جامــــعــة المــــســتــقــبــل
AL MUSTAQBAL UNIVERSITY

قســم الامـــــن الـــــــســيبرانـــــي
DEPARTMENT OF CYBER SECURITY

SUBJECT:

PUBLIC KEY ENCRYPTION

CLASS:

THIRD

LECTURER:

ASST. LECTURER QUSAI AL-DURRAH

LECTURE (5 & 6):

RSA CRYPTOSYSTEM AND DIGITAL SIGNATURES

1. Introduction

The **RSA cryptosystem**, developed by **Rivest, Shamir, and Adleman (1977)**, is the most widely used **public-key encryption algorithm** in modern information security. It supports both **confidentiality** (encryption/decryption) and **authentication** (digital signatures). Its security relies on the **difficulty of factoring very large integers**

Although the RSA scheme was publicly announced in 1977, historical records later revealed that it had been invented earlier (in 1973) at **GCHQ (UK)** by **Clifford Cocks**, but was kept classified.

RSA is a block cipher in which the plaintext and ciphertext are integers between 0 and (n-1) for some (n). Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$$C = M^{e} \bmod n$$
$$M = C^{d} \bmod n$$

Both sender and receiver must know the values of ***n*** and ***e***, and only the receiver knows the value of ***d***. This is a public-key encryption algorithm with a public key of PU = $\{e, n\}$ and a private key of PR = $\{d, n\}$.

In this lecture, we will explore:

- The mathematical basis of RSA.
- The key generation, encryption, and decryption algorithms.
- The RSA signature scheme.
- Security issues and common attacks.
- Code of the RSA algorithm in python.

2. Learning Outcomes

By the end of these two lectures, students will be able to:

1. **Explain** the theoretical foundation of RSA and its reliance on number theory.
2. **Apply** the RSA key generation, encryption, and decryption algorithms to simple examples.
3. **Describe** how RSA provides both encryption and digital signatures.
4. **Demonstrate** how modular arithmetic ensures confidentiality and authenticity.
5. **Analyze** major security threats and attacks on RSA and discuss countermeasures.

3. RSA Algorithm:

❖ **Key Generation:**

• Select *p, q* :	where *p, q* both prime , $p \neq q$
• Calculate *n* :	$n = p \times q$
• Calculate $\Phi(n)$:	$\Phi(n) = (p-1) \times (q-1)$
• Select integer *e*:	$gcd(e, \Phi(n)) = 1$; $1 < e < \Phi(n)$
• Calculate *d* :	$d \times e \bmod \Phi(n) = 1$
Public Key	$PU = \{e, n\}$
Private key	$PR = \{d, n\}$

❖ **Encryption:**

• Plaintext	$M<n$
• Ciphertext	$C = M^e \bmod n$

❖ **Decryption:**

• Ciphertext	C
• Plaintext	$M = C^d \bmod n$

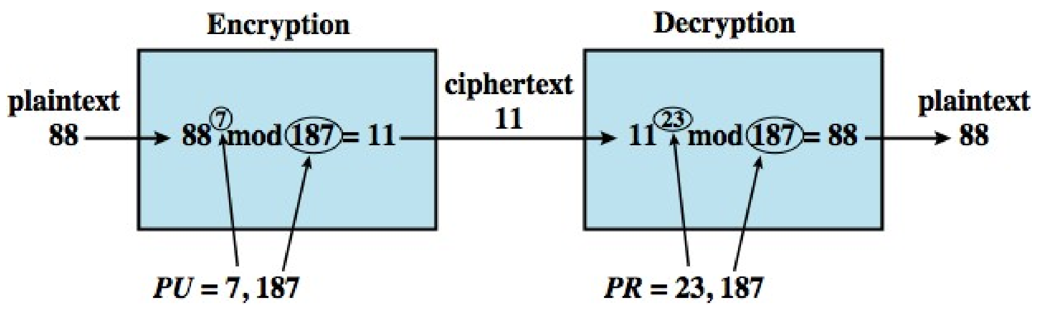

Figure 1: *RSA example*

3.1 RSA Example 1:

An example is shown in Figure 1 For this example, the keys were generated as:

1. Select two prime numbers, $p = 17$ and $q = 11$.

2. Calculate $n = p \times q = 17 \times 11 = 187$.

3. Calculate $\phi(n) = (p-1) \times (q-1) = 16 \times 10 = 160$.

4. Select (e) Where e is relatively prime to $\phi(n) = 160$ and less than $\phi(n)$;

$gcd\ (e\ ,\ \Phi(n)) = 1 \Rightarrow gcd(e\ ,\ 160) = 1$ we choose $e = 7$.

5. Determine d such that $d \times e \bmod 160 = 1$ and $d < 160$.

The correct value is $d = 23$, because $23 \times 7 = 161 \bmod 160 = 1$

The resulting keys are:

Public Key $PU = \{7, 187\}$

Private Key $PR = \{23, 187\}$.

The example shows the use of these keys for a plaintext input of $M = 88$.
If M<n then:

Encryption: $C = 88^7 \bmod 187 = 11$

Decryption: $M = 11^{23} \bmod 187 = 88$.

