Contents

Synopsis:

This is the report for ISEC2000 Fundamental Concepts of Cryptography Assignment 2. It will go through all required aspect mentioned in the assignment pdf. These include:

- Proving gcd(a, b) = gcd(b, a mod b)
- Explaining how RSA digital signatures work, and how they can be forged
- Explaining lessons learnt and difficulties faced while implementing the RSA encryption algorithm
- Explaining how I implemented the source coding part of the RSA implementation

Question 1:

Proof of Linear Combination Common Divisor:

To prove gcd(a, b) = gcd(b, a mod b), I will first explain how linear combination common division works.

Assume all values whole numbers and not equal to 0. D|a read as D evenly divides a.

- If D is a common divisor for a and b, D is also a divisor for any multiples for a and b such that D|ax and D|by.
- This can also be written as a = xD and b = yD.
- If we made a and b into a linear combination, a – b = xD – yD = (x – y)D, or a+b = xD + yD = (x+y)D

As you can see, using linear combinations and getting common divisors is mathematically logical, as D|ax-by and D|ax+by is possible.

Proof gcd(a, b) = gcd(b, a mod b):

Let a = bq + r, where r = a mod b, and q = any integer other than 0.

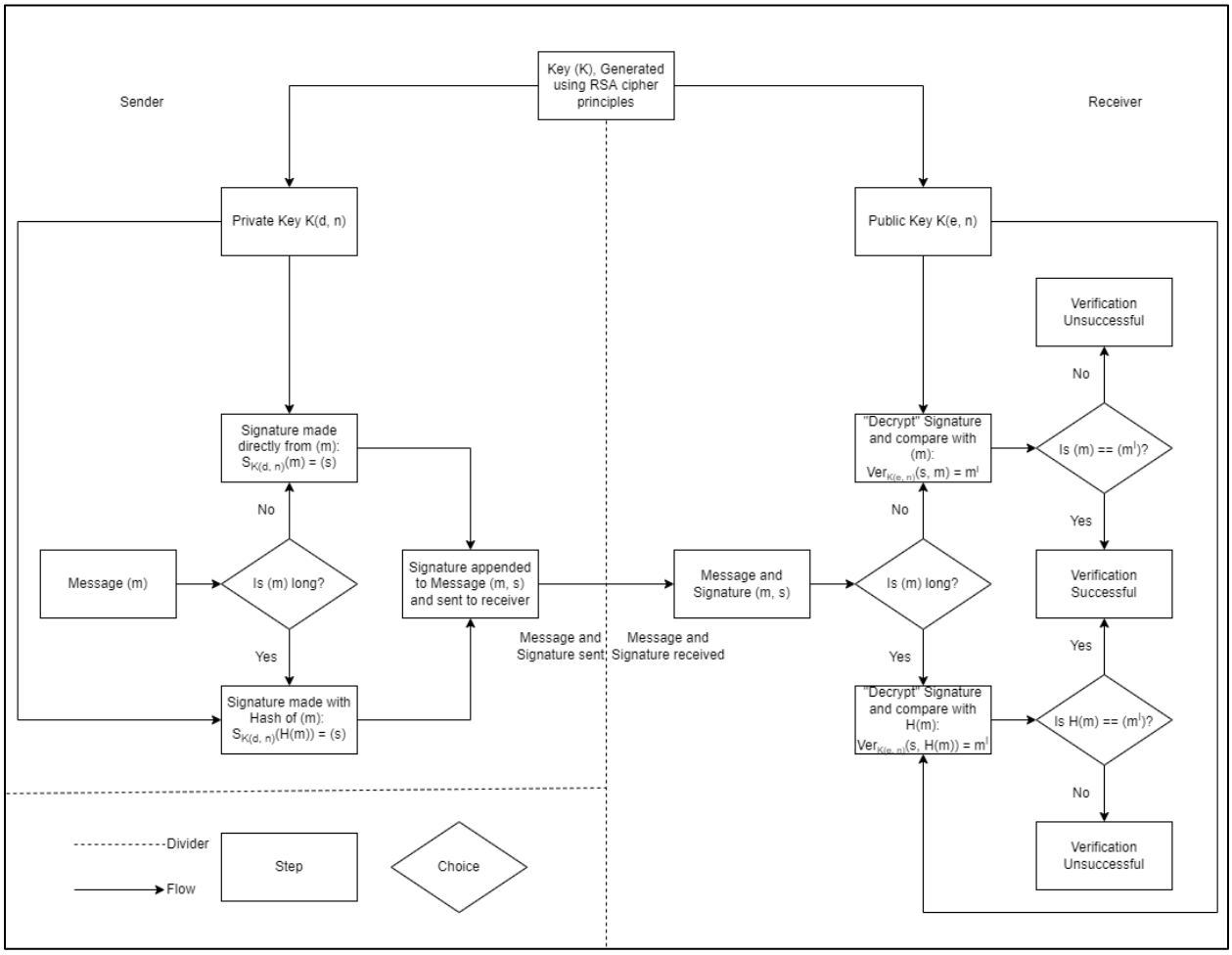
- If d1 = gcd(a, b), d1|a and d1|b. With the proof of the linear combination divisor, we can say that d1|a-bq is possible, where if we're talking in terms of ax-by, we can say that x = 1 and y = q.
- If we notice, d1|a-bq satisfies d1|r, which satisfies d1|a mod b.
- Now if d2 = gcd(b, r), d2|b and d2|r. Using the same logic from the linear combination proof, we can say d2|bq+r is possible, where if we're talking in terms of ax+by, we can say that x = q and y = 1
- Now notice, d2|bq+r satisfies d2|a.
- This means that d1|a, d1|b, d1|r, and d2|a, d2|b, d2|r.
- There are a few conclusions we can make from this.
    - Firstly:
        - Since d1|b and d1|r, it is a common divisor of b and r.
        - This means d1 <= d2, as d2 is the "greatest" common divisor for b and r.
    - Secondly:
        - Since d2|a and d2|b, it is a common divisor of a and b.
        - This means d2 <= d1, as d1 is the "greatest" common divisor for a and b.
    - Since d1 <= d2 and d2 <= d1, we can conclusively say the d1 = d2.
    - If d1 = d2, gcd(a, b) = gcd(b, r)

