Week 10 Homework due on November 23, 18:59 Hour

Group 5

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

Exercise 1

Prove lcm(a, b) = ab/gcd(a; b), where a and b are integers, lcm = the Least Common Multiple, gcd = the Greatest Common Divisor.

Answers:

Proof: First a

Lemma: If $m > 0$, lcm (ma, mb) = $m \times$ lcm (a, b).

Since lcm(ma, mb) is a multiple of ma, which is a multiple of m, we have m | lcm (ma, mb).

Let mh_1 = lcm(ma, mb), and set h_2 = lcm(a, b).

Then $ma \mid mh_1 \Rightarrow a \mid h_1$ and $mb \mid mh_1 \Rightarrow b \mid h_1$.

That says h_1 is a common multiple of a and b; but h_2 is the least common multiple, so

$$h_1 \geq h_2 . \quad (1)$$

Next, $a \mid h_2 \Rightarrow am \mid mh_2$ and $b \mid h_2 \Rightarrow bm \mid mh_2$.

Since mh_2 is a common multiple of ma and mb, and mh_1 = lcm(ma, mb), we have $mh_2 \geq mh_1$,i.e.

$$h_2 \geq h_1 . \quad (2)$$

From (1) and (2), $h_1 = h_2$.

Therefore, lcm(ma, mb) = $mh_1 = mh_2 = m \times$ lcm(a, b); proving the Lemma.

Conclusion of Proof of Theorem:

Let g = gcd(a, b). Since g | a, g | b, let a = gc and b = gd.

From a result in the text, gcd(c, d) = gcd(a/g, b/g) = 1.

Now we will prove that lcm(c, d) = cd. (3)

Since c | lcm(c, d), let lcm(c, d) = kc.

Since d | kc and gcd(c, d) = 1, d | k and so $dc \leq kc$.

However, kc is the least common multiple and dc is a common multiple, so $kc \leq dc$.

Hence kc = dc, i.e. lcm(c, d) = cd.

Finally, using the Lemma and (3), we have:

lcm(a, b) x gcd(a, b) = lcm(gc, gd) x g = g x lcm(c, d) × g = gcdg = (gc)(gd) = ab.

Exercise 2

Compute the result of 12358 * 1854 * 14303 (mod 29101) in two ways and verify the equivalence: by reducing modulo 29101 after each multiplication and by computing the entire product first and then reducing modulo 29101.

Answers:

The codes are written in Python 3 as per below:

```
a= (12358*1854) % 29101
print("The first product and mod result is", a)

b= (a*14303) % 29101
print("The second product and mod result is", b)

print("The final result after each multiplication and mod is", b)


c = 12358*1854*14303
c = c % 29101
print("The result after computing the entire product and then mod is", c)
```

The resulting output is:

```
The first product and mod result is 9245
The second product and mod result is 25392
The final result after each multiplication and mod is 25392
The result after computing the entire product and then mod is 25392
```

Therefore, given integers x, y & z, the result of [x*y (mod p)]*z (mod p) is equal and similar to the full product and mod = x*y*z (mod p).

Exercise 3

What are the subgroups generated by 3, 7, and 10 in the multiplicative group of integers modulo p = 11?
Answers:

3^0 mod 11=1 3^1 mod 11=3 3^2 mod 11=9 3^3 mod 11=5 3^4 mod 11=4 3^5 mod 11=1 3^6 mod 11=3 3^7 mod 11=9 3^8 mod 11=5 3^9 mod 11=4 Sub groups = 1,3,9,5,4	7^0 mod 11=1 7^1 mod 11=7 7^2 mod 11=5 7^3 mod 11=2 7^4 mod 11=3 7^5 mod 11=10 7^6 mod 11=4 7^7 mod 11=6 7^8 mod 11=9 7^9 mod 11=8 7^{10} mod 11= 1 7^{11} mod 11 =7 7^{12} mod 11 = 5 7^{13} mod 11 = 2 7^{14} mod 11 = 3 7^{15} mod 11 = 10 7^{16} mod 11 = 4 7^{17} mod 11 = 6 7^{18} mod 11 = 9 Sub groups=1,7,5, 2,3,10,4,6,9,8	10^0 mod 11=1 10^1 mod 11=10 10^2 mod 11=1 10^3 mod 11=10 10^4 mod 11=1 10^5 mod 11=10 10^6 mod 11=1 10^7 mod 11=10 10^8 mod 11=1 10^9 mod 11=10 Sub groups = 1,10

Exercise 4

Let p = 71; q = 89; n = pq; e = 3. First find the corresponding private RSA key d. Then compute the signature on $m_1 = 5416$, $m_2 = 2397$, and $m_3 = m_1 m_2 \pmod n$ using the basic RSA operation. Show that the third signature is equivalent to the product of the first two signatures.

