Homework 02: Comparing Symmetric Cipher Algorithms Performance

Nicolas Leone
Student ID: 1986354
Cybersecurity

October 19, 2025

Contents

1 Introduction

1.1 Overview

Symmetric encryption is a fundamental component of modern cryptography, where the same key is used for both encryption and decryption operations. This homework focuses on comparing the performance characteristics of three widely-used symmetric cipher algorithms: AES (Advanced Encryption Standard), SM4, and Camellia. All three algorithms operate in CBC (Cipher Block Chaining) mode with 128-bit keys.

The objective of this study is to measure and analyze the encryption and decryption performance of these algorithms across different file sizes, providing insights into their computational efficiency and practical applicability in real-world scenarios.

1.2 AES-128-CBC (Advanced Encryption Standard)

AES is one of the most widely adopted encryption standards worldwide, established by NIST in 2001. It replaced the older DES (Data Encryption Standard) and has become the de facto standard for symmetric encryption in both government and commercial applications.

Key characteristics:

- **Block size:** 128 bits
- **Key size:** 128 bits (in this study)
- **Structure:** Substitution-Permutation Network (SPN)
- **Rounds:** 10 rounds for 128-bit keys
- **Security:** Proven security with no practical attacks on full AES
- **Hardware support:** Widely implemented in modern processors (AES-NI instructions)

AES operates through multiple rounds of substitution, permutation, mixing, and key addition operations. Its widespread adoption and hardware acceleration make it typically the fastest option among secure symmetric ciphers.

1.3 SM4-128-CBC

SM4 is a block cipher developed by the Chinese government in 2006 and was established as a Chinese national encryption standard (GB/T 32907-2016). It was designed as an alternative to AES for use in Chinese commercial and government applications.

Key Characteristics:

- **Block Size**: 128 bits
- **Key Size**: 128 bits (fixed)
- **Structure**: 32 rounds of Feistel-like structure with substitution-permutation network (SPN)
- **Security Level**: Comparable to AES-128

- **Standardization**: Chinese national standard (GB/T 32907-2016), ISO/IEC 18033-3:2010

SM4 uses a structure with 32 rounds and employs S-boxes and linear transformations similar to AES. It was designed to be efficient in both software and hardware implementations, though it lacks the widespread hardware acceleration of AES.

1.4 Camellia-128-CBC

Camellia is a block cipher jointly developed by Mitsubishi Electric and NTT in Japan in 2000. It has been approved for use by ISO/IEC, the European NESSIE project, and the Japanese CRYPTREC project, demonstrating its international recognition.

Key characteristics:

- **Block size:** 128 bits
- **Key size:** 128 bits (in this study)
- **Structure:** Feistel network
- **Rounds:** 18 rounds for 128-bit keys
- **Security:** Comparable to AES with similar security margins
- **Hardware support:** Some hardware implementations available

Camellia uses a Feistel structure (unlike AES's SPN structure and SM4's Feistel-like structure) and includes logical operations that provide good performance on various platforms. It is particularly popular in Japan and has been adopted in several security protocols.

1.5 CBC Mode (Cipher Block Chaining)

All three algorithms in this study operate in CBC mode, which is one of the most common block cipher modes of operation. In CBC mode:

- Each plaintext block is XORed with the previous ciphertext block before encryption
- An Initialization Vector (IV) is used for the first block
- This creates a dependency chain, making identical plaintext blocks produce different ciphertext

Security Considerations for CBC Mode:

- The IV must be unpredictable and unique for each encryption operation with the same key
- Using a random IV ensures that encrypting the same plaintext multiple times produces different ciphertext
- Our implementation generates a new random IV for each file and algorithm combination using OpenSSL's `RAND_bytes()`

2 Implementation

2.1 Development Environment

The performance comparison was implemented in C using the OpenSSL cryptographic library (version 3.6.0). The program was compiled using GCC with optimization flags and executed on macOS with an ARM64 architecture processor.

