Introduction to Cryptography

MAT354 - Cryptography Course

Instructor: Adil Akhmetov
University: SDU
Week 1

Press Space for next page →

What is Cryptography?

Definition

Cryptography is the practice and study of techniques for secure communication in the presence of adversarial behavior.

Key Concepts

- **Confidentiality** - Only authorized parties can read
- **Integrity** - Data hasn't been tampered with
- **Authentication** - Verify identity of parties
- **Non-repudiation** - Cannot deny sending/receiving

Why Programmers Need It?

- **Web Security** - HTTPS, authentication
- **Data Protection** - Encrypt sensitive data
- **API Security** - Secure communication
- **Mobile Apps** - Protect user data
- **Blockchain** - Digital signatures, hashing

The CIA Triad

🔒 Confidentiality

Information is accessible only to authorized parties

- Data encryption
- Access controls
- User authentication

✅ Integrity

Information remains accurate and unmodified

- Hash functions
- Digital signatures
- Checksums

🚀 Availability

Information is accessible when needed

- Redundancy
- Backup systems
- DDoS protection

Historical Context

Ancient Times

- **Caesar Cipher** (100 BC) - Julius Caesar
- **Scytale** - Spartan military communication
- **Atbash Cipher** - Hebrew alphabet substitution

Middle Ages

- **Vigenère Cipher** (1553) - Polyalphabetic substitution
- **Frequency Analysis** - Al-Kindi’s breakthrough
- **Steganography** - Hidden messages

Modern Era

- **Enigma Machine** (WWII) - German encryption
- **DES** (1977) - First widely used standard
- **RSA** (1977) - Public key cryptography
- **AES** (2001) - Current standard

Fun Fact: The word "cryptography" comes from Greek: "kryptos" (hidden) + "graphein" (to write)

Real-World Examples

Everyday Applications

- 🔐 **Password Storage** - Hashed, not plain text
- 💳 **Online Banking** - TLS/SSL encryption
- 📱 **Mobile Payments** - Apple Pay, Google Pay
- 🔒 **File Encryption** - BitLocker, FileVault

Enterprise Systems

- 🌐 **VPN Connections** - Secure remote access
- 📧 **Email Security** - PGP, S/MIME
- 🗄️ **Database Encryption** - At-rest and in-transit
- ☁️ **Cloud Security** - AWS, Azure, GCP

Did you know? Every time you visit a website with HTTPS, you're using cryptography!

Cryptographic Terminology

Basic Terms

- **Plaintext** - Original readable message
- **Ciphertext** - Encrypted message
- **Key** - Secret used for encryption/decryption
- **Algorithm** - Mathematical process for encryption

Security Terms

- **Cipher** - Encryption/decryption algorithm
- **Cryptanalysis** - Study of breaking ciphers
- **Key Space** - All possible keys
- **Entropy** - Randomness in keys/passwords

Remember: Security through obscurity is NOT security. Always assume attackers know your algorithm!

Cryptographic Primitives

Types of Cryptographic Primitives

Symmetric Cryptography

- **Same key** for encryption and decryption
- **Fast** and efficient
- **Examples:** AES, ChaCha20, DES
- **Use cases:** File encryption, database encryption

Asymmetric Cryptography

- **Different keys** for encryption and decryption
- **Slower** but more flexible
- **Examples:** RSA, ECDSA, Diffie-Hellman
- **Use cases:** Key exchange, digital signatures

Hash Functions

- **One-way** transformation
- **Fixed output** size
- **Examples:** SHA-256, MD5, BLAKE2
- **Use cases:** Password storage, data integrity

Digital Signatures

- **Authenticate** message sender
- **Ensure** message integrity
- **Examples:** RSA signatures, ECDSA
- **Use cases:** Software distribution, contracts

Symmetric Cryptography Deep Dive

How It Works

- **Same key** for encryption and decryption
- **Shared secret** between parties
- **Fast** and efficient for large data
- **Examples:** AES, ChaCha20, DES

Use Cases

- **File encryption** - Large data protection
- **Database encryption** - At-rest data
- **Streaming data** - Real-time encryption
- **Disk encryption** - Full disk protection

Key Management

- **Key distribution** is the main challenge
- **Key rotation** for security
- **Secure storage** of keys
- **Key derivation** from passwords

Security Considerations

- **Key length** matters (128, 192, 256 bits)
- **Key generation** must be random
- **Key storage** must be secure
- **Algorithm choice** is critical

Asymmetric Cryptography Deep Dive

How It Works

- **Public key** - Can be shared openly
- **Private key** - Must be kept secret
- **Mathematical relationship** between keys
- **Examples:** RSA, ECDSA, Ed25519

