Block Ciphers and Applications

MAT364 - Cryptography Course

Instructor: Adil Akhmetov **University:** SDU **Week 5**

Press Space for next page →

What Are Block Ciphers?

Definition

Block ciphers encrypt data in fixed-size blocks (typically 64, 128, or 256 bits) using a secret key.

Key Characteristics

- **Fixed block size** - Always encrypt same amount of data
- **Deterministic** - Same input always produces same output
- **Reversible** - Can decrypt to get original plaintext
- **Key-dependent** - Output depends on secret key

How They Work

- **Divide plaintext** into fixed-size blocks
- **Apply encryption function** to each block
- **Use same key** for all blocks
- **Combine blocks** to form ciphertext

Common Block Sizes

- **64 bits** - Older ciphers (DES, 3DES)
- **128 bits** - Modern standard (AES)
- **256 bits** - High security applications
- **Variable** - Some ciphers support multiple sizes

Block Cipher Process Visualization

Plaintext
"HELLO WORLD 123"

→

128-bit Blocks
16 bytes each

→

Ciphertext
Encrypted blocks

🎯 Student Task: Block Cipher Basics

Task: Understanding Block Sizes

Given:

- Message: "CRYPTOGRAPHY ROCKS"
- Block size: 128 bits (16 bytes)
- Each character = 1 byte

Your Task:

1. How many bytes is the message?
2. How many complete blocks can be formed?
3. How many bytes are left over?
4. What needs to be done with the leftover bytes?

Hint: Count the characters including spaces!

Take 2 minutes to figure this out!

✅ Solution: Block Cipher Task

Step-by-Step Solution

Step 1: Count the bytes

```
"CRYPTOGRAPHY ROCKS" = 18 characters = 18 bytes
```

Step 2: Calculate complete blocks

```
Block size: 16 bytes
Complete blocks: 18 ÷ 16 = 1 complete block
```

Step 3: Calculate leftover bytes

```
Leftover: 18 - 16 = 2 bytes
```

Step 4: Handle leftover bytes

```
Need padding! Add 14 bytes to make it 16 bytes total.
Common method: PKCS#7 padding
```

Key Insight: Block ciphers require padding when data doesn't fit perfectly into blocks!

Block Cipher Design Principles

Confusion and Diffusion

Confusion

- **Obscure relationship** between key and ciphertext
- **Make it hard** to find the key from ciphertext
- **Implemented through** substitution operations
- **Example:** S-boxes in AES

Diffusion

- **Spread influence** of each plaintext bit
- **Small change** in input → big change in output
- **Implemented through** permutation operations
- **Example:** MixColumns in AES

How Confusion Works

```
# S-box substitution (simplified)
def substitute_byte(byte_value):
    s_box = [0×63, 0×7C, 0×77, 0×7B, ...]  # 256 values
    return s_box[byte_value]

# Each input byte maps to different output byte
input_byte = 0×53
output_byte = substitute_byte(input_byte)  # 0×ED
```

How Diffusion Works

```
# Simple diffusion (shift and mix)
def diffusion(data):
    # Shift bits around
    shifted = ((data << 1) | (data >> 7)) & 0×FF

    # Mix with itself
    mixed = shifted ^ (shifted >> 4)

    return mixed

# Small change creates big difference
input1 = 0b10101010  # 170
```

Feistel Networks

What is a Feistel Network?

- **Symmetric structure** for block ciphers
- **Splits input** into left and right halves
- **Applies round function** to one half
- **XORs result** with other half
- **Swaps halves** and repeats

Why Feistel Networks?