Answers:

p = 71, q = 89, n = 6319, e = 3, $\Phi(n) = 70*88 = 6160$

We then compute $d = e^{-1} \bmod(\Phi(n)) \Rightarrow e*d = 1 \pmod{\Phi(n)}$

From the textbook chapter 10.3.5 The Extended Euclidean Algorithm or week 10 lecture notes page 11,

$e*d = 1 + k\,\Phi(n) \Rightarrow 3 * d = 1 + k * 6160 \rightarrow (1)$

Using Euclidean algorithm, we calculate by:

$6160 = 3(2053) + 1$
$3(2053) = -1 + 6160$
$3(-2053) = 1 + (-1) * (6160)$ compared it to equation (1)

We obtain k = -1 and d = -2053 => which is in fact 4107 mod (6160) since -2053 + 6160 = 4107
Hence, d = 4107

We also wrote a Python program as follows and obtained the d of value of 4107.

```
from fractions import gcd
from Crypto.Hash import SHA256

def calculateD(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

p = 71
q = 89
n = p * q
```

```
m  (p-1) * (q-1)
e = 3
d = calculateD(e, m)
print d
```

The public key is therefore (n, e) or (6319, 3) and the private key is (p, q, Φ, d) or (71, 89, 6160, 4107).

According to the text book Chapter 12.4.1 Digital Signatures with RSA - to sign a message m, the owner of the private key computes s := m^(1/e) mod n. The pair (m, s) is now a signed message. To verify the signature, anyone who knows the public key can verify that s^e = m (mod n). For convenience, we often write c^(1/e) mod n instead of c^d mod n. The exponents of a modulo n computation are all taken modulo t, because x^t = 1 (mod n), so multiples of t in the exponent do not affect the result. And we computed d as the inverse of e modulo t, so writing d as 1/e is natural.

For this exercise, we first state the messages of m1 m2 and m3. We then use s := m^(1/e) mod n to generate the signature(s) (but we use d to replace 1/e in the inverse modulo formulation). Finally, we obtain the signature of m3 by (sig1*sig2) mod n = sig3.

Finally, we reverse the process to verify the original messages of m1, m2 and m3. We complete this by using s^e = m (mod n). From the results, we successfully verify that the signature(s) produce the original messages of m1, m2 and m3.

As a result, we have produced the signatures for m1, m2 and m3. We have also shown that the product of signatures for m1 and m2 is equal to the signature for m3.

The codes in Python 2.7 are shown below.

```
p = 71
q = 89
n = p * q
m = (p-1) * (q-1)
e = 3
d = 4107

m1 = 5416
m2 = 2397
m3 = (m1*m2) % n
print 'm1:', m1
print 'm2:', m2
print 'm3:', m3

sig1 = (m1**d) % n
sig2 = (m2**d) % n
sig3 = (m3**d) % n
```

```
print 'signature for m1:', sig1
print 'signature for m2:', sig2
print 'signature for m3', sig3
print 'The product of signature 1 & 2 is equal to signature 3?', (sig1 * sig2) % n ==
sig3

verify1 = (sig1**e) % n
verify2 = (sig2**e) % n
verify3 = (sig3**e) % n
print 'm1 obtained from signature 1:', verify1
print 'm2 obtained from signature 2:', verify2
print 'm3 obtained from signature 3:', verify3
```

The outputs are:

```
m1: 5416
m2: 2397
m3: 2926
signature for m1: 1876
signature for m2: 2206
signature for m3 5830
The product of signature 1 & 2 is equal to signature 3? True
m1 obtained from signature 1: 5416
m2 obtained from signature 2: 2397
m3 obtained from signature 3: 2926
```

Exercise 5

Try to conduct timing attacks against your implementation of the RSA encryption: measure time that is needed to encrypt messages with different sizes and contents. What can an adversary deduct about a message given only the execution time of encrypting it? Repeat the measurement for different key sizes.

