CVS8301:Cryptography Theory and Applications - Lecture 3

Habeebah Adamu Kakudi (Mrs) PhD

2025-08-24

Table of contents

1 Modern Symmetric Encryption

Focus: Block Ciphers and Stream Ciphers

Modern symmetric encryption is a cornerstone of secure digital communication. Unlike classical ciphers, which operate on individual characters, modern symmetric algorithms process data in structured units—either fixed-size blocks or continuous streams. These methods are fast, efficient, and widely used in applications ranging from secure messaging to file encryption and network protocols. Symmetric encryption uses the **same key** for both encryption and decryption, making it suitable for high-speed, bulk data operations. However, its security depends heavily on how the encryption is applied—especially in block cipher modes. Two major types of symmetric ciphers are:

- **Block Ciphers**
- **Stream Ciphers**

Both aim to ensure confidentiality, but they differ in how they process data.

1.1 Block Ciphers

Block ciphers encrypt data in fixed-size blocks (e.g., 64 or 128 bits) using a symmetric key.

Block ciphers are a fundamental building block of modern symmetric encryption. Unlike stream ciphers that process data bit-by-bit or byte-by-byte, block ciphers operate on **fixed-size chunks** of data, transforming them into ciphertext using a shared secret key. They are widely used in securing digital communications, file encryption, and cryptographic protocols.

1.1.1 Definition

A **block cipher** is a symmetric key cipher that encrypts plaintext in fixed-size blocks (e.g., 64 or 128 bits). Each block is transformed into ciphertext using a deterministic algorithm and a secret key.

1.1.2 Formal Representation:

- **Plaintext Block**: (m) - The input data to be encrypted.
- **Ciphertext Block**: (c) - The encrypted output.
- **Encryption Function**: $(c = e_k(m))$
- **Decryption Function**: $(m = d_k(c))$
- **Key**: (k), shared secret between sender and receiver for both encryption and decryption

The same key is used for both encryption and decryption, making it symmetric.

1.1.3 Key Concepts

- **Deterministic Transformation**: For a given key and plaintext block, the ciphertext is always the same.
- **Fixed Block Size**: Common sizes include 64 bits (DES) and 128 bits (AES).
- **Key Length**: Determines the strength of the cipher; longer keys offer better security.
- **Round Functions**: Encryption is performed in multiple rounds, each applying transformations to increase diffusion and confusion.

1.1.4 Characteristics:

- Operate on fixed-length blocks
- Require padding for messages shorter than block size
- Support multiple modes of operation
- Examples: AES, DES

1.1.5 Internal Structures

Block ciphers typically follow one of two design paradigms: **Feistel Network** or **Substitution-Permutation Network (SPN)**.

1. **Feistel Network**

- Splits the block into two halves: Left (L) and Right (R).
- Applies a round function to one half and combines it with the other using XOR.
- Swaps halves after each round.
- Used in **DES**, **Blowfish**, **Triple DES**.

2. **Substitution-Permutation Network (SPN)**

- Applies layers of **substitution** (S-boxes) and **permutation** (P-boxes).
- Designed for strong diffusion and confusion.
- Used in **AES**, **Rijndael**, **Serpent**.

These structures ensure that small changes in the plaintext or key produce vastly different ciphertexts—a property known as the **avalanche effect**.

1.1.6 Modes of Operation

Block ciphers alone can only encrypt one block at a time. Block ciphers require modes to handle data larger than one block. To handle larger messages, **modes of operation** are used:

Mode	Description	Strengths	Weaknesses
ECB (Electronic Codebook)	Encrypts each block independently	Simple, parallelizable	Identical plaintext blocks produce identical ciphertext blocks—reveals patterns
CBC (Cipher Block Chaining)	XORs each plaintext block with the previous ciphertext block	Hides patterns, good for file encryption	Requires sequential processing, error propagation
CFB (Cipher Feedback)	Converts block cipher into stream-like behavior	Self-synchronizing, suitable for streaming	Slower than CTR
OFB (Output Feedback)	Uses keystream independent of plaintext	No error propagation	Vulnerable to IV reuse
CTR (Counter Mode)	Encrypts a counter value to produce keystream	Fast, parallelizable, stream-like	Requires unique nonce/counter per encryption

These modes allow block ciphers to encrypt data of arbitrary length securely and efficiently.

1.1.7 Examples of Block Ciphers

1. **DES (Data Encryption Standard)**

- Developed in the 1970s.
- 64-bit block size, 56-bit key.
- Uses 16 Feistel rounds.
- Now considered **insecure** due to short key length and vulnerability to brute-force attacks.

2. **AES (Advanced Encryption Standard)**

- Adopted as a U.S. federal standard in 2001.
- 128-bit block size.
- Key sizes: 128, 192, or 256 bits.
- Uses 10, 12, or 14 rounds depending on key size.
- Based on SPN structure.
- Widely used in TLS, VPNs, disk encryption, and secure messaging.

Block ciphers encrypt data in fixed-size blocks (commonly 64 or 128 bits). Each block is processed independently or in combination with others, depending on the mode of operation. Block ciphers are essential tools in symmetric encryption, offering robust security when used with proper modes of operation. They provide confidentiality for fixed-size data blocks and can be extended to larger datasets using chaining techniques. Understanding their structure and behavior is key to mastering modern cryptographic systems.

1.1.8 Data Encryption Standard (DES)

The Origins and Legacy of the Data Encryption Standard (DES)

In 1972, the U.S. National Bureau of Standards (NBS)—now known as the National Institute of Standards and Technology (NIST)—initiated a groundbreaking move in the field of cryptography. For the first time, a government agency publicly called for proposals to establish a standardized encryption algorithm that could be used across various sectors. This marked a significant departure from the traditional secrecy surrounding cryptographic research, which had long been considered a matter of national security and confined to military and intelligence circles.

The push for a public standard was driven by the growing demand for secure communication in commercial domains, particularly banking and finance. By the early 1970s, the need for robust encryption in civilian applications had become too urgent to ignore, prompting NBS to seek a solution that could serve both public and private interests.

In 1974, IBM responded with a promising candidate based on its earlier cipher known as Lucifer. Developed by Horst Feistel in the late 1960s, Lucifer was among the first block ciphers designed to operate on digital data. It featured a Feistel structure, encrypting 64-bit blocks using a 128-bit key. Recognizing the potential of IBM's submission, NBS enlisted the expertise of the National Security Agency (NSA) to evaluate its security. At the time, the NSA operated with a high level of secrecy and had not publicly acknowledged its involvement in cryptographic standards.