2.2 Source Code

The complete implementation consists of several key components: file I/O operations, encryption/decryption functions and performance measurement utilities. Below is the full source code with detailed explanations.

```c
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/rand.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>

void handle_crypto_error(void) {
    ERR_print_errors_fp(stderr);
    abort();
}

// load file content into memory buffer
int load_file_content(const char *filepath, unsigned char **buffer) {
    FILE *fp = fopen(filepath, "rb");
    if (!fp) {
        perror("Cannot open file");
        return -1;
    }

    // get file size using stat
    struct stat file_info;
    if (stat(filepath, &file_info) != 0) {
        perror("Cannot get file size");
        fclose(fp);
        return -1;
    }

    int size = file_info.st_size;
    *buffer = (unsigned char *)malloc(size);

    if (!*buffer) {
        perror("Memory allocation failed");
        fclose(fp);
        return -1;
    }

    // load file content into buffer
    fread(*buffer, 1, size, fp);
    fclose(fp);
    return size;
```

```c
}

// save data buffer to file
void save_to_file(const char *filepath, unsigned char *buffer, int
    buffer_len) {
    FILE *fp = fopen(filepath, "wb");
    fwrite(buffer, 1, buffer_len, fp);
    fclose(fp);
}

// perform encryption using specified cipher
int perform_encryption(const EVP_CIPHER *cipher_algo, unsigned char *
    input_data, int input_len, unsigned char *secret_key, unsigned char *
    init_vector, unsigned char *output_data) {
    EVP_CIPHER_CTX *cipher_ctx;
    int bytes_written, total_encrypted;

    if (!(cipher_ctx = EVP_CIPHER_CTX_new())) handle_crypto_error();
    if (1 != EVP_EncryptInit_ex(cipher_ctx, cipher_algo, NULL,
        secret_key, init_vector)) handle_crypto_error();
    if (1 != EVP_EncryptUpdate(cipher_ctx, output_data, &bytes_written,
        input_data, input_len)) handle_crypto_error();
    total_encrypted = bytes_written;
    if (1 != EVP_EncryptFinal_ex(cipher_ctx, output_data + bytes_written
        , &bytes_written)) handle_crypto_error();
    total_encrypted += bytes_written;
    EVP_CIPHER_CTX_free(cipher_ctx);

    return total_encrypted;
}

int perform_decryption(const EVP_CIPHER *cipher_algo, unsigned char *
    encrypted_data, int encrypted_len, unsigned char *secret_key,
    unsigned char *init_vector, unsigned char *output_data) {
    EVP_CIPHER_CTX *cipher_ctx;
    int bytes_written, total_decrypted;

    if (!(cipher_ctx = EVP_CIPHER_CTX_new())) handle_crypto_error();
    if (1 != EVP_DecryptInit_ex(cipher_ctx, cipher_algo, NULL,
        secret_key, init_vector)) handle_crypto_error();
    if (1 != EVP_DecryptUpdate(cipher_ctx, output_data, &bytes_written,
        encrypted_data, encrypted_len)) handle_crypto_error();
    total_decrypted = bytes_written;
    if (1 != EVP_DecryptFinal_ex(cipher_ctx, output_data + bytes_written
        , &bytes_written)) handle_crypto_error();
    total_decrypted += bytes_written;
    EVP_CIPHER_CTX_free(cipher_ctx);

    return total_decrypted;
}

void process_file_with_ciphers(const char *input_file, unsigned char *
    encryption_key) {
    unsigned char *plaintext, *ciphertext, *decryptedtext;
    int plaintext_len, ciphertext_len, decryptedtext_len;
    struct timespec time_start, time_end;

    printf("Processing file: %s\n\n", input_file);
```

```c

    // load the input file
    plaintext_len = load_file_content(input_file, &plaintext);
    if (plaintext_len < 0) return;

