CSCI 561: Programming Project & Research Project

STFX Student Name: Rahul Ananda Bijai

STFX Student ID: 202006042

STFX Email ID: x2020fvr@stfx.ca

PROGRAMMING PROJECT: 1. RSA

The following example of RSA cipher text is presented. You task is to decrypt it. The public parameters of the systems are n=31313 and e=4913. In order to translate the plaintext back into ordinary English text, you need to know how alphabetic characters are encoded as elements in Zn. Each element of Zn represents three alphabetic characters as in the following examples: DOG: 3x262 + 14x26 + 6 = 2398 CAT: 2x262 + 0x26 + 19 = 1371 ZZZ: 25x262 + 25x26 + 25 = 17575 6340 8309 14010 8936 27358 25023 16481 25809 23614 7135 24996 30590 27570 26486 30388 9395 27584 14999 4517 12146 29421 26439 1606 17881 25774 7647 23901 7372 25774 18436 12056 13547 7908 8635 2149 1908 22076 7372 8686 1304 4082 11803 5314 107 7359 22470 7372 22827 15698 30317 4685 14696 30388 8671 29956 15705 1417 26905 25809 28347 26277 7897 20240 21519 12437 1108 27106 18743 24144 10685 25234 30155 23005 8267 9917 7994 9694 2149 10042 27705 15930 29748 8635 23645 11738 24591 20240 27212 27486 9741 2149 29329 2149 5501 14015 30155 18154 22319 27705 20321 23254 13624 3249 5443 2149 16975 16087 14600 27705 19386 7325 26277 19554 23614 7553 4734 8091 23973 14015 107 3183 17347 25234 4595 21498 6360 19837 8463 6000 31280 29413 2066 369 23204 8425 7792 25973 4477 30989 You will have to invert this process as the final step in your program.

Code Written on Python:

#Initializing e and n

```
e= 4913
n= 31313
```

#Calculating p and q which would be the factors of n

```
factors= []
def getfactors(n) :
    f= 0
    i = 1
    while i <= n :
        if n % i==0:
          if i!=1:
            if i!=n:
```

```
            factors.insert(f, i),
            f = f + 1
        i = i + 1

getfactors(n)
p= factors[0]
q= factors[1]
print('p=',p)
print('q=',q)
```

#Calculating phi(n) that is required for calculating the private key d

```
phin = (p-1)*(q-1)
print('phin=',phin)
```

#Function for Inverse of a under modulo m --https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/

```
def modInverse(a, m):

    for x in range(1, m):
        if (((a%m) * (x%m)) % m == 1):
            return x
    return -1
```

#Calculating priviate key d as per our formula : e^-1 mod phi(n)

```
d = modInverse(e, phin)
print('d=',d)
priv_key = d
```

#Initializing the Ciphertext as an array from the given question

```
ciphertext = [6340, 8309, 14010, 8936, 27358, 25023, 16481, 25809, 23614, 7135, 24996, 30590, 27570,
26486, 30388, 9395, 27584, 14999, 4517, 12146, 29421, 26439, 1606, 17881, 25774, 7647, 23901, 7372,
```

25774, 18436, 12056, 13547, 7908, 8635, 2149, 1908, 22076, 7372, 8686, 1304, 4082, 11803, 5314, 107, 7359, 22470, 7372, 22827, 15698, 30317, 4685, 14696, 30388, 8671, 29956, 15705, 1417, 26905, 25809, 28347, 26277, 7897, 20240, 21519, 12437, 1108, 27106, 18743, 24144, 10685, 25234, 30155, 23005, 8267, 9917, 7994, 9694, 2149, 10042, 27705, 15930, 29748, 8635, 23645, 11738, 24591, 20240, 27212, 27486, 9741, 2149, 29329, 2149, 5501, 14015, 30155, 18154, 22319, 27705, 20321, 23254, 13624, 3249, 5443, 2149, 16975, 16087, 14600, 27705, 19386, 7325, 26277, 19554, 23614, 7553, 4734, 8091, 23973, 14015, 107, 3183, 17347, 25234, 4595, 21498, 6360, 19837, 8463, 6000, 31280, 29413, 2066, 369, 23204, 8425, 7792, 25973, 4477, 30989]

```
print('ciphertext =', ciphertext)
```

#Funtion To compute modular power --https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/

```
def power(x, y, p) :
    res = 1     # Initialize result

    # Update x if it is more
    # than or equal to p
    x = x % p

    if (x == 0) :
        return 0

    while (y > 0) :

        # If y is odd, multiply
        # x with result
        if ((y & 1) == 1) :
            res = (res * x) % p

        # y must be even now
        y = y >> 1      # y = y/2
```

```
        x = (x * x) % p

    return res
```