The NSA's influence on the development of the final cipher was substantial. The algorithm was renamed the Data Encryption Standard (DES), and several modifications were introduced. Notably, DES was engineered to resist differential cryptanalysis—a sophisticated attack method that was not publicly known until 1990. Whether IBM independently discovered this technique or was guided by the NSA remains unclear. Another significant change was the reduction of the key size from 128 bits to 56 bits, a decision that raised concerns about the cipher's vulnerability to brute-force attacks.

This reduction in key length, along with the NSA's involvement, sparked widespread speculation. Critics feared the possibility of a hidden backdoor—an undisclosed mathematical flaw that could allow the NSA to break the cipher. Others worried that the shortened key made DES susceptible to exhaustive key searches, especially by entities with access to advanced computing resources. Over time, however, many of these fears proved to be unfounded.

Despite the controversy, NBS officially released the specifications for DES in 1977 under the designation FIPS PUB 46. The standard provided a detailed description of the algorithm down to the bit level, although the rationale behind certain design choices—particularly the selection of substitution boxes (S-boxes)—was never disclosed. As personal computing became more widespread in the 1980s, the cryptographic community gained greater access to DES and began to analyze its internal mechanisms. This period saw a surge in civilian cryptographic research, and DES was subjected to intense scrutiny. Yet, until 1990, no major vulnerabilities were discovered.

Originally intended to serve as a federal standard for a decade, DES remained in use well beyond its initial expiration in 1987. Due to its widespread adoption and the absence of critical security flaws, NIST continued to endorse DES until 1999. That year marked the end of its official tenure, as DES was formally replaced by the more advanced and secure **Advanced Encryption Standard (AES)**—a cipher designed to meet the evolving demands of digital security in the 21st century.

The **Data Encryption Standard (DES)** is one of the earliest symmetric-key block ciphers and was a cornerstone of cryptographic systems for decades. Developed in the

1970s by IBM and adopted by the U.S. National Institute of Standards and Technology (NIST), DES was widely used in banking, government, and commercial applications.

Despite its historical significance, DES is now considered **insecure** due to its short key length and susceptibility to brute-force attacks. However, understanding DES is essential for grasping the evolution of modern cryptography.

1.1.8.1 Key Features of DES

Feature	Value
Block Size	64 bits
Key Size	56 bits (plus 8 parity bits)
Rounds	16
Structure	Feistel Network
Status	Deprecated

1.1.8.2 Internal Structure of DES

DES operates on a 64-bit block of plaintext and transforms it into a 64-bit ciphertext using a 56-bit key. The encryption process involves:

1. **Initial Permutation (IP)**

- Rearranges the bits of the plaintext block.

2. **Feistel Rounds (16 rounds)** Each round includes:

- Expansion of 32-bit half to 48 bits.
- XOR with round key.
- Substitution using 8 S-boxes.
- Permutation (P-box).
- XOR with the other half.

3. **Final Permutation (IP ^{1})**

- Inverse of the initial permutation.

Feistel Structure

- Splits the block into Left (L) and Right (R) halves.
- In each round:
  - ($L_{i+1} = R_i$)
  - ($R_{i+1} = L_i \oplus F(R_i, K_i)$)

1.1.8.3 Key Schedule

- DES generates 16 **round keys** from the original 56-bit key.
- Each round key is 48 bits.
- Uses **permuted choice** and **left shifts** to derive keys.

1.1.8.4 Security Limitations

- **Short Key Length**: 56-bit keys are vulnerable to brute-force attacks.
- **Known Attacks**:
  - **Brute-force**: Can be cracked in hours using modern hardware.
  - **Differential Cryptanalysis**
  - **Linear Cryptanalysis**
- **Successor**: **Triple DES (3DES)** and later **AES** replaced DES in secure systems.

1.1.8.5 Manual Exercise

Encrypt the 64-bit plaintext `0x0123456789ABCDEF` using the 56-bit key `0x133457799BBCDFF1`.

Steps:

1. Convert plaintext and key to binary.

Break the Hex into Individual Digits

Hexadecimal: `0x0123456789ABCDEF`
Split into: `0 1 2 3 4 5 6 7 8 9 A B C D E F`

Converting Each Hex Digit to 4-bit Binary

Hex	Binary	Hex	Binary
0	0000	9	1001
1	0001	A	1010
2	0010	B	1011
3	0011	C	1100
4	0100	D	1101
5	0101	E	1110
6	0110	F	1111
7	0111		
8	1000		

2. Apply initial permutation.
3. Split into L and R halves.
4. Generate round keys.
5. Perform 16 Feistel rounds.
6. Apply final permutation.
7. Convert result to hexadecimal.

   Tip: Use a DES diagram to trace each round manually. This helps visualize how bits are transformed.

1.1.8.6 DES Encryption Step-by-Step

Step 1: Convert Plaintext and Key to Binary

- **Plaintext**: `0x0123456789ABCDEF` → Binary:

  ```
  00000001 00100011 01000101 01100111 10001001 10101100 11001101
  11101111
  ```

- **Key**: `0x133457799BBCDFF1` → Binary:

  ```
  00010011 00110100 01010111 01111001 10011011 10111100 11011111
  11110001
  ```

  (Only 56 bits are used; 8 bits are parity bits and discarded in key scheduling.)

Step 2: Apply Initial Permutation (IP)

DES uses a fixed initial permutation table. Applying IP to the plaintext rearranges the bits. This step is tedious manually, but conceptually:

- Rearrange the 64 bits using the IP table.
- Result: 64-bit permuted block.

Step 3: Split 64-bit permuted block into L and R Halves

After IP:

- **Left Half (L0)**: First 32 bits
- **Right Half (R0)**: Last 32 bits

These halves will be processed through 16 rounds.

Step 4: Generate Round Keys

DES generates **16 round keys**, each 48 bits, from the original 56-bit key using:

- **Permuted Choice 1 (PC-1)**: Drops parity bits.
- **Left shifts**: Vary per round.
- **Permuted Choice 2 (PC-2)**: Selects 48 bits for each round.

This process is complex manually but handled automatically in code.

