Stream Ciphers and Modern Symmetric Encryption

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 4

Press Space for next page →

What Are Stream Ciphers?

Definition

Stream ciphers encrypt data bit-by-bit or byte-by-byte using a keystream generated from a secret key.

How They Work

- **Generate keystream** from secret key
- **XOR plaintext** with keystream
- **Same keystream** for decryption
- **Key determines** keystream generation

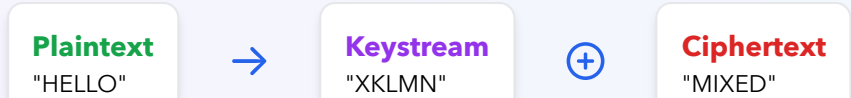
Key Characteristics

- **Synchronous** - Keystream independent of plaintext
- **Asynchronous** - Keystream depends on plaintext
- **Fast** - Suitable for real-time applications
- **Simple** - Easy to implement in hardware

Advantages

- **High speed** - Very fast encryption
- **Low latency** - Real-time processing
- **Hardware friendly** - Easy to implement
- **Memory efficient** - Minimal storage needed

Stream Cipher Process Visualization



🎯 Student Task: Stream Cipher Basics

Task: Simple XOR Stream Cipher

Given:

- Plaintext: "CRYPTO"
- Key: 5 (single byte)
- Operation: XOR each character with the key

Your Task:

1. Convert each letter to its ASCII value
2. XOR each ASCII value with the key (5)
3. Convert back to characters
4. What is the ciphertext?

Hint: A = 65, B = 66, C = 67, etc.

Take 2 minutes to work this out!

✅ Solution: Stream Cipher Task

Step-by-Step Solution

Step 1: Convert to ASCII

```
C = 67, R = 82, Y = 89, P = 80, T = 84, O = 79
```

Step 2: XOR with key (5)

```
67 ⊕ 5 = 70 (F)
82 ⊕ 5 = 87 (W)
89 ⊕ 5 = 92 (\)  ← Special character!
80 ⊕ 5 = 85 (U)
84 ⊕ 5 = 81 (Q)
79 ⊕ 5 = 74 (J)
```

Step 3: Result

```
Ciphertext: "FW\UQJ"
```

Key Insight: XOR is reversible! To decrypt, just XOR again with the same key.

Stream Cipher vs Block Cipher

Stream Ciphers

- **Encrypt bit/byte** at a time
- **Keystream** generated from key
- **XOR operation** with plaintext
- **Examples:** RC4, ChaCha20, A5/1

Block Ciphers

- **Encrypt fixed-size blocks** (64, 128 bits)
- **Same key** for all blocks
- **Complex operations** - Substitution, permutation
- **Examples:** AES, DES, Blowfish

Advantages

- **Fast** - Very high speed
- **Low latency** - Real-time processing
- **Simple** - Easy implementation
- **Memory efficient** - Minimal storage

Advantages

- **Secure** - Well-analyzed algorithms
- **Standardized** - NIST approved
- **Flexible** - Multiple modes of operation
- **Widely used** - Industry standard

Key Insight: Stream ciphers are like a continuous flow, while block ciphers work in discrete chunks!

Stream Cipher Design

Linear Feedback Shift Registers (LFSR)

How LFSR Works

- **Shift register** with feedback
- **Linear function** of register bits
- **Generates** pseudo-random sequence
- **Period** depends on register length

Example: 4-bit LFSR

```
Register: [1,0,1,1]
Feedback: XOR of bits 3 and 1
Output: 1 (rightmost bit)
Next: [0,1,0,1] (shift right, feedback in)
```

Implementation

```python
class LFSR:
    def __init__(self, seed, taps):
        self.register = seed
        self.taps = taps  # Positions to XOR

    def step(self):
        # Calculate feedback
        feedback = 0
        for tap in self.taps:
            feedback ^= (self.register >> tap) & 1

        # Shift and insert feedback
        self.register = (self.register >> 1) | (feedbac

        # Return output bit
```

LFSR Animation: 4-bit Register

🎯 Student Task: LFSR Keystream

Task: Generate LFSR Keystream

Given:

- 3-bit LFSR with seed: [1, 0, 1]
- Feedback taps: positions 2 and 0 (XOR of bits 2 and 0)
- Generate 8 bits of keystream

Your Task:

1. Start with register [1, 0, 1]
2. Output the rightmost bit (1)
3. Calculate feedback: bit[2] ⊕ bit[0] = 1 ⊕ 1 = 0
4. Shift right and insert feedback: [0, 1, 0]
5. Repeat for 8 steps

What is the 8-bit keystream?