Answers:

We will implement RSA encryption on a number of messages to measure the difference in timing in order to launch a timing attack. We vary the messages' sizes by creating messages of size in 4 bits, 8 bits, 12 bits and 16 bits. We vary the contents in each message size by creating two types of contents: all "1" binary values and half "1" and half "0" binary values. For example, for a 8-bit message, the contents are: 0xff (or 11111111) or 0xf0 (or 11110000). For the key size, we use two different key sizes of 10-bit and 20-bit to conduct the experiments. The results are presented in the below four figures. The Python codes are enclosed at the end of the answers.

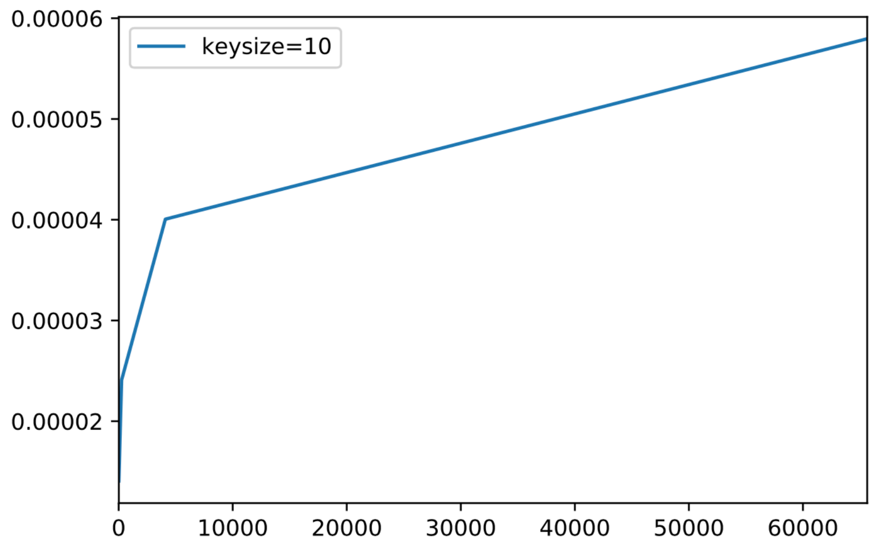

Figure 1 – key size 10, all "1"s binary

For a key size=10, all "1" binary value messages of sizes 4 bit, 8 bit, 12 bit and 16 bit, the time taken to encrypt the messages is ranging from 0.00004 seconds to almost 0.00006 seconds. This is shown in Figure 1.

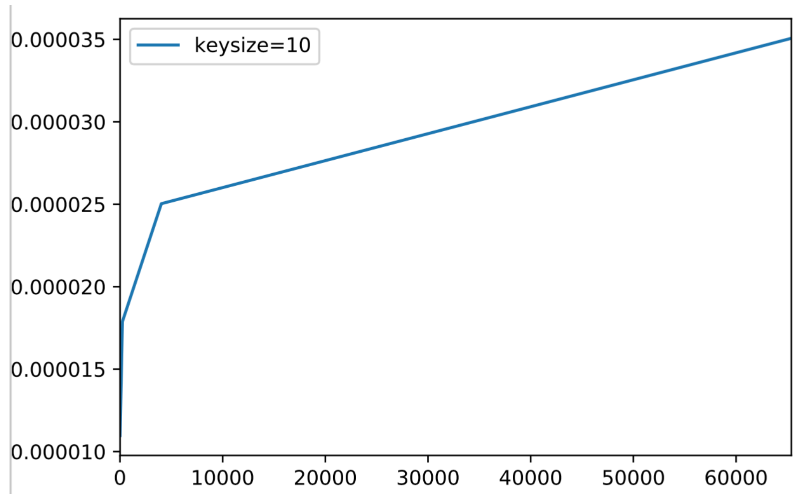

Figure 2 – key size 10, half "1"s and half"0"s binary

For a key size=10, half "1" and half "0" binary value messages of sizes 4 bit, 8 bit, 12 bit and 16 bit, the time taken to encrypt the messages is ranging from 0.000025 seconds to almost 0.000035 seconds. This is shown in Figure 2.