- **Encryption/decryption** use same structure
- **Round function** doesn't need to be invertible
- **Easy to implement** in hardware/software
- **Proven security** when properly designed

Feistel Round Function

```python
def feistel_round(left, right, round_key):
    """Single round of Feistel network"""
    # Apply round function to right half
    f_output = round_function(right, round_key)

    # XOR with left half
    new_right = left ^ f_output

    # Swap halves
    new_left = right

    return new_left, new_right

def round_function(data, key):
    """Round function (can be any function)"""
```

🎯 Student Task: Feistel Network

Task: Trace Feistel Rounds

Given:

- Initial block: $L_0 = 1010$, $R_0 = 1100$
- Round function: $F(R, K) = R \oplus K$ (simple XOR)
- Round keys: $K_1 = 1001$, $K_2 = 0110$

Your Task: Trace through 2 rounds of the Feistel network:

Round 1:

- $F(R_0, K_1) = ?$
- $L_1 = ?$
- $R_1 = ?$

Round 2:

- $F(R_1, K_2) = ?$
- $L_2 = ?$
- $R_2 = ?$

Remember: $L_1 = R_0$ and $R_1 = L_0 \oplus F(R_0, K_1)$

✅ Solution: Feistel Network

Step-by-Step Solution

Initial State:

$L_0 = 1010,\ R_0 = 1100$

Round 1:

$F(R_0, K_1) = 1100 \oplus 1001 = 0101$
$L_1 = R_0 = 1100$
$R_1 = L_0 \oplus F(R_0, K_1) = 1010 \oplus 0101 = 1111$

Round 2:

$F(R_1, K_2) = 1111 \oplus 0110 = 1001$
$L_2 = R_1 = 1111$
$R_2 = L_1 \oplus F(R_1, K_2) = 1100 \oplus 1001 = 0101$

Final Result: $L_2R_2 = 11110101$

Key Insight: Feistel networks are elegant because the same structure works for both encryption and decryption!

Advanced Encryption Standard (AES)

AES Overview

AES History

- **Originally Rijndael** - Designed by Joan Daemen and Vincent Rijmen
- **NIST competition** - Selected in 2001
- **Replaced DES** - Much stronger than previous standard
- **Widely adopted** - Used everywhere today

AES Specifications

- **Block size:** 128 bits (16 bytes)
- **Key sizes:** 128, 192, 256 bits
- **Rounds:** 10, 12, 14 (depending on key size)
- **Structure:** Substitution-Permutation Network (not Feistel)

AES Round Functions

1. **SubBytes** - Byte substitution using S-box
2. **ShiftRows** - Cyclically shift rows
3. **MixColumns** - Mix data within columns
4. **AddRoundKey** - XOR with round key

AES Implementation

```
from cryptography.hazmat.primitives.ciphers import Ciph
from cryptography.hazmat.primitives import padding
import os

def aes_encrypt(plaintext, key):
    """AES encryption with CBC mode"""
    # Generate random IV
    iv = os.urandom(16)

    # Create cipher
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv)
    encryptor = cipher.encryptor()
```

AES Round Function Detail

SubBytes Operation

```
# AES S-box (partial)
S_BOX = [
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
    0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    # ... (256 total values)
]

def sub_bytes(state):
    """Apply S-box substitution"""
    for i in range(4):
        for j in range(4):
            state[i][j] = S_BOX[state[i][j]]
    return state
```

ShiftRows Operation

```
def shift_rows(state):
    """Shift rows cyclically"""
    # Row 0: no shift
    # Row 1: shift left by 1
```