Use Cases

- **Key exchange** - Establish shared secrets
- **Digital signatures** - Authentication
- **Email encryption** - PGP, S/MIME
- **SSL/TLS** - Web security

Key Pairs

- **RSA** - Based on factoring large numbers
- **ECC** - Based on elliptic curves
- **Ed25519** - Modern, efficient curve
- **Key sizes** vary by algorithm

Security Considerations

- **Key size** must be appropriate
- **Key generation** requires good randomness
- **Private key** protection is critical
- **Algorithm choice** affects security

Symmetric vs Asymmetric

Symmetric Cryptography

```
# Same key for both operations
key = "secret_key_123"
encrypted = encrypt(message, key)
decrypted = decrypt(encrypted, key)
```

Advantages:

- Fast performance
- Simple implementation
- Low computational overhead

Disadvantages:

- Key distribution problem
- Doesn't scale well

Asymmetric Cryptography

```
# Different keys for each operation
public_key, private_key = generate_key_pair()
encrypted = encrypt(message, public_key)
decrypted = decrypt(encrypted, private_key)
```

Advantages:

- Solves key distribution
- Enables digital signatures
- Scales to many users

Disadvantages:

- Slower performance
- More complex implementation
- Higher computational overhead

Symmetric vs Asymmetric Theory

Why Symmetric is Fast

Mathematical simplicity:

- **Bit operations** - XOR, shifts, substitutions
- **Linear operations** - Fast on modern CPUs
- **Small key sizes** - 128-256 bits typical
- **Stream processing** - Can encrypt data as it arrives

Key distribution problem:

- Both parties need the same secret key
- How do you securely share the key?
- Doesn’t scale to many users
- Requires secure channel for key exchange

Why Asymmetric is Slow

Mathematical complexity:

- **Large number operations** - Exponentiation, modular arithmetic
- **Large key sizes** - 2048+ bits for RSA
- **Complex algorithms** - Elliptic curve operations
- **CPU intensive** - Not suitable for large data

Key distribution solution:

- Public keys can be shared openly
- Private keys stay secret
- Scales to unlimited users
- Enables digital signatures

Hash Functions Deep Dive

Properties

- **One-way** - Easy to compute, hard to reverse
- **Deterministic** - Same input = same output
- **Fixed output** - Always same length
- **Avalanche effect** - Small change = big difference

Hash Functions Theory

Why Hash Functions Work

One-way property comes from:

- **Mathematical complexity** - No efficient algorithm to reverse
- **Information loss** - Output smaller than input
- **Avalanche effect** - Each bit affects many output bits
- **Non-linear operations** - Mixing functions prevent patterns

Avalanche Effect

- Changing 1 bit in input changes ~50% of output bits
- Makes it impossible to predict output from similar inputs
- Ensures uniform distribution of hash values
- Prevents pattern recognition attacks

Hash Functions Security

Mathematical Foundation

Hash functions are based on:

- **Compression functions** - Reduce input size
- **Mixing functions** - Distribute bits uniformly
- **Iterative construction** - Process input in blocks
- **Finalization** - Ensure all input affects output

Security Properties

Security depends on:

- **Collision resistance** - Hard to find two inputs with same hash
- **Preimage resistance** - Hard to find input for given hash
- **Second preimage** - Hard to find different input with same hash

Key Insight: Hash functions are designed to be computationally irreversible. The security comes from the mathematical difficulty of finding collisions or preimages!

Hash Functions Algorithms

Common Algorithms

- **MD5** - 128-bit, broken (don't use!)
- **SHA-1** - 160-bit, deprecated
- **SHA-256** - 256-bit, widely used
- **SHA-3** - 256-bit, newer standard

Use Cases

- **Password storage** - Never store plain text
- **File integrity** - Detect tampering
- **Digital signatures** - Sign hash, not data
- **Blockchain** - Block verification

Security Considerations

- **Collision resistance** - Hard to find same hash
- **Preimage resistance** - Hard to find input
- **Second preimage** - Hard to find different input
- **Algorithm choice** matters

Attack Models and Security

Types of Attacks

- **Ciphertext-only** - Attacker has only encrypted data
- **Known-plaintext** - Attacker has some plaintext-ciphertext pairs
- **Chosen-plaintext** - Attacker can choose plaintexts to encrypt
- **Chosen-ciphertext** - Attacker can choose ciphertexts to decrypt

Security Properties

- **Confidentiality** - Data remains secret
- **Integrity** - Data hasn't been modified
- **Availability** - Data is accessible when needed
- **Authenticity** - Data comes from claimed source

Important: Perfect security doesn't exist. We aim for computational security - making attacks computationally infeasible.