Work through this step by step!

✅ Solution: LFSR Keystream

Step-by-Step Solution

Step 1: [1,0,1] → Output: 1, Feedback: 1⊕1 = 0 → Next: [0,1,0] **Step 2:** [0,1,0] → Output: 0, Feedback: 0⊕0 = 0 → Next: [0,0,1]
Step 3: [0,0,1] → Output: 1, Feedback: 0⊕1 = 1 → Next: [1,0,0] **Step 4:** [1,0,0] → Output: 0, Feedback: 1⊕0 = 1 → Next: [1,1,0]
Step 5: [1,1,0] → Output: 0, Feedback: 1⊕1 = 0 → Next: [0,1,1] **Step 6:** [0,1,1] → Output: 1, Feedback: 0⊕1 = 1 → Next: [1,0,1]
Step 7: [1,0,1] → Output: 1, Feedback: 1⊕1 = 0 → Next: [0,1,0] **Step 8:** [0,1,0] → Output: 0, Feedback: 0⊕0 = 0 → Next: [0,0,1]

Answer: `10100110`

Notice the pattern repeats after 7 steps (period = 7)

RC4 Stream Cipher

How RC4 Works

1. **Key Scheduling Algorithm (KSA)** - Initialize S-box
2. **Pseudo-Random Generation Algorithm (PRGA)** - Generate keystream
3. **XOR** plaintext with keystream
4. **Same process** for decryption

Key Features

- **Variable key length** - 1-256 bytes
- **Simple implementation** - Easy to code
- **Fast** - Very efficient
- **Widely used** - SSL/TLS, WEP

Implementation

```
class RC4:
    def __init__(self, key):
        self.key = key
        self.S = list(range(256))
        self._ksa()

    def _ksa(self):
        """Key Scheduling Algorithm"""
        j = 0
        for i in range(256):
            j = (j + self.S[i] + self.key[i % len(self
            self.S[i], self.S[j] = self.S[j], self.S[i

    def _prga(self, length):
        """Pseudo-Random Generation Algorithm"""
```

ChaCha20 Stream Cipher

Why ChaCha20?

- **RC4 vulnerabilities** - Weak key scheduling
- **AES timing attacks** - Side-channel vulnerabilities
- **ChaCha20 advantages** - Constant-time, fast, secure
- **Modern standard** - Used in TLS 1.3

Design Principles

- **ARX operations** - Add, Rotate, XOR
- **Constant-time** - No timing attacks
- **Fast software** - Optimized for CPUs
- **Simple design** - Easy to analyze

ChaCha20 Structure

```
def chacha20_quarter_round(state, a, b, c, d):
    """Quarter round function"""
    state[a] = (state[a] + state[b]) & 0×FFFFFFFF
    state[d] = state[d] ^ state[a]
    state[d] = ((state[d] << 16) | (state[d] >> 16)) &

    state[c] = (state[c] + state[d]) & 0×FFFFFFFF
    state[b] = state[b] ^ state[c]
    state[b] = ((state[b] << 12) | (state[b] >> 20)) &

    state[a] = (state[a] + state[b]) & 0×FFFFFFFF
    state[d] = state[d] ^ state[a]
    state[d] = ((state[d] << 8) | (state[d] >> 24)) & (

    state[c] = (state[c] + state[d]) & 0×FFFFFFFF
```

Block Cipher Modes of Operation

Electronic Codebook (ECB) Mode

How ECB Works

- **Each block** encrypted independently
- **Same plaintext** → same ciphertext
- **No chaining** between blocks
- **Parallel processing** possible

Example

```
Plaintext:  HELLO WORLD
Blocks:     HELL | O WO | RLD
Encrypt:    AES(HELL) | AES(O WO) | AES(RLD)
Ciphertext: XKLM | YZAB | CDE
```

Problems with ECB

- **Pattern leakage** - Identical blocks produce identical ciphertext
- **Not secure** - Reveals structure of plaintext
- **Example vulnerability:**

```
Image: [BLACK][WHITE][BLACK][WHITE]
ECB:   [ENCR1][ENCR2][ENCR1][ENCR2]
Result: Pattern still visible!
```

When to Use

- **Never for real data** - Too insecure
- **Educational purposes** - Understanding block ciphers
- **Single block** - Only one block to encrypt

ECB Mode Visualization

🎯 Student Task: Block Cipher Modes

Task: Identify the Problem

Scenario: You're encrypting an image with a block cipher.