Step 5: Perform 16 Feistel Rounds $X_2 := X_1 \oplus RK_1$ $X_3 := X_2 \oplus RK_2$ Each round:

- Expand R to 48 bits.
- XOR with round key.
- Substitute using 8 S-boxes.
- Permute using P-box.
- XOR with L.
- Swap halves.

After 16 rounds: - Combine final R and L (note: no swap in final round).

Step 6: Apply Final Permutation (IP ¹)

Apply the inverse of the initial permutation to the combined block to get the final ciphertext.

Step 7: Convert Result to Hexadecimal

Let's do this using Python to get the actual ciphertext.

Python Code Using PyCryptodome Package

```python
from Crypto.Cipher import DES

# Step 1: Define key and plaintext
key = bytes.fromhex('133457799BBCDFF1')
plaintext = bytes.fromhex('0123456789ABCDEF')

# Step 2: Create DES cipher in ECB mode
cipher = DES.new(key, DES.MODE_ECB)

# Step 3: Encrypt
ciphertext = cipher.encrypt(plaintext)

# Step 4: Output result
print("Ciphertext (hex):", ciphertext.hex().upper())
```

Output

```
Ciphertext (hex): 85E813540F0AB405
```

Python Code Example 2:Encryption and Decryption

You can use the `pycryptodome` library to implement DES in Python.

1. Installation

```
pip install pycryptodome
```

2. Code

```python
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

# 8-byte key (56 bits + 8 parity bits)
key = b'12345678'
cipher = DES.new(key, DES.MODE_ECB)

# Plaintext (must be multiple of 8 bytes)
plaintext = b'HelloDES'
padded_text = pad(plaintext, DES.block_size)

# Encrypt
ciphertext = cipher.encrypt(padded_text)
print("Ciphertext (hex):", ciphertext.hex())

# Decrypt
decrypted = unpad(cipher.decrypt(ciphertext), DES.block_size)
print("Decrypted:", decrypted.decode())
```

3. Output

```
Ciphertext (hex): e.g. 8b8a3b9c2e3f1a7d
Decrypted: HelloDES
```

- DES transforms the plaintext `0x0123456789ABCDEF` into ciphertext `0x85E813540F0AB405` using the key `0x133457799BBCDFF1`.
- The process involves permutations, key scheduling, and 16 rounds of Feistel operations.
- Python simplifies the implementation while preserving the cryptographic logic.

1.1.8.7 Summary

- DES was a pioneering block cipher that introduced key cryptographic concepts.
- Its Feistel structure and round-based design influenced many later ciphers.
- Though insecure today, DES remains a valuable teaching tool.
- Manual exercises and code help reinforce understanding of its internal mechanics.

1.1.9 Advance Encryption Standard (AES)

The Evolution from DES to AES: A Cryptographic Milestone

By the late 1990s, the limitations of the Data Encryption Standard (DES) had become increasingly apparent. Originally developed by IBM and adopted by the U.S. National Institute of Standards and Technology (NIST) in the 1970s, DES had served as a foundational block cipher for decades. However, its 56-bit key length rendered it vulnerable to brute-force attacks, especially as computational power advanced. In 1999, NIST formally recommended that DES be reserved for legacy systems and that Triple DES (3DES) be used as a transitional solution.

While 3DES offered improved resistance to brute-force attacks by applying DES encryption three times with separate keys, it was far from ideal. The algorithm suffered from significant inefficiencies, particularly in software implementations. DES itself was not optimized for software, and 3DES, being three times slower, compounded this issue. Moreover, the 64-bit block size of DES and 3DES posed limitations in modern cryptographic applications, such as constructing secure hash functions. These constraints, coupled with emerging concerns about the future threat of quantum computing—which could render current key lengths insufficient—prompted NIST to seek a more robust and forward-looking encryption standard.

In 1997, NIST initiated an open call for proposals to develop a new block cipher that would become the Advanced Encryption Standard (AES). Unlike the closed development process of DES, the AES selection was transparent and collaborative, involving extensive input from the global cryptographic community. Over the course of three rigorous evaluation rounds, experts assessed the security, performance, and implementation feasibility of the submitted algorithms.

The process culminated in 2001 with the selection of Rijndael, a cipher designed by Belgian cryptographers Vincent Rijmen and Joan Daemen. Rijndael was officially published as AES under FIPS Publication 197. Its design addressed the shortcomings of DES and 3DES, offering a modern substitution-permutation network structure, a fixed block size of 128 bits, and support for three key lengths: 128, 192, and 256 bits. These specifications ensured both flexibility and resilience against current and anticipated cryptographic threats.

The criteria for AES candidates were clearly defined: each submission had to be a block cipher with a 128-bit block size, support key lengths of 128, 192, and 256 bits, demonstrate security strength relative to other contenders, and exhibit efficiency in both software and hardware environments. Rijndael met and exceeded these requirements, marking a significant advancement in the field of symmetric encryption and setting a new global standard for secure digital communication.

AES is designed for speed, security, and flexibility. It supports multiple key lengths, resists known cryptanalytic attacks, and is efficient in both hardware and software environments.

1.1.9.1 Key Features of AES

Feature	Value
Block Size	128 bits
Key Sizes	128, 192, or 256 bits
Rounds	10 (AES-128), 12 (AES-192), 14 (AES-256)
Structure	Substitution-Permutation Network (SPN)
Status	Active and Secure

1.1.9.2 AES vs. DES: Structural Differences

Unlike its predecessor, the **Data Encryption Standard (DES)**, AES does not use a **Feistel structure**. In a Feistel network, only half of the data block is processed in each round, with the other half being modified indirectly. For example, DES encrypts 32 bits of its 64-bit block per round. AES, in contrast, processes the entire 128-bit block in every round. This full-block processing contributes to AES's efficiency and allows it to achieve strong security with fewer rounds.

1.1.9.3 Key Schedule and Round Key Generation

AES uses a **key schedule algorithm** to generate a series of round keys from the original key. These keys are denoted as $(k_0, k_1, \dots, k_{nr})$, where (nr) is the number of rounds. The key schedule involves operations such as byte substitution, rotation, and XOR with round constants. This process ensures that each round key is unique and cryptographically strong.

1.1.9.4 Mathematical Foundations: Galois Fields

To fully understand AES, one must be familiar with **Galois field arithmetic**, particularly operations in the finite field ($GF(2^8)$). AES performs many of its internal transformations—especially in the MixColumns and S-box layers—using arithmetic over this field. Galois fields allow for well-defined addition and multiplication operations on byte-level data, which are essential for achieving the desired cryptographic properties of confusion and diffusion.