Common Attack Vectors

Cryptographic Attacks

- **Brute Force** - Try all possible keys
- **Frequency Analysis** - Analyze patterns
- **Timing Attacks** - Measure execution time
- **Side-channel** - Power, electromagnetic analysis

Implementation Attacks

- **Buffer Overflows** - Memory corruption
- **Injection Attacks** - SQL, XSS, etc.
- **Social Engineering** - Human manipulation
- **Physical Access** - Hardware compromise

Security Principle: The weakest link determines overall security. A perfect algorithm with poor implementation is insecure!

Programming Languages for Cryptography

🐍 Python

Best for: Learning, prototyping, web apps

- cryptography library
- pycryptodome
- Easy to read
- Great documentation

☕ Java

Best for: Enterprise, Android

- Built-in crypto APIs
- Bouncy Castle
- Cross-platform
- Strong typing

🦀 Rust

Best for: Performance, security

- ring library
- Memory safety
- Zero-cost abstractions
- Growing ecosystem

Recommendation: Start with Python for learning, then explore other languages based on your needs!

Practical Programming Examples

Simple XOR Cipher Implementation

```
def xor_cipher(text, key):
    """Simple XOR cipher implementation"""
    result = ""
    for char in text:
        # XOR each character with the key
        result += chr(ord(char) ^ key)
    return result

# Usage
message = "Hello World"
key = 5
encrypted = xor_cipher(message, key)
decrypted = xor_cipher(encrypted, key)
print(f"Original: {message}")
print(f"Encrypted: {encrypted}")
print(f"Decrypted: {decrypted}")
```

Why XOR Works

- **Reversible** - Same operation for encryption/decryption
- **Simple** - Easy to understand and implement
- **Fast** - Very efficient operation
- **Foundation** - Used in many modern ciphers

Security Issues

- **Weak** - Vulnerable to frequency analysis
- **Key reuse** - Same key for multiple messages
- **Patterns** - Can reveal information about plaintext

XOR Cipher Theory

Mathematical Properties

XOR (⊕) is its own inverse:

- `A ⊕ B = C`
- `C ⊕ B = A` (decryption)

Why this works:

- XOR is **associative**: `(A ⊕ B) ⊕ B = A ⊕ (B ⊕ B)`
- `B ⊕ B = 0` (any value XOR itself equals zero)
- `A ⊕ 0 = A` (any value XOR zero equals itself)

Example Walkthrough

```
Original: 'H' = 72 (binary: 01001000)
Key: 5 = 5 (binary: 00000101)
XOR: 72 ⊕ 5 = 77 (binary: 01001101) = 'M'

Decrypt: 'M' = 77 (binary: 01001101)
Key: 5 = 5 (binary: 00000101)
XOR: 77 ⊕ 5 = 72 (binary: 01001000) = 'H'
```

Key Insight: XOR's self-inverse property makes it perfect for symmetric encryption - the same operation encrypts and decrypts!

Caesar Cipher

Implementation

```
def caesar_cipher(text, shift):
    """Caesar cipher implementation"""
    result = ""
    for char in text:
        if char.isalpha():
            # Determine if uppercase or lowercase
            ascii_offset = 65 if char.isupper() else 97
            # Apply shift and wrap around alphabet
            shifted = (ord(char) - ascii_offset + shift) %
            result += chr(shifted + ascii_offset)
        else:
            result += char
    return result

# Usage
message = "Hello World"
shift = 3
encrypted = caesar_cipher(message, shift)
decrypted = caesar_cipher(encrypted, -shift)
```

Historical Context

- **Julius Caesar** used this cipher
- **Shift of 3** was common
- **Military communication** in ancient Rome
- **Foundation** for more complex ciphers

Breaking Caesar Cipher

- **Brute force** - Try all 25 possible shifts
- **Frequency analysis** - Analyze letter frequencies
- **Pattern recognition** - Look for common words

Caesar Cipher Theory

Mathematical Foundation

Modular Arithmetic:

- Maps letters to numbers: A=0, B=1, …, Z=25
- **Encryption:** `E(x) = (x + k) mod 26`
- **Decryption:** `D(y) = (y - k) mod 26`

Why it works:

- `D(E(x)) = ((x + k) - k) mod 26 = x mod 26 = x`
- Modular arithmetic ensures we stay within alphabet bounds
- The inverse operation is simply subtraction

