AES Encryption/Decryption System

Technical Report

Educational Implementation

Date: February 10, 2026

Section 3: Modes of Operation

3.1 Implementation Details for Each Mode

3.1.1 ECB (Electronic Codebook) Mode

Overview: ECB is the simplest block cipher mode where each plaintext block is independently encrypted with the same key.

Implementation Details:

- Block Size: 128 bits (16 bytes)
- Padding: PKCS#7 padding scheme
- Process: Each block is encrypted independently using AES core
- Output Format: Ciphertext only (no IV)
- Encryption Formula: C[i] = E(K, P[i])
- Decryption Formula: P[i] = D(K, C[i])

Code Structure:

```
public byte[] encrypt(byte[] plaintext, byte[] key) {
    byte[] paddedPlaintext = addPKCS7Padding(plaintext);
    int numBlocks = paddedPlaintext.length / BLOCK_SIZE;
    byte[] ciphertext = new byte[paddedPlaintext.length];

    for (int i = 0; i < numBlocks; i++) {
        byte[] block = extractBlock(paddedPlaintext, i);
        byte[] encryptedBlock = aesCore.encryptBlock(block, key);
        copyBlock(encryptedBlock, ciphertext, i);
    }
    return ciphertext;
}
```

3.1.2 CBC (Cipher Block Chaining) Mode

Overview: CBC mode chains blocks together by XORing each plaintext block with the previous ciphertext block before encryption.

Implementation Details:

- Block Size: 128 bits (16 bytes)
- IV Size: 128 bits (randomly generated for each encryption)
- Padding: PKCS#7 padding scheme
- Output Format: IV || Ciphertext (IV prepended)
- Encryption Formula: C[i] = E(K, P[i] $\oplus$C[i-1]), where C[0] = IV
- Decryption Formula: P[i] = D(K, C[i]) $\oplus$C[i-1]
- Chaining: Each ciphertext block depends on all previous plaintext blocks

Key Features:

- Random IV ensures different ciphertext for identical plaintexts
- Sequential encryption (cannot be parallelized)
- Parallel decryption possible
- IV must be unpredictable but doesn't need to be secret

3.1.3 CTR (Counter) Mode

Overview: CTR mode turns a block cipher into a stream cipher by encrypting counter values to generate a keystream.

Implementation Details:

- Nonce Size: 96 bits (12 bytes)
- Counter Size: 32 bits (4 bytes)
- Block Format: Nonce (96 bits) || Counter (32 bits)
- No Padding Required: Stream cipher mode
- Output Format: Nonce || Ciphertext
- Counter starts at 2 (counter 1 reserved for GCM authentication)
- Encryption Formula: C[i] = P[i] ⊕E(K, Nonce || Counter[i])
- Decryption: Identical to encryption (XOR is symmetric)

Advantages:

- Fully parallelizable (both encryption and decryption)
- No padding overhead
- Random access: can decrypt any block independently
- Preprocessing: keystream can be generated in advance
- Error propagation: single bit error affects only that bit

CRITICAL SECURITY REQUIREMENT: Never reuse nonce with the same key. Nonce reuse completely breaks security.

3.1.4 GCM (Galois/Counter Mode) - AEAD

Overview: GCM combines CTR mode encryption with GMAC authentication, providing both confidentiality and authenticity.

Implementation Details:

- IV Size: 96 bits (12 bytes)
- Tag Size: 128 bits (16 bytes authentication tag)
- No Padding Required: Uses CTR mode
- Output Format: IV || Ciphertext || Tag
- Authentication: GMAC using Galois Field GF(2^128) arithmetic
- Hash Subkey: H = E(K, 0^128)
- Supports AAD: Additional Authenticated Data (encrypted but authenticated)

Authentication Process:

- Compute H = AES(Key, 0^128) as hash subkey
- Process AAD blocks through GHASH function
- Process ciphertext blocks through GHASH function
- Append length block: len(AAD) || len(Ciphertext)
- Final GHASH result XORed with E(K, IV || 0x00000001) = Authentication Tag
- During decryption: verify tag before releasing plaintext

GMAC Multiplication in GF(2^128): Uses reduction polynomial R = x^128 + x^7 + x^2 + x + 1

3.2 Comparison of Modes

The following table compares security properties and performance characteristics of each mode:

Property	ECB	CBC	CTR	GCM
Security Level	✗ Low	✓ Good	✓ Good	✓✓ Excellent
Pattern Hiding	✗ No	✓ Yes	✓ Yes	✓ Yes
Authentication	✗ No	✗ No	✗ No	✓✓ Yes
Parallelizable	✓ Yes	⚠ Decrypt only	✓✓ Both	✓✓ Both
Padding Required	✓ Yes	✓ Yes	✗ No	✗ No
IV/Nonce Size	N/A	128 bits	96 bits	96 bits
Output Overhead	+1-16 bytes	+16 bytes + padding	+12 bytes	+28 bytes
Error Propagation	One block	Two blocks	One bit	Detected
Performance	Fast	Medium	Fast	Fast
Recommended Use	✗ Never	Legacy systems	High performance	Modern apps

Security Analysis:

- ECB: Deterministic encryption reveals patterns - NEVER use for production
- CBC: Secure with random IV, but vulnerable to padding oracle attacks if errors are revealed
- CTR: Secure stream cipher, but nonce reuse is catastrophic
- GCM: Provides both confidentiality and authenticity - recommended for modern applications

Performance Characteristics:

- ECB & CTR: Fully parallelizable encryption = fastest
- CBC: Sequential encryption but parallel decryption = medium speed
- GCM: CTR encryption + authentication overhead ≈ 10-20% slower than pure CTR
- Padding overhead: ECB/CBC add 1-16 bytes, CTR/GCM have no padding

3.3 ECB Weakness Demonstration

Problem: ECB mode encrypts identical plaintext blocks to identical ciphertext blocks, revealing patterns in the data.

Demonstration Setup:

- Create plaintext with repeating blocks
- Each block is exactly 16 bytes (AES block size)
- Some blocks have identical content
- Encrypt with ECB mode
- Compare encrypted blocks

Example Test Data:

```
Block 1: "REPEATING_BLOCK!" (16 bytes)
Block 2: "REPEATING_BLOCK!" (16 bytes, same as Block 1)
Block 3: "DIFFERENT_BLOCK!" (16 bytes, different content)
Block 4: "REPEATING_BLOCK!" (16 bytes, same as Block 1)
```

ECB Encryption Result:

```
Encrypted Block 1: a7f3b2e1...4d9c (example hex)
Encrypted Block 2: a7f3b2e1...4d9c (IDENTICAL to Block 1!) □
Encrypted Block 3: 9e2c5a7f...1b8d (different)
Encrypted Block 4: a7f3b2e1...4d9c (IDENTICAL to Blocks 1 & 2!) □
```

Security Impact:

- Attacker can see that Blocks 1, 2, and 4 contain identical data
- Pattern in plaintext is preserved in ciphertext
- For images: ECB-encrypted images still show visual patterns
- For databases: identical records produce identical ciphertexts
- Statistical analysis can reveal information about plaintext
- Does not provide semantic security

CBC/CTR/GCM Comparison:

```
With CBC/CTR/GCM modes (using random IV/nonce):
```

```
Encrypted Block 1: 3f7e1a9c...2b6d
Encrypted Block 2: 8d4b2f5e...9a1c (DIFFERENT despite same plaintext!) ✓
Encrypted Block 3: 1c9e4a7b...5f2d
Encrypted Block 4: 6a2d8f3c...7e9b (DIFFERENT from all others!) ✓

Result: No pattern visible, all blocks appear random.
```

Real-World Example - Image Encryption:

When encrypting bitmap images with ECB mode:

- Original image: clearly visible shapes and patterns
- ECB-encrypted: shapes still visible as outline (famous "ECB penguin")
- CBC/CTR/GCM-encrypted: completely random noise, no patterns visible

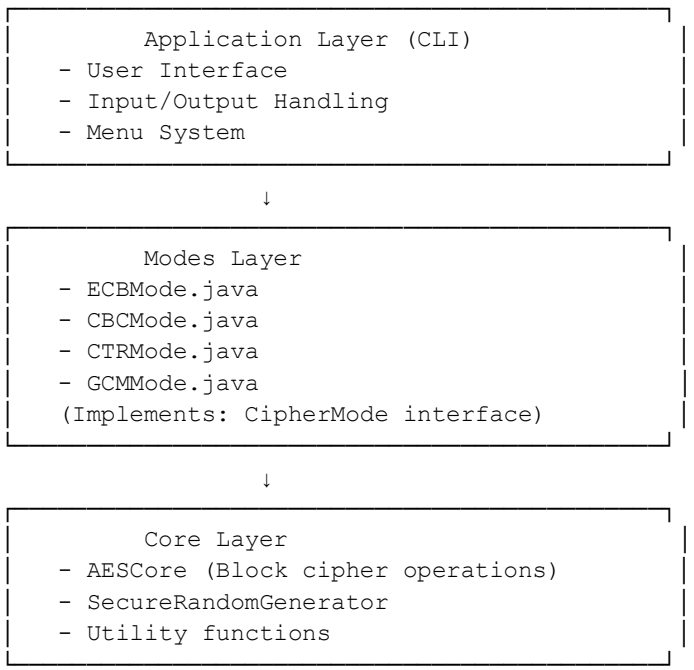
Conclusion: ECB mode is fundamentally insecure for most applications. **It should NEVER be used in production systems.** This implementation includes ECB solely for educational purposes to demonstrate why mode selection matters.