Image Pattern:

```
[BLACK][WHITE][BLACK][WHITE]
[WHITE][BLACK][WHITE][BLACK]
[BLACK][WHITE][BLACK][WHITE]
[WHITE][BLACK][WHITE][BLACK]
```

ECB Encryption Result:

```
[ENCR1][ENCR2][ENCR1][ENCR2]
[ENCR2][ENCR1][ENCR2][ENCR1]
[ENCR1][ENCR2][ENCR1][ENCR2]
[ENCR2][ENCR1][ENCR2][ENCR1]
```

Questions:

1. What security problem does this demonstrate?
2. Why is this a problem for real applications?
3. What mode would you use instead?

✅ Solution: Block Cipher Modes

Answers

1. Security Problem:

- **Pattern leakage** - Identical plaintext blocks produce identical ciphertext blocks
- **Structure preservation** - The encrypted image still shows the original pattern
- **Information disclosure** - Attacker can see the structure without knowing the key

2. Why This Matters:

- **Images** - Patterns in photos, logos, diagrams are visible
- **Documents** - Repeated headers, footers, formatting revealed
- **Databases** - Duplicate records can be identified
- **Real-world impact** - Complete loss of confidentiality

3. Better Solution:

- **CBC mode** - Each block XORed with previous ciphertext
- **GCM mode** - Authenticated encryption with random IV
- **CTR mode** - Counter-based encryption
- **Any mode with chaining** - Breaks the pattern

Cipher Block Chaining (CBC) Mode

How CBC Works

- **XOR each block** with previous ciphertext
- **First block** XORed with IV (Initialization Vector)
- **Chaining** prevents pattern leakage
- **Sequential** - Cannot parallelize encryption

Encryption Process

```
P1 = Plaintext block 1
P2 = Plaintext block 2
IV = Initialization Vector

C1 = Encrypt(P1 ⊕ IV)
C2 = Encrypt(P2 ⊕ C1)
C3 = Encrypt(P3 ⊕ C2)
```

Decryption Process

```
C1 = Ciphertext block 1
C2 = Ciphertext block 2

P1 = Decrypt(C1) ⊕ IV
P2 = Decrypt(C2) ⊕ C1
P3 = Decrypt(C3) ⊕ C2
```

Implementation

```python
def cbc_encrypt(plaintext, key, iv):
    """CBC mode encryption"""
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b''
    prev_block = iv

    for i in range(0, len(plaintext), 16):
        block = plaintext[i:i+16]
        # Pad if necessary
        if len(block) < 16:
            block = pad(block, 16)
```

Galois/Counter Mode (GCM)

Why GCM?

- **Authenticated encryption** - Provides both confidentiality and integrity
- **Parallelizable** - Can encrypt blocks in parallel
- **Efficient** - Fast implementation
- **Widely used** - TLS 1.3, IPsec

Key Features

- **CTR mode** for encryption
- **GHASH** for authentication
- **No padding** required
- **Associated data** support

GCM Structure

```
def gcm_encrypt(plaintext, key, iv, aad=b''):
    """GCM mode encryption with authentication"""
    # Generate counter blocks
    counter_blocks = generate_counters(iv, len(plainte

    # Encrypt counter blocks
    cipher = AES.new(key, AES.MODE_ECB)
    keystream = b''.join(cipher.encrypt(ctr) for ctr in

    # XOR with plaintext
    ciphertext = bytes(a ^ b for a, b in zip(plaintext,

    # Calculate authentication tag
    tag = ghash(ciphertext, aad, key)
```

Advanced Symmetric Encryption

AES (Advanced Encryption Standard)

AES Overview

- **Rijndael algorithm** - Winner of AES competition
- **Block size** - 128 bits (16 bytes)
- **Key sizes** - 128, 192, 256 bits
- **Rounds** - 10, 12, 14 (depends on key size)

AES Structure

- **SubBytes** - Non-linear substitution
- **ShiftRows** - Transposition
- **MixColumns** - Linear transformation
- **AddRoundKey** - XOR with round key

AES Implementation

```python
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

def aes_encrypt(plaintext, key):
    """AES encryption"""
    # Generate random IV
    iv = get_random_bytes(16)

    # Create cipher
    cipher = AES.new(key, AES.MODE_CBC, iv)

    # Pad plaintext
    padded = pad(plaintext, AES.block_size)

    # Encrypt
```

AES Round Function

🎯 Student Task: AES Key Sizes

Task: Choose the Right AES Key Size

Scenario: You're designing a secure messaging app.

