Homework 6

Task 1 - RSA Modulus Generation (10 points)

(a)

Given that $N = PQ$, $P = R - i$ and $Q = R + j$, where R is a k-bit integer and (since primes are dense) i, j are relatively small integers, we have two scenarios:
1. If R is not a prime, then P and Q are neighboring primes in the set of k-bit numbers.
2. If R is a prime, then P and Q are immediate prime neighbors of R.

In both scenarios, since P and Q are close to each other, the value of N can be efficiently factorized. One approach can be to search for primes near the square root of N, which would be approximately $\sqrt{N} \approx R$. This search is feasible in polynomial time relative to k. Once an attacker can factor N and find P and Q, they can efficiently compute the private key by using the public key $PK(N, e)$ and looking for a decryption exponent d in $\mathbb{Z}^*_{(P-1)(Q-1)}$, such that $ed = 1$ (mod $(P-1)(Q-1)$).

The proximity of P and Q ($|P - Q|$ is small) significantly reduces the complexity of the factorization problem. Since RSA security relies on the difficulty of factoring N, the method mentioned above of generating P and Q compromises the security of RSA modulus.

(b)

$$P = 3512301459123013912301193312031222319871623812391823111938 2061$$
$$Q = 35123014591230139123011933120312223198716238123918231119382447$$

```python
from sympy import isprime
from timeit import default_timer as timer
import math

# The RSA modulus N
N = 12336261539757652568320691057196254494530050076556470009232333
    67120767290238588667397052161653352801437540471197470570083267

def factor(N):
    # Approximate square root of N
    approx_sqrt_N = int(math.isqrt(N))

    # Search for prime factors near the square root of N
    for i in range(approx_sqrt_N, 1, -1):
        print("Trying i = ", i)
        if N % i == 0 and isprime(i):
            # double check
            if isprime(N // i) and i * (N // i) == N:
                return i, N // i

    print("Error: no factors found")
    return None, None

# Find P and Q
timer_start = timer()
P, Q = factor(N)
timer_end = timer()
print("P = ", P, "Q = ", Q, "Time = ", timer_end - timer_start)
```

Task 2 - ElGamal and DDH (15 points)

(a)

Decryption Algorithm:

procedure Dec(SK, $C = (C_1, C_2)$) :
if $C_1^{\mathsf{SK}} = C_2$ **then**
return 0
else
return 1

Correctness when $b = 0$:
- During encryption, we have $C_1 = g^y$ and $C_2 = PK^y = (g^x)^y = g^{xy}$.
- During decryption, we have $C_1^{\mathsf{SK}} = (g^y)^x = g^{yx} = g^{xy} = C_2$.
- Thus, when $b = 0$, the decryption algorithm will always correctly output 0.

Correctness when $b = 1$:
- During encryption, we have $C_1 = g^y$ and $C_2 = g^z$, where $y \xleftarrow{\$} \mathbb{Z}_p$ and $z \xleftarrow{\$} \mathbb{Z}_p$.
- During decryption, we have $C_1^{\mathsf{SK}} = (g^y)^x = g^{yx}$. Since y and z are independently and uniformly chosen from $\mathbb{Z}_p$and group $\mathbb{Z}_p$ has prime order p, the probability that $g^{yx} = g^z$ (i.e., $C_1^{\mathsf{SK}} = C_2$) is

$$\Pr[yx \equiv z \pmod{p}] = \frac{1}{p}$$

- Thus, the probability of the decryption algorithm incorrectly return 0 is $\frac{1}{p}$, which is very small when the prime order p is sufficiently large.

Therefore, we have shown that the correctness requirement of the scheme may sometimes not hold, but only with very small probability.

(b)

<u>**oracle** $\mathsf{O}^{\mathsf{DDH}_b[\mathbb{G},g]}$:</u>

<u>private **procedure** Init(X, Y, Z):</u> // $(X, Y, Z) \leftarrow \mathsf{DDH}_b[\mathbb{G}, g]$
(PK, Y, Z) $\leftarrow (X, Y, Z)$
queried $\leftarrow 0$
return PK

<u>public **procedure** Encrypt(M^0, M^1):</u> // $M^0, M^1 \in \mathcal{M} = \{0, 1\}$
if $|M^0| \neq |M^1|$ **or** queried $= 1$ **return** $\perp$
if $M^b = b$ **then**
    $C \leftarrow (Y, Z)$ // When $M^0 = 0$, $C = (g^y, g^{xy})$. When $M^1 = 1$, $C = (g^y, g^z)$
else if $M^b = 0$
    $y' \xleftarrow{\$} \mathbb{Z}_p$
    $C \leftarrow (g^{y'}, \mathsf{PK}^{y'})$ // Here $M^1 = 0$, $Z = g^z$, so we need a new y' s.t.$C = (g^{y'}, g^{xy'})$
else
    $z' \xleftarrow{\$} \mathbb{Z}_p$
    $C \leftarrow (Y, g^{z'})$ // Here $M^0 = 1$, $Z = g^{xy}$, so we need a new z' s.t.$C = (g^y, g^{z'})$
queried $\leftarrow 1$
return C

(c)

Using the oracle O from part (b), for any distinguisher D, setting $D' = D^{\mathsf{O}}$ we have:

$$\mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G},g}(D) = \mathsf{Adv}^{\mathsf{1\text{-}ind\text{-}cpa}}_{\Pi}(D')$$

Further, if D is polynomial time, because O's procedures all run in polynomial time, we also have that D' is polynomial time.
Then, because we assume DDH assumption holds for $\mathbb{G}$ with respect to g, $\mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G},g}(D)$ is negligible, which means that $\mathsf{Adv}^{\mathsf{1\text{-}ind\text{-}cpa}}_{\Pi}(D')$ is also negligible (since the input of D' first passed through the Init procedure of O, which takes the output oracle $\mathsf{DDH}_b[\mathbb{G}, g]$ as input).
Therefore, we have shown that if DDH assumption holds for $\mathbb{G}$ with respect to g, then Π is one-time IND-CPA secure.

Task 3 - Chosen-Ciphertext Security (10 points)

(a)

(b)

(c)

Task 4 - AES-Based Signatures (15 points)

(a)

(b)

(c)