Public Key Cryptography and Asymmetric Encryption

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 5

Press Space for next page →

What is Public Key Cryptography?

Definition

Public key cryptography uses two different but mathematically related keys - a public key and a private key.

Key Characteristics

- **Asymmetric** - Different keys for encryption/decryption
- **Public key** - Can be shared openly
- **Private key** - Must be kept secret
- **Mathematical relationship** - Keys are related but one cannot be derived from the other

How It Works

- **Encryption** - Use recipient's public key
- **Decryption** - Use your own private key
- **Digital signatures** - Sign with private key, verify with public key
- **Key exchange** - Establish shared secrets securely

Advantages

- **Solves key distribution** - No need to share secret keys
- **Digital signatures** - Authentication and non-repudiation
- **Scalable** - Works with many users
- **Enables secure communication** - Even with strangers

Revolutionary: Public key cryptography solved the key distribution problem that had plagued cryptography for centuries!

The Key Distribution Problem

Symmetric Cryptography Problem

- **Both parties** need the same secret key
- **How to share** the key securely?
- **Chicken and egg** - Need secure channel to share key
- **Doesn't scale** - n(n-1)/2 keys for n users

Public Key Solution

- **Publish public keys** - Put them on websites, directories
- **Keep private keys** - Never share them
- **Anyone can encrypt** - Using your public key
- **Only you can decrypt** - Using your private key

Example: 1000 Users

- **Symmetric:** Need 499,500 different keys
- **Public key:** Each user has 1 key pair = 1000 keys total
- **Key management** nightmare vs. simple solution

Real-World Analogy

- **Public key** = Your mailbox (anyone can put mail in)
- **Private key** = Your mailbox key (only you can open it)
- **Encryption** = Putting mail in someone's mailbox
- **Decryption** = Opening your own mailbox

RSA Algorithm

RSA: The First Public Key Algorithm

History

- **1977** - Rivest, Shamir, Adleman (RSA)
- **First practical** public key system
- **Based on** integer factorization
- **Still widely used** today

Key Generation

1. **Choose two primes** p and q
2. **Calculate n** = p × q
3. **Calculate φ(n)** = (p-1)(q-1)
4. **Choose e** such that gcd(e, φ(n)) = 1
5. **Calculate d** such that e × d ≡ 1 (mod φ(n))

Keys

- **Public key:** (n, e)
- **Private key:** (n, d)
- **n:** Modulus (product of two primes)
- **e:** Public exponent (usually 65537)
- **d:** Private exponent (calculated)

Security

- **Based on** difficulty of factoring large numbers
- **Breaking RSA** = Factoring n into p and q
- **Current recommendation:** 2048-bit keys minimum
- **Quantum threat:** Shor's algorithm can break RSA

RSA Encryption and Decryption

Encryption

Formula: c = m^e (mod n)

- **m:** Plaintext message (as integer)
- **e:** Public exponent
- **n:** Modulus
- **c:** Ciphertext

Decryption

Formula: m = c^d (mod n)

- **c:** Ciphertext
- **d:** Private exponent
- **n:** Modulus
- **m:** Original plaintext

Example

```
p = 61, q = 53
n = 61 × 53 = 3233
φ(n) = 60 × 52 = 3120
e = 17 (chosen)
d = 2753 (calculated)

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

Verification

```
Decrypt: m = 2790^2753 mod 3233 = 65 ✓
```

Implementation

```python
def rsa_encrypt(message, public_key):
    n, e = public_key
    return pow(message, e, n)

def rsa_decrypt(ciphertext, private_key):
```

🎯 Student Task: RSA Key Generation

Task: Generate RSA Keys

Given:

- $p = 7, q = 11$
- $e = 3$

Your Task:

1. Calculate $n = p \times q$
2. Calculate $\varphi(n) = (p-1)(q-1)$
3. Verify that $\gcd(e, \varphi(n)) = 1$
4. Calculate d such that $e \times d \equiv 1 \pmod{\varphi(n)}$
5. What are the public and private keys?

Hint: Use extended Euclidean algorithm for step 4

Work through this step by step!

✅ Solution: RSA Key Generation

Step-by-Step Solution

Step 1: $n = p \times q = 7 \times 11 = 77$

Step 2: $\varphi(n) = (p-1)(q-1) = 6 \times 10 = 60$

Step 3: $\gcd(3, 60) = 3 \neq 1$ ❌

Problem: e = 3 is not coprime with $\varphi(n) = 60$

Solution: Choose a different e

Let e = 7: $\gcd(7, 60) = 1$ ✓

Step 4: Find d such that $7 \times d \equiv 1 \pmod{60}$

Using extended Euclidean algorithm: d = 43

Verification: $7 \times 43 = 301 \equiv 1 \pmod{60}$ ✓

Keys:

Elliptic Curve Cryptography

What are Elliptic Curves?

Mathematical Definition

General form: $y^2 = x^3 + ax + b$

- **a, b:** Curve parameters
- **Points (x,y)** that satisfy the equation
- **Special point O** - point at infinity
- **Group operation** - point addition

Example: $y^2 = x^3 - x + 1$