#Generating Plaintext from the ciphertext above using our formula: C^d mod n

```
y = d; p = n
plaintext= []
# Getting length of list
length = len(ciphertext)
i = 0
# Iterating using while loop
while i < length:
    x = ciphertext[i]
    result= power(x, y, p)
    plaintext.insert(i, result)
    i += 1

print('Plaintext is',plaintext)
```

#Function for Converting numbers to Alphabets

```
def Alphabet_Generation(n):
    if n == 0:
        n = 'A'
        return n
    elif n == 1:
        n = 'B'
        return n
    elif n == 2:
        n = 'C'
        return n
```

```
elif n == 3:
    n = 'D'
    return n
elif n == 4:
    n = 'E'
    return n
elif n == 5:
    n = 'F'
    return n
elif n == 6:
    n = 'G'
    return n
elif n == 7:
    n = 'H'
    return n
elif n == 8:
    n = 'I'
    return n
elif n == 9:
    n = 'J'
    return n
elif n == 10:
    n = 'K'
    return n
elif n == 11:
    n = 'L'
    return n
elif n == 12:
    n = 'M'
    return n
elif n == 13:
```

```
    n = 'N'
    return n
elif n == 14:
    n = 'O'
    return n
elif n == 15:
    n = 'P'
    return n
elif n == 16:
    n = 'Q'
    return n
elif n == 17:
    n = 'R'
    return n
elif n == 18:
    n = 'S'
    return n
elif n == 19:
    n = 'T'
    return n
elif n == 20:
    n = 'U'
    return n
elif n == 21:
    n = 'V'
    return n
elif n == 22:
    n = 'W'
    return n
elif n == 23:
    n = 'X'
```

```
        return n
    elif n == 24:
        n = 'Y'
        return n
    else:
        n = 'Z'
        return n
```

#Creating a nested loop for finding the 3 numbers(x,y,z) from the plain text as instructed in the question (x*26^2) + (y*26) + z = plaintext number

#Once we get the x,y,z; we will call the Alphabet generation function to assign the appropriate Alphabets and Concatenating the final plaintext

```
final = ' Plaintext After Converting to Alphabets =  '
pt = 0
length = len(plaintext)
while pt < length:
  for i in range(26):
      for j in range(26):
          for k in range(26):
              results= (26*26*i)+(26*j)+k
              if results == plaintext[pt]:
                  i_value = Alphabet_Generation(i)
                  j_value = Alphabet_Generation(j)
                  k_value = Alphabet_Generation(k)
                  final += i_value + j_value + k_value
                  break
          else:
              continue
          break
      else:
          continue
```

```
    break
  pt +=1


print(final)
```

OUTPUT:

p= 173

q= 181

phin= 30960

d= 6497

ciphertext = [6340, 8309, 14010, 8936, 27358, 25023, 16481, 25809, 23614, 7135, 24996, 30590, 27570, 26486, 30388, 9395, 27584, 14999, 4517, 12146, 29421, 26439, 1606, 17881, 25774, 7647, 23901, 7372, 25774, 18436, 12056, 13547, 7908, 8635, 2149, 1908, 22076, 7372, 8686, 1304, 4082, 11803, 5314, 107, 7359, 22470, 7372, 22827, 15698, 30317, 4685, 14696, 30388, 8671, 29956, 15705, 1417, 26905, 25809, 28347, 26277, 7897, 20240, 21519, 12437, 1108, 27106, 18743, 24144, 10685, 25234, 30155, 23005, 8267, 9917, 7994, 9694, 2149, 10042, 27705, 15930, 29748, 8635, 23645, 11738, 24591, 20240, 27212, 27486, 9741, 2149, 29329, 2149, 5501, 14015, 30155, 18154, 22319, 27705, 20321, 23254, 13624, 3249, 5443, 2149, 16975, 16087, 14600, 27705, 19386, 7325, 26277, 19554, 23614, 7553, 4734, 8091, 23973, 14015, 107, 3183, 17347, 25234, 4595, 21498, 6360, 19837, 8463, 6000, 31280, 29413, 2066, 369, 23204, 8425, 7792, 25973, 4477, 30989]