1.1.9.5 Internal Structure of AES

AES operates on a 128-bit block of plaintext and transforms it into ciphertext using a series of well-defined operations. The core data structure is a **4×4 byte matrix** called the **state**. Each round of AES (except the first and last) applies a series of transformations to this state. These transformations are organized into **three distinct layers**, each serving a specific cryptographic purpose:

1. Key Addition Layer

- In this layer, a **round key** (also called a subkey) is XORed ($\oplus$) with the current state.
- The round key is derived from the original encryption key through a process known as the **key schedule**.
- This operation introduces key-dependent variability into the encryption process.

2. Byte Substitution Layer (S-Box)

- Each byte in the state is replaced using a **substitution box (S-box)**—a nonlinear lookup table designed with strong mathematical properties.
- This layer introduces **confusion**, meaning it obscures the relationship between the plaintext and ciphertext.
- The S-box ensures that even small changes in the input lead to significant changes in the output, enhancing security.

3. Diffusion Layer This layer spreads the influence of each input byte across the entire state. It consists of two sublayers:

- **ShiftRows**: Each row of the state matrix is cyclically shifted by a different number of bytes. This rearrangement disrupts the byte alignment and contributes to diffusion.
- **MixColumns**: Each column of the state is transformed using a **matrix multiplication** over a finite field (specifically, a Galois field). This mixes the bytes within each column, further enhancing diffusion.

It's important to note that the **final round of AES omits the MixColumns step**, making the encryption and decryption processes symmetric in structure.

1.1.9.6 AES Encryption Steps

1. **Key Expansion**
    - Derives round keys from the original key using Rijndael's key schedule.
    - Each round key is 128 bits.
    - AES uses a deterministic key schedule to generate round keys:
        - **Rcon** constants are used for non-linearity.
        - **RotWord** and **SubWord** operations are applied.
        - Round keys are derived from previous words using XOR and substitution.
2. **Initial Round**
    - **AddRoundKey**: XOR the state with the first round key.
3. **Main Rounds (9, 11, or 13 rounds depending on key size)**
    - **SubBytes**: Non-linear substitution using S-box.
        - Each byte in the state is replaced using a fixed 16×16 S-box.
        - Provides non-linearity and confusion.
    - **ShiftRows**: Cyclically shift rows of the state.
        - Row 0: No shift
        - Row 1: Left shift by 1 byte
        - Row 2: Left shift by 2 bytes
        - Row 3: Left shift by 3 bytes
        - Enhances diffusion across the state.
    - **MixColumns**: Mix data within each column using matrix multiplication in ^8F(2^8).
        - Each column is transformed using matrix multiplication over a finite field.
        - Spreads byte influence across the column.
    - **AddRoundKey**: XOR with the round key.

    - XORs the current state with the round key.
    - Introduces key-dependent transformation.

4. **Final Round**
    - SubBytes
    - ShiftRows
    - AddRoundKey (no MixColumns)

1.1.9.7 Security Strengths

- **Large Key Sizes**: AES-256 offers 2^{256} possible keys.
- **Resistance to Known Attacks**:
    - No practical differential or linear cryptanalysis
    - No known structural weaknesses
- **Efficient Implementation**:
    - Fast in both hardware and software
    - Supported by modern CPUs with AES-NI instructions

1.1.9.8 Limitations

- AES is vulnerable to **side-channel attacks** (e.g., timing, power analysis) if not implemented securely.
- ECB mode (Electronic Codebook) is insecure for patterns—use CBC, GCM, or CTR instead.

1.1.9.9 Manual Exercise

Encrypt the 128-bit plaintext `0x00112233445566778899AABBCCDDEEFF` using the 128-bit key `0x000102030405060708090A0B0C0D0E0F`.

Step-by-Step Solution Breakdown:

1. Convert plaintext and key to 4×4 byte matrices.
2. Apply initial AddRoundKey.
3. Perform 9 rounds of SubBytes, ShiftRows, MixColumns, AddRoundKey.
4. Final round: SubBytes, ShiftRows, AddRoundKey.
5. Convert final state to hexadecimal ciphertext.

