



**Princess Sumaya** جامعة  
**University** الأميرة سميرة  
**for Technology** للتكنولوجيا

Princess Sumaya University for Technology  
Department of Data Science  
Fall Semester 2023/2024

# Public-Key Cryptosystems

**RSA & RSA Attack**

**Prepared By:**

Sondos Ali 20200438  
Raghad Joudeh 20200777  
Jessica Abueledam 20200137

**Instructor:**

Dr. Mustafa Al-fayoumi

## **Introduction**

Public-key cryptography is also called asymmetric cryptography, a technique used to encrypt or sign data with two non-identical keys. The first key is the public key which means that anyone can access or use it. On the other hand, the second key is known as the private key. Both the sender and the receiver must have two keys, one public and the other private. The public key is used to encrypt the message, then the sender sends the encrypted message using the receiver's public key to the receiver. Only the receiver can decrypt the message with his own private key.

Encryption of data has some challenges such as the need of extra resources, cost of implementation and maintenance, and increases complexity. Nevertheless, data encryption has a positively high impact on making data more secure offering the CIA (confidentiality, integrity, authentication) security benefits.

Applications such as digital signature, symmetric key distribution, and encryption of secret keys use public key cryptography. Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications.

## **Public key cryptosystems**

RSA is one of the algorithms used in public key cryptography. A strong RSA algorithm depends on the size of the key, so if we doubled the key size the strength of encryption increases exponentially. The RSA algorithm is still being used by cryptographers with key sizes of 1024 or 2048 bits long.

## **Prototype to the RSA Algorithm**

In this project, we will work on a python implementation of the RSA algorithm. First, we will discuss what processes it contains, prototype the algorithm, and what variables it needs.

The RSA algorithm contains generating keypairs (public and private), encryption and decryption of the message.

➤ Steps to generating keypairs are listed below:

- 1) Select two large prime numbers and assign them to P and Q variables.
- 2) The first part of the public key can be generated by multiplying P by Q. Assigning the product to variable n. The product of the two prime numbers must be larger than the message we want to encrypt.

- 3) Then assign an integer number to variable  $e$ , a small exponent. Variable  $e$  must be greater than one and less than  $\Phi(n)$  and  $\gcd(e, \Phi(n)) = 1$ , where  $\Phi(n) = (p-1)*(q-1)$ . The value of  $e$  will be the public key exponent.
- 4) To compute the private key exponent  $d$ , we need to find an integer  $d$  such that  $d*e = 1 \pmod{\Phi(n)}$ . This can be done using the extended Euclidean algorithm.

In conclusion, our public key is  $(n, e)$  and our private key is  $(n, d)$ .

➤ The methodology of data encryption is discussed below:

- 1) The message is of datatype string and in order to encrypt the message, it has to be an integer between 0 and  $n-1$ . Converting the message into integer can be done using ASCII or UTF-8.
- 2) Having the integer representation of the message, we compute the ciphertext,  $c$ , using modular exponentiation algorithms, where  $c = m^e \pmod{n}$ .

Encryption and decryption steps are similar. In ciphertext decryption, we compute the message as  $m = c^d \pmod{n}$ . Again, we can use modular exponentiation algorithms to do this efficiently.

### **Question 5: Implementation of RSA Algorithm (Keypair Generation, Encryption & Decryption)**

The language used to implement the public-key encryption method is Python3. A detailed description of the code is discussed as follows:

The first step is to import all the libraries that we might need during our implementation.

```
>> import random
```

Defining two functions to find the Greatest Common Factor. GCD1 function returns the GCD using the Extended Euclidean Algorithm, while GCD2 simply returns the GCD using the basic Euclidean Algorithm. GCD2 is used to compute the public key, while GCD1 is used to compute the private key.

```
>>def GCD1(a, b):  
>> if a == 0:  
>> return (b, 0, 1)
```

```

>> else:
>>     g, y, x = GCD1(b % a, a)
>>     return (g, x - (b // a) * y, y)

>>def GCD2(a, b):
>>     if (b == 0):
>>         return a;
>>     return GCD2(b, a % b)