Plaintext is [7446, 3290, 786, 9810, 12494, 12673, 16628, 9924, 341, 16706, 5694, 8870, 14317, 16707, 11713, 4557, 2808, 11993, 4836, 14318, 13910, 8914, 1808, 10509, 11858, 7243, 5698, 12537, 11858, 7247, 3047, 2827, 4946, 4953, 13030, 1733, 8909, 12537, 3592, 7532, 10350, 6867, 1306, 3046, 11511, 5785, 12537, 14000, 9822, 8229, 13331, 9848, 11713, 2566, 13310, 10001, 340, 3191, 9924, 1548, 12291, 4851, 7438, 3206, 11599, 3836, 8613, 2929, 7813, 4304, 13975, 3346, 83, 9913, 554, 8955, 2853, 13030, 8495, 4853, 7449, 2058, 4953, 2041, 2529, 5891, 7438, 3150, 8333, 2135, 13030, 8481, 13030, 11614, 9965, 3346, 3217, 7543, 4853, 2848, 11652, 13346, 8963, 4751, 13030, 16406, 2318, 3893, 4853, 2734, 1372, 12291, 4847, 341, 14890, 9300, 9634, 9560, 9965, 3046, 15166, 3503, 13975, 2796, 14033, 9494, 300, 13138, 6883, 4850, 7518, 15604, 10653, 4858, 5765, 3164, 446, 15253, 12297]

Plaintext After Converting to Alphabets=
LAKEWOBEGONISMOSTLYPOORSANDYSOILANDEVERYSPRINGTHEEARTHHEAVESUPAN
EWCROPOFROCKSPILESOFROCKSTENFEETHIGHINTHECORNERSOFFIELDSPICKEDBYGEN
ERATIONSOFUSMONUMENTSTOOURINDUSTRYOURANCESTORSCHOSETHEPLACETIREDF
ROMTHEIRLONGJOURNEYSADFORHAVINGLEFTTHEMOTHERLANDBEHINDANDTHISPLAC
EREMINDEDTHEMOFTHERESOTHEYSETTLEDHEREFORGETTINGTHATTHEYHADLEFTTHE
REBECAUSETHELANDWASNTSOGOODSOTHENEWLIFETURNEDOUTTOBEALOTLIKETHEO
LDEXCEPTTHEWINTERSAREWORSEZ

PROGRAMMING PROJECT: 2. ElGamal

Decrypt the ElGamal (see page 317) ciphertext presented in the following table. The parameters of the system are the prime number p=31847, primitive root e1=5, e2=18074. Each element of Zn represents three alphabetic characters as in the above problem. ElGamal Ciphertext (3781, 14409) (31552, 3930) (27214, 15442) (5809, 30274) (5400, 31486) (19936, 721) (27765, 29284) (29820, 7710) (31590, 26470) (3781, 14409) (15898, 30844) (19048, 12914) (16160, 3129) (301, 17252) (24689, 7776) (28856, 15720) (30555, 24611) (20501, 2922) (13659, 5015) (5740, 31233) (1616, 14170) (4294, 2307) (2320, 29174) (3036, 20132) (14130, 22010) (25910, 19663) (19557, 10145) (18899, 27609) (26004, 25056) (5400, 31486) (9526, 3019) (12962, 15189) (29538, 5408) (3149, 7400) (9396, 3058) (27149, 20535) (1777, 8737) (26117, 14251) (7129, 18195) (25302, 10248) (23258, 3468) (26052, 20545) (21958, 5713) (346, 31194) (8836, 25898) (8794, 17358) (1777, 8737) (25038, 12483) (10422, 5552) (1777, 8737) (3780, 16360) (11685, 133) (25115, 10840) (14130, 22010) (16081, 16414) (28580, 20845) (23418, 22058) (24139, 9580) (173, 17075) (2016, 18131) (19886, 22344) (21600, 25505) (27119, 19921) (23312.16906) (21563, 7891) (28250, 21321) (28327, 19237) (15313, 28649) (24271, 8480) (26592, 25457) (9660, 7939) (10267, 20623) (30499, 14423) (5839, 24179) (12846, 6598) (9284, 27858) (24875, 17641) (1777, 8737) (18825, 19671) (31306, 11929) (3576, 4630) (26664, 27572) (27011, 29164) (22763, 8992) (3149, 7400) (8951, 29435) (2059, 3977) (16258, 30341) (21541, 19004) (5865, 29526) (10536, 6941) (1777, 8737) (17561, 11884) (2209, 6107) (10422, 5552) (19371, 21005) (26521, 5803) (14884, 14280) (4328, 8635) (28250, 21321) (28327, 19237) (15313, 28649)

Code Written on Python:

#Initializing e1, e2 and p

p = 31847

e1 = 5

e2 = 18074

#Initializing Cipher Text C1 and C2

C1 = [3781, 31552, 27214, 5809, 5400, 19936, 27765, 29820, 31590, 3781, 15898, 19048, 16160, 301, 24689, 28856, 30555, 20501, 13659, 5740, 1616, 4294, 2320, 3036, 14130, 25910, 19557, 18899, 26004, 5400, 9526, 12962, 29538, 3149, 9396, 27149, 1777, 26117, 7129, 25302, 23258, 26052, 21958, 346, 8836, 8794, 1777, 25038, 10422, 1777, 3780, 11685, 25115, 14130, 16081, 28580, 23418, 24139, 173, 2016, 19886, 21600, 27119, 23312, 21563, 28250, 28327, 15313, 24271, 26592, 9660, 10267, 30499, 5839, 12846, 9284, 24875, 1777, 18825, 31306, 3576, 26664, 27011, 22763, 3149, 8951, 2059, 16258, 21541, 5865, 10536, 1777, 17561, 2209, 10422, 19371, 26521, 14884, 4328, 28250, 28327, 15313]

C2 = [14409, 3930, 15442, 30274, 31486, 721, 29284, 7710, 26470, 14409, 30844, 12914, 3129, 17252, 7776, 15720, 24611, 2922, 5015, 31233, 14170, 2307, 29174, 20132, 22010, 19663, 10145, 27609, 25056, 31486, 3019, 15189, 5408, 7400, 3058, 20535, 8737, 14251, 18195, 10248, 3468, 20545, 5713, 31194, 25898, 17358, 8737, 12483, 5552, 8737, 16360, 133, 10840, 22010, 16414, 20845, 22058, 9580, 17075, 18131, 22344, 25505, 19921, 16906, 7891, 21321, 19237, 28649, 8480, 25457, 7939, 20623, 14423, 24179, 6598, 27858, 17641, 8737, 19671, 11929, 4630, 27572, 29164, 8992, 7400, 29435, 6941, 8737, 19004, 29526, 6941, 8737, 11884, 6107, 5552, 21005, 5803, 14280, 8635, 21321, 19237, 28649]

#To compute modular power --https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/

```
def power(x, y, p) :
    res = 1     # Initialize result

    # Update x if it is more
    # than or equal to p
    x = x % p

    if (x == 0) :
        return 0

    while (y > 0) :

        # If y is odd, multiply
        # x with result
        if ((y & 1) == 1) :
            res = (res * x) % p

        # y must be even now
        y = y >> 1      # y = y/2
        x = (x * x) % p

    return res
```

#Finding the correct value for d, using formula e2 = e1^d mod p. We have e1, e2 and p values given. Using a loop to try all the numbers till n such that e1^i mod p would give us a result e2

```
for i in range(1,31847):
```

```
  results= power(e1, i, p)
  if results == 18074:
     d= i
     break

print('Private key D=',d)
```

#Generating Plaintext from the ciphertext using our formula : P = [C2 * C1^(p-1-d)] mod p

```
y = p-1-d
z = 1
plaintext= []
# Getting length of list
length = len(C1)
i = 0
# Iterating using while loop
while i < length:
    x = C1[i]
    result= power(x, y, p)
    result = result * C2[i]
    final_result = power(result, z, p)
    plaintext.insert(i, final_result)
    i += 1

print('Plaintext is',plaintext)
```

#Function for Converting numbers to Alphabets

```
def Alphabet_Generation(n):
    if n == 0:
        n = 'A'
        return n
    elif n == 1:
```

```
    n = 'B'
    return n
elif n == 2:
    n = 'C'
    return n
elif n == 3:
    n = 'D'
    return n
elif n == 4:
    n = 'E'
    return n
elif n == 5:
    n = 'F'
    return n
elif n == 6:
    n = 'G'
    return n
elif n == 7:
    n = 'H'
    return n
elif n == 8:
    n = 'I'
    return n
elif n == 9:
    n = 'J'
    return n
elif n == 10:
    n = 'K'
    return n
elif n == 11:
    n = 'L'
```

```
    return n
elif n == 12:
    n = 'M'
    return n
elif n == 13:
    n = 'N'
    return n
elif n == 14:
    n = 'O'
    return n
elif n == 15:
    n = 'P'
    return n
elif n == 16:
    n = 'Q'
    return n
elif n == 17:
    n = 'R'
    return n
elif n == 18:
    n = 'S'
    return n
elif n == 19:
    n = 'T'
    return n
elif n == 20:
    n = 'U'
    return n
elif n == 21:
    n = 'V'
    return n
```

```
    elif n == 22:
        n = 'W'
        return n
    elif n == 23:
        n = 'X'
        return n
    elif n == 24:
        n = 'Y'
        return n
    else:
        n = 'Z'
        return n
```

