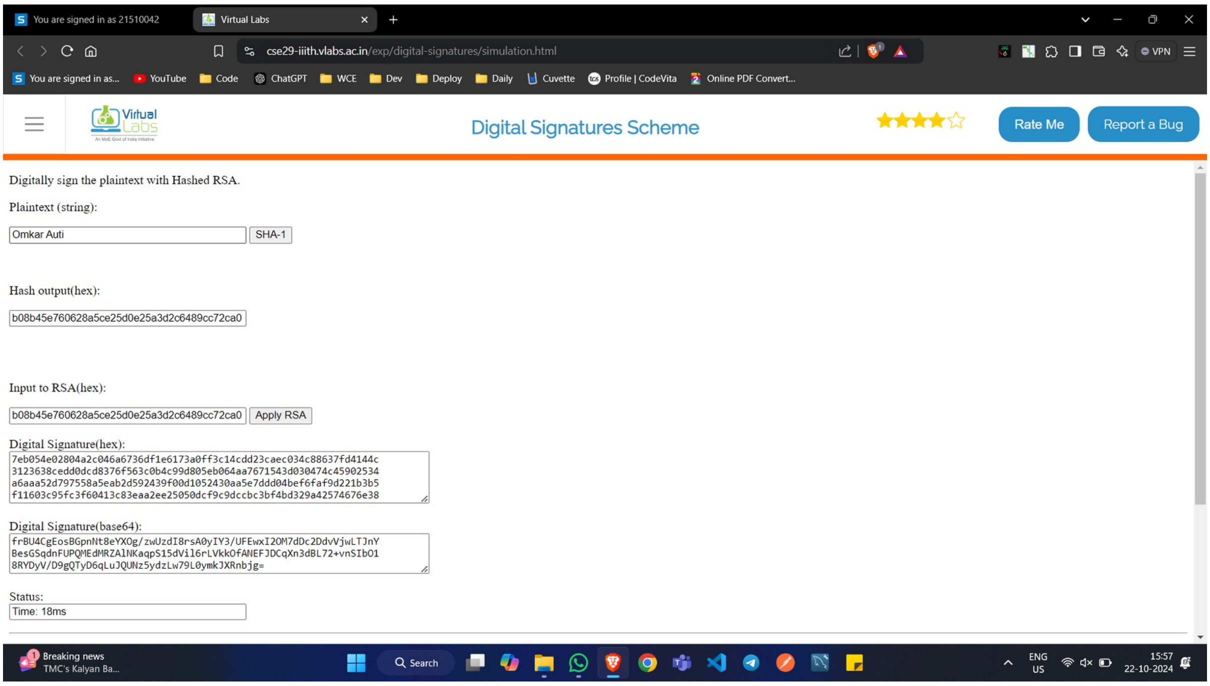
```
if __name__ == "__main__":
    menu()
```

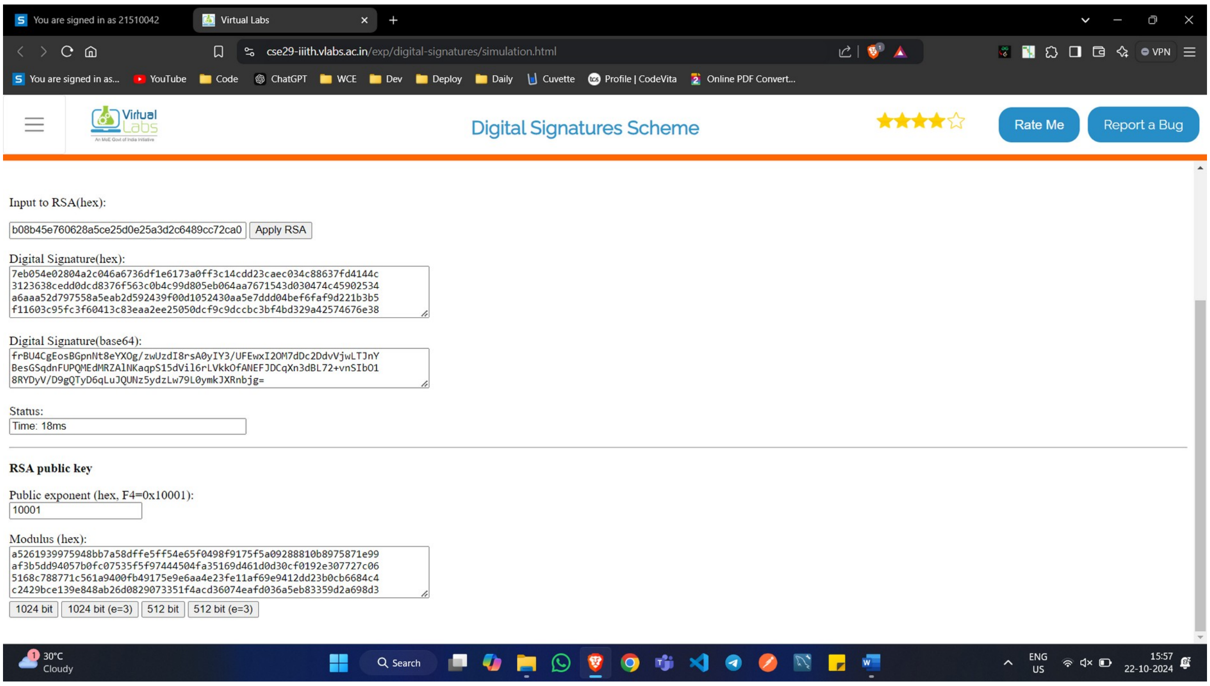
Output:

```
===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5): 1

RSA Keys Generated!

===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5): 2
Enter the message to sign: Omkar Auti
Hash of the message: d5eadc6ba2bc54d3df9a539bbf8ab494750a54a5b9af176b6bc3c69018665df5

Message signed successfully!
```

```
Message signed successfully!

===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5): 3
Enter the hash of the message to verify: d5eadc6ba2bc54d3df9a539bbf8ab494750a54a5b9af176b6bc3c69018665df5

Signature verified successfully! The message is authentic.

===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5):
```