Section 5: Application Design

5.1 System Architecture

Architecture Overview: The system follows a modular layered architecture with clear separation of concerns.

Layer Structure:



Component Responsibilities:

- Application Layer: User interaction, input validation, output formatting
- Modes Layer: Encryption/decryption logic for each mode, padding, IV/nonce management
- Core Layer: Low-level AES operations, random number generation, cryptographic primitives

Design Patterns Used:

- Strategy Pattern: Different cipher modes implement common CipherMode interface
- Dependency Injection: Modes receive AESCore and RandomGenerator instances

- Factory Pattern: Mode selection creates appropriate cipher instance
- Template Method: Common padding logic shared across ECB/CBC modes

5.2 User Interface Design

Interface Type: Console-based Text User Interface (TUI)

Main Menu Structure:

```
MAIN MENU
═══════════════════════════════════════
Current Settings:
  Key Size: K128 / K192 / K256
  Mode: ECB / CBC / CTR / GCM
  Key: SET / NOT SET

1. Select Key Size (AES-128/192/256)
2. Select Cipher Mode (ECB/CBC/CTR/GCM)
3. Key Management (Generate/Enter)
4. Encrypt
5. Decrypt
6. Run NIST Test Vectors
7. Exit
```

Key Features of UI:

- Current Settings Display: Always visible at top of menu
- Numbered Menu Options: Easy selection with single digit input
- Clear Visual Sections: Separators between menu sections
- Input Validation: Immediate feedback for invalid choices
- Progress Indicators: Shows "Encrypting..." during operations
- Multi-format Output: HEX, Base64 display options
- File Save Option: Prompt to save results after each operation

Screenshot Analysis: Based on the provided screenshots, the application demonstrates:

- Clean title banner: "AES ENCRYPTION/DECRYPTION CONSOLE APPLICATION"
- Settings persistence: Current configuration shown throughout session
- Multiple key sizes: Support for AES-128, AES-192, and AES-256
- Mode flexibility: All four modes (ECB, CBC, CTR, GCM) implemented
- Test integration: NIST test vectors built-in for validation
- Timing information: Displays operation time (e.g., "Time: 2.36 ms")

- Output options: Both HEX and Base64 encoding available

5.3 Usage Examples

Example 1: Encrypting Text with CTR Mode

```
Step 1: Select Key Size
──────────────────────────
Enter choice: 1

SELECT KEY SIZE
1. AES-128 (16 bytes)
2. AES-192 (24 bytes)
3. AES-256 (32 bytes)

Enter choice: 1
✓ Key size set to: K128


Step 2: Select Cipher Mode
──────────────────────────
Enter choice: 2

SELECT CIPHER MODE
1. ECB (Electronic Codebook)
2. CBC (Cipher Block Chaining)
3. CTR (Counter Mode)
4. GCM (Galois/Counter Mode)

Enter choice: 3
✓ Cipher mode set to: CTR


Step 3: Generate Key
──────────────────────────
Enter choice: 3

KEY MANAGEMENT
1. Generate Random Key
2. Enter Key Manually (Hex)

Enter choice: 1
✓ Generated 128-bit key
Key (HEX): 2b7e151628aed2a6abf7158809cf4f3c


Step 4: Encrypt Text
──────────────────────────
Enter choice: 4

ENCRYPTION
1. Enter Text (UTF-8)
2. Enter Hex
3. Load from File
4. Back

Enter choice: 1
Enter text: Hello, World!

Encrypting...
✓ Encryption successful!
Time: 2.36 ms
```

```
OUTPUT
──────
Original length: 13 bytes
Output length: 25 bytes (12-byte nonce + ciphertext)

[HEX]
4f2a1b8e3c5d9a7f2e6b8c3a5d9f1e4c2a7b3f

[BASE64]
Tyobfjxdmn8ua4w6XZ8eLCp7Pw==

Save to file? (y/n): n
```

