Asymmetric Cryptography – RSA Deep Dive

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 7

Press Space for next page →

RSA Refresher and Overview

Core Idea

- Key pair: **Public (n, e)** and **Private (n, d)**
- Security: Hardness of factoring $n = p \times q$
- Operations: $c = m^e \bmod n$; $m = c^d \bmod n$

Parameters

- $n = p \times q$ (1024/2048/3072/4096-bit)
- e: public exponent (use 65537)
- d: modular inverse of e mod $\varphi(n)$

Where We Go Deeper Today

- Number theory foundations
- Secure key generation
- Padding (OAEP/PSS)
- Attacks and defenses

Performance

- Use **CRT** to speed up decryption/signing
- Avoid side-channel leaks (constant-time code)

Number Theory for RSA

Modular Arithmetic Essentials

Concepts

- $\varphi(n) = (p-1)(q-1)$
- Modular inverse: $d \equiv e^{-1} \pmod{\varphi(n)}$
- Euler's theorem: $a^{\varphi(n)} \equiv 1 \pmod{n}$ for $\gcd(a, n) = 1$

```python
from math import gcd

def egcd(a: int, b: int):
    if b == 0:
        return (a, 1, 0)
    g, x1, y1 = egcd(b, a % b)
    return (g, y1, x1 - (a // b) * y1)

def modinv(a: int, m: int) -> int:
    g, x, _ = egcd(a, m)
    if g != 1:
        raise ValueError('inverse does not exist')
    return x % m
```

Secure Key Generation

Steps

1. Generate random large primes p, q
2. $n = p \times q$; $\varphi(n) = (p-1)(q-1)$
3. Choose e = 65537
4. Compute d = e^{-1} mod φ(n)
5. Validate (gcd(e, φ(n)) = 1, key tests)

Pitfalls

- p and q too close → Fermat factorization
- Reused primes → catastrophic compromise
- Weak RNG → predictable keys
- Small d (Wiener's attack)

Toy Implementation (Educational)

```python
import secrets

def generate_prime(bits: int) → int:
    # Placeholder primality; use robust tests (e.g., Mi
    def is_probable_prime(n: int) → bool:
        if n % 2 == 0: return False
        # ... omitted: implement Miller–Rabin rounds ..
        return True
    while True:
        candidate = secrets.randbits(bits) | 1 | (1 <<
        if is_probable_prime(candidate):
            return candidate

def generate_rsa(bits: int = 2048):
    p = generate_prime(bits // 2)
```

Best Practice

- Use vetted libraries for production keygen

Padding and Safe RSA

Why Padding?

- Textbook RSA is deterministic and malleable
- Enables chosen-ciphertext attacks
- Use standardized padding: **OAEP** (encryption), **PSS** (signatures)

Encryption (OAEP)

- Randomized mask generation (MGF1)
- Semantic security under RSA assumption

Python `cryptography` Example

```
from cryptography.hazmat.primitives.asymmetric import r
from cryptography.hazmat.primitives import hashes

private_key = rsa.generate_private_key(public_exponent=
public_key = private_key.public_key()

ciphertext = public_key.encrypt(
    b"secret",
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None,
    ),
)
```

Signatures (PSS)

- Probabilistic padding; mitigates forgery structures

Attacks and Defenses

Classical Attacks

- **Small e broadcast (Håstad)**: same message, different moduli
- **Padding oracles (Bleichenbacher)**: PKCS#1 v1.5
- **Wiener's attack**: small private exponent d
- **Fault/side-channel**: timing, power, cache

Mitigations

- Use OAEP/PSS; avoid PKCS#1 v1.5
- Constant-time implementations and blinding
- Robust key sizes (≥2048-bit)
- Side-channel hardening

RSA Blinding (Concept)

- Multiply ciphertext by r^e mod n before decrypt
- Remove r after exponentiation
- Breaks timing correlation with input

Do/Don't

- Do: authenticate ciphertexts (KEM + DEM)
- Don't: encrypt raw data with textbook RSA

Practical RSA (Toy)

Minimal Demo (Educational Only)

```
def rsa_encrypt_int(m: int, pub: tuple[int,int]) -> int:
    n, e = pub
    if m >= n:
        raise ValueError('message too large')
    return pow(m, e, n)

def rsa_decrypt_int(c: int, priv: tuple[int,int]) -> int
    n, d = priv
    return pow(c, d, n)
```

🎯 Student Task: Modular Inverse and CRT

Task A: Modular Inverse

Given $e = 17$ and $\varphi(n) = 3120$, compute d such that $e \cdot d \equiv 1 \pmod{3120}$.

Task B: CRT Speedup

Given $p = 61$, $q = 53$, $n = 3233$, $d = 2753$, and $c = 2790$, compute m using CRT steps (dp, dq, qinv).

Deliverables

- Value of d
- Step-by-step CRT recombination

✅ Solution Sketch

Task A

- d = 2753 (since 17 × 2753 = 46801 ≡ 1 (mod 3120))

Task B (Outline)

1. dp = d mod (p-1) = 2753 mod 60 = 53
2. dq = d mod (q-1) = 2753 mod 52 = 49
3. qinv = q^{-1} mod p = 53^{-1} mod 61 = 38
4. m1 = c^{dp} mod p; m2 = c^{dq} mod q
5. h = (qinv × (m1 - m2)) mod p
6. m = m2 + h × q (mod n)

Real-World RSA

Usage Patterns

- TLS key exchange (historically); modern TLS prefers ECDHE
- Code signing and package signing
- Document signing (PDF, XMLDSig)
- PKI and certificates (X.509)

Best Practices Recap

- Use 2048–3072-bit keys (or ECC alternative)
- OAEP for encryption, PSS for signatures
- e = 65537; enforce key validation
- Prefer ECDHE for key exchange, RSA for signatures

Questions?

Let's discuss RSA in depth! 💬

Next Week: We'll study key exchange protocols in practice (DH, ECDH, authenticated key exchange).

Assignment: Implement RSA with OAEP/PSS using a standard library, and measure CRT speedups.