1.1.9.10 Python Code Example (Using PyCryptodome)

1. Installation

```
pip install pycryptodome
```

2. Code

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

# 16-byte key (AES-128)
key = bytes.fromhex('000102030405060708090A0B0C0D0E0F')
cipher = AES.new(key, AES.MODE_ECB)

# 16-byte plaintext
plaintext = bytes.fromhex('00112233445566778899AABBCCDDEEFF')

# Encrypt
ciphertext = cipher.encrypt(plaintext)
print("Ciphertext (hex):", ciphertext.hex().upper())

# Decrypt
decrypted = cipher.decrypt(ciphertext)
print("Decrypted (hex):", decrypted.hex().upper())
```

3. Output

```
Ciphertext (hex): 69C4E0D86A7B0430D8CDB78070B4C55A
Decrypted (hex): 00112233445566778899AABBCCDDEEFF
```

1.1.9.11 Applications of AES

Domain	Use Case
Web Security	HTTPS encryption (TLS)
File Encryption	BitLocker, VeraCrypt
Mobile Security	Android full-disk encryption
Cloud Storage	AWS S3, Azure Blob encryption
Messaging Apps	Signal, WhatsApp

Domain	Use Case
VPNs	Secure tunneling protocols

1.1.9.12 Summary

AES represents a significant advancement in block cipher design. Its layered structure, full-block processing, and mathematically grounded transformations make it both secure and efficient. By supporting multiple key lengths and operating on a fixed 128-bit block size, AES offers flexibility and scalability for a wide range of applications—from securing web traffic to encrypting sensitive data in mobile devices and cloud environments.

Understanding the internal mechanics of AES—especially its use of substitution, permutation, and finite field arithmetic—is crucial for students and professionals aiming to master modern cryptography. In the next section, we will explore the detailed implementation of each AES layer and examine how these components work together to produce secure ciphertext.

- AES is the gold standard for symmetric encryption.
- Its SPN structure offers strong security and performance.
- Understanding AES is essential for modern cryptographic systems.
- Practical implementation via libraries like PyCryptodome reinforces theoretical knowledge.

1.1.9.13 Handling Multiple Blocks in AES

AES is a **block cipher**, meaning it encrypts data in fixed-size blocks—specifically **128 bits**. However, AES by itself only defines how to encrypt a single block. To encrypt larger messages, we need a **mode of operation** that tells AES how to handle multiple blocks. That's where modes of operations like **ECB** comes in.

Electronic Codebook (ECB) is one of the five standard modes of operation for block ciphers like AES. In ECB mode:

- The plaintext is divided into fixed-size blocks (128 bits for AES).
- Each block is encrypted **independently** using the same key.
- The same plaintext block will always produce the same ciphertext block.

ECB as a Mode of AES

- **ECB (Electronic Codebook)** is the **simplest mode** of operation for AES.

- It divides plaintext into 128-bit blocks and encrypts **each block independently** using AES and the same key.
- ECB is one of the five standard modes defined by NIST for block ciphers like AES.

How AES Works in ECB Mode

Here's the step-by-step process:

1. **Divide the plaintext** into 128-bit blocks.
2. **Encrypt each block** separately using the AES algorithm and the same key.
3. **Concatenate the ciphertext blocks** to form the final encrypted message.

There's **no chaining or feedback** between blocks, which makes ECB easy to implement and fast to execute.

Security Implications

While ECB is straightforward, it has **major security flaws**:

- **Pattern leakage**: Identical plaintext blocks produce identical ciphertext blocks. This reveals structure in the data.
- **No diffusion across blocks**: A change in one block doesn't affect others.
- **Vulnerable to replay and substitution attacks**: Attackers can manipulate ciphertext blocks without breaking the encryption.

A famous example is the "ECB penguin" image, where encrypting a bitmap image in ECB mode preserves the visual outline—proving that ECB doesn't hide data patterns effectively.

When Is ECB Acceptable?

Despite its weaknesses, ECB can be used safely in **very limited scenarios**, such as:

- Encrypting **short, random values** (e.g., keys or nonces)
- When **data patterns don't matter** and performance is critical
- In **testing or educational contexts**

For most real-world applications, **CBC**, **CTR**, or **GCM** modes are preferred due to their stronger security properties.

1.1.9.14 Python Example Using ECB Mode

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

key = b'ThisIsA16ByteKey'  # 128-bit key
plaintext = b'Confidential Data Block'
cipher = AES.new(key, AES.MODE_ECB)

# Pad plaintext to 16-byte boundary
padded = pad(plaintext, AES.block_size)

# Encrypt
ciphertext = cipher.encrypt(padded)
print("Ciphertext:", ciphertext.hex())

# Decrypt
decrypted = unpad(cipher.decrypt(ciphertext), AES.block_size)
print("Decrypted:", decrypted.decode())
```

1.1.9.15 AES in CBC Mode (Cipher Block Chaining)

Why ECB Falls Short

While AES is a powerful encryption algorithm, its effectiveness depends heavily on the mode of operation used. ECB mode, though simple, is rarely suitable for encrypting structured or repetitive data due to its lack of pattern resistance and integrity checks. More advanced modes like CBC, CTR, and GCM offer stronger security guarantees and are preferred in modern cryptographic systems.

Before we compare, let's quickly recap ECB's main flaw:

- **Deterministic encryption**: Identical plaintext blocks produce identical ciphertext blocks.
- **Pattern leakage**: This can expose structure in the data (e.g., images, repeated headers).
- **No integrity check**: ECB only encrypts—there's no way to verify if data was tampered with.

Now, let's see how CBC and GCM address these issues.

A. How CBC Works:

- Each plaintext block is **XORed with the previous ciphertext block** before encryption.
- The first block uses a **random Initialization Vector (IV)**.
- Encryption becomes **dependent on previous blocks**, creating a chain.

B. Improvements Over ECB:

- **Pattern resistance**: Even identical plaintext blocks produce different ciphertexts due to chaining.
- **Randomness via IV**: Prevents predictable outputs.
- **Error propagation**: A change in one block affects subsequent blocks—good for tamper detection.

C. Key Concepts:

- Requires a **16-byte Initialization Vector (IV)**
- Needs **padding** since AES works on fixed-size blocks
- Provides **confidentiality**, but **no authentication**

D. CBC Mode Example:

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

# Generate a 16-byte AES key and IV
key = get_random_bytes(16)  # AES-128
iv = get_random_bytes(16)

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

# Encrypt
plaintext = b'This is a secret message.'
padded_data = pad(plaintext, AES.block_size)
ciphertext = cipher.encrypt(padded_data)

print("CBC Ciphertext (hex):", ciphertext.hex())

# Decrypt
decipher = AES.new(key, AES.MODE_CBC, iv)
decrypted = unpad(decipher.decrypt(ciphertext), AES.block_size)
```

```
print("CBC Decrypted:", decrypted.decode())
```

E. Limitations of ECB

- **No built-in authentication**: You still need a separate MAC (Message Authentication Code).
- **Sequential processing**: Encryption must be done in order—no parallelism.

1.1.9.16 AES in GCM Mode (Galois/Counter Mode)

A. How GCM Works:

- GCM is built on **CTR (Counter) mode**, which turns AES into a stream cipher.
- Each block is XORed with a keystream derived from a counter and key.
- GCM adds **authentication** using Galois field multiplication to produce a tag.

B. Improvements Over ECB and CBC:

- **Confidentiality + Integrity**: GCM encrypts and authenticates in one pass.
- **Parallelizable**: Both encryption and authentication can be done in parallel—great for performance.
- **Tamper detection**: If ciphertext is altered, the authentication tag fails.

C. Key Concepts:

- Uses a **nonce** (recommended: 12 bytes)
- Provides **confidentiality + authentication**
- Generates an **authentication tag** to verify integrity

D. GCM Mode Example:

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

# Generate a 16-byte AES key and 12-byte nonce
key = get_random_bytes(16)  # AES-128
nonce = get_random_bytes(12)

# Create AES cipher in GCM mode
cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
```

```
# Encrypt and generate authentication tag
plaintext = b'This is a secure message.'
ciphertext, tag = cipher.encrypt_and_digest(plaintext)

print("GCM Ciphertext (hex):", ciphertext.hex())
print("GCM Tag (hex):", tag.hex())

# Decrypt and verify
decipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
decrypted = decipher.decrypt_and_verify(ciphertext, tag)

print("GCM Decrypted:", decrypted.decode())
```

E. Considerations:

- **IV management is critical**: Reusing IVs in GCM can break security.
- **More complex implementation**: Requires careful handling of nonce, tag, and key.