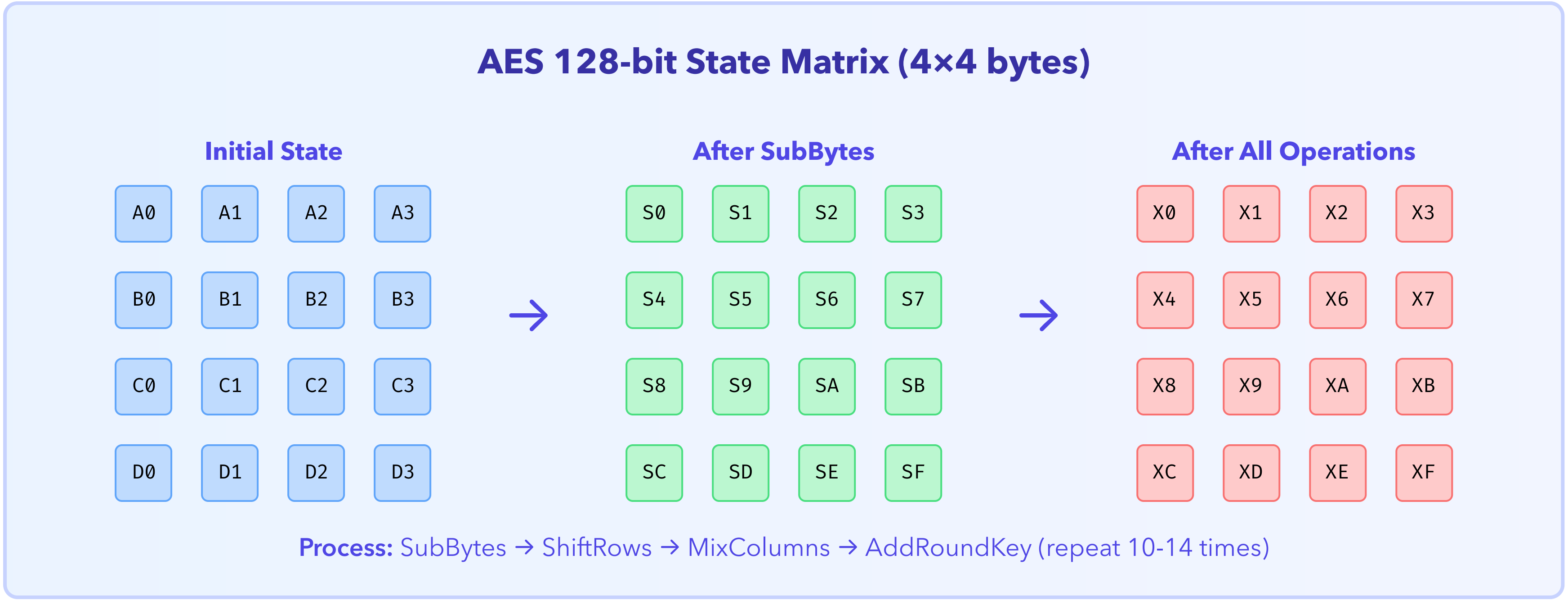
MixColumns Operation

```
def mix_columns(state):
    """Mix columns using matrix multiplication"""
    # Multiplication matrix for AES
    mix_matrix = [
        [2, 3, 1, 1],
        [1, 2, 3, 1],
        [1, 1, 2, 3],
        [3, 1, 1, 2]
    ]

    for col in range(4):
        column = [state[row][col] for row in range(4)]
        mixed = matrix_multiply_gf(mix_matrix, column)
        for row in range(4):
            state[row][col] = mixed[row]
```

Complete AES Round

```
def aes_round(state, round_key):
    """One complete AES round"""
    state = sub_bytes(state)
```

AES State Matrix



Process: SubBytes → ShiftRows → MixColumns → AddRoundKey (repeat 10-14 times)

🎯 Student Task: AES Key Sizes

Task: Choose the Right AES Configuration

Scenario: You're building a secure file storage system.

Requirements:

- Store personal financial documents
- Must be secure for at least 30 years
- Users upload files frequently (performance matters)
- Government compliance required

Options:

- **AES-128:** 128-bit key, 10 rounds, fastest
- **AES-192:** 192-bit key, 12 rounds, medium speed
- **AES-256:** 256-bit key, 14 rounds, slowest

Consider:

- Quantum computers may break AES-128 in the future
- Government standards require AES-256 for classified data
- Performance difference is about 40% between AES-128 and AES-256

Questions:

✅ Solution: AES Configuration

Recommended Solution: AES-256

1. Why AES-256:

- **Future-proof:** Resistant to quantum attacks (effectively becomes AES-128 post-quantum)
- **Compliance:** Meets government standards for sensitive data
- **30-year security:** Strong enough for long-term protection
- **Industry standard:** Widely adopted for financial applications

2. Reasoning:

- **Security priority:** Financial documents require maximum protection
- **Regulatory compliance:** Government standards mandate AES-256
- **Future threats:** Quantum computers will weaken all current encryption
- **Cost of breach:** Much higher than performance overhead

3. Performance Trade-offs:

- **Accept 40% slower encryption** for much better security
- **Use hardware acceleration** (AES-NI instructions on modern CPUs)
- **Optimize implementation** with established libraries
- **Consider hybrid approach:** AES-256 for data, AES-128 for temporary operations

Block Cipher Modes of Operation

Why Do We Need Modes?