```
Points on curve:
(0,1), (0,-1), (1,1), (1,-1), (2,2.65), (2,-2.65), ...
```

Why Elliptic Curves?

- **Smaller keys** - 256-bit ECC = 3072-bit RSA security
- **Faster operations** - More efficient than RSA
- **Less memory** - Smaller key storage
- **Mobile friendly** - Better for constrained devices

Security

- **Based on** discrete logarithm problem
- **Harder to break** than RSA for same security level
- **Quantum resistant** - No known quantum algorithm
- **Future-proof** - Will remain secure longer

Elliptic Curve Operations

Point Addition

Rule: P + Q = R

- **P, Q:** Points on curve
- **R:** Result point
- **Geometric:** Draw line through P and Q, find third intersection
- **Algebraic:** Use formulas for coordinates

Scalar Multiplication

Rule: kP = P + P + … + P (k times)

- **k:** Integer (scalar)
- **P:** Point on curve
- **Result:** Another point on curve
- **Used for** key generation and encryption

Point Doubling

Rule: 2P = P + P

- **P:** Point on curve
- **Geometric:** Draw tangent at P, find second intersection
- **Special case** of point addition

Implementation

```
def point_add(P, Q, a, p):
    """Add two points on elliptic curve"""
    if P == O: return Q
    if Q == O: return P
    if P == Q: return point_double(P, a, p)

    x1, y1 = P
    x2, y2 = Q
```

Digital Signatures

What are Digital Signatures?

Purpose

- **Authentication** - Prove who sent the message
- **Integrity** - Ensure message wasn't modified
- **Non-repudiation** - Sender cannot deny sending
- **Legal validity** - Equivalent to handwritten signature

Properties

- **Unforgeable** - Only private key holder can sign
- **Verifiable** - Anyone can verify with public key
- **Non-reusable** - Each message has unique signature
- **Non-deniable** - Cannot claim signature is fake

How They Work

1. **Hash the message** - Create message digest
2. **Sign the hash** - Use private key
3. **Send message + signature** - Both together
4. **Verify signature** - Use public key

Real-World Uses

- **Software distribution** - Verify authenticity
- **Email security** - PGP, S/MIME
- **Blockchain** - Transaction authentication
- **Legal documents** - Electronic contracts

RSA Digital Signatures

Signing Process

1. **Hash message:** h = H(m)
2. **Sign hash:** s = h^d (mod n)
3. **Send:** (message, signature)

Verification Process

1. **Hash message:** h = H(m)
2. **Verify signature:** h′ = s^e (mod n)
3. **Compare:** h == h′ ✓

Implementation

```python
import hashlib

def rsa_sign(message, private_key):
    """Sign message using RSA"""
    n, d = private_key

    # Hash the message
    message_hash = int(hashlib.sha256(message).hexdiges

    # Sign the hash
    signature = pow(message_hash, d, n)

    return signature

def rsa_verify(message, signature, public_key):
```

Key Exchange Protocols

Diffie-Hellman Key Exchange

The Problem

- **Two parties** want to establish shared secret
- **No secure channel** available
- **Eavesdropper** can see all communication
- **Solution:** Use public key cryptography

How It Works

1. **Alice and Bob** agree on public parameters (p, g)
2. **Alice** chooses private key a, sends g^a (mod p)
3. **Bob** chooses private key b, sends g^b (mod p)
4. **Both calculate** shared secret: g^(ab) (mod p)

Security

- **Eavesdropper** sees g^a and g^b
- **Cannot calculate** g^(ab) without a or b
- **Based on** discrete logarithm problem
- **Man-in-the-middle** attacks possible

Example

```
p = 23, g = 5
Alice: a = 6, sends 5^6 mod 23 = 8
Bob: b = 15, sends 5^15 mod 23 = 19
Shared secret: 8^15 mod 23 = 19^6 mod 23 = 2
```

Implementation

```python
def diffie_hellman(p, g, private_key):
    """Diffie-Hellman key exchange"""
    return pow(g, private_key, p)
```

Elliptic Curve Diffie-Hellman (ECDH)

Why ECDH?

- **Smaller keys** - 256-bit ECC = 3072-bit RSA
- **Faster** - More efficient than regular DH
- **Mobile friendly** - Better for constrained devices
- **Same security** - Equivalent to DH but smaller

How It Works

1. **Agree on curve** and base point G
2. **Alice** chooses private key a, sends aG
3. **Bob** chooses private key b, sends bG
4. **Shared secret** = abG = baG

Implementation