```

Function keypairs calculates and returns a pair of keys, public key and private key. Comments are added throughout the code implementation to a more understanding methodology.

```

>>def keypairs():
# Ask the user to enter two different prime numbers
>> p = int(input("Enter a prime number p : "))
>> q = int(input("Enter another prime number q other than p : "))
# Validation
>> while p == q:
>>     print("p must not equal q!")
>>     q = int(input("Please enter another prime number q other than p : "))

# Finding variables n and phi(n)
>> n = p * q
>> phi = (p-1) * (q-1)
# Ask user to enter value for e
>> e = int(input("Enter the value of e(choose e coprime to n and 1 < e < phi(n) : "))

# Ensure the GCD of e and phi(n) = 1
>> g = GCD2(e,phi)
>> while g != 1:
>>     e = random.randint(1, phi)
>>     g = GCD2(e, phi)

>> print("\nThe value first part of the public key n = ",n)
>> print("The second part of the public key e = ",e," and phi(n) = ",phi)

# Computing the private key
>> d = GCD1(e, phi)[1]

```

```

>> d = d % phi
>> if(d < 0):
>>     d += phi

>> return ((e,n),(d,n))

```

Defining functions `textdecrypt` and `textencrypt` to decrypt and encrypt the messages or the ciphertext respectively. `Textdecrypt` function takes a ciphertext and a private key tuple and returns the plaintext using  $m = c^d \pmod n$ . The `textencrypt` function takes the plaintext and the public key and returns the ciphertext using  $c = m^e \pmod n$ .

```

>>def textdecrypt(ctext, pri_key):
>> try:
>>     key, n = pri_key
>>     text = [chr(pow(char, key, n)) for char in ctext]
>>     return "".join(text)
>> except TypeError as e:
>>     print(e)

>>def textencrypt(plaintext, public_key):
>> key,n = public_key
>> ctext = [(ord(char) ** key) % n for char in plaintext]
>> return ctext

```

The figure below shows a sample output when running the RSA Algorithm, encrypting the word 'hello'.

---

```

Enter a prime number p : 61
Enter another prime number q other than p : 53
Enter the value of e(choose e coprime to n and 1 < e < phi(n)) : 17

The value first part of the public key n = 3233
The second part of the public key e = 17 and phi(n) = 3120

Public key: (17, 3233)
Private key: (2753, 3233)

Enter text to be encrypted: hello
Encrypted text = 217013137457452185
Decrypted text = hello

```

Figure 1: Output sample to running RSA Algorithm

## Question 6: Implementation of Math-Based Key Recovery Attacks on RSA

If a cryptanalyst knows the value of  $\Phi(n)$ , then he can factor  $n$  and break the system. In other words, computing  $\Phi(n)$  is no easier than factoring  $n$ .

Three possible approaches:

1. Factor  $n = pq$
2. Determine  $\Phi(n)$
3. Find the private key  $d$  directly

In fact, knowing both  $n$  and  $\Phi(n)$ , one knows

- $n = pq$
- $\Phi(n) = (p-1)(q-1) = pq - p - q + 1 = n - p - n/p + 1$
- $p\Phi(n) = np - (p^2) - n + p$
- $p^2 - np + \Phi(n)p - p + n = 0$
- $p^2 - (n - \Phi(n) + 1)p + n = 0$

Example: suppose the cryptanalyst has learned that  $n = 84773093$  and  $\Phi(n) = 84754668$ .

Find out the two factors of  $n$ :

- Equation:  $p^2 - 18426p + 84773093 = 0$
- Solutions: 9539 and 8887

We have used two libraries for a simple GUI and here is the code:

```
>>import tkinter as tk

>>from tkinter import messagebox

>># GUI setup

>>root = tk.Tk()

>>root.title("RSA Decryptor")

>># UI elements
```

```

>>label_n = tk.Label(root, text="Enter the value of n:")

>>label_n.pack()

>>entry_n = tk.Entry(root)

>>entry_n.pack()

>>label_e = tk.Label(root, text="Enter the value of e:")

>>label_e.pack()

>>entry_e = tk.Entry(root)

>>entry_e.pack()

>>label_cipher = tk.Label(root, text="Enter the cipher text (4 characters):")

>>label_cipher.pack()

>>entry_cipher = tk.Entry(root)

>>entry_cipher.pack()

>>decrypt_button = tk.Button(root, text="Decrypt", command=find_pq_and_decrypt)

>>decrypt_button.pack()

>># Start the GUI event loop

>>root.mainloop()

```

Function find\_pq takes n and e as parameters then it calculates two prime numbers p and q and returns them. Find\_pq uses is\_prime function to check if the number is prime or not.

```

>>def find_pq(n, e):

>>    p, q = None, None

>>    for i in range(2, n):

>>        if n % i == 0 and is_prime(i):

```

```

>>     q = i
>>     p = n // i
>>     break
>> return p, q

# Is_prime function takes a number and returns true if it is prime and false otherwise.

>>def is_prime(num):
>> for i in range(2, int(num**0.5) + 1):
>>     if num % i == 0:
>>         return False
>> return True