The Problem

- **Block ciphers** only encrypt fixed-size blocks
- **Real data** is usually larger than one block
- **Naive approach** - encrypt each block independently
- **Security issues** with independent encryption

Security Issues

- **Pattern leakage** - Identical blocks show patterns
- **No randomness** - Predictable outputs
- **Vulnerable to analysis** - Easy to detect structure
- **Real-world impact** - Can reveal image patterns, repeated data

ECB Mode Problems

```python
# ECB Mode (Electronic Codebook) - INSECURE!
def ecb_encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    blocks = split_into_blocks(plaintext, 16)
    ciphertext = b''

    for block in blocks:
        ciphertext += cipher.encrypt(block)

    return ciphertext

# Problem: Same plaintext block = Same ciphertext bloc
```

Example: Image Encryption

```
Original Image: [BLACK][WHITE][BLACK][WHITE]
ECB Result:     [ENCR1][ENCR2][ENCR1][ENCR2]
                  ↑      ↑      ↑      ↑
                Same!  Same!  Same!  Same!

Problem: The pattern is still visible!
```

Cipher Block Chaining (CBC) Mode

How CBC Works

- **XOR each block** with previous ciphertext
- **First block** XORed with random IV
- **Creates dependency** between blocks
- **Same plaintext** → different ciphertext (due to IV)

CBC Encryption

```python
def cbc_encrypt(plaintext, key, iv):
    """CBC mode encryption"""
    cipher = AES.new(key, AES.MODE_ECB)
    blocks = split_into_blocks(plaintext, 16)
    ciphertext = b''
    prev_block = iv  # Start with IV

    for block in blocks:
        # XOR with previous ciphertext
        xored = xor_bytes(block, prev_block)

        # Encrypt the XORed result
```

CBC Decryption

```python
def cbc_decrypt(ciphertext, key):
    """CBC mode decryption"""
    # Extract IV
    iv = ciphertext[:16]
    encrypted_blocks = ciphertext[16:]

    cipher = AES.new(key, AES.MODE_ECB)
    blocks = split_into_blocks(encrypted_blocks, 16)
    plaintext = b''
    prev_block = iv

    for block in blocks:
        # Decrypt the block
        decrypted = cipher.decrypt(block)
```

CBC Properties

- **Random IV** required for each encryption
- **Sequential** - Cannot parallelize encryption
- **Self-synchronizing** - Errors don't propagate

Counter (CTR) Mode

How CTR Works

- **Turns block cipher** into stream cipher
- **Encrypt counter values** to create keystream
- **XOR keystream** with plaintext
- **Parallelizable** - Can encrypt blocks independently

CTR Advantages

- **Parallelizable** - Can process blocks in parallel
- **Random access** - Can decrypt any block independently
- **No padding** - Works with any data length
- **Stream cipher properties** - XOR-based operation

CTR Implementation

```
def ctr_encrypt(plaintext, key, nonce):
    """Counter mode encryption"""
    cipher = AES.new(key, AES.MODE_ECB)
    blocks = split_into_blocks(plaintext, 16)
    ciphertext = b''
    counter = 0

    for block in blocks:
        # Create counter block
        counter_block = nonce + counter.to_bytes(8, 'bi

        # Encrypt counter to get keystream
```

CTR Requirements

- **Unique nonce** for each encryption
- **Counter must not repeat** with same key
- **Nonce + counter** must be unique
- **No authentication** - needs additional MAC

CTR Security

```
# Counter construction (common approach)
def create_counter_block(nonce, counter):
```

Galois/Counter Mode (GCM)

What is GCM?