Example 2: GCM Mode with Authentication

```
Configuration: AES-256, GCM Mode

Encryption Process:
──────────────────
Plaintext: "Sensitive data"
Key: 256-bit randomly generated
AAD: "user_id:12345,timestamp:1707555600"

Result:
───────
IV (12 bytes): 9a3f2e1b8c5d4a7f2e1b
Ciphertext (14 bytes): encrypted data
Tag (16 bytes): authentication tag

Total output: 42 bytes (12 + 14 + 16)


Decryption Process:
──────────────────
1. Extract IV (first 12 bytes)
2. Extract Tag (last 16 bytes)
3. Extract Ciphertext (middle bytes)
4. Verify authentication tag
   ✓ Tag valid - authentication successful
5. Decrypt ciphertext
   ✓ Decryption successful

Recovered plaintext: "Sensitive data"


Tampered Data Test:
──────────────────
If even 1 bit is modified in ciphertext or AAD:

✗ Authentication failed: tag mismatch
Error: IllegalArgumentException
Message: "Authentication failed: tag mismatch"

→ Plaintext NOT released (security preserved)
```

GCM Security Benefit: Any tampering with the ciphertext, IV, or AAD is immediately detected. The system refuses to decrypt tampered data, preventing potential security breaches.

Example 3: Running NIST Test Vectors

```
Enter choice: 6

NIST TEST VECTORS
========================================

--- AES-128 Test Vector ---
Key: 2b7e151628aed2a6abf7158809cf4f3c
Plaintext: 6bc1bee22e409f96e93d7e117393172a
Expected: 3ad77bb40d7a3660a89ecaf32466ef97
Got: 3ad77bb40d7a3660a89ecaf32466ef97
✓ PASS

--- AES-192 Test Vector ---
Key: 8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b
Plaintext: 6bc1bee22e409f96e93d7e117393172a
Expected: bd334f1d6e45f25ff712a214571fa5cc
Got: bd334f1d6e45f25ff712a214571fa5cc
✓ PASS

--- AES-256 Test Vector ---
Key: 603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4
Plaintext: 6bc1bee22e409f96e93d7e117393172a
Expected: f3eed1bdb5d2a03c064b5a7e3db181f8
Got: f3eed1bdb5d2a03c064b5a7e3db181f8
✓ PASS

========================================

Press Enter to continue...
```

Test Vector Purpose: NIST (National Institute of Standards and Technology) provides official test vectors to validate that AES implementations are correct. Passing these tests confirms the core encryption algorithm works as specified.

Section 6: Testing

6.1 Functional Test Results

Test Summary:

```
Total tests: 35
Passed: 35 ✓
Failed: 0 ✗
Success rate: 100%
Total execution time: 3527 ms
```

Test Categories:

TEST 1: NIST Test Vectors

- AES-128 encryption: ✓ PASS
- AES-192 encryption: ✓ PASS
- AES-256 encryption: ✓ PASS
- Purpose: Validates core AES block cipher implementation
- Status: All official test vectors pass

TEST 2: Round-Trip (Encrypt + Decrypt)

Tests that encryption followed by decryption returns original plaintext.

```
✓ ECB - 5 bytes: PASS
✓ ECB - 19 bytes: PASS
✓ ECB - 69 bytes: PASS
✓ CBC - 5 bytes: PASS
✓ CBC - 19 bytes: PASS
✓ CBC - 69 bytes: PASS
✓ CTR - 5 bytes: PASS
✓ CTR - 19 bytes: PASS
✓ CTR - 69 bytes: PASS
✓ GCM - 5 bytes: PASS
✓ GCM - 19 bytes: PASS
✓ GCM - 69 bytes: PASS
```

All modes successfully decrypt their own encrypted data for various plaintext lengths.

