Week 10 Classwork due on Friday Nov 16, 22:00 Hour

Group 5

Wong Ann Yi (1004000)
Liu Bowen (1004028)
Tan Chin Leong Leonard (1004041)

Exercise 1

Implement the Diffie-Hellman protocol.

Answers:

The Diffie-Hellman protocol is used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

The implementation of the protocol uses the multiplicative group of integers modulo p, where p is prime, and g is a primitive root modulo p. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to p–1. The below is a step by step explanation of the key exchange process between Alice and Bob.

ALICE	BOB
Public Keys available = P, G	Public Keys available = P, G
Private Key Selected = a	Private Key Selected = b
Key generated x = G^a Mod P	Key generated y = G^b Mod P
Exchange of generated keys takes place	
Key received = y	key received = x
Generated Secret Key k_a = y^a Mod P	Generated Secret Key k_b = x^b Mod P
Algebraically it can be shown that $k_a = k_b$ – therefore both Alice and Bob have a symmetric key to encrypt.	

The implementation in the form of codes are written as follows:

```
p = 23    # P
g = 5     # G
alice = 6     # a
bob = 15      # b

print( "p: " , p )
print( "g: " , g )

# Alice Sends Bob A = g^a mod p
A = pow(g, alice) % p
print( "Alice sending message: " , A )

# Bob Sends Alice B = g^b mod p
B = pow(g, bob) % p
print( "Bob sending message: ", B )

print( "==========================" )
# Alice Computes Shared Secret: s = B^a mod p
aliceSecret = (B ** alice) % p
print( "Alice Shared info: ", aliceSecret )

# Bob Computes Shared Secret: s = A^b mod p
bobSecret = (A**bob) % p
print( "Bob Shared info: ", bobSecret )
```

++

The output is:

```
('p: ', 23)
('g: ', 5)
('Alice sending message: ', 8)
('Bob sending message: ', 19)
==========================
('Alice Shared info: ', 2)
('Bob Shared info: ', 2)
```

Exercise 2

Implement the RSA encryption scheme from scratch. Use the following interface:

- Gen(minPrime) generates a public/private keypair (512 bits) where p, q > minPrime.
- Enc(pubKey, msg) returns ctxt (integer).
- Dec(privKey, ctxt) returns msg (integer).

Answers:

We will implement the RSA encryption by using the following steps:

Gen(minPrime)

- Select (large) random prime numbers p, q ($p \neq q$, but with almost equal size)
- Compute modulus n = pq
- Compute Φ = (p-1)(q-1)
- Select public exponente,1 < e < Φ,such that gcd(e, Φ) = 1
- Compute private exponent d = e-1mod Φ
- Return public key (n, e), and private key (p, q, Φ,d)

Enc(pubKey=e, msg=m)

- Return m^e mod n = c

Dec(privKey=d, ctxt=c)

- Return c^d mod n = m

The codes are written as follows (in Python 2.7):

```
import random

def gcd(a, b):
    while a != 0:
        a, b = b % a, a
    return b

#calculate d
def findModInverse(a, m):
    # Returns the modular inverse of a % m, which is
    # the number x such that a*x % m = 1
    if gcd(a, m) != 1:
        return None
```

```
    # Calculate using the Extended Euclidean Algorithm:
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return u1 % m

def generatePrime(keysize):
    while True:
        num = random.randrange(2**(keysize-1), 2**(keysize))
        if isPrime(num):
            return num

def isPrime(n):
        if n <=1: return False
        i = 2
        while i*i <= n:
                if n%i == 0 : return False
                i += 1
        return True

def Gen(minPrime, keySize):

    # Step 1: Create two prime numbers, p and q. Calculate n = p * q.
    p = generatePrime(keySize)
    if p < minPrime:
        return;
    q = generatePrime(keySize)
    if q < minPrime:
        return;
    n = p * q
    print 'p is:', p
    print 'q is:', q

    # Step 2: Create a number e that is relatively prime to (p-1)*(q-1).
    while True:
        # Keep trying random numbers for e until one is valid.
```

```
        e = random.randrange(2 ** (keySize - 1), 2 ** (keySize))
        if gcd(e, (p - 1) * (q - 1)) == 1:
            break
    print 'e is:', e

    # Step 3: Calculate d, the mod inverse of e.
    d = findModInverse(e, (p - 1) * (q - 1))
    print 'd is:', d

    publicKey = (n, e)
    privateKey = (n, d)
    print('Public key:', publicKey)
    print('Private key:', privateKey)
    return (publicKey, privateKey)

def Enc(pubKey, msg):
    return pow(msg, pubKey[1]) % pubKey[0]

def Dec(privKey, ctxt):
    return pow(ctxt, privKey[1]) % privKey[0]

def main():
    minPrime = 2
    publicKey, privateKey = Gen(minPrime, 10) # set key to 10 bits
    msg = 10000; # set message to be 10000
    ciphertext =  Enc(publicKey, msg)
    print 'ciphertext is:', ciphertext
    print 'plaintext is:', Dec(privateKey, ciphertext)

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

The output is:

```
p is: 919
q is: 601
e is: 623
d is: 251087
```

```
('Public key:', (552319, 623))
('Private key:', (552319, 251087))
ciphertext is: 164526
plaintext is: 10000
```

Exercise 3

Implement the RSA signature scheme from scratch. Use the following interface:

- Gen(minPrime) generates a public/private keypair (512 bits) where p, q > minPrime.
- Sign(privKey, msg) returns a signature (integer).
- Verify(pubKey, msg, signature) returns boolean.

Both Sign() and Verify() take msg as integer and use SHA-512.