#Creating a nested loop for finding the 3 numbers(x,y,z) from the plain text as instructed in the question (x*26^2) + (y*26) + z = plaintext number

#Once we get the x,y,z; we will call the Alphabet generation function to assign the appropriate Alphabets and Concatenating the final plaintext

```
final = 'Plaintext After Converting to Alphabets=  '
pt = 0
length = len(plaintext)
while pt < length:
  for i in range(26):
      for j in range(26):
          for k in range(26):
              results= (26*26*i)+(26*j)+k
              if results == plaintext[pt]:
                  i_value = Alphabet_Generation(i)
                  j_value = Alphabet_Generation(j)
                  k_value = Alphabet_Generation(k)
                  final += i_value + j_value + k_value
                  break
```

```
        else:
            continue
        break
    else:
        continue
    break
 pt +=1

print(final)
```

Output:

```
Private key D= 7899
Plaintext is [12354, 12662, 8884, 13918, 9289, 2860, 11574, 9367, 3150, 12354, 4750, 784, 9374, 11760,
8944, 8877, 9838, 12389, 13227, 4839, 5895, 340, 3179, 2886, 12640, 9260, 2030, 4745, 4168, 9289,
3280, 501, 3165, 4853, 2930, 352, 13030, 11668, 12676, 3960, 8866, 1198, 7050, 9607, 9690, 2925,
13030, 225, 341, 13030, 12855, 7512, 10586, 12640, 12658, 642, 4855, 13975, 12181, 2354, 14318,
13917, 5705, 13230, 445, 12669, 2900, 13992, 1646, 8146, 8958, 14317, 300, 15444, 7727, 2819, 5752,
13030, 3605, 12665, 11871, 12537, 11500, 9165, 4853, 693, 22294, 24471, 4838, 11874, 12290, 13030,
7806, 4264, 341, 11344, 1623, 16341, 12965, 12669, 2900, 13992]
Plaintext After Converting to Alphabets=
SHESTANDSUPINTHEGARDENWHERESHEHASBEENWORKINGANDLOOKSINTOTHEDISTA
NCESHEHASSENSEDACHANGEINTHEWEATHERTHEREISANOTHERGUSTOFWINDABUCKL
EOFNOISEINTHEAIRANDTHETALLCYPRESSESSWAYSHETURNSANDMOVESUPHILLTOWA
RDSTHEHOUSECLIMBINGOVERALOWWALLFEELINGTHEFIRSTDROPSOFRAINONHERBAR
HECROSSESTHELOGGIAANDQUICKLYENTERSTHEHOUSE
```

CSCI 561 Research Project: Attacks on low-exponent RSA

1. Introduction:

The RSA cryptosystem, named after R. Rivest, A. Shamir, and L. Adleman, who developed it in 1978, is the most commonly used public-key cryptosystem. It can be used for both confidentiality and digital signatures. Its security is also dependent on the integer factorization problem's intractability [2].

In a public-key cryptosystem, each user stores an encryption method in a public record E. That is, the public file is a file that maintains each user's encryption method. The individual keeps the specifics of his decryption method D a secret.

The four properties of these procedures are as follows [8]:

- Deciphering a message's enciphered form yields M. D(E(M)) = M in formal terms.
- E and D are both simple to calculate.
- The user does not provide a simple way to compute D by disclosing E publicly. In general, this means that only he can decrypt messages encrypted with E or efficiently calculate D.
- M is the product of deciphering and then enciphering a message M. E(D(M)) = M in formal terms.

A general procedure and an encryption key are usually used in an encryption or decryption operation. With control of the key, the general method encrypts a message M to obtain the ciphertext C, which is the encoded form of the message. All should use the same general method; the security of a process will be determined by the key's security. The key is revealed when an encryption algorithm is revealed. When the user shows E, he presents a very inefficient way of calculating D(C): checking all possible messages M until it finds E(M) = C. The number of messages to evaluate will be so high that this method will be inefficient [8].

The modulus N of the RSA cryptosystem is the product of two big prime numbers p and q. The encryption and decryption exponents are represented by the integers e and d, respectively. The relation $ed \equiv 1 \pmod{\phi(N)}$, where $\phi(N) = (p-1)(q-1)$ is called the Euler's totient function, which interrelates the exponents e and d [2].

