K.N.Toosi university of technology

A Brief Journey Through

# RSA ENCRYPTION & DECRYPTION

With Mathematical Analysis & Real-World Project

Submitted By:
- **Mahdi Jorati**
- **Mohammad Amin Abdullahi**

# Table of Contents

# 1. Introduction

## 1.1 What is RSA?

RSA (Rivest–Shamir–Adleman) is one of the most widely used public-key cryptographic systems in the world. Developed in 1977, RSA revolutionized digital security by introducing a method for secure communication that doesn't require the sender and receiver to share a secret key in advance.

In RSA-based cryptography, a user's *private key*—which can be used to sign messages, or decrypt messages sent to that user—is a pair of large random prime numbers chosen and kept secret. A user's *public key*—which can be used to verify messages from the user, or encrypt messages so that only that user can decrypt them—is the product of the prime numbers.

The security of RSA is paying off with the difficulty of factoring the product of two large prime numbers, also called "factoring problem". Breaking RSA encryption is known as the RSA problem. Whether it is as difficult as the factoring problem is an still-standing question. There are no published methods to defeat the system if a large enough key is used.

## 1.2 Origins of RSA Encryption

The origin of RSA returns to a system that was developed secretly in 1973 at Government Communications Headquarters (GCHQ), the British signal intelligence agency, by the English mathematician Clifford Cocks.

Later in 1977, Ron Rivest, Adi Shamir and Leonard Adleman, publicly described the algorithm. They also introduced digital signatures and attempted to apply number theory. However, they left open the problem of realizing a one-way function, possibly because the difficulty of factoring was not well-studied at the time.

Adi Shamir, co-inventor of RSA

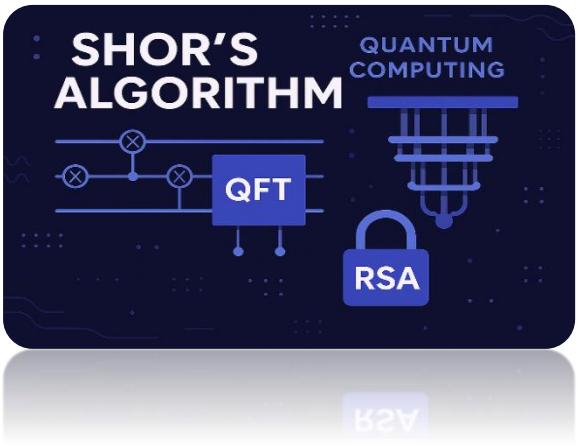
In addition, due to the significant computational resources required to implement RSA encryption during its early development, it was largely regarded as a theoretical curiosity rather than a practical solution. At the time, the necessary hardware was so expensive and far beyond the reach of most institutions, making real-world deployment virtually impossible. As a result, despite its groundbreaking potential, RSA remained confined to classified research environments.

## 1.3 RSA: Opportunities and threats

While still secure against traditional attacks with sufficient key lengths (2048 bits or more), its public-key encryption foundation faces future threats from quantum computing. Additionally, the complexity of secure RSA implementation makes it vulnerable to attacks like padding oracle attacks, leading to its deprecation in newer systems like TLS 1.3.

For its original purpose of secure data transmission, RSA with 2048-bit keys or longer is currently considered secure against conventional computing power.

Quantum computers, using algorithms like Shor's algorithm, could theoretically break RSA by factoring large numbers, making the algorithm obsolete. While no sufficiently powerful quantum computer exists yet, it's a recognized future threat.




The complexity of correctly implementing RSA, especially its padding scheme, creates a large attack surface. Errors in implementation, even if the underlying math is sound, can lead to vulnerabilities such as padding oracle attacks, which can compromise security.

# 2. Mathematics

## 2.1 Miller-Rabin Primality Test

The Miller–Rabin primality test or Rabin–Miller primality test is a probabilistic primality test, an algorithm which determines whether a given number is likely to be prime. For odd composite $n$ > 1, over 75% of numbers from to 2 to $n$ − 1 are witnesses in the Miller–Rabin test for $n$.

The property is the following. For a given odd integer $n$ > 2, let's write *n-1* as $2^s . d$ where *s* is a positive integer and *d* is an odd positive integer. Let's consider an integer *a*, called a *base*, which is coprime to $n$. Then, $n$ is said to be a strong probable prime to base *a* if one of these relations holds:

- $a^d \equiv 1 \ (mod \ n)$
- 2. $a^{2^r d} \equiv -1 \ (mod \ n)$ for some $0 \leq r < s$.