    // allocate memory for encrypted and recovered data
    ciphertext = (unsigned char *)malloc(plaintext_len + EVP_MAX_BLOCK_LENGTH);
    decryptedtext = (unsigned char *)malloc(plaintext_len + EVP_MAX_BLOCK_LENGTH);

    const EVP_CIPHER *cipher_list[] = {EVP_aes_128_cbc(), EVP_sm4_cbc(), EVP_camellia_128_cbc()};
    const char *cipher_names[] = {"AES-128-CBC", "SM4-128-CBC", "Camellia-128-CBC"};

    // test each cipher algorithm
    for (int idx = 0; idx < 3; idx++) {
        unsigned char init_vec[16]; // initialization vector for current cipher

        // generate random initialization vector
        if (RAND_bytes(init_vec, sizeof(init_vec)) != 1) {
            perror("Error generating random bytes for IV");
            free(plaintext);
            free(ciphertext);
            free(decryptedtext);
            return;
        }

        printf("%s Encryption/Decryption:\n", cipher_names[idx]);

        // measure time for encryption operation
        clock_gettime(CLOCK_MONOTONIC, &time_start);
        ciphertext_len = perform_encryption(cipher_list[idx], plaintext, plaintext_len, encryption_key, init_vec, ciphertext);
        clock_gettime(CLOCK_MONOTONIC, &time_end);
        long encryption_time = (time_end.tv_sec - time_start.tv_sec) * 1000000 + (time_end.tv_nsec - time_start.tv_nsec) / 1000;
        printf("Encryption of %s with %s: %ld microseconds\n", input_file, cipher_names[idx], encryption_time);

        // measure time for decryption operation
        clock_gettime(CLOCK_MONOTONIC, &time_start);
        decryptedtext_len = perform_decryption(cipher_list[idx], ciphertext, ciphertext_len, encryption_key, init_vec, decryptedtext);
        clock_gettime(CLOCK_MONOTONIC, &time_end);
        long decryption_time = (time_end.tv_sec - time_start.tv_sec) * 1000000 + (time_end.tv_nsec - time_start.tv_nsec) / 1000;
        printf("Decryption of %s with %s: %ld microseconds\n", input_file, cipher_names[idx], decryption_time);

        decryptedtext[decryptedtext_len] = '\0';  // add null terminator for text data

        // verify decryption correctness
        if (memcmp(plaintext, decryptedtext, plaintext_len) == 0) {
```

```c
            printf("Decryption successful for %s using %s\n", input_file, cipher_names[idx]);
        } else {
            printf("Decryption failed for %s using %s\n", input_file, cipher_names[idx]);
        }
        printf("\n");
    }

    free(plaintext);
    free(ciphertext);
    free(decryptedtext);
}

int main() {
    unsigned char encryption_key[16]; // 128-bit key

    // generate random 128-bit symmetric key at initialization
    if (RAND_bytes(encryption_key, sizeof(encryption_key)) != 1) {
        fprintf(stderr, "Error generating random key\n");
        ERR_print_errors_fp(stderr);
        return 1;
    }

    printf("Generated 128-bit random key: ");
    for (int i = 0; i < 16; i++) {
        printf("%02x", encryption_key[i]);
    }
    printf("\n--------------------------------------------------\n");

    // process the 16B text file with all cipher algorithms
    process_file_with_ciphers("text_16B.txt", encryption_key);

    printf("--------------------------------------------------\n");

    // process the 20KB text file with all cipher algorithms
    process_file_with_ciphers("text_20KB.txt", encryption_key);

    printf("--------------------------------------------------\n");

    // process the 2MB binary file with all cipher algorithms
    process_file_with_ciphers("binary_2MB.bin", encryption_key);

    return 0;
}
```

