Week 3: Public-Key Cryptography – RSA

What is RSA?

RSA is a **public-key encryption system**. It allows **secure communication** between people who have **never met before**, by using two keys:

- A **public key** (shared with everyone)
- A **private key** (kept secret)

You use the **public key to lock** a message, but **only the private key can unlock it**.

Why Two Keys?

- Anyone can **lock** a box and send it to you (using your public key).
- But only you can **unlock** it (because you have the private key).

How is RSA used?

RSA has two main uses:

1. **Encrypting**: To send private messages.
2. **Digital Signatures**: To prove a message really came from you.

Basic Idea Behind RSA

- RSA relies on the fact that some problems are **easy in one direction but hard in the other**.
- It's easy to **multiply** two numbers, even big ones.
- But it's very hard to **factor** the result back into the original two numbers if they're large enough (like hundreds of digits).
- RSA is built around this one-way trapdoor function.

Why is RSA considered secure?

- Even if everyone knows your public key, they can't figure out your private key unless they can break a very hard problem (factoring a huge number).
- As long as your private key is safe and the math problem stays hard, RSA works.

What makes RSA vulnerable?

RSA can be insecure if:

- Keys are too short (e.g., 512-bit keys are now breakable).
- Bad randomness is used when generating keys.

- RSA is used without proper padding (which protects against certain attacks).
- Computers get fast enough (or quantum) to break it.

Where is RSA used?

- **Web browsers** (HTTPS)
- **Email encryption**
- **Software signing**
- **Banking**
- **Cryptocurrencies** (sometimes)

MATH FOUNDATION

Prime, Coprime and Composite Numbers

A **prime number** is a natural number greater than 1 that has no positive divisors other than 1 and itself.

A **composite number** is any number that **has more than two factors**.

Two numbers are **coprime** if their greatest common divisor (GCD) is 1.

▸ In RSA:

You choose **two large prime numbers** to generate your keys.

In RSA, the number $n=p \times q$ (product of two primes) is a composite number that forms part of the public key.

We choose a number 'e' that is **coprime to ϕ(n)** to ensure it can have a modular inverse — essential for decrypting messages.

GCD (Greatest Common Divisor)

The **GCD** of two numbers is the largest number that divides both without leaving a remainder.

▸ RSA Relevance:

We check that the public exponent 'e' is coprime with $\phi(n)$ using GCD.

Modular Arithmetic

This is the heart of RSA. All encryption and decryption in RSA happens **modulo a number.** So understanding modular arithmetic is absolutely essential.

Why Modular Arithmetic Matters in RSA

RSA encrypts messages using: $C = m^e \bmod n$

And decrypts using: $M = c^d \bmod n$

So **everything depends on doing exponentiation modulo nnn**!

Modular Exponentiation

Fast Exponentiation (a.k.a. Binary Exponentiation)

This algorithm calculates a^b mod m **efficiently**, using squaring and reducing along the way.

C++ CODE :

```
int modexp (int a, int b, int m) {
   int result = 1;
   a = a % m;
   while (b > 0) {
      if (b % 2 == 1)
         result = (1LL * result * a) % m;
      a = (1LL * a * a) % m;
      b /= 2;
   }
   return result;
}
```

Euler's Totient Function ϕ(n)

Euler's Totient Function **ϕ(n)** counts how many numbers **less than n** are **coprime to n**.

If p is a **prime number :** $\phi(p) = p-1$

Euler's Totient for Product of Two Primes

This is **exactly what we need for RSA**, since $n=p\times q$

If p and q are **distinct primes**:

$$\phi(n) = \phi(p \cdot q) = (p-1)(q-1)$$

This works because of a special property: $\phi(a \cdot b)=\phi(a) \cdot \phi(b)$ when **a and b are coprime**

When generating RSA keys, we must choose a **public exponent e** such that:

$1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$

That is, e must be:

- **Greater than 1**
- **Less than $\phi(n)$**
- **Coprime to $\phi(n)$**

The number 65537 is very commonly used in practice

In RSA:

- You have public exponent e
- You've computed $\phi(n)$

Now you compute private key d as:

$$d \equiv e^{-1} \bmod \phi(n)$$

In other words, **d is the modular inverse of e mod $\phi(n)$**

RSA Key Generation — The Full Process

1. Choose Two Large Prime Numbers p and q

- Pick two distinct prime numbers, p and q
- For strong security, these are very large primes (hundreds of digits) — but for learning, small primes work fine!