- **Authenticated encryption** - Provides confidentiality AND integrity
- **Combines CTR mode** with Galois field authentication
- **Industry standard** - Used in TLS 1.3, IPsec
- **High performance** - Hardware acceleration available

GCM Components

- **CTR mode** for encryption
- **GHASH** for authentication
- **Additional Associated Data (AAD)** support
- **Authentication tag** verifies integrity

GCM Implementation

```
from cryptography.hazmat.primitives.ciphers.aead impor
import os

def gcm_encrypt(plaintext, key, aad=b''):
    """GCM mode encryption with authentication"""
    # Generate random nonce
    nonce = os.urandom(12)  # 96-bit nonce for GCM

    # Create AESGCM cipher
    aesgcm = AESGCM(key)

    # Encrypt and authenticate
    ciphertext = aesgcm.encrypt(nonce, plaintext, aad)

    return nonce + ciphertext
```

🎯 Student Task: Mode Comparison

Task: Choose the Right Mode

Scenario 1: Video Streaming Service

- Need to encrypt video files for streaming
- Users may seek to different parts of the video
- Performance is critical
- Authentication not required (separate signature)

Scenario 2: Secure Messaging App

- Encrypt chat messages
- Need both confidentiality and integrity
- Messages are small (< 1KB typically)
- Security is more important than performance

Scenario 3: Database Encryption

- Encrypt database records
- Need to decrypt individual records randomly
- Performance matters for queries
- Integrity verification needed

✅ Solution: Mode Comparison

Optimal Mode Choices

Scenario 1 → CTR Mode

- **Random access:** Can seek to any part of video
- **Parallelizable:** Can decrypt multiple blocks simultaneously
- **No padding:** Works perfectly with video frame boundaries
- **High performance:** Excellent for streaming applications

Scenario 2 → GCM Mode

- **Authenticated encryption:** Provides both confidentiality and integrity
- **Small messages:** Overhead is acceptable for small data
- **Security priority:** Perfect for messaging applications
- **Standard choice:** Used in Signal, WhatsApp, etc.

Scenario 3 → GCM Mode

- **Random access:** Can decrypt individual records
- **Authentication:** Verifies database integrity
- **Performance:** Good for database applications
- **Standard compliance:** Meets enterprise security requirements

Padding and Data Handling

PKCS#7 Padding

Why Padding?

- **Block ciphers** require complete blocks
- **Real data** rarely fits perfectly
- **Must pad** incomplete blocks
- **Padding must be removable** after decryption

PKCS#7 Algorithm

- **Add N bytes** each with value N
- **If block is complete** - add full block of padding
- **Always unambiguous** - Can always remove correctly
- **Most common** padding scheme

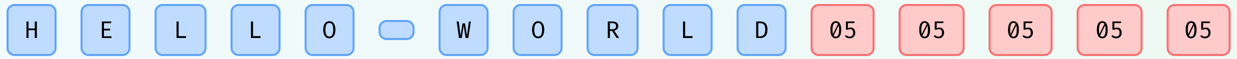
PKCS#7 Implementation

```python
def pkcs7_pad(data, block_size):
    """Add PKCS#7 padding to data"""
    padding_length = block_size - (len(data) % block_si
    padding = bytes([padding_length] * padding_length)
    return data + padding

def pkcs7_unpad(padded_data):
    """Remove PKCS#7 padding from data"""
    if len(padded_data) == 0:
        raise ValueError("Empty data")

    padding_length = padded_data[-1]

    # Validate padding
    if padding_length == 0 or padding_length > len(pad
```