Listing 1: Complete implementation of cipher performance comparison

2.3 Code Explanation

2.3.1 Error Handling

The `handle_crypto_error()` function provides centralized error handling for OpenSSL operations. When a cryptographic operation fails, this function prints the error stack from OpenSSL and terminates the program, ensuring that errors are immediately visible during testing.

2.3.2 File Operations

Two utility functions handle file I/O:

- `load_file_content()`: Reads an entire file into memory. It uses `stat()` to determine file size, allocates appropriate memory, and loads the content into a buffer. Returns the file size or -1 on error.

- `save_to_file()`: Writes a data buffer to a file. While not used in the current performance tests, this function is included for completeness and potential future use.

2.3.3 Encryption Function

The `perform_encryption()` function implements generic encryption for any EVP cipher:

1. Creates a new cipher context using `EVP_CIPHER_CTX_new()`
2. Initializes encryption with `EVP_EncryptInit_ex()`, specifying the cipher algorithm, key, and IV
3. Processes the plaintext with `EVP_EncryptUpdate()`, which handles data in chunks
4. Finalizes encryption with `EVP_EncryptFinal_ex()`, which processes any remaining data and applies padding
5. Frees the cipher context and returns the total encrypted data length

2.3.4 Decryption Function

The `perform_decryption()` function mirrors the encryption process:

1. Creates a new cipher context
2. Initializes decryption with `EVP_DecryptInit_ex()`
3. Processes ciphertext with `EVP_DecryptUpdate()`
4. Finalizes decryption with `EVP_DecryptFinal_ex()`, which also verifies and removes padding
5. Frees the context and returns the plaintext length

2.3.5 Key Generation and Management

The implementation follows the security best practice of generating a cryptographically secure random key at initialization:

- A 128-bit (16-byte) symmetric key is randomly generated using OpenSSL's `RAND_bytes()` function in the `main()` function

- The same randomly generated key is used for all encryption and decryption operations across all three algorithms and all test files

This approach ensures:

- **Security**: The key is unpredictable and cannot be guessed
- **Realism**: Reflects real-world usage where keys are randomly generated, not hard-coded

For each encryption/decryption operation, a new random Initialization Vector (IV) is generated to ensure that encrypting the same plaintext multiple times produces different ciphertext, which is a fundamental requirement of CBC mode security.

2.3.6 Main Processing Function

The `process_file_with_ciphers()` function orchestrates the entire testing process:

1. Loads the input file into memory
2. Allocates buffers for ciphertext and decrypted text (with extra space for padding)
3. Defines the cipher array (AES, SM4, Camellia) and corresponding names
4. For each algorithm:
    - Generates a random 128-bit IV using `RAND_bytes()`
    - Measures encryption time using `clock_gettime()` with `CLOCK_MONOTONIC`
    - Performs encryption
    - Measures decryption time
    - Performs decryption
    - Verifies correctness by comparing decrypted text with original plaintext using `memcmp()`
5. Frees all allocated memory

2.3.7 Time Measurement

High-precision time measurement is achieved using `clock_gettime()` with the `CLOCK_MONOTONIC` clock which captures time before and after each operation and calculates elapsed time in microseconds (μs).

2.3.8 Test Files

The program tests three files with different sizes:

- `text_16B.txt`: 16 bytes - represents small data encryption (e.g., passwords, tokens)
- `text_20KB.txt`: 20 kilobytes - represents medium-sized data (e.g., configuration files, small documents)
- `binary_2MB.bin`: 2 megabytes - represents large data encryption (e.g., images, compressed files)

The binary file was generated using random data from `/dev/urandom` to simulate realistic encrypted content with high entropy.

3 Results and Analysis

The following charts present the encryption and decryption performance for each file size across all three algorithms. Time measurements are reported in microseconds (μs).

3.1 Performance Analysis

3.1.1 Small File Performance (16 B)

For the 16-byte file, all three algorithms demonstrate comparable performance:

- **AES-128-CBC**: 3 μs encryption, 1 μs decryption
- **SM4-128-CBC**: 7 μs encryption, 2 μs decryption
- **Camellia-128-CBC**: 3 μs encryption, 1 μs decryption