Requirements:

- Messages contain sensitive financial data
- Must be secure for at least 20 years
- Performance is important but security is critical
- Must comply with government standards

Available Options:

- **AES-128** - 128-bit key, 10 rounds
- **AES-192** - 192-bit key, 12 rounds
- **AES-256** - 256-bit key, 14 rounds

Questions:

1. Which key size would you choose and why?
2. What are the trade-offs between security and performance?
3. How does the number of rounds affect security?

✅ Solution: AES Key Sizes

Recommended Answer

1. Choose AES-256

- **Future-proof** - 256-bit keys provide 2^256 security level
- **Government standard** - Required for classified information
- **Long-term security** - Will remain secure for decades
- **Minimal performance cost** - Only 40% slower than AES-128

2. Trade-offs:

- **AES-128**: Fast, but may become vulnerable to quantum computers
- **AES-192**: Good middle ground, but not widely supported
- **AES-256**: Best security, slightly slower, widely supported

3. Round Impact:

- **More rounds** = More security through confusion and diffusion
- **AES-128**: 10 rounds (sufficient for 128-bit security)
- **AES-256**: 14 rounds (necessary for 256-bit security)
- **Each round** adds exponential complexity for attackers

Key Derivation Functions

Why Key Derivation?

- **Password-based** encryption
- **Key stretching** - Slow down brute force
- **Salt** - Prevent rainbow table attacks
- **Multiple keys** from one password

PBKDF2

- **Password-Based Key Derivation Function 2**
- **HMAC** as pseudorandom function
- **Configurable iterations** - 100,000+ recommended
- **Salt** - Random data to prevent attacks

Implementation

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import P
import secrets

def derive_key(password, salt=None, iterations=100000):
    """Derive key from password using PBKDF2"""
    if salt is None:
        salt = secrets.token_bytes(16)

    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,  # 256-bit key
        salt=salt,
        iterations=iterations,
    )
```

Practical Implementation

Complete Encryption Suite

Stream Cipher Class

```
class StreamCipher:
    def __init__(self, key):
        self.key = key
        self.position = 0

    def encrypt(self, plaintext):
        """Encrypt plaintext using stream cipher"""
        keystream = self._generate_keystream(len(plaint
        return bytes(a ^ b for a, b in zip(plaintext, k

    def decrypt(self, ciphertext):
        """Decrypt ciphertext (same as encryption)"""
        return self.encrypt(ciphertext)

    def _generate_keystream(self, length):
```

Block Cipher Class

```
class BlockCipher:
    def __init__(self, key, mode='CBC'):
        self.key = key
        self.mode = mode

    def encrypt(self, plaintext, iv=None):
        """Encrypt plaintext using block cipher"""
        if self.mode == 'ECB':
            return self._ecb_encrypt(plaintext)
        elif self.mode == 'CBC':
            return self._cbc_encrypt(plaintext, iv)
        elif self.mode == 'GCM':
            return self._gcm_encrypt(plaintext, iv)

    def decrypt(self, ciphertext, iv=None, tag=None):
```

Security Considerations

Key Management

- **Generate keys** using secure random
- **Store keys** securely (HSM, key vault)
- **Rotate keys** regularly
- **Use different keys** for different purposes

Common Mistakes

- ❌ **Reusing keys** across different contexts
- ❌ **Weak random** number generation
- ❌ **Predictable IVs** - Use random IVs
- ❌ **Not authenticating** - Use authenticated encryption

Implementation Security

- **Constant-time** operations
- **Avoid timing attacks** - Use constant-time comparisons
- **Clear sensitive data** from memory
- **Validate inputs** properly

Best Practices

- ✅ **Use established** libraries
- ✅ **Follow** security guidelines
- ✅ **Test thoroughly** for vulnerabilities
- ✅ **Keep libraries** updated

Practice Tasks

Task 1: Implement Stream Cipher

Requirements

Create a complete stream cipher implementation:

1. **LFSR-based** keystream generator
2. **RC4-style** cipher
3. **ChaCha20** implementation
4. **Interactive** testing interface

Features

- **Key generation** from passwords
- **File encryption/decryption**
- **Performance testing**
- **Security analysis**

Implementation Guide

```
# LFSR Stream Cipher
class LFSRCipher:
    def __init__(self, key):
        self.lfsr = LFSR(key, [3, 1])  # 4-bit LFSR

    def encrypt(self, data):
        keystream = self.lfsr.generate_keystream(len(da
        result = []

        for i, byte in enumerate(data):
            byte_bits = [(byte >> j) & 1 for j in range
            keystream_bits = keystream[i*8:(i+1)*8]

            encrypted_bits = [a ^ b for a, b in zip(byt
            encrypted_byte = sum(bit << j for j, bit in
```

Task 2: Block Cipher Modes

Requirements

Implement different block cipher modes:

1. **ECB mode** - Educational purposes
2. **CBC mode** - Secure encryption
3. **CTR mode** - Counter mode
4. **GCM mode** - Authenticated encryption

Features

- **AES implementation** using libraries
- **Padding schemes** (PKCS7)
- **IV generation** and management
- **Authentication** for GCM mode

Implementation Guide