2. Compute $n=p \times q$

- This n is part of the public and private keys
- It is called the **modulus**

3. Compute Euler's Totient $\phi(n)=(p-1)(q-1)$

- This number is crucial for key generation
- It represents the count of numbers coprime with n

4. Choose Public Exponent e

- $1 < e < \phi(n)$
- e must be coprime with $\phi(n)$
- Common choices: 65537, 17, or 3 (prefer 65537 in practice)

5. Compute Private Key d

- $d \equiv e^{-1} \bmod \phi(n)$
- Use the **modular inverse** to find d

6. Public Key = (e, n)

- This key is shared publicly — used to encrypt messages

7. Private Key = (d, n)

- This key is kept secret — used to decrypt messages

Example: Small Numbers for Clarity

Let's generate keys with small primes:

- $p=5, q=11$
- $n=5\times 11=55$
- $\phi(n)=(5-1)(11-1)=4\times 10=40$
- Choose e = 3 (check gcd(3, 40) = 1, so valid)
- Compute $d \equiv e^{-1} \bmod 40$

Using modular inverse:

- $3 \times d \equiv 1 \bmod 40$
- d = 27 (because $3\times 27=81\equiv 1 \bmod 40$)

Final Keys:

- **Public Key:** $(e,n)=(3,55)$
- **Private Key:** $(d,n)=(27,55)$

RSA ENCRYPTION AND DECRYPTION

Encrypting a Message

- You have a **plaintext message** mmm, where $0 \le m < n$
- Use the **public key** (e, n)
- Compute the **ciphertext** c: $c = m^e \bmod n$

This c is the encrypted message.

Decrypting the Message

- You have the **ciphertext** c
- Use the **private key** (d,n)
- Compute the original message m: $m = c^d \bmod n$

This recovers the original plaintext.

Why Does This Work?

Because of a mathematical property (Euler's theorem) that guarantees:

(m^e)^d ≡ m mod n

when e and d are chosen as modular inverses mod $\phi(n)$.

Let's walk through a tiny RSA example:

- Choose $p = 3$, $q = 11$
- So, $n = 33$, $\phi(n) = (3-1)(11-1) = 2 \times 10 = 20$
- Choose $e=3$ (must be coprime with 20)
- Find d such that $d \cdot e \equiv 1 \bmod 20$
- $d=7$, because $3 \cdot 7 = 21 \equiv 1 \bmod 20$

Now:

- **Public key**: $(e = 3, n = 33)$
- **Private key**: $(d = 7, n = 33)$

Encrypt message **m = 4**:

$c = 4^3 \bmod 33 = 64 \bmod 33 = 64 - 33 = 31$

So ciphertext is c=31

Now **decrypt:**

$m = 31^7 \bmod 33 = 27512614111 \bmod 33 = 4$

We got back the original message!

Real World

- You’d never use such small primes in real RSA — you’d use primes with **hundreds or thousands of bits**.

- Encryption/decryption in real systems uses **modular exponentiation** with optimized algorithms like **square-and-multiply** (also known as binary exponentiation).

SECURITY ASPECTS

The Hardness of Integer Factorization

RSA’s security is based on a **hard mathematical problem**:

➔ **Given n=p×q, it is very difficult to find p and q — the prime factors of n.**

This is known as the **Integer Factorization Problem (IFP)**.

Why is it hard?

- If n is a 2048-bit number (≈ 617 decimal digits), even the **fastest classical computers** would take **centuries** to factor it.
- The best known classical factoring algorithm is the **General Number Field Sieve (GNFS)**, and its time complexity grows **sub-exponentially** with the number of bits.

Implication:

- **If you can't factor n**, then you can't compute ϕ(n), so you **can't compute d** (the private key), which keeps the system secure.

Importance of Key Length

Key Length = Number of bits in n = p × q

Key Size (bits)	Status (as of 2025)	Notes
512	**Broken** in seconds	Factorable easily
1024	**Weak**	Can be cracked with effort
2048	**Standard**	Secure against classical attacks
3072+	**Stronger**	For future-proofing

Longer keys = More security, but **slower operations**

RSA with **2048 bits** is currently the chosen option between speed and security

Attacks on RSA

a. Mathematical Attacks

- **Factoring n:** Direct attack, as explained above.
- **Small e attack:** If e is too small (like 3) and the message is small, m^e < n, then no mod is applied → message is easily recovered.
- **Common modulus attack:** If two users share the same n but have different e, attackers can sometimes recover the message.

b. Timing Attacks

➔ If RSA decryption (modular exponentiation) takes different times depending on the input, attackers can measure time and infer bits of the key.

c. Chosen Ciphertext Attack (CCA)

➔ An attacker submits specially crafted ciphertexts and observes the decrypted output to infer information about the key or plaintext.

d. Side-channel Attacks

➔ Attacks using physical observations (timing, power usage, etc.) to extract secret key data.