VLAB


cse29-iiith.vlabs.ac.in/exp/digital-signatures/simulation.html
Digital Signatures Scheme
Rate Me
Report a Bug
Digitally sign the plaintext with Hashed RSA.
Plaintext (string):
Omkar Auti
SHA-1
Hash output(hex):
b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
Input to RSA(hex):
b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
Apply RSA
Digital Signature(hex):
7eb054e02804a2c046a6736df1e6173a0ff3c14cdd23caec034c88637fd4144c
3123638cedd0dcd8376f563c0b4c99d805eb064aa7671543d030474c45902534
a6aaa52d797558a5eab2d592439f00d1052430aa5e7ddd04bef6faf9d221b3b5
f11603c95fc3f60413c83eaa2ee25050dcf9c9dccbc3bf4bd329a42574676e38
Digital Signature(base64):
frBU4CgEosBGpnNt8eYXOg/zwUzdI8rsA0yIY3/UFEwxI2OM7dDc2DdvVjwLTJnY
BesGSqdnFUPQMEdMRZAlNKaqpS15dVil6rLVkkOfANEFJDCqXn3dBL72+vnSIbO1
8RYDyV/D9gQTyD6qLuJQUNz5ydzLw79L0ymkJXRnbjg=
Status:
Time: 18ms


cse29-iiith.vlabs.ac.in/exp/digital-signatures/simulation.html
Digital Signatures Scheme
Rate Me
Report a Bug
Input to RSA(hex):
b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
Apply RSA
Digital Signature(hex):
7eb054e02804a2c046a6736df1e6173a0ff3c14cdd23caec034c88637fd4144c
3123638cedd0dcd8376f563c0b4c99d805eb064aa7671543d030474c45902534
a6aaa52d797558a5eab2d592439f00d1052430aa5e7ddd04bef6faf9d221b3b5
f11603c95fc3f60413c83eaa2ee25050dcf9c9dccbc3bf4bd329a42574676e38
Digital Signature(base64):
frBU4CgEosBGpnNt8eYXOg/zwUzdI8rsA0yIY3/UFEwxI2OM7dDc2DdvVjwLTJnY
BesGSqdnFUPQMEdMRZAlNKaqpS15dVil6rLVkkOfANEFJDCqXn3dBL72+vnSIbO1
8RYDyV/D9gQTyD6qLuJQUNz5ydzLw79L0ymkJXRnbjg=
Status:
Time: 18ms
RSA public key
Public exponent (hex, F4=0x10001):
10001
Modulus (hex):
a5261939975948bb7a58dffe5ff54e65f0498f9175f5a09288810b8975871e99
af3b5dd94057b0fc07535f5f97444504fa35169d461d0d30cf0192e307727c06
5168c788771c561a9400fb49175e9e6aa4e23fe11af69e9412dd23b0cb6684c4
c2429bce139e848ab26d0829073351f4acd36074eafd036a5eb83359d2a698d3
1024 bit
1024 bit (e=3)
512 bit
512 bit (e=3)