This simplifies to first checking for $a^d \ mod \ n = 1$ and then $a^{2^r d} = n - 1$ for successive values of $r$.The idea beneath this test is that when $n$ is an odd prime, it passes the test because of two facts:

- $a^{n-1} \equiv 1$
- the only square roots of 1 modulo $n$ are 1 and −1.

if $n$ is not a strong probable prime to base $a$, then $n$ is definitely composite, and $a$ is called a **witness** for the compositeness of $n$.

However, if $n$ is composite, it may nonetheless be a strong probable prime to base $a$, in which case it is called a **strong pseudoprime**, and $a$ is a **strong liar**.

## 2.2 Euler's Totient Function

In number theory, **Euler's totient function** counts the positive integers up to a given integer *n* that are relatively prime to $n$.

As we know from discrete math, Euler's product formula equals to: $\varphi(n) = n\prod_{p|n}(1-\frac{1}{p})$, where the product is over the distinct prime numbers dividing $n$.

An equivalent formulation is:

$$\varphi(n) = p_1^{k_1-1}(p_1-1)\,p_2^{k_2-1}(p_2-1)\cdots p_r^{k_r-1}(p_r-1)$$

Where $n = p_1^{k_1}p_2^{k_2}\cdots p_r^{k_r}$ is the prime factorization of $n$. So, if $p$ is prime and $k \geq 1$, then:

$$\varphi\left(p^k\right) = p^k - p^{k-1} = p^{k-1}(p-1) = p^k\left(1-\tfrac{1}{p}\right).$$

Since $p$ is a prime number, the only possible values of $\gcd(p^k, m)$ are $1, p, p^2, \ldots, p^k$, and the only way to have

$\gcd(p^k, m) > 1$ is if $m$ is multiply of $p$, that means $m \in \{p, 2p, 3p, \dots, p^{k-1}p = p^k\}$, and there are $p^{k-1}$ such multiplies not greater than $p^k$.In result, the other $p^k - p^{k-1}$ numbers are all relatively prime to $p^k$.

## 2.3 Extended Euclidean Function

The Euclidean algorithm is used to find the greatest common divisor (gcd) of two integers $a$ and $b$:

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

This repeats until the remainder is zero.

### 2.3.1 Example2.1

$\gcd(99, 78)$

$$99 = 78 \cdot 1 + 21 \Rightarrow \gcd(99,78) = \gcd(78,21)$$

$$78 = 21 \cdot 3 + 15 \Rightarrow \gcd(78,21) = \gcd(21,15)$$

$$21 = 15 \cdot 1 + 6 \Rightarrow \gcd(21,15) = \gcd(15,6)$$

$$15 = 6 \cdot 2 + 3 \Rightarrow \gcd(15,6) = \gcd(6,3)$$

$$6 = 3 \cdot 2 + 0 \Rightarrow gcd(6,3) = 3$$

So $\gcd(99, 78) = 3$

The extended version does more:
It not only finds $\gcd(a, b)$,but also finds integers $s, t$ such that:

$$at + bs = \gcd(a, b)$$

From division formula we know:

$$a = bq + r \Rightarrow r = a - bq$$

If $gcd(b, r) = d$, then there exist integers $t_1$ , $s_1$ such that:

$$bs_1 + rt_1 = d$$

Replace $r = a - bq$:

$$bs_1 + (a - bq)t_1 = d$$

$$at_1 + b(s_1 - qt_1) = d$$

This recursion lets us compute $s, t$ back up the chain.

### 2.3.2 Example2.2

We already saw $gcd\ (99,78) = 3$. Now let's find $t, s$:

1. $99 = 78 \cdot 1 + 21 \Rightarrow 21 = 99 - 78 \cdot 1$
2. $78 = 21 \cdot 3 + 15 \Rightarrow 15 = 78 - 21 \cdot 3$
3. $21 = 15 \cdot 1 + 6 \Rightarrow 6 = 21 - 15 \cdot 1$
4. $15 = 6 \cdot 2 + 3 \Rightarrow 3 = 15 - 6 \cdot 2$
   - From (4): $3 = 15 - 6 \cdot 2$
   - Replace 6 from (3): $6 = 21 - 15$

     $$3 = 15 - (21 - 15) \cdot 2 = 15 \cdot 3 - 21 \cdot 2$$

   - Replace 15 from (2): $15 = 78 - 21 \cdot 3$

     $$3 = (78 - 21 \cdot 3) \cdot 3 - 21 \cdot 2 = 78 \cdot 3 - 21 \cdot 11$$

   - Replace 21 from (1): $21 = 99 - 78$

     $$3 = 78 \cdot 3 - (99 - 78) \cdot 11 = 78 \cdot 14 - 99 \cdot 11$$

   - Final result:

     $$3 = (-11) \cdot 99 + (14) \cdot 78$$

     So: $t = -11, s = 14$

## 2.4 Modular Inverse

For integers $a$ and $m$, the modular inverse of $a\ (mod\ m)$ is an integer $t$ such that:

$$a \cdot t \equiv 1\ (mod\ m)$$

This only exists if:

$$\gcd(\mathrm{a}, \mathrm{m}) = 1$$

From the Extended Euclidean Algorithm, we know:

$$a.t + m.s = gcd\ (a, m)$$

$gcd(a, m) = 1$, then:

$$a.t + m.s = 1$$

Taking this equation $(mod\ m)$:

$$a.t \equiv 1(mod\ m)$$

So, the coefficient $t$ is the modular inverse of $a\ (mod\ m)$.

### 2.4.1 Example2.3

We want: $3x \equiv 1(mod 26)$

Step 1: Run Extended Euclidean Algorithm

- $26 = 3 \cdot 8 + 2 \Rightarrow 2 = 26 - 3 \cdot 8$
- $3 = 2 \cdot 1 + 1 \Rightarrow 1 = 3 - 2 \cdot 1$

Step 2: Back-substitute

- From 2nd equation: $1 = 3 - 2 \cdot 1$
- Replace $2 = 26 - 3 \cdot 8$:

$1 = 3 - (26 - 3 \cdot 8) \cdot 1 = 3 + 3 \cdot 8 - 26$

$1 = 3 \cdot 9 - 26 \cdot 1$

Step 3: Interpret Result

We have: $1 = 3 \cdot 9 + 26 \cdot (-1)$

So: $3 \cdot 9 \equiv 1(mod 26)$

The modular inverse is: $3^{-1} \equiv 9(mod 26)$

# 3. Real World Project

## 3.1 Step one: Generating prime numbers

**Goal:** pick two large primes $p$ and $q$ using the Miller–Rabin test, so RSA remains secure.

**Where in code:** $gen\_prime(bits)$, $miller\_rabin(n, k = 40)$, and the $single\_test(n, a)$.

**How it works** (tied to 2.1 Miller–Rabin):

1. For a candidate odd $n$, write $n - 1 = 2^s \cdot d$ with $d$ odd. The code does this by repeatedly halving $n - 1$ to count the factor of two (d and s). This is exactly the decomposition used by Miller–Rabin.
2. Pick a random base $a \in [2, n - 2]$
   (a = self.rand.range(2, n -1)). Compute $x \equiv a^d \ mod(n)$.
   If $x \ \in \{1, n - 1\}$, the round passes. Otherwise square up to $s - 1$ times. If any $x \equiv -1 \ mod(n)$, the round passes; if not, $n$ is composite (the base $a$ is a *witness*). This matches discussion we talked about: strong probable prime or strong liar.

```python
def single_test(self, n, a):
    # First, finding 'd' and 's' such n - 1 = 2^s * d
    d = n - 1
    s = 0
    while d % 2 == 0:
        s += 1
        d >>= 1
    # Now 'd' is the odd part of n-1

    # Calculate x = a^d mod n.
    x = pow(a, d, n)

    # If x is 1 or n-1, the number passes the test for this 'a'.
    if x == 1 or x == n - 1:
        return True
    for i in range(s - 1):
        x = pow(x, 2, n)
        if x == n - 1:
            return True
    return False
```

```python
def miller_rabin(self, n, k=40):

    # Handling numbers lower than 4.
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0:
        return False

    # Applying the test 'k' times(usually 40).
    for i in range(k):
        a = self.rand.randrange(2, n - 1)
        if not self.single_test(n, a):
            return False
    return True
```

3. We repeat this process for 40 times. As the chance of a single Miller-Rabin test fail is ¼, this lowers the error possibility to $2^{-80}$!
4. One of the fastest way that we found for finding large prime number, was to use random.getrandbits(bits). This function randomly generates a bits-length zeros and ones. As the bits are chosen randomly, it's possible that the highest bit be zero. Thus, we set it to one manually. For making sure our number is odd, we also have to set the first bit to 1.

```python
def gen_prime(self, bits):
    while True:
        # Generating "bits" number of digits, randomly chosen between 1 and 0.
        a = random.getrandbits(bits)

        # Making sure 'a' has its highest bit set to 1.
        a |= (1 << bits - 1)

        # Making the least significant bit 1.
        # Ensures owr number is odd.
        a |= 1
        if self.miller_rabin(a):
            return a
```

## 3.2 Step two: Computing $n$ and Euler's totient $\varphi(n)$

**Goal:** build the RSA modulus $n = pq$ and $\varphi(n) = (p-1)(q-1)$ (since $p, q$ are prime).

**Where in code:** $calc\_n(p, q)$ and $calc\_phi\_n(p, q)$.

**Math link to** 2.2 Euler's Totient Function**:** Our 2.2 presents Euler's product formula and the special case for primes. Because $n = pq$ with distinct primes, $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$). The code applies exactly this identity.

```python
# Calculate n (used for the public and private keys)
def calc_n(self, p, q):
    return p * q

# Calculating phi_n(n)
# Number of "numbers" that are coprime with n.
def calc_phi_n(self, p, q):
    return (p - 1) * (q - 1)
```

## 3.3 Step three: Choosing the public exponent $e$

**Goal:** select a small, fixed public exponent coprime to $\varphi(n)$, typically $e = 65537$, to balance speed and security.