Example Walkthrough

```
Original: 'H' = 7 (H is 8th letter, so 7 in 0-indexed)
Shift: 3
Encrypt: (7 + 3) mod 26 = 10 = 'K'

Decrypt: 'K' = 10
Shift: 3
Decrypt: (10 - 3) mod 26 = 7 = 'H'
```

Cryptographic Libraries Comparison

Python Libraries

- **cryptography** - High-level, secure by default
- **pycryptodome** - Low-level, more control
- **cryptodome** - Fork of pycrypto
- **hashlib** - Built-in hash functions

Other Languages

- **Java** - Built-in crypto APIs
- **C#** - System.Security.Cryptography
- **Go** - crypto package
- **Rust** - ring, openssl

Features

- **Symmetric encryption** - AES, ChaCha20
- **Asymmetric encryption** - RSA, ECC
- **Hash functions** - SHA-256, SHA-3
- **Digital signatures** - ECDSA, Ed25519

Best Practices

- **Use established libraries**
- **Keep libraries updated**
- **Follow security guidelines**
- **Test your implementations**

Modern Cryptographic Libraries

Python - cryptography

```python
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import rsa

# Generate key
key = Fernet.generate_key()
f = Fernet(key)

# Encrypt/Decrypt
encrypted = f.encrypt(b"Hello World")
decrypted = f.decrypt(encrypted)
```

Python - pycryptodome

```python
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

# Generate key and IV
key = get_random_bytes(16)
iv = get_random_bytes(16)

# Encrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
encrypted = cipher.encrypt(pad(b"Hello World", 16))
```

Best Practice: Always use well-tested cryptographic libraries. Never implement cryptographic algorithms from scratch for production use.

Hash Functions in Practice

Password Hashing

```python
import hashlib
import os

def hash_password(password, salt=None):
    if salt is None:
        salt = os.urandom(32)

    # Use PBKDF2 for key derivation
    key = hashlib.pbkdf2_hmac(
        'sha256',
        password.encode('utf-8'),
        salt,
        100000  # iterations
    )
    return salt + key

def verify_password(password, hashed):
    salt = hashed[:32]
    key = hashed[32:]
    return hash_password(password, salt) == hashed
```

File Integrity

```python
def calculate_file_hash(filename):
    hash_sha256 = hashlib.sha256()
    with open(filename, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            hash_sha256.update(chunk)
    return hash_sha256.hexdigest()

# Usage
file_hash = calculate_file_hash("document.pdf")
print(f"SHA-256: {file_hash}")
```

Digital Signatures Example

Creating Signatures

```python
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import rsa

def create_signature(message, private_key):
    signature = private_key.sign(
        message.encode(),
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature
```

Verifying Signatures

```python
def verify_signature(message, signature, public_key):
    try:
        public_key.verify(
            signature,
            message.encode(),
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except:
        return False
```

Use Case: Digital signatures ensure message authenticity and integrity - perfect for software distribution and contracts!

Practical Assignment

Task 1: Simple Cipher Implementation

Create a Python program that implements and demonstrates:

1. **XOR Cipher** with different keys
2. **Caesar Cipher** with different shifts
3. **Frequency Analysis** tool
4. **Brute Force** attack on Caesar cipher

Requirements

- **Clean, documented code**
- **Error handling** for edge cases
- **Interactive menu** for user selection
- **Test cases** with different inputs
- **Git repository** with proper commits

Goal: Understand how basic ciphers work and why they can be broken **Submission:** GitHub repository link **Focus:** Learning through implementation and experimentation

Development Environment Setup

Required Software

```
# Python 3.8+
python --version

# Install required packages
pip install cryptography pycryptodome requests

# Install development tools
pip install pytest black flake8
```

IDE Setup

- **Visual Studio Code** with Python extension
- **PyCharm** (Community or Professional)
- **Jupyter Notebook** for experimentation
- **Git** for version control

Project Structure

```
cryptography-course/
├── assignments/
│   └── assignment1/
├── lectures/
├── projects/
└── README.md
```

Common Vulnerabilities to Avoid

Programming Mistakes

- **Hardcoded keys** - Never embed secrets in code
- **Weak randomness** - Use cryptographically secure random generators
- **Key reuse** - Don't reuse keys across different contexts
- **Timing attacks** - Be aware of timing-based vulnerabilities

Implementation Issues

- **Padding attacks** - Use proper padding schemes
- **Side-channel attacks** - Protect against power/timing analysis
- **Buffer overflows** - Validate input lengths
- **Memory leaks** - Clear sensitive data from memory

Security Principle: Security through obscurity is not security. Always assume attackers know your implementation details.