Padding Examples

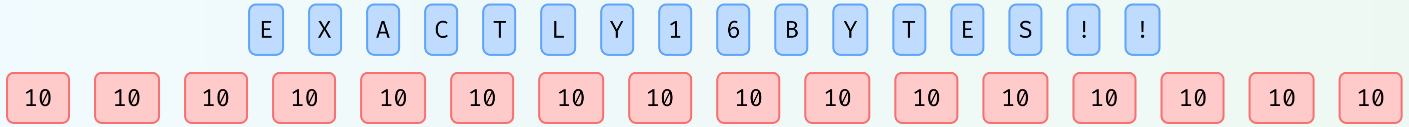
PKCS#7 Padding Examples

Example 1: "HELLO WORLD" (11 bytes) → 16-byte block



Need 5 padding bytes, so add five 0x05 bytes

Example 2: "EXACTLY16BYTES!!" (16 bytes) → Need another block



Complete block needs full 16-byte padding block (sixteen 0x10 bytes)

Key Rule: Padding length = value of padding bytes. This makes removal unambiguous!

Real-World Applications

File Encryption System

Complete File Encryption

```python
import os
from cryptography.hazmat.primitives.ciphers import Ciph
from cryptography.hazmat.primitives import padding, has
from cryptography.hazmat.primitives.kdf.pbkdf2 import P

class FileEncryption:
    def __init__(self, password):
        self.password = password.encode()

    def _derive_key(self, salt):
        """Derive key from password"""
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,  # 256-bit key
            salt=salt
```

File Decryption

```python
def decrypt_file(self, input_file, output_file):
    """Decrypt a file using AES-CBC"""
    with open(input_file, 'rb') as infile:
        # Read salt and IV
        salt = infile.read(16)
        iv = infile.read(16)

        # Derive key from password
        key = self._derive_key(salt)

        # Create cipher
        cipher = Cipher(algorithms.AES(key), modes.CBC(
        decryptor = cipher.decryptor()

        # Read encrypted data
```

Database Encryption

Column-Level Encryption

```
import sqlite3
from cryptography.hazmat.primitives.ciphers.aead import
import os
import base64

class EncryptedDatabase:
    def __init__(self, db_path, encryption_key):
        self.conn = sqlite3.connect(db_path)
        self.aesgcm = AESGCM(encryption_key)
        self.setup_tables()

    def setup_tables(self):
        """Create tables with encrypted columns"""
        self.conn.execute('''
            CREATE TABLE IF NOT EXISTS users (
```

Database Operations

```
def add_user(self, username, email, ssn):
    """Add user with encrypted sensitive data"""
    encrypted_email = self.encrypt_field(email)
    encrypted_ssn = self.encrypt_field(ssn)

    self.conn.execute('''
        INSERT INTO users (username, email_encrypted,
        VALUES (?, ?, ?)
    ''', (username, encrypted_email, encrypted_ssn))
    self.conn.commit()

def get_user(self, user_id):
    """Get user and decrypt sensitive data"""
    cursor = self.conn.execute('''
        SELECT id, username, email_encrypted, ssn_encr
```

Web API Encryption

API Payload Encryption

```
from flask import Flask, request, jsonify
from cryptography.hazmat.primitives.ciphers.aead import
import json
import base64
import os

app = Flask(__name__)

class APIEncryption:
    def __init__(self, key):
        self.aesgcm = AESGCM(key)

    def encrypt_payload(self, data):
        """Encrypt JSON payload"""
        # Convert to JSON string
```

Client-Side Usage

```
import requests
import json

class SecureAPIClient:
    def __init__(self, api_url, encryption_key):
        self.api_url = api_url
        self.api_crypto = APIEncryption(encryption_key

    def send_secure_request(self, data):
        """Send encrypted request to API"""
        # Encrypt payload
        encrypted_payload = self.api_crypto.encrypt_pay

        # Send request
        response = requests.post(
```

Performance and Security

Block Cipher Performance

Performance Factors

- **Algorithm choice** - AES vs alternatives
- **Key size** - 128 vs 256 bits
- **Mode of operation** - CBC vs CTR vs GCM
- **Implementation** - Software vs hardware
- **Data size** - Small vs large blocks

Hardware Acceleration

- **AES-NI** - Intel/AMD processors
- **ARMv8 Crypto** - ARM processors
- **Dedicated chips** - Hardware security modules
- **GPU acceleration** - Parallel processing