What is Padding?

Padding means **adding extra data** to a message before encrypting it — usually to make it:

1. The **correct size** for encryption
2. **More secure** by adding **randomness** or structure

Why is Padding Needed in RSA?

- Raw RSA is **deterministic**:
  - Same message → same ciphertext every time!
- Also vulnerable to **chosen ciphertext attacks** and **structure-based attacks**
- Padding introduces **randomness** and **structure**.

PKCS#1 (Public-Key Cryptography Standards) (Legacy Standard)

- Adds non-zero random padding bytes before the message
- Still used, but has known vulnerabilities (Bleichenbacher attack)

The padded message (called **EM**, for *encoded message*) looks like this:

```
EM = 0x00 || 0x02 || PS || 0x00 || M
```

Part	Name	Description
00	Header byte	Always 0x00 — separates RSA padding from other types
02	Block type	"02" means this is for encryption (other values are used for signatures)
PS	Padding string	Random non-zero bytes, at least 8 bytes long
00	Separator	Marks end of padding, start of message
M	Message	The original plaintext message

Example Use

Assume:

- RSA key size: **1024 bits = 128 bytes**
- Message M: `"Hello"` (5 bytes)

Then:

- You need: **128 - 3 - 5 = 120 bytes of padding**
- The `PS` string is filled with **random, non-zero bytes** (e.g., `0x8A`, `0xF1`, etc.)

The structure becomes:

```
00 || 02 || [120 random bytes] || 00 || "Hello"
```

You encrypt this whole block using RSA: **c = (EM)^e mod n**

OAEP (Optimal Asymmetric Encryption Padding)

- **Modern, secure** padding scheme
- Combines the message with:
  - A **mask generation function (MGF)**
  - A **hash function** (e.g., SHA-256)
  - **Random seed**
- Provides **semantic security** — i.e., an attacker can't distinguish between ciphertexts of different messages.

Key Properties:

- **Prevents deterministic output**
- **Resists chosen ciphertext attacks**
- **Used in RSAES-OAEP (RSA Encryption Scheme)**

IMPLEMENTING RSA FROM SCRATCH

Step-by-Step Process

1. **Choose two large primes**: p and q
2. Compute **n = p × q**
3. Compute **ϕ(n) = (p−1)(q−1)**
4. Choose public exponent **e: usually 65537**
5. Compute private exponent d, such that **d × e ≡ 1 mod ϕ(n)**
6. Encryption: **c = m^e mod n**
7. Decryption: **m = c^d mod n**

Chinese Remainder Theorem (CRT) Optimization

RSA decryption is slow. But we can **speed it up 4x using CRT**.

CRT Decryption Formula:

Instead of computing: m = c^d mod n

We do:

- m1 = c^{d \mod (p-1)} \mod p
- m2 = c^{d \mod (q-1)} \mod q
- Combine results using CRT to get final m

Benefits:

- Works with smaller numbers (faster exponentiation)
- 3x – 5x speed-up in decryption

RSA in Libraries (Python Example)

Instead of building everything from scratch, use tested libraries.

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes

# Key generation
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()
# Encryption
ciphertext = public_key.encrypt(
    b"Hello Khushi!",
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
# Decryption
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
print(plaintext.decode())
```

RSA in the Real World

RSA is **everywhere**, especially in:

1. HTTPS (SSL/TLS)

- RSA is used to securely **exchange symmetric keys**
- The browser encrypts the session key using the server's **RSA public key**
- Only the server (with private key) can decrypt it

2. Digital Signatures

- A sender signs a hash of a message with their **private RSA key**
- Anyone can verify the signature using their **public key**
- Used in:
  - Emails (S/MIME)
  - Software updates
  - Blockchain transactions

3. SSH Keys

- You can generate RSA key pairs for logging into servers securely.

4. JWTs and Authentication

- Tokens are signed using RSA to verify authenticity

Week 3 : Resources Used - The Mathematics of Secrets Cryptography from Caesar Ciphers to Digital Encryption (Joshua Holden) | Introduction To Modern Cryptography (Jonathan Katz, Yehuda Lindell)