Real-World Applications

Web Security

- **HTTPS** - TLS/SSL encryption - **JWT Tokens** - Stateless authentication - **OAuth 2.0** - Authorization framework - **CSRF Protection** - Cross-site request forgery

Mobile Apps

- **Keychain** - Secure storage - **Biometric Auth** - Fingerprint/Face ID - **Certificate Pinning** - Prevent MITM attacks - **App Transport Security** - iOS/Android security

Blockchain

- **Digital Signatures** - Transaction authentication - **Hash Functions** - Block integrity - **Merkle Trees** - Data structure integrity - **Consensus Algorithms** - Network agreement

Next Week Preview

Week 2: Classical Ciphers

- **Substitution Ciphers** - Monoalphabetic and polyalphabetic
- **Transposition Ciphers** - Columnar and rail fence
- **Frequency Analysis** - Breaking substitution ciphers
- **Vigenère Cipher** - Polyalphabetic substitution

Practical Focus

- **Implement** various classical ciphers
- **Create** frequency analysis tools
- **Break** ciphers using statistical methods
- **Compare** security of different approaches

Reading Assignment: Review the classical ciphers chapter and prepare questions about implementation challenges.

Security Best Practices

Key Management

- **Never hardcode keys** in source code
- **Use environment variables** for secrets
- **Rotate keys regularly** for security
- **Use key management services** (AWS KMS, Azure Key Vault)

Implementation Security

- **Validate all inputs** before processing
- **Use constant-time algorithms** to prevent timing attacks
- **Clear sensitive data** from memory
- **Use secure coding practices**

Random Number Generation

- **Use cryptographically secure** random generators
- **Avoid predictable patterns** in keys
- **Use proper entropy sources** (hardware RNG)
- **Test randomness quality** regularly

Monitoring & Testing

- **Log security events** for analysis
- **Regular security audits** of code
- **Penetration testing** of applications
- **Keep libraries updated** with security patches

Common Mistakes to Avoid

Programming Errors

- ❌ **Hardcoded secrets** in code
- ❌ **Weak random number generation**
- ❌ **Reusing keys** across different contexts
- ❌ **Not validating inputs**

Implementation Issues

- ❌ **Timing attacks** vulnerabilities
- ❌ **Memory leaks** with sensitive data
- ❌ **Side-channel attacks** exposure
- ❌ **Not handling errors** properly

Algorithm Misuse

- ❌ **Using broken algorithms** (MD5, DES)
- ❌ **Wrong key sizes** for algorithms
- ❌ **Improper padding** schemes
- ❌ **Not using authenticated encryption**

Security Oversights

- ❌ **Security through obscurity**
- ❌ **Not updating dependencies**
- ❌ **Poor key management**
- ❌ **Inadequate testing**

Golden Rule: When in doubt, consult security experts and use established, well-tested libraries!

Career Opportunities in Cryptography

Job Roles

- **Security Engineer** - Implement security solutions
- **Cryptographer** - Research new algorithms
- **Penetration Tester** - Find vulnerabilities
- **Security Architect** - Design secure systems

Skills Needed

- **Programming** - Python, Java, C++, Rust
- **Mathematics** - Number theory, algebra
- **Security Knowledge** - Attack vectors, defenses
- **Communication** - Explain complex concepts

Industries

- **Financial Services** - Banking, payments
- **Technology Companies** - Cloud, software
- **Government** - Intelligence, defense
- **Healthcare** - Medical data protection

Certifications

- **CISSP** - Certified Information Systems Security Professional
- **CISM** - Certified Information Security Manager
- **CEH** - Certified Ethical Hacker
- **CISSP** - Certified Information Systems Security Professional

Resources for Learning

Online Courses

- **Coursera** - Cryptography I (Dan Boneh)
- **edX** - Introduction to Cryptography
- **Udemy** - Practical Cryptography
- **Khan Academy** - Computer Science

Practice Platforms

- **CryptoHack** - Interactive challenges
- **Cryptopals** - Programming challenges
- **CTFtime** - Capture the flag competitions
- **HackTheBox** - Penetration testing practice

Books

- **"Real-World Cryptography"** - David Wong
- **"Cryptography Engineering"** - Ferguson, Schneier
- **"Serious Cryptography"** - Jean-Philippe Aumasson
- **"Applied Cryptography"** - Bruce Schneier

Tools & Libraries

- **Wireshark** - Network analysis
- **Burp Suite** - Web application testing
- **John the Ripper** - Password cracking
- **OpenSSL** - Command-line crypto tools

Questions?

Let's discuss cryptography! 💬

Next Week: We'll dive into classical ciphers and learn how to implement them from scratch!

Assignment: Create your first cipher implementation and try to break it!