3.2 RSA Example 2:

Key Generation:

- **P =** 2357, **q =** 2551
- **n =**$2357 \times 2551 = 6{,}012{,}707$
- **φ(n) =** 6,007,800
- Choose **e =** 3,674,911
- Compute **d=** 422,191 such that $e \times d \equiv 1 (\bmod\ \varphi(n))$

Public Key: (n= 6,012,707, e= 3,674,911)
Private Key: (n= 6,012,707, d= 422,191)
Encryption:
For message m= 5,234,673:
$c = m^e \bmod n = 5{,}234{,}673^{3{,}674{,}911} \bmod 6{,}012{,}707 = 3{,}650{,}502$

Decryption:
$m=c^d \bmod n=3{,}650{,}502^{422{,}191} \bmod 6{,}012{,}707=5{,}234{,}673$

The decrypted message matches the original plaintext.

4. RSA Signature Scheme:

RSA can also provide **digital signatures**—ensuring *authentication, integrity,* and *non-repudiation*.

4.1 Signing and verification Processes

- Sender A computes: $s \equiv m^{d_A} \pmod{n_A}$

  where d_A is A’s private key.

- The receiver verifies by: $m \equiv s^{e_A} \pmod{n_A}$

If the result matches, the signature is valid.

4.2 RSA Signature Example

- $p=11,\ q=17 \Rightarrow n= p*q= 187$
- $\phi(n)=160$
- $e_A=27$, $d_A=3$
- $m=55$

Signing:

$s=55^3 \bmod 187 = 132$

Verification:

$m=132^{27} \bmod 187 = 55$

5. Security of RSA:

Four possible approaches to attacking the RSA algorithm are:

- Brute Force Attack
  - trying all possible private keys
  - The defense against this is to use a large key space, but then slower
- Mathematical Attacks (factoring n)
  - Factoring attacks used the quadratic sieve (QS), The recent attack on RSA-130 used the generalized number field sieve (GNFS). We can expect further refinements in GNFS, and the use of an even better algorithm, such as the special number field sieve (SNFS). It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better. see improving algorithms (QS, GNFS, SNFS)
  - Currently 1024-2048-bit keys seem secure
- Timing Attacks (on implementation)
  - These depend on the running time of the decryption algorithm.
  - use - constant time, random delays, blinding
- Chosen Ciphertext Attacks (on RSA props)
  - This type of attack exploits properties of the RSA algorithm

6. RSA Algorithm Implementation in Python:

This section demonstrates how the RSA encryption and decryption process can be implemented in Python. The code follows the same mathematical logic described earlier — including key generation, modular exponentiation, and text encryption/decryption

```
import math

# Function to find GCD (Greatest Common Divisor)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Function to find e (public exponent)
def find_e(phi):
    for e in range(2, phi):
        if gcd(e, phi) == 1:
            return e

# Function to find d (private exponent)
def find_d(e, phi):
    # Using the Extended Euclidean Algorithm
    d = pow(e, -1, phi)
    return d

# RSA encryption/decryption function
def rsa_encrypt_decrypt(key, msg, n):
    result = ""
    for ch in msg:
        m = ord(ch)
        c = pow(m, key, n)
        result += chr(c % 256)   # keeps result in readable ASCII range
    return result

# --------------------------------------------------------
# Step 1: Select two prime numbers
```

```
p = 17
q = 11

# Step 2: Compute modulus and totient
n = p * q
phi = (p - 1) * (q - 1)

# Step 3: Select e and compute d
e = find_e(phi)
d = find_d(e, phi)

print("Public Key (e, n): <", e, ",", n, ">")
print("Private Key (d, n): <", d, ",", n, ">")

# -----------------------------------------------------------
# Step 4: Input message and perform encryption/decryption
msg = input("Enter a short message: ")

# Encrypt message
cipher = rsa_encrypt_decrypt(e, msg, n)
print("\nCipher Text:", cipher)

# Decrypt message
plain = rsa_encrypt_decrypt(d, cipher, n)
print("Decrypted Text:", plain)

# -----------------------------------------------------------
# Validation
if plain == msg:
    print("\n RSA Encryption/Decryption Successful!")
else:
    print("\n Decryption Error - Check Implementation")
```

Explanation:

Step	**Operation**	**Purpose**
1	Choose primes p, q	Generate modulus n and totient ϕ(n)
2	Find e	Public key exponent coprime to ϕ(n)
3	Compute d	Private exponent using modular inverse
4	Encryption: c = m^e mod n	Converts plaintext into ciphertext
5	Decryption: m = c^d mod n	Restores original message

Output

```
Public Key (e, n): < 7 , 187 >
Private Key (d, n): < 23 , 187 >
Enter a short message: HI
Cipher Text: .9
Decrypted Text: HI
RSA Encryption/Decryption Successful!
```

7. Summary

The **RSA algorithm** is one of the most fundamental public-key encryption techniques used to ensure **data confidentiality and authentication**.
It operates on simple mathematical principles of **modular arithmetic** and the difficulty of **factoring large prime numbers**.
In this lecture, students learned the complete RSA process:

- How to generate keys using two prime numbers.
- How encryption and decryption are performed using modular exponentiation.

- How RSA also supports **digital signatures** for data integrity and authentication.
  The Python implementation demonstrated how theoretical RSA concepts can be applied programmatically to secure messages.

RSA remains essential in many modern security applications, such as **digital certificates, secure emails, and SSL/TLS protocols**.