Carnegie Mellon University: Cryptography

Solution and insights for lab-5 problems

Author's name: Abel Jotie

Date: Dec. 7, 2024

1. **Post-quantum cryptography:**

   Key encapsulation mechanisms are sets of protocols for exchanging secret messages in a secure manner. The underlying hard problem for these exchange mechanisms is based on problems that are, up to current knowledge, known to be quantum secure. The code implementation uses OQS python binding for the open quantum-safe project.

   The key encapsulation method used is the kyber512. The kyber512 KEM is a quantum-resistant key encapsulation method based on the learning with errors problem and is the application of lattice cryptography. The key sizes are:

   - Public Key: 800 bytes
   - Secret Key: 1,632 bytes
   - Ciphertext: 768 bytes

   The overview of the Key encapsulation process includes key generation, encapsulation, and, decapsulation algorithms. The above algorithms can be thought of as running in two sides, client and server side of the application. The client side of the application generates a public and private key. The client secret is stored in one file, while the public key generated is made accessible to the server. The server computes a shared key and cipher text of the shared key and shares the cipher text with the client, which computes the shared secret key using the private key.

   Learning with error problems as discussed in class comes with the possibility of error in the computation of the shared secret. However, this probability is low and the system can also check the matching of the generated secret keys.

   Code:

   All the codes provided assume you are in the src path of assignment5.

   The code part of this question contains three separate programs. All the programs are named as given in the question.

   The first program kem_gen_keys takes file arguments to store the public key and secret key. This program is runs on the client side and stores the generated public key and secret into a file. The keys are formatted in base64 and written to the files given.

   For example, to generate keys run:

   ```
   python3 key_encapsulation/kem_gen_keys.py client/pub_key1.b64 client/client_sec1.b64
   ```

   All the key encapsulation codes are given in the src/key_encapsulation folder and all files related to the client are stored in the client folders, while the server-related data is stored in the server folder.

The second program runs on the server and generates a cipher text given a public key file and outputs the generated secret key for the server's use. The cipher text program, using the perk public key as the public key file can be run as:

```
python3 key_encapsulation/kem_gen_ciphertext.py client/key_pub_perk.b64
server/cipher_output1.b64 server/shared_secret1.b64.
```

The last program kem_get_shared_sec takes the client secret key and cipher text to generate a share key output for the client. The client object is re-instantiated using the client secret key provided and key file sizes are all checked for validity as in the above program. The shared secret can be generated on the client side using the command shown below as an example:

```
python3 kem_get_shared_sec.py client/client_sec1.b64 server/cipher_output1.b64
client/shared_sec.b64
```

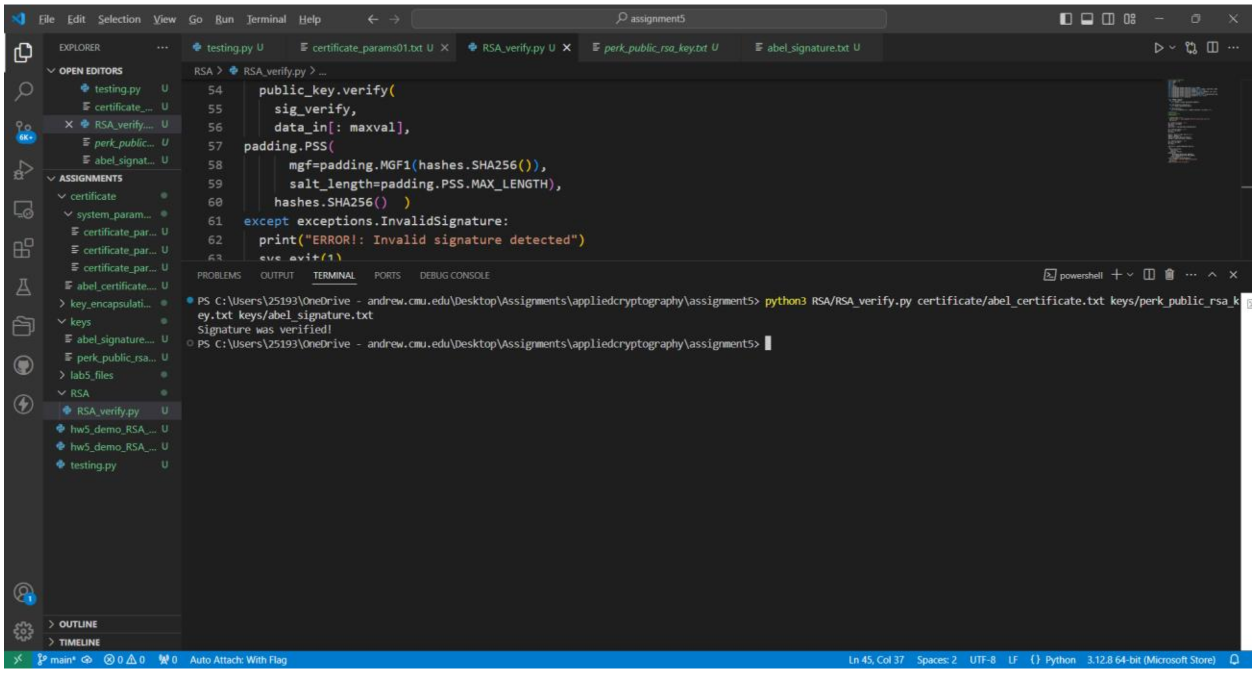
All files in the folder server are included in answering question 4a.

The public key can be found in the client/public_key1.b64.