The integer factorization problem is related to the protection of RSA: find the prime factorization of a positive integer N. Recovering A's private key d or factoring its RSA modulus N is a more optimistic adversary goal. The primary goal of an adversary wishing to attack RSA is to recover plaintext from ciphertext intended for some other entity A, i.e., to solve the RSA problem. If this is accomplished, the RSA cryptosystem is said to have been broken informally since the adversary would be able to decode all ciphertext sent to A [1].

2. Low private Exponent

2.1 Wiener attack: If $p < q < 2p$, $e < pq$ and $d < N^{1/4}$, then by using continued fraction expansion $\frac{e}{N}$, we can recover the decryption exponent d from the public key pair (e, N). Since $ed \equiv 1 \pmod{\phi(N)}$, there is a $k \in Z$ such that $ed = 1 + k\phi(N)$ [2]. Hence, we get

$$\left|\frac{e}{\phi(N)} - \frac{k}{d}\right| = \frac{1}{d\phi(N)}$$

As a result, we have $\frac{k}{d} \approx \frac{e}{\phi(N)}$. Furthermore, the modulus N is a close approximation to ϕ(N). Therefore, we have $\frac{k}{d} \approx \frac{e}{n}$. An attacker can use N to approximate ϕ (N), Because of ϕ(N) = N −(p+q−1) and p+q−1 < 3 $\sqrt{N}$, we have |N − ϕ(N)| < 3 $\sqrt{N}$ and [2]

$$\left|\frac{e}{N} - \frac{k}{d}\right| \leq \frac{3k}{d\sqrt{N}} \leq \frac{1}{2d^2}$$

So $\frac{k}{d}$ is a convergent of the continued fraction expansion of $\frac{e}{n}$ [2]

The Euclidean algorithm is used to calculate the continued fraction representation of a positive rational number [2].

For example, suppose N has a standard size of n=1024 bits, or 309 decimal digits and two large primes of the same size (n/2 bits each), then, in order to avoid this attack, since N is 1024 bits, d must be at least 256 bits long [3].

2.2 Lattice attacks:

Phong Q. Nguyen proposed a method for recovering the private exponent d for a two-dimensional lattice based on the solution of the shortest problem finding. Using two-dimension, d < $N^{1/4}$, he demonstrated that this is just a heuristic. Assume that p and q are two prime numbers of the same bit length. There is a k ∈ Z such that ed =1+ k ϕ(N) when ed ≡ 1 (mod ϕ(N)) [2].

Consider the 2-rank lattice L, which is spanned by two vectors u = (e, $\sqrt{N}$) and v = (N, 0), then L contains vector t = d × u – k × v = (ed – kN, d$\sqrt{N}$). The norm of $\|t\| = \sqrt{(ed - kN)^2 + (d\sqrt{N})^2}$ ≈ $d\sqrt{N}$, while $\sqrt{vol(L)} = N^{3/4}$. If $d\sqrt{N} < \sqrt{vol(L)} = N^{3/4}$, vector t may be the shortest vector. It denotes the d < $N^{1/4}$. We use Gaussian's lattice reduction algorithm on two vectors, u and v, to find t. The algorithm finishes and produces the vectors u^* and v^*, where $\|u\| < \|v\|$ [2].

It should be noted that vector u^* is the smallest non-zero lattice L vector. The lattice reduction algorithm of Gaussian will stop in at most $\left|log_1 + \sqrt{2}^{\frac{\|u\|}{\lambda_2}}\right|$ + 3 iterations, where λ_2 is the second minima of L. If u^* = ± t, then our prediction is most likely right, and the value of the decryption exponent d can be calculated using the vector u [2].

3. Low public exponent

A small public exponent e is commonly used to minimize encryption or signature-verification time. The smallest possible value is 3, but the value e = 2^{16} + 1 is suggested to defeat such attacks. Actually 17 multiplications are necessary for signature verification when the value 2^{16} + 1 is used, comparison to roughly 1000 when a random e < ϕ(N) is used. Apart from attacks based on a low private exponent, attacks based on a small e are far from complete [3][4].