```

Defining find\_pq\_and\_decrypt function that uses find\_pq function to find the two prime numbers by factoring the given value of n, and it uses calculate\_private\_key function that takes p, q, and e as parameters to compute a tuple consisting of d (private\_key) and n (p\*q). Find\_pd\_and\_decrypt also uses text\_decrypt function to find the plaintext when given the ciphertext and the private key computed from calculate\_private\_key function. Text\_decrypt implementation is the same as the decryption process mentioned in Question 5.

```

>>def find_pq_and_decrypt():
>> try:
>>     n = int(entry_n.get())
>>     e = int(entry_e.get())
>>     cipher_text = entry_cipher.get()
>>     cipher_text = cipher_text.split(",")
>>     cipher_text = list(map(int, cipher_text))
>>     p, q = find_pq(n, e)
>>     if p is not None and q is not None:

```



```

>> # Decrypt the cipher text

>>     private_key = calculate_private_key(p, q, e)

>>     decrypted_text = text_decrypt(cipher_text, private_key)

>>     messagebox.showinfo("Decryption Successful", f"Decrypted text:
{decrypted_text}")

>>     else:

>>         messagebox.showerror("Error", "Unable to find p and q.")

>> except ValueError:

>>         messagebox.showerror("Error", "Invalid input. Please enter valid integers.")

```

```

>>def calculate_private_key(p, q, e):

>>    phi = (p - 1) * (q - 1)

>># gcd_extended function is of the same implementation of GCD1 function mentioned in
Question 5.

>>    d = gcd_extended(e, phi)[1]

>>    d = d % phi if d < 0 else d + phi

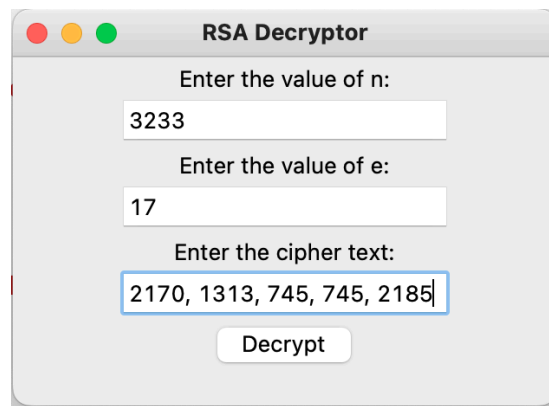
>>    return d, p * q

```

The figures below show the GUI and the output of the plaintext.



Figure 2: The Design of the GUI of RSA Decryption

A screenshot of a software window titled "RSA Decryptor". It contains three input fields: "Enter the value of n:" with the value "3233", "Enter the value of e:" with the value "17", and "Enter the cipher text:" with the value "2170, 1313, 745, 745, 2185". Below the input fields is a button labeled "Decrypt".

RSA Decryptor

Enter the value of n:  
3233

Enter the value of e:  
17

Enter the cipher text:  
2170, 1313, 745, 745, 2185

Decrypt

Figure 3: Inserting the values of  $n$ ,  $e$  and a list of cipher text

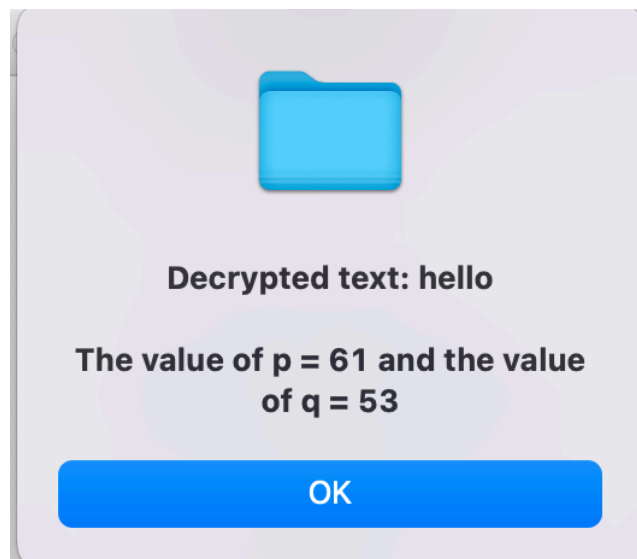


Figure 4: The output of the decrypted text, the value of  $p$  and the value of  $q$

## Conclusion

In conclusion, the security of RSA encryption's scheme depends on the hardness of the RSA problem. If factoring is hard, so is finding out  $d$ . RSA-640 bits, Factored Nov.2 2005. RSA-200 (663 bits) factored in May 2005. RSA-768 has 232 decimal digits and was factored on December 12, 2009, latest.