This proves, mathematically, that gcd(a, b) = gcd(b, a mod b). Using this constraint, the Euclidean algorithm effectively finds the greatest common divisor for two numbers.

Question 2:

Signature Structure:

The RSA digital signature is an asymmetric technique, used for authentication and verification of the sender by a receiver. The following diagram explains how this works:



1. The private and public keys are generated using the same scheme used for generating the keys in the RSA encryption, by:
    a. Getting primes p and q
    b. Getting n = p * q
    c. Getting phiN = (p-1) * (q-1)
    d. Getting encryption exponent (e) such that gcd(e, phiN) = 1
    e. Getting modular inverse of (e) which gives decryption exponent (d)
    f. With that we have our private key (d, n) and public key (e, n)
2. After getting the keys, the RSA will use these for getting the digital signatures and their verification

3. Let's say message = m. If the size of the message is:
    a. Not long:
        - Use the signature function S(), that uses the private key(d, n), to get the signature (s) using the calculation: $s = S_{K(d, n)}(m) = m^d \bmod n$
    b. Long:
        - Run message through a Hash function H(), and use the hashed value of m in the signature function, instead of the actual m. The calculation then looks something like this: $s = S_{K(d, n)}(H(m)) = (H(m))^d \bmod n$
4. After getting the signature s, we append this to m and send them together as (m, s)
5. After receiving, we run the verification function Ver(), that uses the public key(e, n), on the signature. Calculation: $m^l = Ver_{K(e, n)}(s) = s^e \bmod n$
6. We again check for the size. If size of the message m is:
    a. Not Long:
        - Compare m^l directly with m.
            - If m^l == m, the verification is successful, and sender is authenticated.
            - If m^l != m, the verification is unsuccessful, and message is not trusted.
    b. Long:
        - Compare m^l with the hash of m, H(m).
            - If m^l == H(m), the verification is successful, and sender is authenticated.
            - If m^l != H(m), the verification is unsuccessful, and message is not trusted.

This is the summary on how the RSA digital signature scheme functions.

Different Message but Same Hash:

"Alice signed a document m using the RSA signature scheme. The signature is sent to Bob. Accidentally Bob found one message m0 (m != m0) such that H(m) = H(m0), where H() is the hash function used in the signature scheme"

If Bob finds two different messages that have the same result after a hash function, means that he has found a hash collision case. This is a major find as he can then send this message to someone else, with Alices signature, and m0 will completely get verified as something that Alice sent. Here how:

- Alice sends Bob (m, s).
- Bob hashes m with hash function H(m) and runs the signature through the verification function Ver(s).
- Bob compares H(m) and Ver(s), and if H(m) = Ver(s), Bob can confirm that it's Alice who sent the message.

- Bob then finds m0, which is different from m sent by Alice, but has the same hash H(m0) as H(m).
- With this discovery, if Bob wants to send m0 to someone, for example Charlie, he can send Charlie (m0, s) while acting as Alice. (s being the same signature he got from the delivery from Alice)
- When Charlie gets (m0, s), he runs H(m0) and Ver(s) and compares the results. This will be verified as H(m) = H(m0) = Ver(s). With this, Bob has successfully forged a message by Alice, as Charlie has verified that m0 had the digital signature from Alice.

Question 3:

(a) Lessons and Difficulties:

Lessons:

Eventhough the theory was relatively simple for the RSA, after the implementation there were some notable things that I learnt from the experience.

1. **Prime Numbers**: At a low level, prime numbers are pretty easy to generate and calculate. But for bigger numbers, this line between being a prime and composite number becomes increasingly blurry. Theoretically, this was difficult to understand, but once I worked on the problem practically, it became clear as to why this happens, and how even checking if a number is a prime number becomes difficult, and only a probability determines whether the number is prime or not. I used Lehmann's algorithm to check for if the number generated is a prime number or not.