```
def ecdh_key_exchange(private_key, other_public, curve
    """ECDH key exchange"""
    # Calculate shared secret
    shared_point = scalar_multiply(private_key, other_p

    # Extract x-coordinate as shared secret
    shared_secret = shared_point[0]

    return shared_secret

def ecdh_public_key(private_key, base_point, curve):
    """Generate ECDH public key"""
    return scalar_multiply(private_key, base_point, cur
```

Practical Implementation

Complete RSA Implementation

Key Generation

```python
import random
from math import gcd

def generate_rsa_keys(bit_length=2048):
    """Generate RSA key pair"""
    # Generate two large primes
    p = generate_prime(bit_length // 2)
    q = generate_prime(bit_length // 2)

    # Calculate n and φ(n)
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Choose public exponent
    e = 65537  # Common choice
```

Encryption/Decryption

```python
def rsa_encrypt(message, public_key):
    """RSA encryption"""
    n, e = public_key

    # Convert message to integer
    if isinstance(message, str):
        message = message.encode()

    # Convert to integer
    m = int.from_bytes(message, 'big')

    # Encrypt
    c = pow(m, e, n)

    return c
```

Security Considerations

RSA Security

- **Key size** - Use at least 2048 bits
- **Prime generation** - Use cryptographically secure random
- **Padding** - Use OAEP padding, not PKCS#1 v1.5
- **Timing attacks** - Use constant-time implementations

Best Practices

- **Use established libraries** - Don't implement from scratch
- **Keep keys secure** - Store private keys safely
- **Rotate keys** - Change keys regularly
- **Use hybrid systems** - RSA for key exchange, AES for data

Common Attacks

- **Factorization** - Breaking n into p and q
- **Timing attacks** - Measuring execution time
- **Side-channel** - Power analysis, cache attacks
- **Padding oracle** - Exploiting padding errors

Real-World Usage

- **TLS/SSL** - Web security
- **Email encryption** - PGP, S/MIME
- **Digital certificates** - PKI infrastructure
- **Blockchain** - Bitcoin, Ethereum

🎯 Student Task: Implement RSA

Task: Complete RSA Implementation

Requirements:

1. **Generate RSA keys** (small primes for testing)
2. **Implement encryption/decryption**
3. **Add digital signature** functionality
4. **Test with sample messages**
5. **Handle edge cases** (message too large, etc.)

Bonus:

- Add padding schemes
- Implement key validation
- Add performance testing
- Create interactive demo

Focus on understanding the mathematics behind RSA!

Common Vulnerabilities

Implementation Errors

- ❌ **Weak random** - Predictable primes
- ❌ **Small key sizes** - Easy to factor
- ❌ **No padding** - Vulnerable to attacks
- ❌ **Timing attacks** - Leak information

Best Practices

- ✅ **Use established libraries** - Tested implementations
- ✅ **Proper key sizes** - 2048+ bits for RSA
- ✅ **Secure random** - Cryptographically secure PRNG
- ✅ **Constant time** - Prevent timing attacks

Protocol Issues

- ❌ **Key reuse** - Same key for different purposes
- ❌ **Weak parameters** - Small primes, bad curves
- ❌ **No authentication** - Man-in-the-middle attacks
- ❌ **Side channels** - Power, timing, cache attacks

Modern Alternatives

- **Elliptic curves** - Smaller keys, faster
- **Post-quantum** - Quantum-resistant algorithms
- **Hybrid systems** - Best of both worlds
- **Hardware security** - HSM, TPM

Real-World Applications

Web Security

- **HTTPS** - TLS/SSL certificates
- **Email security** - S/MIME, PGP
- **VPN** - IPsec, OpenVPN
- **SSH** - Secure shell connections

Enterprise

- **PKI** - Public Key Infrastructure
- **Digital certificates** - X.509 standard
- **Code signing** - Software authenticity
- **Document signing** - PDF, Office documents

Blockchain

- **Bitcoin** - ECDSA signatures
- **Ethereum** - ECDSA + ECDH
- **Smart contracts** - Cryptographic verification
- **Digital wallets** - Key management

Future Trends

- **Post-quantum crypto** - Quantum-resistant algorithms
- **Homomorphic encryption** - Compute on encrypted data
- **Zero-knowledge proofs** - Prove without revealing
- **Multi-party computation** - Secure collaborative computing

Questions?

Let's discuss public key cryptography! 💬

Next Week: We'll explore hash functions and learn about SHA, MD5, and their applications!

Assignment: Implement RSA and ECDH to understand asymmetric cryptography!