Performance Benchmark

```
import time
from cryptography.hazmat.primitives.ciphers import Ciph
import os

def benchmark_aes_modes():
    """Benchmark different AES modes"""
    key = os.urandom(32)  # 256-bit key
    data = os.urandom(1024 * 1024)  # 1MB test data

    modes_to_test = [
        ('CBC', modes.CBC(os.urandom(16))),
        ('CTR', modes.CTR(os.urandom(16))),
        ('GCM', modes.GCM(os.urandom(12))),
    ]
```

Security Best Practices

Key Management

- **Use strong keys** - 256-bit minimum
- **Generate securely** - Cryptographically secure random
- **Store safely** - Hardware security modules, key vaults
- **Rotate regularly** - Change keys periodically
- **Separate keys** - Different keys for different purposes

Implementation Security

- **Use established libraries** - Don't implement crypto yourself
- **Validate inputs** - Check all parameters
- **Handle errors** - Don't leak information through errors
- **Clear memory** - Wipe sensitive data after use

Common Mistakes

- ❌ **Weak randomness** - Using `random()` instead of `secrets`
- ❌ **Key reuse** - Using same key for different purposes
- ❌ **Predictable IVs** - Not using random IVs
- ❌ **ECB mode** - Never use ECB for real data
- ❌ **Hardcoded keys** - Never embed keys in code

Security Checklist

- ✅ **Use AES-256** with secure modes (GCM preferred)
- ✅ **Generate random IVs** for each encryption
- ✅ **Implement proper padding** (PKCS#7)
- ✅ **Use authenticated encryption** (GCM mode)
- ✅ **Test thoroughly** including edge cases
- ✅ **Keep libraries updated** for security patches

Common Vulnerabilities

Padding Oracle Attacks

- **Attack on CBC mode** with padding validation
- **Information leakage** through error messages
- **Can decrypt** without knowing the key
- **Mitigation:** Use authenticated encryption (GCM)

Timing Attacks

- **Measure execution time** to leak information
- **Affects key comparison** and validation
- **Can extract keys** bit by bit
- **Mitigation:** Constant-time implementations

Bit-flipping Attacks

- **Attack on CBC mode** - Modify ciphertext to change plaintext
- **Exploits XOR** properties of CBC
- **Can modify** specific plaintext bits
- **Mitigation:** Use authenticated encryption

Example: Secure Comparison

```python
import hmac

def secure_compare(a, b):
    """Constant-time string comparison"""
    return hmac.compare_digest(a, b)

def insecure_compare(a, b):
    """Vulnerable to timing attacks"""
    if len(a) ≠ len(b):
        return False

    for i in range(len(a)):
```

Practical Tasks

Task 1: Complete AES Implementation

Requirements

Create a complete AES encryption system:

1. **Key derivation** from passwords using PBKDF2
2. **Multiple modes** - CBC, CTR, GCM
3. **Proper padding** - PKCS#7 implementation
4. **File encryption** - Handle large files efficiently
5. **Error handling** - Robust error management

Features to Implement

- **Password-based encryption**
- **Salt generation and storage**
- **IV generation and management**
- **Integrity verification** (for GCM mode)
- **Performance measurement**

Implementation Framework

```
class AdvancedAES:
    def __init__(self, password=None, key=None):
        if password:
            self.key = self._derive_key_from_password(p
        elif key:
            self.key = key
        else:
            self.key = os.urandom(32)  # Generate rando

    def encrypt_cbc(self, plaintext):
        # Implement CBC mode encryption
        pass

    def decrypt_cbc(self, ciphertext):
        # Implement CBC mode decryption
```

Task 2: Mode Comparison Tool

Requirements

Build a tool to compare block cipher modes:

1. **Performance testing** - Measure speed of different modes
2. **Security analysis** - Demonstrate ECB vulnerabilities
3. **Pattern detection** - Show pattern leakage in ECB
4. **Visual comparison** - Create charts/graphs
5. **Report generation** - Comprehensive analysis