3.1 Theorem 1: Coppersmith theorem

A Copper-smith theorem incorporates the most efficient attacks on RSA with a low public exponent. Let N be an integer and f ϵ Z[x] be a 'd' degree monic polynomial. For any ϵ ≥ 0, make X = $N^{1/d-\epsilon}$. The attacker can then efficiently find all integers $|x_0|$ < X satisfying f(x_0) = 0 mod N given (N, f). The time it takes to run the LLL algorithm on a lattice of dimension O(w) with w = min(1/ϵ, log_2N) dominates the running time. The theorem presents an important algorithm to determine all roots of f modulo N that are less than X = $N^{1/d}$. The algorithm's execution time decreases when X gets smaller. The potential of this theorem to find small roots of polynomials modulo a composite N [3][4].

3.2 Theorem 2: Hastad's broadcast attack

Hastad's Broadcast Attack is the first use of Coppersmith's theorem and an improvement on an old attack. Let's say Bob wants to send an encrypted message M to a group of people named P_1, P_2,..... P_k. Each people seem to have its own RSA key (N_i, e_i). We'll say M is less than the number of all N_i 's. To send M, Bob encrypts it with each of the public keys and sends it to p_i using the i^{th} ciphertext. Out of Bob's sight, an intruder named Eve may eavesdrop on the communication and collect the *k* transmitted ciphertexts [3][4][5].

Assume that all public exponents e_i are equal to three. Eve will recover M if *k* is less than three. Bob does indeed get C_1, C_2 and C_3, where C_1 = M^3 mod N_1, C_2 = M^3 mod N_2, C_3 = M^3 mod N_3. Since Eve can factor any of the N_i's, assume that gcd(N_i, N_j) = 1 for all i ≠ j. As a result, the Chinese Remainder Theorem (CRT) applied to C_1, C_2 and C_3 yields a C' ϵ $Z_{N_1N_2N_3}$ satisfying C' = M^3 mod $N_1N_2N_3$. We have M^3 < $N_1N_2N_3$ since M is less than all the N_i 's. Then C' = M^3 holds over the integers. As a result, Eve will recover M by computing C' real cube root. In general, Eve will recover M as soon as *k* > e if all public exponents are equal to e. Only when a small e is used is the attack possible [3][4][5].

Let's take a look at some numbers. Let's say someone uses trivial example of RSA to send the message M=102 to three different people, with moduli N_1=377, N_2=391 and N_3=589.[6]

C_1 =102^3 mod 377= 330

C_2 =102^3 mod 391= 34

C_3 =102^3 mod 589= 419

As a result, the attacker attempts to solve the following congruence problem:

C ≡ 330 (mod 377)

C ≡ 34 (mod 391)

C ≡ 419 (mod 589)

Using Chinese Remainder Theorem CRT, we can calculate the value of C [7].

We need to calculate the value of n = $N_1 * N_2 * N_3$= 377 * 391 * 589 = 86822723

Therefore, n_1= n/N_1 = 86822723 ÷ 377 = 230299

n_2= n/N_2 = 86822723 ÷ 391 = 222053

n_3= n/N_3 = 86822723 ÷ 589 = 147407

Now Calculating the inverse of ${n_1}^{-1}$, ${n_2}^{-1}$ and ${n_3}^{-1}$

Therefore, ${n_1}^{-1} = 230299^{-1}$ mod 377 = 322

${n_2}^{-1} = 222053^{-1}$ mod 391 = 67

${n_3}^{-1} = 147407^{-1}$ mod 589 = 574

As per CRT,

C= ($a_1 * n_1 * {n_1}^{-1} + a_2 * n_2 * {n_2}^{-1} + a_3 * n_3 * {n_3}^{-1}$) mod n

C= [(330 * 230299 * 322) + (34 * 222053 * 67) + (419 * 147407 * 574)] mod n

C= (24471571740 + 505836734 + 35452267942) mod 86822723

C = 60429676416 mod 86822723 = 1061208

Therefore C = 1061208

Hence, we can calculate M by M = $\sqrt[3]{C}$

M = $\sqrt[3]{1061208}$ = 102 which is the original message

Given a large enough group of people, RSA will function on any exponent; however, e=3 is the most practical setting.

3.3 Theorem 3: Franklin-Reiter

When Bob sends Alice related encrypted messages using the same modulus, Franklin and Reiter discovered a sophisticated approach to attack it. Let's call Alice's public key (N, e). Assume that M_1, M_2 are two separate messages, with M_1 = f (M_2) mod N. We will illustrate how an attacker can easily recover M_1 and M_2 if Bob encrypts the messages and transmits the resulting cyphers C_1 and C_2 [3][4].

Let (N, e) be an RSA public key with e = 3. . Let M_1 != M_2 satisfy M_1 = f (M_2) mod N for some linear polynomial f = ax + b with b != 0. Then the attacker can then recover M_1 and M_2 in time quadratic in log N given (N, e, C_1, C_2, f) [3][4].