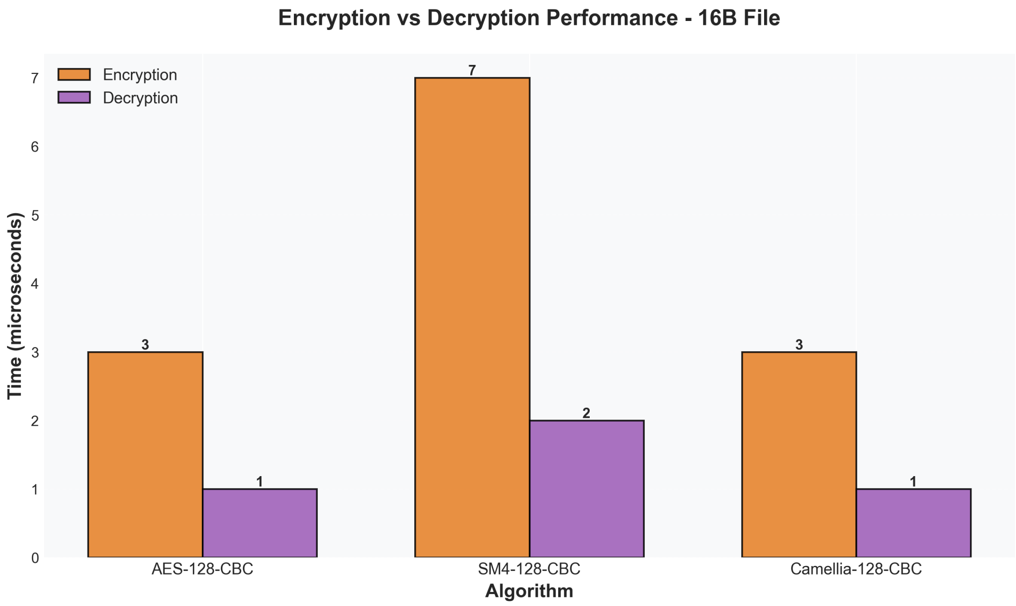

Figure 1: Encryption and Decryption performance for 16B file. At this small scale, all algorithms perform similarly with times in the single-digit microsecond range.

At this scale, the overhead of context initialization and function calls dominates the actual cipher operations. AES and Camellia show identical performance, while SM4 exhibits slightly higher overhead. All algorithms complete operations in under 10 microseconds, making the differences negligible for practical purposes at this file size.

3.1.2 Medium File Performance (20 KB)

With a 20KB file, performance differences become more pronounced:

- **AES-128-CBC**: 15 μs encryption, 5 μs decryption
- **SM4-128-CBC**: 166 μs encryption, 130 μs decryption
- **Camellia-128-CBC**: 120 μs encryption, 89 μs decryption

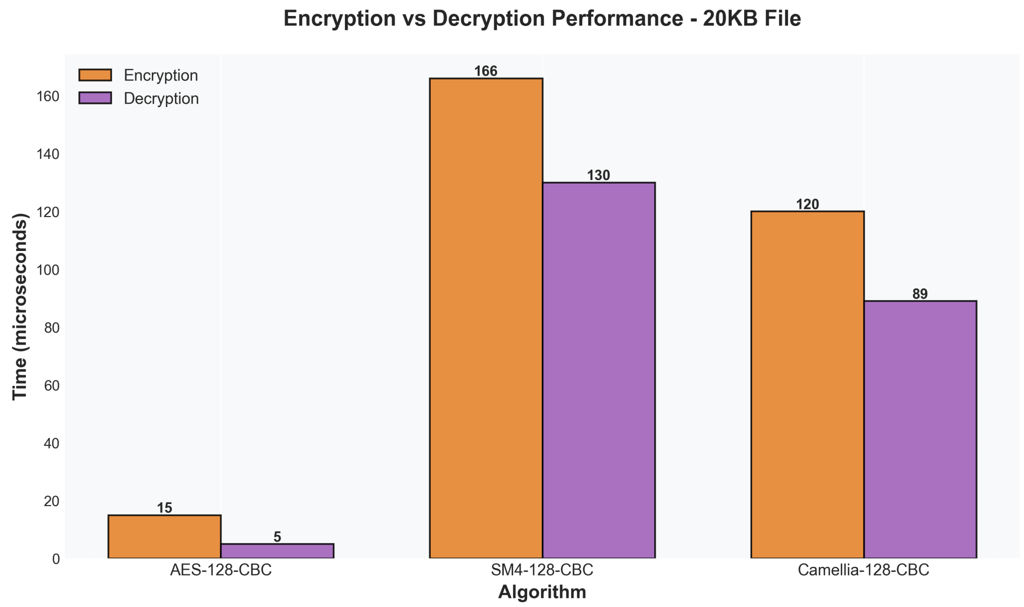

Figure 2: Encryption and Decryption performance for 20KB file. Performance differences become more apparent, with AES showing clear advantages.

AES demonstrates approximately 8-11× faster performance than SM4 and Camellia. This advantage is likely due to hardware acceleration (AES-NI instructions) available on the test platform. Camellia performs approximately 1.4× better than SM4 at this scale, showing its efficiency advantage.