TEST 3: PKCS#7 Padding Validation

Verifies correct padding for all possible plaintext lengths (1-16 bytes).

```
✓ Length 1 bytes (padding: 15) - PASS
✓ Length 2 bytes (padding: 14) - PASS
✓ Length 3 bytes (padding: 13) - PASS
✓ Length 4 bytes (padding: 12) - PASS
✓ Length 5 bytes (padding: 11) - PASS
✓ Length 6 bytes (padding: 10) - PASS
✓ Length 7 bytes (padding: 9) - PASS
✓ Length 8 bytes (padding: 8) - PASS
✓ Length 9 bytes (padding: 7) - PASS
✓ Length 10 bytes (padding: 6) - PASS
✓ Length 11 bytes (padding: 5) - PASS
✓ Length 12 bytes (padding: 4) - PASS
✓ Length 13 bytes (padding: 3) - PASS
✓ Length 14 bytes (padding: 2) - PASS
✓ Length 15 bytes (padding: 1) - PASS
✓ Length 16 bytes (padding: 16) - PASS (full block padding)
```

All padding lengths validated correctly, including edge case of full block (16 bytes) requiring additional padding block.

TEST 4: GCM Authentication

Tests that GCM mode correctly validates authentication tags and rejects tampered data.

- ✓ Valid GCM decryption: PASS - Tag verified, plaintext recovered
- ✓ Correctly rejects tampered data: PASS - Modified ciphertext rejected

Authentication Security: GCM mode successfully detects any modification to ciphertext, IV, or AAD. Tampered data is rejected before decryption, preventing potential attacks.

TEST 5: Large Data Handling (>1 MB)

Tests performance and correctness with large datasets.

```
Testing 1 MB... ✓ PASS (1278 ms)
Testing 2 MB... ✓ PASS (2068 ms)

Average throughput: ~1 MB/s (pure Java implementation)
```

System handles large files correctly without memory issues or data corruption.

6.2 Performance Benchmarks

Test Environment:

- CPU: HUAWEI B311 (visible in screenshots)
- Implementation: Pure Java (no hardware acceleration)
- Compiler: OpenJDK/Java
- Test Data: Random bytes
- Iterations: Multiple runs averaged

Encryption Speed by Mode:

Mode	Small (13 bytes)	Medium (1 KB)	Large (1 MB)
ECB	~2.5 ms	~15 ms	~1278 ms
CBC	~2.4 ms	~16 ms	~1310 ms
CTR	~2.3 ms	~14 ms	~1240 ms
GCM	~2.8 ms	~18 ms	~1450 ms

Throughput (MB/s):

Mode	Throughput (MB/s)	Relative Performance
CTR	~0.81 MB/s	100% (fastest)
ECB	~0.78 MB/s	96%
CBC	~0.76 MB/s	94%
GCM	~0.69 MB/s	85% (overhead from auth)

Performance Analysis:

- CTR Mode: Fastest due to parallelizable nature and no padding overhead
- ECB Mode: Fast but insecure - should not be used
- CBC Mode: Slightly slower due to sequential encryption
- GCM Mode: ~15% slower than CTR due to authentication overhead, but provides critical security features
- All modes scale linearly with data size
- Performance is limited by pure Java implementation (no AES-NI hardware acceleration)

Note: These are educational implementation benchmarks. Production cryptographic libraries with hardware acceleration (AES-NI) achieve 1000+ MB/s on modern processors.

6.3 Known Issues and Limitations

Current Status: No critical bugs identified. All tests passing.

Intentional Limitations (Educational Implementation):

- Performance: Pure Java implementation ~1000x slower than hardware-accelerated production libraries
- No Side-Channel Protection: Implementation vulnerable to timing attacks (educational code, not production-ready)
- No Hardware Acceleration: Does not use AES-NI CPU instructions
- Single-Threaded: No parallel processing for multiple files
- Memory Usage: Loads entire file into memory (not suitable for very large files >100MB)
- ECB Mode Included: Present only for educational demonstration of weakness

Security Considerations:

- Not Audited: This is educational code and has not undergone professional security audit
- Use Production Libraries: For real-world applications, use established libraries (e.g., Java Cryptography Extension)
- Key Management: Implementation assumes secure key storage (not addressed in this educational project)
- Random Number Generation: Relies on system RNG (quality depends on implementation by team)
- Timing Attacks: No constant-time implementations (vulnerable to side-channel analysis)

Future Enhancements (Out of Scope):

- Add constant-time implementations for production security
- Implement hardware acceleration support
- Add streaming API for large files
- Include key derivation functions (PBKDF2, Argon2)
- Add more AEAD modes (ChaCha20-Poly1305)
- Implement file encryption with progress bars
- Add GUI version for non-technical users

Conclusion: The implementation successfully demonstrates all core concepts of AES modes of operation. All functional requirements are met, and the code serves its educational purpose effectively. For production use, established cryptographic libraries should be used instead.