Homework #1: Mini-RSA

Due: ~~Sep 21, 11:59 PM~~ Sep 26, 11:59 PM

1 Assignment Description

In this programming assignment, you will implement the RSA encryption and digital signature algorithms in Python 3. You will create a program, `rsa_program.py`, that allows users to generate RSA key pairs, encrypt and decrypt messages using RSA, and create and verify digital signatures.

2 Submission

- Late submission will be assessed a penalty of 10% per day (We will only accept late submissions of up to 3 days).
- Submit one ZIP file (`<id>-hw1.zip`) to Blackbored.
- Do not import and use 3rd-party modules (e.g., `pycrypto`). We will evaluate your program without doing any `pip install`.
- Compress the following files into a single ZIP archive `<id>-hw1.zip`:
  - `rsa_program.py`: This file will contain the program for implementing the RSA cryptosystem.
  - `coding_document.pdf`: This document should provide a clear and concise explanation of the RSA implementation, including how the program works, the purpose of each function, and any important algorithms or mathematical concepts involved. The coding document can be written in either English or Korean.

3 Guidelines

Several updated notes:

- Both standard and file output must not contain spaces, including new line, at the end of the text.
- For encryption, decryption, and signing implementation, you only need to consider processing one line of text. Don't consider multiple lines of text.

3.1 RSA Key Generation

1. Implement a program that generates RSA key pairs.
2. Allow the user to input prime numbers p and q.
3. Log and display the following key components:
   - Modulus n $(n = p \times q)$
   - Public exponent e
   - Private exponent d

- Totient function $\phi(n)$ ($\phi(n) = (p-1)(q-1)$)

4. Ensure that e is chosen deterministically based on the input p and q. Specifically:
   - Calculate $(p-1)(q-1)$.
   - Choose e as the *smallest* prime number greater than 2 and less than $(p-1)(q-1)$ (hint: use the Euclidean Algorithm).

5. Your program should be invoked from the command line with the following syntax:

   ```
   $python3 rsa_program.py --generate-key --p <prime_p> --q <prime_q>
   ```

   - Output #1 (stdout): displayed in the terminal

     ```
     RSA key pair generated:
     n=<modulus>
     e=<public exponent>
     d=<private exponent>
     phi=<Totient function>
     ```

   - Output #2 (file): `public_key.txt` and `private_key.txt`
     - Public Key (`public_key.txt`)

       ```
       n=<modulus in integer format>
       e=<public exponent in integer format>
       ```

     - Private Key (`private_key.txt`)

       ```
       n=<modulus in integer format>
       d=<private exponent in integer format>
       ```

3.2 RSA Encryption and Decryption

Implement functions for (1) RSA encryption and (2) RSA decryption:

- Allow users to use their generated keys (`public_key.txt` and `private_key.txt`).

- For encryption, your program should read the content of the `<plaintext_file>` and encrypt it using the `<public_key_file>`. You should convert each character in the message to its corresponding ASCII (decimal) value, encrypt each ASCII value using the public key, and then display the results in a hexadecimal sequence (Refer to §4.2):

  ```
  $python3 rsa_program.py --encrypt <plaintext_file> --public-key <public_key_file> --output
  <ciphertext_file>
  ```

  - Output #1 (stdout): displayed in the terminal

    ```
    Ciphertext: <Ciphertext in hexadecimal format>
    ```

  - Output #2 (`<ciphertext_file>`)

    ```
    <Ciphertext in hexadecimal format>
    ```

- For decryption, your program should read the content of the `<ciphertext_file>` and decrypt it using the `<private_key_file>`:

  ```
  $python3 rsa_program.py --decrypt <ciphertext_file> --private-key <private_key_file> --output
  <plaintext_file>
  ```

  - Output #1 (stdout): displayed in the terminal

    ```
    Decrypted plaintext: <Plaintext>
    ```

  - Output #2 (`<plaintext_file>`)

    ```
    <Plaintext>
    ```

3.3 Digital Signatures

Implement functions for (1) creating and (2) verifying digital signatures using RSA:

- To create a digital signature, sign the message using the private key and show the signature.
- For creating signature, your program should follow the following format. You should convert each character in the message to its corresponding ASCII value, encrypit each ASCII value using the private key, and then present it as a hexadecimal sequence (Refer to §4.3):

  ```
  $python3 rsa_program.py --sign <verification_message> --private-key <private_key_file>
  --signature <signature_file>
  ```

  - Output #1 (stdout): displayed in the terminal

    ```
    Signature: <Signature in hexadecimal format>
    ```

  - Output #2 (`<signature_file>`)

    ```
    <Signature in hexadecimal format>
    ```

- To verify a digital signature, verify it using the public key and show whether the given verification message is same with the signed message or not.
- For verifying signature, your program should follow the following format:

  ```
  $python3 rsa_program.py --verify <verification_message> --signature <signature_file>
  --public-key <public_key_file>
  ```

  - Output (stdout): displayed in the terminal
    If the signature is valid, show the following content:

    ```
    Signature is valid
    ```

    If the signature is invalid, show the following content:

    ```
    Signature is invalid
    ```

4 Examples

Here are examples of the expected standard output, file output, and command-line usage for each part of the assignment:

4.1 Generate RSA Key Pairs

- **Command-line usage:**

  ```
  $python3 rsa_program.py --generate-key --p 61 --q 53
  ```

  - **Expected standard output:**

    ```
    RSA key pair generated:
    n=3233
    e=7
    d=1783
    phi=3120
    ```

- **Expected file output (`public_key.txt`):**

  ```
  n=3233
  e=7
  ```

- **Expected file output (`private_key.txt`):**

  ```
  n=3233
  d=1783
  ```

4.2 RSA Encryption and Decryption

- **Command-line usage (encryption) - In this example, let's assume that the content of the `plaintext.txt` is "`Hello, RSA!`".**

```
$python3 rsa_program.py --encrypt plaintext.txt --public-key public_key.txt
--output ciphertext.txt
```

  - **Expected standard output (encryption, e.g., `H` is mapped to `0x43f`):**

    ```
    Ciphertext: 0x43f 0xbff 0x755 0x755 0xc6f 0x469 0xad6 0x435 0x721 0x525 0x971
    ```

  - **Expected file output (`ciphertext.txt`):**

    ```
    0x43f 0xbff 0x755 0x755 0xc6f 0x469 0xad6 0x435 0x721 0x525 0x971
    ```

- **Command-line usage (decryption):**

```
$python3 rsa_program.py --decrypt ciphertext.txt --private-key private_key.txt
--output decrypted.txt
```

  - **Expected standard output (decryption):**

    ```
    Decrypted plaintext: Hello, RSA!
    ```

  - **Expected file output (`decrypted.txt`):**

    ```
    Hello, RSA!
    ```

4.3 Digital Signatures

- **Command-line usage (signature creation):**

```
$python3 rsa_program.py --sign ThisIsMySign --private-key private_key.txt
--signature signature.txt
```

  - **Expected standard output (signature creation):**

    ```
    Signature: 0x8b0 0xfb 0x561 0x8cb 0xb0f 0x8cb 0x314 0x392 0x4c 0x561 0x76f 0x4e8
    ```

  - **Expected file output (`signature.txt`):**

    ```
    0x8b0 0xfb 0x561 0x8cb 0xb0f 0x8cb 0x314 0x392 0x4c 0x561 0x76f 0x4e8
    ```

- **Command-line usage (signature verification):**

```
$python3 rsa_program.py --verify ThisIsMySign --signature signature.txt
--public-key public_key.txt
```

  - **Expected standard output (signature verification):**

    ```
    Signature is valid
    ```