Test Cases

- **Different data sizes** - 1KB to 100MB
- **Different patterns** - Repeated blocks, images
- **Performance metrics** - Throughput, latency
- **Security metrics** - Pattern detection, entropy

Analysis Framework

```
class ModeAnalyzer:
    def __init__(self):
        self.key = os.urandom(32)
        self.test_data_sizes = [1024, 10240, 102400, 1(

    def performance_test(self, mode, data_size):
        """Test encryption/decryption performance"""
        test_data = os.urandom(data_size)

        # Measure encryption time
        start_time = time.time()
        encrypted = self.encrypt_with_mode(test_data, r
        encryption_time = time.time() - start_time

        # Measure decryption time
```

Task 3: Secure Database System

Requirements

Create a secure database with encrypted columns:

1. **Field-level encryption** - Encrypt sensitive columns
2. **Searchable encryption** - Support some queries on encrypted data
3. **Key management** - Secure key storage and rotation
4. **Audit logging** - Track all encryption/decryption operations
5. **Performance optimization** - Efficient encryption/decryption

Features

- **Multiple encryption keys** for different data types
- **Encrypted indexes** for search functionality
- **Backup encryption** - Secure database backups
- **User access control** - Role-based encryption access

Database Schema

```
class SecureDatabase:
    def __init__(self, db_path, master_key):
        self.conn = sqlite3.connect(db_path)
        self.crypto = self._setup_encryption(master_ke
        self.audit_log = AuditLogger()
        self.setup_schema()

    def setup_schema(self):
        """Create tables with encrypted fields"""
        self.conn.execute('''
            CREATE TABLE IF NOT EXISTS customers (
                id INTEGER PRIMARY KEY,
                name TEXT NOT NULL,
                email_encrypted BLOB,
                ssn_encrypted BLOB
```

Task 4: Performance Benchmarking

Requirements

Create comprehensive performance benchmarking:

1. **Algorithm comparison** - AES vs ChaCha20 vs others
2. **Mode comparison** - CBC vs CTR vs GCM
3. **Key size impact** - 128 vs 192 vs 256 bits
4. **Hardware utilization** - CPU vs hardware acceleration
5. **Memory usage** - RAM consumption analysis

Metrics to Measure

- **Throughput** - MB/s for encryption/decryption
- **Latency** - Time per operation
- **CPU usage** - Processor utilization
- **Memory usage** - RAM consumption
- **Scalability** - Performance with data size

Benchmark Framework

```python
import psutil
import matplotlib.pyplot as plt
from contextlib import contextmanager

class CryptoBenchmark:
    def __init__(self):
        self.results = {}
        self.data_sizes = [1024, 10*1024, 100*1024, 102

    @contextmanager
    def measure_resources(self):
        """Context manager to measure CPU and memory"""
        process = psutil.Process()

        # Initial measurements
```

Best Practices Summary

Algorithm Selection

- **Use AES-256** for most applications
- **Choose GCM mode** for authenticated encryption
- **Consider ChaCha20** for software-only implementations
- **Avoid deprecated** algorithms (DES, 3DES, RC4)

Key Management

- **Generate keys** using cryptographically secure random
- **Use key derivation** (PBKDF2, Argon2) for passwords
- **Store keys securely** (HSM, key vaults)
- **Rotate keys** regularly
- **Use different keys** for different purposes

Implementation Guidelines

- **Use established libraries** - Don't implement crypto yourself
- **Generate random IVs** for each encryption
- **Implement proper error handling**
- **Clear sensitive data** from memory
- **Use constant-time operations** when possible

Security Checklist

- ✅ **Never use ECB mode**
- ✅ **Always use random IVs**
- ✅ **Implement proper padding**
- ✅ **Use authenticated encryption**
- ✅ **Validate all inputs**
- ✅ **Test thoroughly**
- ✅ **Keep libraries updated**

Questions?

Let's discuss block ciphers! 💬

Next Week: We'll explore hash functions and data integrity verification!

Assignment: Implement a complete block cipher system with multiple modes and analyze their security properties!