When we compare Figure 1 and 2, we observed that the encryption time for a full "1" binary value message is longer than the similar size message of half "1" and half "0" binary value message (when using the same key size).

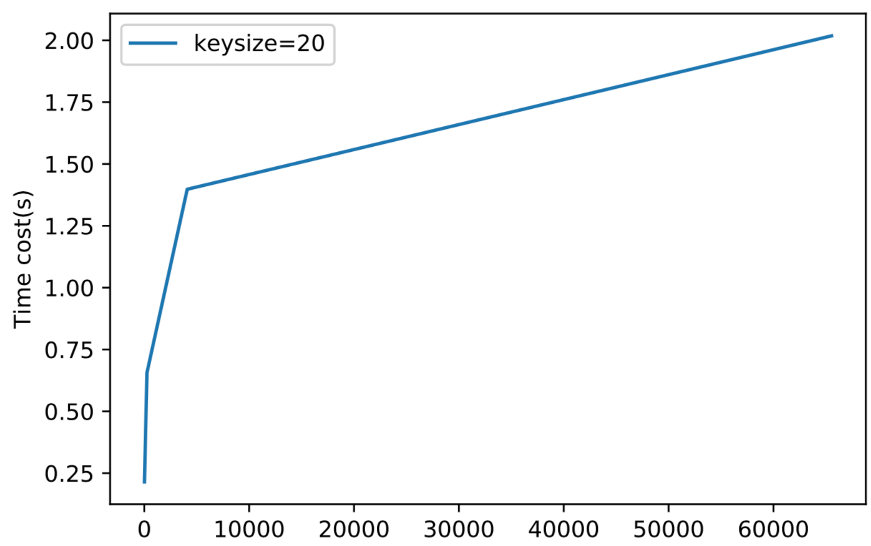

Figure 3 - key size 20, all "1"s binary

For a key size=20, all "1" binary value messages of sizes 4 bit, 8 bit, 12 bit and 16 bit, the time taken to encrypt the messages is ranging from 1.4 seconds to almost 2 seconds. This is shown in Figure 3.

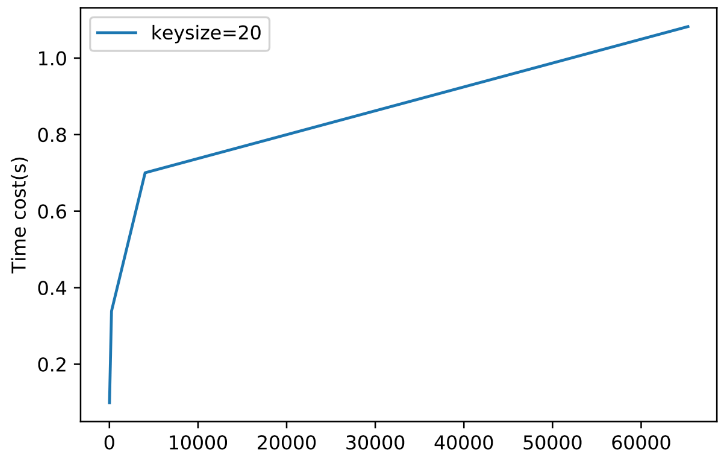

Figure 4 - key size 20, half "1"s and half"0"s binary

For a key size=20, half "1" and half "0" binary value messages of sizes 4 bit, 8 bit, 12 bit and 16 bit, the time taken to encrypt the messages is ranging from 0.7 seconds to almost 1.1 seconds. This is shown in Figure 4. Again, the comparison of Figure 3 and 4 shows a similar result as the comparison of Figure 1 and 2.

Again, when we compare Figure 3 and 4, we observed that the encryption time for a full "1" binary value message is longer than the similar size message of half "1" and half "0" binary value message.

From the above graphs and observations, we conclude that key size and message size will have an impact to the encryption time of the message. The contents of the message also affect the encryption time. A message comprising of all "1"s takes a longer time to encrypt than a message of half "1"s and half "0"s. We suspect that the modulo operation performed on 1 takes a longer time than the modulo operation performed on 0. We did not investigate further into the cause.