**Where in code:** $find\_e(phi\_n)$ and helper $gcd(a, b)$.

**How it works**:

1. Start with $e = 65537$ (as number: $2^{16} - 1$, as bits: 1000...0001) a fast exponentiation and widely used in practice. The code first checks $gcd\ (e, \varphi(n)) = 1$.

```
# We call public exponent in RSA encryption 'e'.
# 'e' must satisfy two conditions: (1 < e < phi_n) & (being coprime with phi_n).
# The number 65537 is a common choice for 'e'. More details in PDF.
def find_e(self, phi_n):
    e = 65537
    if self.gcd(e, phi_n) == 1:
        return e

    # Making sure e is coprime with phi_n.
    while self.gcd(e, phi_n) != 1:
        e += 2
    return e
```

2. If $gcd\ \neq 1$, increment by 2 until coprime (staying odd). This guarantees existence of a modular inverse later.

## 3.4 Step four: Computing the private exponent $d$

**Goal:** compute $d \equiv e^{-1}(mod\varphi(n))$ using the Extended Euclidean Algorithm.

**Where in code:** $extended\_Euclid\_gcd(a, b)$ and $modinv(a, m)$

How it works (mathematically ties to 2.3 & 2.4):

1. The extended Euclidean algorithm finds integers $s, t$ such that $as + bt = gcd\ (a, b)$. Our 2.3 builds this identity by

back-substitution. The code computes it iteratively with (old_r, r), (old_s, s), (old_t, t) updates.

```python
# Extended Euclidean Algorithm
def extended_Euclid_gcd(self, a, b):
    # Initializing the variables
    old_r, r = a, b
    old_s, s = 1, 0
    old_t, t = 0, 1

    # Loop until remainder becomes zero
    while r != 0:
        quotient = old_r // r

        # Update remainders
        old_r, r = r, old_r - quotient * r

        # Update s coefficients
        old_s, s = s, old_s - quotient * s

        # Update t coefficients
        old_t, t = t, old_t - quotient * t

    # Final result will be: gcd = old_r, coefficients = old_s, old_t
    return old_r, old_s, old_t
```

2. For the modular inverse, we set $a = e, m = \varphi(n)$. If $gcd\ (e, \varphi(n)) = 1$, the coefficient $s$ satisfies $es + \varphi(n) \cdot t = 1$ so $es \equiv 1(mod\varphi(n))$ and $d \equiv s\ (mod\,\varphi(n))$. The code returns s % m and raises an exception if the gcd is not 1 (no inverse). This matches our 2.4 derivation and Example 2.3.

```python
def modinv(self, a, m):
    # Calculating the modular inverse of a mod m.
    # This finds 's' such that (a * s) % m = 1.
    g, s, t = self.extended_Euclid_gcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    return s % m
```

## 3.5 Step five: Encryption & decryption

**Goal:** implement RSA maps $c \equiv m^e (mod n)$ & $m \equiv c^d (mod n)$.

**Where in code:** $encrypt(plaintext: str)$,
$decrypt(ciphertext: int)$.

**Message ↔ integer:** The code encodes a string using UTF-8 to get $m$. It requires $0 \leq m < n$. After decryption, it converts the integer back to bytes and decodes UTF-8.

**Core operations:** Encryption is pow(msg_int, self.e, self.n), which means, $c = m^e \ mod \ n$; decryption is pow(ciphertext, self.d, self.n),which also means, $m = c^d mod \ n$. We handled fast modular exponentiation by Python's built-in pow.

```python
def encrypt(self, plaintext: str):
    # Using UTF-8 character encoding standard.
    msg_int = int.from_bytes(plaintext.encode('utf-8'), 'big')
    cipher_int = pow(msg_int, self.e, self.n)
    return cipher_int

def decrypt(self, ciphertext: int):
    msg_int = pow(ciphertext, self.d, self.n)
    length = (msg_int.bit_length() + 7) // 8
    return msg_int.to_bytes(length, 'big').decode('utf-8')
```