```
# CBC Mode Implementation
def cbc_encrypt(plaintext, key, iv):
    """CBC mode encryption"""
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b''
    prev_block = iv

    for i in range(0, len(plaintext), 16):
        block = plaintext[i:i+16]
        if len(block) < 16:
            block = pad(block, 16)

        # XOR with previous ciphertext
        xored = bytes(a ^ b for a, b in zip(block, prev
        encrypted = cipher.encrypt(xored)
```

Task 3: Security Analysis

Requirements

Analyze security of different ciphers:

1. **Frequency analysis** on stream ciphers
2. **Pattern detection** in ECB mode
3. **Timing attacks** on implementations
4. **Key recovery** from weak implementations

Tools

- **Statistical analysis** - Letter frequencies
- **Pattern recognition** - Block patterns
- **Timing measurements** - Execution time analysis
- **Cryptanalysis** - Breaking weak ciphers

Analysis Framework

```
class SecurityAnalyzer:
    def __init__(self):
        self.english_freq = self._load_english_frequen

    def analyze_stream_cipher(self, ciphertext):
        """Analyze stream cipher security"""
        # Frequency analysis
        freq = self._frequency_analysis(ciphertext)

        # Chi-squared test
        chi_squared = self._chi_squared_test(freq)

        # Autocorrelation
        autocorr = self._autocorrelation(ciphertext)
```

Task 4: Performance Testing

Requirements

Compare performance of different ciphers:

1. **Speed testing** - Encryption/decryption speed
2. **Memory usage** - RAM consumption
3. **Throughput** - Data processed per second
4. **Scalability** - Performance with large files

Metrics

- **Encryption speed** - MB/s
- **Memory usage** - Peak RAM
- **CPU usage** - Processor utilization
- **Latency** - Time per operation

Performance Testing Framework

```python
import time
import psutil
import os

class PerformanceTester:
    def __init__(self):
        self.results = {}

    def test_cipher_performance(self, cipher_class, dat
        """Test cipher performance across different dat
        results = {}

        for size_mb in data_sizes:
            # Generate test data
            test_data = os.urandom(size_mb * 1024 * 10
```

Common Vulnerabilities

Stream Cipher Issues

- **Weak key scheduling** - RC4 vulnerabilities
- **Key reuse** - Same keystream for different messages
- **Predictable IVs** - Weak initialization
- **Linear feedback** - LFSR predictability

Block Cipher Issues

- **ECB mode** - Pattern leakage
- **Weak padding** - Padding oracle attacks
- **IV reuse** - CBC mode vulnerabilities
- **Timing attacks** - Implementation flaws

Implementation Mistakes

- ❌ **Hardcoded keys** - Never embed secrets
- ❌ **Weak randomness** - Use secure random
- ❌ **Reusing IVs** - Generate random IVs
- ❌ **Not authenticating** - Use authenticated encryption

Best Practices

- ✅ **Use established** libraries
- ✅ **Follow** security guidelines
- ✅ **Test thoroughly** for vulnerabilities
- ✅ **Keep libraries** updated

Real-World Applications

Stream Ciphers

- **TLS/SSL** - ChaCha20 in TLS 1.3
- **WiFi** - WPA3 uses ChaCha20
- **VPN** - WireGuard uses ChaCha20
- **Disk encryption** - Some implementations

Block Ciphers

- **TLS/SSL** - AES in all versions
- **Disk encryption** - BitLocker, FileVault
- **Database encryption** - Transparent Data Encryption
- **File encryption** - PGP, S/MIME

Modern Trends

- **Authenticated encryption** - GCM mode preferred
- **Post-quantum** - Preparing for quantum computers
- **Hardware acceleration** - AES-NI instructions
- **Cloud security** - Key management services

Future Directions

- **Quantum-resistant** algorithms
- **Homomorphic encryption** - Computation on encrypted data
- **Zero-knowledge** proofs - Prove without revealing
- **Secure multi-party** computation

Questions?

Let's discuss stream ciphers! 💬

Next Week: We'll explore public key cryptography and learn about RSA and elliptic curves!

Assignment: Implement and test different stream and block ciphers to understand their strengths and weaknesses!