The image decoded in question 1 part 5 a:

2. **PKI:**
The public key infrastructure comprises the necessary tools for managing digital keys and certificates. Public key infrastructures form the basis of secure communication over the internet. This problem examines the use of system certificates, user certificates, digital signature, El-Gamal asymmetric encryption/decryption.

Firstly, trusted authorities are needed to securely access public key information over the internet. It verifies that the public key is authentic and can only be decoded using the private of the correct recipient. Trusted authorities public key is shared securely (could be compiled in browsers) and certificates are signed using the private key of the trusted authorities. The signed part is the part that includes information for which the certificate is issued, including El-Gamal public key in this case. User id information, issuer, and date of expiry are also included as part of the certificate data. The associated data are hashed and encrypted using the private key of the trusted authority as part of the use of digital signatures. However, in addition to certificates the receiver must be authenticated for having the right private key associated with the public key to form a complete secure session.
Question 2a: This part verifies by taking the body of the certificate and decrypting using the public RSA key of the certified authority (perks public RSA key).
The signature verification output result is shown in the image below:



```
    public_key.verify(
      sig_verify,
      data_in[: maxval],
  padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH),
      hashes.SHA256()  )
  except exceptions.InvalidSignature:
    print("ERROR!: Invalid signature detected")
```

```
PS C:\Users\25193\OneDrive - andrew.cmu.edu\Desktop\Assignments\appliedcryptography\assignment5> python3 RSA/RSA_verify.py certificate/abel_certificate.txt keys/perk_public_rsa_key.txt keys/abel_signature.txt
Signature was verified!
PS C:\Users\25193\OneDrive - andrew.cmu.edu\Desktop\Assignments\appliedcryptography\assignment5>
```

The RSA verify function with SHA256 is used to verify the certificate. The command for running the above code is:

python3 certificate_verification/RSA_verify.py certificate/abel_certificate.txt keys/perk_public_rsa_key.txt keys/abel_signature.txt

Certificates are stored in the certificate folder. Signature files and keys are stored in the keys folder as shown above.

2b. The verification step follows the above and considers all the files in the folder certificate/system_parameters folder. The verified system parameter certificate is system_parameters/certificate_params03.txt. System parameter certificates store the properties needed for the El-Gamal encryption to work, the large prime number p and primitive number alpha.

2c. The encrypt_gamal function works by taking each byte of the key and encoding it to a base64 file. After that, a b value which is a random number between 1 and p-1 selected to serve as a nonce and private exponent of the sender. The sender computes the shared secret using the beta value it got from the certified authorities. The shared secret is then multiplied with the current byte mod p to generate the y2 value. Y1 value is set to the computed public key associated with the b selection. Each of the y1 and y2 values are 4 bytes and are generated for each of the bytes of the key. The code for the encrypt logic can be run by generating a random AES key using the command:

```
python3 elgamal/encrypt_gamal.py
```

2d. The decryption function works by reading 8 bytes of the decrypted file and computing the secret by y1**a mod p. The inverse of that secret is calculated and multiplied by y2 mod p to get the plain text byte. The above code can be tested using the command:

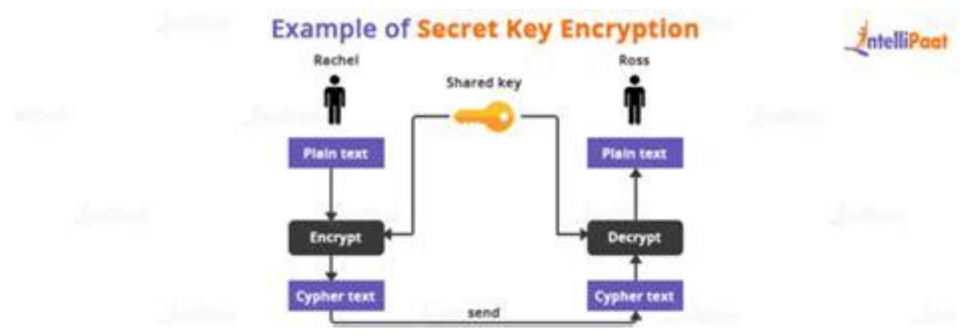
```
python3 elgamal/decrypt_gamal.py
```

To discuss some high-level details, the El Gamal public key cryptography scheme is based on applying the Diffie Hellman key exchange multiple times and depends on the hardness of solving the discrete logarithm problem. One thing to notice for the el-gamal program is that the y1 and y2 values are taken to be 4 bytes. These values represent a smaller range than the normal cryptographic numbers. Even doing that the size of the encrypted files has grown 8 times. This indicates how asymmetric cryptographic systems are costly in terms of space and computation. This is the reason behind the use of asymmetric cryptographic tools at the initial step of the communication while continuous encryption/decryption is done using more performant symmetric encryption algorithms.

3. **Hacking:**

a) To find someone's secret exponent the way to go would be to try and solve the discrete logarithm problem. By taking the public key from the certificates testing for values for a between 1 and p-1, such that alpha**a mod p = the public key. This problem cannot be solved in a time-feasible manner in traditional computers. This result is further shown below.

b) The output of the Landon's image is shown below.



The above process was a time-intensive task even for a smaller prime number selected. In real scenarios, with large cryptographic numbers iterating through the whole selection of the prime numbers would not be feasible in terms of time in traditional computers, even using supercomputers. However, quantum computers can solve the integer factorization and discrete logarithm problem efficiently. The code for the above test can be found in elgamal/decrypt_landon.py file.