2. **Modular Exponentiation**: This was one of the things that confused me most during the process. The assignment asked for a modular exponentiation function, but I thought it wouldn't be required, as the simple ** python operator was doing the trick for small exponents. It soon became clear, when I started using larger exponents, that it took a great amount of time for the calculations and the ** operator was not effective enough for the job. Hence, implementing the binary modular exponentiation function was very useful.

Difficulties:

Though it was theoretically simple to implement, the practical implementation of the RSA proved to have some unforeseen difficulties.

1. **Extended Euclidean Algorithm**: This was something I invested a lot of my time to. Doing this on paper was a pretty logical and simple process, but I had a very difficult time programming the systematic extended part of the algorithm. It took a hefty amount of time to understand and implement the algorithm in a recursive manner, which allowed the procurement of the GCD and the linear combination quotients simultaneously.

2. **Negative Decryption Exponent**: This was one of the biggest problems I faced when working on the RSA. During the key generation part, after the encryption exponent (e) is found, finding the modular inverse of the e, to be used as the decryption exponent d, is easy using the Extended Euclidean Algorithm. The problem arises when the modular inverse is a negative number. This leads to the decryption not functioning correctly. The solution I found was to add the phi value of the modulus (n) to the modular inverse of e and treat the resulting value as the decryption exponent (d). This works, as the negative values will give the same result if a multiple of the modulus is added or subtracted.

3. **Source Coding**: The strategy for source coding took a large chunk of the implementation time too, as thinking of a strategy that would satisfy the assignment's requirements was not simple. Since we are working with large numbers for the key (the first primes p and q are greater than 2^{64}), simply converting each character to their ascii values was not enough, as it would be a waste to keep it so simple. But the source coding should still satisfy the $m < n$ constraint for the RSA encryption to function as expected. Finding a strategy that was complex enough to make the message size just large enough to stay within the $m < n$ constraint was troubling. Refer to "(b) Source Coding" section for my solution to this problem, basically adopting the strategy used in question 1 of lab07.

(b) Source Coding:

Encoding:

```
# Converts string to a list of 3 digit integers in the form of strings, with leading zeros, if integer is less than 3 digits
def str2intList(string):
    chars = []
    for i in string:
        dec = str(ord(i))
        if len(dec) < 3:
            dec = '0' + str(dec)
        chars.append(dec)
    return chars
```

```
# Converts "string" to list of numbers, to be used in the RSA calculation
def sourceCode(string):
    s = str2intList(string)
    blocks = []
    for i in range(len(s)):
        if i % 2 == 0:
            if (i == (len(s)-1)):
                blocks.append(s[i] + '000')
            else:
                blocks.append(s[i] + s[i+1])
    return blocks
```

For the source coding, I first converted each character to their ascii values. Then I check if the ascii value is 3 digit long, if it's not, then make it 3 digits by adding leading zeros. This is not possible with conventional integers in Python, so I converted them to strings and added the leading zeros. After getting the list of these 3-digit ascii values, I join every two values into one, so I get blocks of 6-digit integers. If the last block is an odd index, pad it with three zeros. These 6-digit blocks are the ones that are used as the base for the encryption function. Because of the lower values used as m, this source coding technique easily satisfies the m < n constraint of RSA. With this, the encoding part of the source coding is complete.

For String "Shuber?", ascii 3-digit values will look like:

Character	3-digit Ascii
S	083
h	104
u	117
b	098
e	101
r	114
?	063

These values are then joined alternately as “Sh”, “ub”, “er”, “?”. Their blocks will look like:

Characters	6-digit block
Sh	083104
ub	117098
er	101114
?	063000 Padded with three zeros as doesn’t have a pair

Each of these will then be used as the m:

$c1 = 083104^e \bmod n$

$c2 = 117098^e \bmod n$

$c3 = 101114^e \bmod n$

$c4 = 063000^e \bmod n$

Then these are converted to: hex(c1, c2, c3, c4) = h1, h2, h3, h4, and cipher = h1 + h2 + h3 + h4

Decoding:

```
# Decryption: cipherText = string of hexadecimal returned by "Encryption()" function, d = decryption exponent, n = modulus
def Decrypt(cipherText, d, n):
    hexList = cipherText.split('g')
    nums = []
    for m in hexList:
        nums.append(modExp(hex2int(m), d, n))
    numList = []
    for i in nums:
        chars = str(i)
        if len(chars) < 6:
            chars = ('0' * (6 - len(chars))) + chars
        char1, char2 = chars[:3], chars[3:]
        numList.extend((char1, char2))
    return intList2str(numList)
```

For decoding, I do the opposite. Once the hex cipher text is converted back to integers, and decrypted, each of the decrypted numbers are checked to see if they are 6 digits or not. If < 6 digits, make them a 6-digit number by adding leading zeros, similar to the principle used for the encoding part of the source coding. We then have 6-digit blocks of numbers, with each three numbers of the six representing one character. The blocks are then separated to blocks of three as their ascii values (with leading zeros if ascii value less than 3 digits) and these are converted back to strings.