The second observation is that the larger key size will also increase the encryption time significantly. For key size 10, the similar all "1"s binary message encryption time ranges from 0.00004 seconds to 0.00006 seconds while for the key size 20, the time taken is from 1.4 seconds to almost 2 seconds.

From all the graphs, we observe that the encryption time increases linearly as the size of the message increases.

Therefore, in basic RSA encryption process, each message undergoes Modulo computation and the size of message, the contents within the message and the key size will affect the performance of the Modulo operation. Thus, an adversary can try to deduce the private key by measuring the encryption time.

Timing attacks exploit the timing variations in cryptographic operations. Because of performance optimizations, computations performed by a cryptographic algorithm often take different amounts of time depending on the input and the value of the secret parameter. If RSA private key operations can be timed reasonably accurately, in some cases statistical analysis can be applied to recover the secret key involved in the computations.

Below are the Python codes used to generate the above graphs.

```
import random
import time
import matplotlib.pyplot as plt

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

def findModInverse(a, m):
    if gcd(a, m) != 1:
        return None
    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(keySize):

    p = generatePrime(keySize)
    q = generatePrime(keySize)
```

```
    n = p * q
    print 'p is:', p
    print 'q is:', q
    print p * q
    while True:
        e = random.randrange(2 ** (keySize - 1), 2 ** (keySize))
        if gcd(e, (p - 1) * (q - 1)) == 1:
            break
    print 'e is:', 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 draw1():
    publicKey, privateKey = Gen(20)
    msg_size = [4, 8, 12, 16]
    msg_all_1 = [int('f', 16), int('ff', 16), int('fff', 16), int('ffff', 16)]
    time_cost_all = []
    for i in range(len(msg_all_1)):
        start = time.time()
        ciphertext =  Enc(publicKey, msg_all_1[i])
        end = time.time()
        time_cost_all.append(end - start)
    plt.xlabel("Message size(bit)")
    plt.ylabel("Time cost(s)")
    plt.plot(msg_all_1, time_cost_all, label="keysize=20")
    plt.legend()
    plt.savefig('20-all.pdf')

def draw2():
    publicKey, privateKey = Gen(20)
```

```
    msg_half_1 = [int('c', 16), int('f0', 16), int('fc0', 16), int('ff00', 16)]
    time_cost_half = []
    for i in range(len(msg_half_1)):
        start = time.time()
        ciphertext =  Enc(publicKey, msg_half_1[i])
        end = time.time()
        time_cost_half.append(end - start)
    plt.xlabel("Message size(bit)")
    plt.ylabel("Time cost(s)")
    plt.plot(msg_half_1, time_cost_half, label="keysize=20")
    plt.legend()
    plt.savefig('20-half.pdf')

def draw3():
    publicKey, privateKey = Gen(10)
    msg_size = [4, 8, 12, 16]
    msg_all_1 = [int('f', 16), int('ff', 16), int('fff', 16), int('ffff', 16)]
    time_cost_all = []
    for i in range(len(msg_all_1)):
        start = time.time()
        ciphertext =  Enc(publicKey, msg_all_1[i])
        end = time.time()
        time_cost_all.append(end - start)
    plt.xlabel("Message size(bit)")
    plt.ylabel("Time cost(s)")
    plt.xlim(int('f', 16),int('ffff', 16)+100)
    plt.plot(msg_all_1, time_cost_all, label="keysize=10")
    plt.legend()
    plt.savefig('10-all.pdf')

def draw4():
    publicKey, privateKey = Gen(10)
    msg_half_1 = [int('c', 16), int('f0', 16), int('fc0', 16), int('ff00', 16)]
    time_cost_half = []
    for i in range(len(msg_half_1)):
        start = time.time()
        ciphertext =  Enc(publicKey, msg_half_1[i])
        end = time.time()
        time_cost_half.append(end - start)
    plt.xlabel("Message size(bit)")
    plt.ylabel("Time cost(s)")
```

```
    plt.xlim(int('c', 16),int('ff00', 16)+100)
    plt.plot(msg_half_1, time_cost_half, label="keysize=10")
    plt.legend()
    plt.savefig('10-half.pdf')

draw1()
draw2()
draw3()
draw4()
```