We know that M_2 is a root of the polynomial g_1(x) = f(x)^e - C_1 since C1 = M_1^e mod N, and similarly M_2 is a root of g_2(x) = f(x)^e - C_2. Both polynomials are divided by the linear factor x - M_2. As a result, an attacker could compute the gcd of g_1 and g_2 using the Euclidean algorithm. M_2 is found if the gcd turns out to be linear [3][4]

3.4 Theorem 4: Coppersmith's short pad attack

Coppersmith enhanced the attack and produced a significant padding result. Coppersmith demonstrated that RSA encryption is insecure when Hastad's suggested randomized padding is used incorrectly [3][4]. A simple random padding algorithm will pad plaintext M by appending a few random bits to one of the ends. The following attack highlights the dangers of such straightforward padding. Assume Bob sends Alice a correctly padded encryption of M. Eve, an intruder, intercepts the ciphertext and prevents it from reaching its intended recipient. When Bob notices that Alice hasn't replied to his message, he decides to resend M to her. He pads M at random and sends the ciphertext again. Eve now has two ciphertexts, each leading to two separate random pad encryptions of the same message. Eve can recover the plaintext despite not knowing the pads used, as shown by the following theorem [3][4].

4. Calculating d when N & e have small values:

Let us assume e is a low exponent and if N which is the product of two prime numbers p and q, is small. Then we can find the values of p and q using most of the programming languages. Since p and q are the prime numbers, their factors would be 1 and the number itself. Hence, if we multiply these two prime numbers, there would be 4 factors of N. That is, 1, p, q and N. Once we are successfully able to calculate the 2 prime numbers, we can calculate ϕ (N) = (p − 1) (q − 1). With ϕ (N), we can then calculate the private key d using the formula $d = e^{-1} \bmod \phi(N)$. Since the values of e and ϕ (N) are low, the computational time required would be very less.

Let us take a trivial example from CSCI 561 programming project 1, with N= 31313, e = 4913. We will use Python programming language to calculate the factors of N. The Script below is checking for all the numbers from 1 to N and finding the 2 numbers that should be divisible by N ignoring 1 and N.

Code:

```
#Calculating p and q which would be the factors of n
factors= []
def getfactors(n) :
   f= 0
   i = 1
   while i <= n :
      if n % i==0:
        if i!=1:
          if i!=n:
            factors.insert(f, i),
            f = f + 1
      i = i + 1
```

getfactors(n)

p= factors[0]

q= factors[1]

print('p=',p)

print('q=',q)

We get the result a p = 173 and q =181. Hence ϕ (N) = (p − 1) (q − 1) = 30960. Hence, we can calculate d = e^{-1} mod ϕ (N) = 4913^{-1} mod 30960 = 6497. Once we obtain the value of secret key d, we can decrypt any cipher text that was encrypted using e and N.

5. Conclusion:

We may infer that the lattice attack is more successful than the Wiener attack in recovering the private key d from a pair of public keys (e, N) when a low private key d exponent is used [2]. We have also shown that if e and N are having small values, we can easily calculate the private key d. When implemented correctly, RSA will keep its promise of protection, and the majority of documented RSA attacks are simply examples of improper implementation. If RSA is used correctly then it can overcome such attacks [1].

References:

1. "An Application of Low Private Exponent Attack on RSA", Yong-Hui Zheng, Yue-Fei Zhu, Hong Xu: Proceedings of 2009 4th International Conference on Computer Science & Education
2. "Attacks on low private exponent RSA: an experimental study" Thuc D. Nguyen, Than Duc Nguyen, Long D. Tran: 2013 13th International Conference on Computational Science and Its Applications
3. "RSA Attacks", Abdulaziz Alrasheed, Fatima: https://www.utc.edu/center-academic-excellence-cyber-defense/pdfs/course-paper-5600-rsa.pdf
4. Coppersmith's attack: https://en.wikipedia.org/wiki/Coppersmith's_attack
5. "Attack on RSA with Low Public Exponent". :https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/rsa.pdf
6. https://crypto.stackexchange.com/questions/6713/low-public-exponent-attack-for-rsa
7. CSCI 561 Chapter 9 Mathematics of Cryptography: Chinese Remainder Theorem
8. "A Method for Obtaining Digital Signatures and PublicKey Cryptosystems", R.L. Rivest, A. Shamir, L. Adleman: https://people.csail.mit.edu/rivest/Rsapaper.pdf