1.1.10 Summary Comparison

Feature	ECB	CBC	GCM
Pattern Resistance	No	Yes	Yes
Integrity Protection	No	No (MAC needed)	Yes (built-in)
Parallel Processing	Yes	No	Yes
IV Required	No	Yes	Yes
Suitable for Images/Data	No	Yes	Yes
Performance	Fast	Moderate	Fast (with hardware)

1.2 Stream Ciphers

Stream ciphers are a fundamental concept in cryptography, especially useful for encrypting data in real-time applications such as voice communication or streaming media. Unlike block ciphers, which encrypt fixed-size blocks of data, stream ciphers operate on individual bits of plaintext. This bit-by-bit approach offers simplicity and speed, making stream ciphers ideal for scenarios where performance is critical.

1.2.1 How Does Bit-Level Encryption Work?

At the heart of stream cipher encryption lies a surprisingly simple mathematical operation: **modulo 2 addition**, also known as the **exclusive OR (XOR)** operation. Each plaintext bit (x_i) is combined with a corresponding bit (s_i) from a secret key stream. The result is the ciphertext bit (y_i), calculated as:

$$y_i = x_i + s_i \mod 2$$

This operation ensures that if ($s_i = 0$), the plaintext bit remains unchanged, and if ($s_i = 1$), the plaintext bit is flipped. Decryption uses the exact same operation:

$$x_i = y_i + s_i \mod 2$$

This symmetry is one of the elegant features of stream ciphers—**encryption and decryption are performed using the same function**. The XOR operation is its own inverse, meaning applying it twice with the same key stream bit restores the original plaintext.

To fully grasp the mechanics and implications of stream cipher encryption, let's explore three essential points:

1.Encryption and Decryption Are Identical

Unlike many cryptographic systems that use distinct algorithms for encryption and decryption, stream ciphers rely on the same operation—XOR with the key stream. This not only simplifies implementation but also reduces computational overhead. The same logic circuit or software function can be used for both processes, making stream ciphers highly efficient.

2. Why Modulo 2 Addition Works as Encryption

Modulo 2 addition (XOR) is a powerful tool in cryptography because of its reversible nature. When a bit is XORed with another bit, the result is either the same or flipped, depending on the value of the second bit. Importantly, XORing the result again with the same bit restores the original value. This property makes XOR ideal for encryption: it obfuscates the data while allowing easy recovery with the same key stream.

3.The Nature of the Key Stream Bits (s_i)

The security of a stream cipher hinges entirely on the key stream (s_i). These bits must be:

- **Secret**: Known only to the sender and receiver.
- **Random or Pseudorandom**: Ideally unpredictable to an attacker.
- **Synchronized**: Both parties must use the same key stream sequence.

In practice, the key stream is often generated by a **pseudorandom number generator (PRNG)** seeded with a secret key. If the key stream is truly random and never reused (a concept known as a **one-time pad**), the encryption is theoretically unbreakable. However, reusing key streams or using weak PRNGs can lead to vulnerabilities.

In diagrams, the XOR operation is typically represented by a circle with a plus sign inside $\oplus$, indicating modulo 2 addition. This visual cue helps distinguish it from regular arithmetic addition and reinforces the idea that the operation is binary and reversible.

Stream ciphers offer a clean and efficient method for encrypting data at the bit level. Their simplicity belies their power—when implemented correctly, they can provide robust security with minimal computational cost. Understanding the role of XOR, the importance of the key stream, and the symmetry of encryption and decryption lays a strong foundation for exploring more advanced cryptographic systems.

1.2.2 Why Encryption and Decryption Are the Same in Stream Ciphers

One of the most elegant features of stream ciphers is that **the encryption and decryption processes use the exact same mathematical operation**. This might seem surprising at first, especially when compared to more complex cryptographic systems that require distinct algorithms for each direction. But in stream ciphers, this symmetry is not only intentional—it's foundational. Let's explore why this works and how we can prove it.

1.2.3 The Core Operation: Modulo 2 Addition

Stream ciphers encrypt data one bit at a time using a simple operation called **modulo 2 addition**, also known as the **exclusive OR (XOR)**. The encryption formula is:

$$y_i = x_i + s_i \mod 2$$

Here: - (x_i) is the plaintext bit, - (s_i) is the key stream bit, - (y_i) is the resulting ciphertext bit.

To decrypt, we apply the same operation:

$$x_i = y_i + s_i \mod 2$$

This raises a natural question: **How can the same operation recover the original plaintext?**

1.2.4 Step-by-Step Proof of Symmetry

Let's walk through a mathematical proof to show that decryption truly reverses encryption.

We start with the encryption formula:

$$y_i = x_i + s_i \mod 2$$

Now, substitute this into the decryption formula:

$$x_i = y_i + s_i \mod 2$$

$$x_i = (x_i + s_i) + s_i \mod 2$$

$$x_i = x_i + s_i + s_i \mod 2$$

$$x_i = x_i + 2s_i \mod 2$$

Now here's the clever part: $(2s_i \mod 2)$ **is always zero**, regardless of whether (s_i) is 0 or 1.

- If $(s_i = 0)$, then $(2s_i = 0)$, and $(0 \mod 2 = 0)$
- If $(s_i = 1)$, then $(2s_i = 2)$, and $(2 \mod 2 = 0)$

So we simplify:

$$x_i = x_i + 0 \mod 2 = x_i$$

This confirms that the decryption function correctly recovers the original plaintext bit. **Q.E.D.** (which stands for *quod erat demonstrandum*, meaning “that which was to be demonstrated”).

This proof highlights a key advantage of stream ciphers: **efficiency and simplicity**. Because encryption and decryption are the same operation, systems using stream ciphers can be implemented with minimal logic—ideal for hardware, embedded systems, and real-time applications.

Moreover, the use of XOR ensures that the transformation is reversible, provided the key stream is known and synchronized between sender and receiver.

The symmetry of encryption and decryption in stream ciphers isn’t just a mathematical curiosity—it’s a practical feature that makes these ciphers fast, lightweight, and easy to implement. Understanding this property helps build intuition for how cryptographic systems can be both secure and elegant.