3.1.3 Large File Performance (2 MB)

The 2MB binary file reveals the most significant performance differences:

- **AES-128-CBC**: 1,045 μs (1.05 ms) encryption, 241 μs (0.24 ms) decryption
- **SM4-128-CBC**: 15,788 μs (15.79 ms) encryption, 11,365 μs (11.37 ms) decryption
- **Camellia-128-CBC**: 8,843 μs (8.84 ms) encryption, 6,564 μs (6.56 ms) decryption

Key observations:

- AES maintains its performance advantage with approximately 15× faster encryption and 47× faster decryption compared to SM4
- Camellia performs approximately 1.8× better than SM4 for large files
- All algorithms show asymmetric performance, with decryption generally faster than encryption
- AES's exceptional decryption speed (241 μs) suggests highly optimized parallel processing
- The performance gap between hardware-accelerated AES and software-only implementations (SM4, Camellia) becomes dramatic at scale

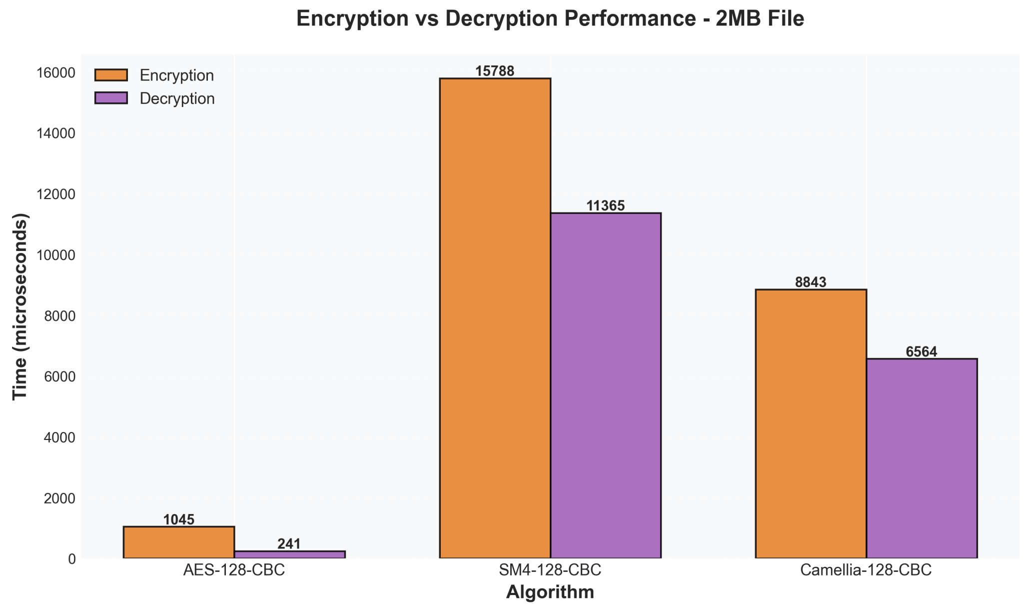

Figure 3: Encryption and Decryption performance for 2MB file. The performance gap widens significantly, with AES demonstrating substantial speed advantages over SM4 and Camellia.

3.1.4 Scalability Analysis

Examining how performance scales with file size:

- **AES**: Scales linearly and efficiently, approximately 1,913 MB/s encryption throughput for the 2MB file
- **SM4**: Shows linear scaling but at a lower throughput, approximately 126 MB/s encryption
- **Camellia**: Better than SM4 with approximately 226 MB/s encryption throughput

The consistent scaling behavior indicates that all algorithms are well-implemented, with the performance differences primarily stemming from algorithmic complexity and hardware optimization rather than implementation inefficiencies.

3.2 Verification Results

All encryption and decryption operations successfully passed verification tests. The `memcmp()` function confirmed that the decrypted data exactly matched the original plaintext for every algorithm and file size combination, demonstrating:

- Correct implementation of encryption/decryption operations
- Proper handling of padding in CBC mode
- Data integrity throughout the encryption/decryption cycle

4 Conclusions

4.1 Final Remarks

This homework demonstrated successfully the real performance factors of the three symmetric encryption algorithms we studied, highlighting that AES clearly dominates in terms of raw speed, while SM4 and Camellia remain viable options for specific use cases and platforms.

It is also very important to remember that the choice of encryption algorithm should consider not only performance (in terms of time) but also factors such as hardware support, standardization requirements and platform constraints.

All of the three algorithms analyzed provide strong cryptographic security and in many real-world scenarios, the performance differences observed here would be negligible compared to other system bottlenecks (disk I/O, network latency, etc.).