Fantastic! Let’s dive into the fascinating world of **key stream generation**, which is the beating heart of any stream cipher. The security of a stream cipher depends entirely on the quality and secrecy of its key stream. So understanding how these streams are generated is crucial for grasping how stream ciphers protect data.

1.2.5 Key Stream

A **key stream** is a sequence of bits $(s_1, s_2, s_3, \dots)$ that is combined with the plaintext bits to produce ciphertext. In a stream cipher, each plaintext bit is encrypted by XORing it with a corresponding key stream bit. The strength of the encryption depends on how unpredictable and unique this key stream is.

There are two main approaches to generating key streams:

1. **True Random Key Stream (One-Time Pad)**

- **Description**: This method uses a truly random sequence of bits that is as long as the message itself.
- **Security**: Perfectly secure if the key stream is never reused and kept secret.
- **Challenge**: Generating and securely distributing long random keys is impractical for most real-world applications.

2. **Pseudorandom Key Stream (Using PRNGs)**

Most modern stream ciphers use **pseudorandom number generators (PRNGs)** or **cryptographically secure pseudorandom number generators (CSPRNGs)** to produce key streams from a short secret key.

1.2.5.1 Basic Process:

1. **Initialization**: Start with a secret key (and often an initialization vector or IV).
2. **State Setup**: The key and IV initialize the internal state of the generator.
3. **Bit Generation**: The generator produces a long sequence of bits that appear random but are deterministically derived from the key.

1.2.6 Examples of Key Stream Generators

1. RC4 (Historical but now deprecated)

- **Mechanism**: Uses a permutation of all 256 possible byte values and a key scheduling algorithm.
- **Weakness**: Vulnerable to several attacks; no longer recommended.

2. Salsa20 / ChaCha20 (Modern and secure)

- **Mechanism**: Uses a combination of addition, rotation, and XOR operations on 32-bit words.
- **Strengths**: Fast, secure, and resistant to known cryptanalytic attacks.
- **Use Case**: Widely used in TLS, VPNs, and secure messaging apps.

3. Linear Feedback Shift Registers (LFSRs)

- **Mechanism**: Uses a shift register and feedback function to generate bits.
- **Use Case**: Common in hardware implementations (e.g., RFID, satellite communication).
- **Weakness**: Not secure on their own; often combined with nonlinear components.

Key Properties of a Good Key Stream Generator

To ensure strong encryption, a key stream generator must:

- Produce bits that are **indistinguishable from true randomness**.
- Be **deterministic** (same key always produces the same stream).
- Be **unpredictable** without knowing the key.
- Avoid **repeating key streams** (which can lead to serious vulnerabilities).

Initialization Vectors (IVs)

Many stream ciphers use an **IV** alongside the key to ensure that even if the same key is reused, the key stream will be different. IVs are typically:

- Public (not secret)
- Unique for each encryption session
- Combined with the key during initialization

Key stream generation is where cryptography meets clever engineering. It's a delicate balance between randomness, efficiency, and security. The better the generator, the harder it is for attackers to guess or reproduce the key stream—and that's what keeps your data safe.

1.2.7 RC4 Stream Cipher: Mechanism and Operation

RC4 is one of the earliest and most widely known stream ciphers in cryptographic history. Developed by Ron Rivest in 1987, RC4 gained popularity due to its simplicity, speed, and ease of implementation in both software and hardware. Although it has since been deprecated due to security vulnerabilities, understanding RC4 provides valuable insight into the principles of stream cipher design and key stream generation.

RC4 is a **symmetric stream cipher**, meaning the same key is used for both encryption and decryption. It operates by generating a **pseudo-random key stream** and XORing it with the plaintext to produce ciphertext. The same key stream is used to decrypt the ciphertext back into plaintext using the same XOR operation.

The cipher consists of two main components:

1. **Key Scheduling Algorithm (KSA)**
2. **Pseudo-Random Generation Algorithm (PRGA)**

1. Key Scheduling Algorithm (KSA)

The KSA is responsible for initializing the internal state of RC4. This state is a permutation of all 256 possible byte values (0 to 255), stored in an array called **S**.

Initialization Steps:

- Start with an array (S) of 256 bytes ($S[0], S[1], ..., S[255]$), initialized to the identity permutation (i.e., ($S[i] = i$)).
- Use the secret key (K) (which can be between 1 and 256 bytes) to shuffle the array.

- A temporary array (T) is created by repeating the key as needed to match the length of (S).
- The array (S) is then scrambled using the key values through a series of swaps.

This process ensures that the internal state is uniquely determined by the secret key, setting the stage for secure key stream generation.

2. Pseudo-Random Generation Algorithm (PRGA)

Once the state array (S) is initialized, the PRGA begins producing the key stream. This is a sequence of bytes that appears random but is deterministically derived from the key.

PRGA Steps:

- Two index variables (i) and (j) are initialized to 0.
- For each byte of output:
  - Increment (i): ($i = (i + 1) \mod 256$)
  - Update (j): ($j = (j + S[i]) \mod 256$)
  - Swap ($S[i]$) and ($S[j]$)
  - Output: ($K = S[(S[i] + S[j]) \mod 256]$)

This output byte (K) becomes part of the key stream and is used to encrypt or decrypt one byte of data.

Encryption and Decryption

RC4 uses the **exclusive OR (XOR)** operation to combine the key stream with the plaintext. The encryption formula is:

$$\text{ciphertext}_i = \text{plaintext}_i \oplus \text{keystream}_i$$

Since XOR is its own inverse, decryption uses the same operation:

$$\text{plaintext}_i = \text{ciphertext}_i \oplus \text{keystream}_i$$

This simplicity allows RC4 to be implemented efficiently, even on constrained devices.

1.2.7.1 Security Considerations

Despite its historical significance, RC4 has several vulnerabilities:

- **Key stream bias**: The initial bytes of the key stream are not uniformly random.
- **Key reuse risks**: Reusing the same key across multiple messages can lead to information leakage.
- **Known attacks**: RC4 has been broken in practical scenarios, especially in protocols like WEP and early versions of TLS.

As a result, RC4 is no longer recommended for secure communications and has been phased out of modern cryptographic standards. RC4 exemplifies the core principles of stream cipher design: generating a pseudo-random key stream and using XOR for encryption and decryption. While its simplicity made it a favorite for decades, its weaknesses highlight the importance of rigorous cryptanalysis and evolving cryptographic practices. Understanding RC4 provides a strong foundation for exploring more secure modern stream ciphers like ChaCha20.

1.2.8 ChaCha20 Stream Cipher: Mechanism and Operation

ChaCha20 is a **symmetric stream cipher** developed by Daniel J. Bernstein as an improvement over Salsa20. It operates on 512-bit blocks and uses a combination of **addition**, **rotation**, and **XOR** operations—collectively known as **ARX operations**—to generate a secure key stream.

Unlike RC4, which works byte-by-byte, ChaCha20 processes data in 32-bit words, making it highly efficient on modern processors.

1. Internal Structure of ChaCha20

ChaCha20's core consists of a **state matrix** and a series of transformation rounds. Let's walk through the key components:

a. State Matrix Initialization

The cipher begins by constructing a 4×4 matrix of 32-bit words (totaling 512 bits). This matrix includes:

Row	Content
1	Four constant words
2	Eight words from the 256-bit key
3	Two words for block counter
4	Three words for nonce (IV)

So the full matrix looks like:

```
[ constants | key | counter | nonce ]
```

The constants are fixed ASCII values: `"expand 32-byte k"` split into four 32-bit words.

b.ChaCha Rounds

ChaCha20 performs **20 rounds** of transformation, grouped into 10 pairs of **column rounds** and **diagonal rounds**. Each round applies a **quarter-round function** to four words in the matrix.

2. Quarter-Round Function

The quarter-round operates on four 32-bit words (a, b, c, d) and updates them using ARX operations:

$$a+=b; d^{=}a; d <<<= 16;$$

$$c+=d; b^{=}c; b <<<= 12;$$

$$a+=b; d^{=}a; d <<<= 8;$$

$$c+=d; b^{=}c; b <<<= 7;$$

- `+=` is 32-bit addition modulo (2^{32})
- `^=` is bitwise XOR
- `<<<=` is left rotation by the specified number of bits

These operations mix the bits thoroughly, ensuring diffusion and non-linearity.

3. Finalization and Output

After 20 rounds:

- The transformed state is **added** to the original state (modulo (2^{32}) addition).
- The resulting matrix is **serialized** into a 64-byte key stream block.
- This key stream is XORed with 64 bytes of plaintext to produce ciphertext.

Each block is indexed by a **block counter**, which increments for each new block, ensuring a unique key stream even with the same key and nonce.

1.2.8.1 Security Features

ChaCha20 is designed to be:

- **Resistant to timing attacks**: All operations take constant time.
- **Secure against known cryptanalysis**: No practical attacks have been found.
- **Efficient on software platforms**: Especially optimized for mobile and embedded devices.

It's often paired with **Poly1305**, a message authentication code, to provide authenticated encryption (AEAD), as in **ChaCha20-Poly1305**.

ChaCha20's internal operations are a masterclass in cryptographic design:

- A simple, elegant structure
- Fast and secure ARX transformations
- Robust key stream generation
- Proven resistance to attacks

It's no wonder ChaCha20 has become the go-to stream cipher for modern secure applications.

1.2.9 RC4: Solved Exercises

Exercise 1: Key Scheduling Algorithm (KSA)
Given: Key = [1, 2, 3]
Task: Perform the first few steps of RC4's KSA on S[0..5]

Solution:

1. Initialize S = [0, 1, 2, 3, 4, 5]
2. Repeat key to match S length: T = [1, 2, 3, 1, 2, 3]
3. For each i from 0 to 5:
   - j = (j + S[i] + T[i]) mod 6
   - Swap S[i] and S[j]

Step-by-step: - i = 0: j = (0 + 0 + 1) mod 6 = 1 → Swap S[0] and S[1] → S = [1, 0, 2, 3, 4, 5] - i = 1: j = (1 + 0 + 2) mod 6 = 3 → Swap S[1] and S[3] → S = [1, 3, 2, 0, 4, 5] - i = 2: j = (3 + 2 + 3) mod 6 = 2 → Swap S[2] and S[2] → No change - i = 3: j = (2 + 0 + 1) mod 6 = 3 → Swap S[3] and S[3] → No change

Partial KSA result: S = [1, 3, 2, 0, 4, 5]

Exercise 2: PRGA Key Stream Generation
Given: S = [1, 3, 2, 0, 4, 5], i = 0, j = 0
Task: Generate the first key stream byte

Solution:

- i = (0 + 1) mod 6 = 1
- j = (0 + S[1]) mod 6 = (0 + 3) = 3
- Swap S[1] and S[3] → S = [1, 0, 2, 3, 4, 5]
- Output = S[(S[1] + S[3]) mod 6] = S[(0 + 3) mod 6] = S[3] = 3

First key stream byte: **3**

Exercise 3: Encryption with RC4
Given: Plaintext byte = 0x6A (binary: 01101010), Key stream byte = 0x3C (binary: 00111100)
Task: Encrypt using XOR

Solution:

```
01101010
 00111100
-----------
01010110
```

Ciphertext byte: **0x56**

1.2.10 ChaCha20: Solved Exercises

Exercise 1: Quarter-Round Function
Given: a = 0x11111111, b = 0x01020304, c = 0x9B8D6F43, d = 0x01234567
Task: Perform one quarter-round (simplified)

Solution (showing first step only):

- a += b → a = 0x11111111 + 0x01020304 = 0x12131415
- d ^= a → d = 0x01234567 ^ 0x12131415 = 0x13355172

- d «< 16 → Rotate left 16 bits → d = 0x51721335

Partial result: a = 0x12131415, d = 0x51721335

Exercise 2: Key Stream Block Generation
Given: Key = 256-bit all zeros, Nonce = 96-bit all zeros, Counter = 0
Task: Generate first 64-byte key stream block

Solution:

- Initialize state matrix with constants, key, counter, nonce
- Apply 20 rounds of ChaCha20
- Add original state to final state
- Serialize to produce 64-byte output

Output: First block (simplified) starts with:
`76b8e0ada0f13d90405d6ae55386bd28...` (standard test vector)

Exercise 3: ChaCha20 Encryption
Given: Plaintext = "Hi" = [0x48, 0x69], Key stream = [0xAA, 0xBB]
Task: Encrypt using XOR

Solution:

- 0x48   0xAA = 0xE2

- 0x69   0xBB = 0xD2

Ciphertext: [0xE2, 0xD2]