HW02

Riccardo Zucchelli 1984963

October 2024

1 Introduction

This homework focuses on showing the programming efficiency of AES, compared to other encryption algorithms like Camellia, researched and designed by Mitsubishi Electronics in 2000 and officially approved by the ISO/IEC commission, and Aria, designed by South Korean engineers in 2003, we'll be especially interested in comparing their performance relative to encryption and decryption speed.

To ensure that the results obtained are valid and reliable, it's essential to adhere to several best practices. First and foremost, using the same input file for each test is crucial to eliminate external variables that could influence the results, like the randomization of files themselves. Additionally, the key and initialization vector (IV) must remain constant across all trials, as well as the chosen mode of operation that in this case has been chosen to be Cipher Block Chaining (CBC). These precautions ensure that any differences in execution times can be attributed solely to the algorithms being tested. It's also important to conduct a significant number of trials for each algorithm, as a single test may not accurately reflect average performance, for factors as the internal CPU scheduler, and state of memory. By calculating the average of the obtained results, a more precise and meaningful assessment of each algorithm's performance can be achieved, while minimizing the impact of any anomalies or temporal fluctuations.

2 Choosing plaintext

To evaluate their speeds we'll be choosing 3 files with which we'll benchmark the algorithms, ensuring to include in its insides binary and plaintext files. This dual approach allows us to comprehensively evaluate the performance and effectiveness of the algorithms under different conditions. Choosing a text file enables us to assess how well the algorithms handle standard character encoding and formatting, which is critical for applications like document security and data transmission. Text files are common in many contexts, and ensuring that encryption and decryption preserve the integrity of the original content is essential. By using a text file, we can also examine how the algorithms manage

variations in plaintext size and structure, as well as their ability to maintain readability after decryption. On the other hand, incorporating a binary file is equally important, as it allows us to evaluate the algorithms' performance with binary files, which can include images, audio, and executable files, that often contain complex structures that differ significantly from text files. There is also usually a higher amount of entropy, which may impact their performance. We'll be using:

1. A 1Kb text file, containing the *Lorem Ipsum* text;
2. A 10Kb text file, taken from part of the *Bee Movie* script;
3. A 10Mb binary file, taken from the binaries of the VPN program *Tailscale*, cut to 10Mb according to the specifications.

The C language will be used, choosen for it's high efficiency and very low overhead, along with the library `time.h` will be used to accurately keep track of how much time has passed, and the library `openssl`, the leading de facto standard of free and open-source software for general-purpose cryptography and secure communication. Source code will be provided at the end of the document.

3 starting the test

The test will start on the 1kb file.

```
$ HW02 ./encrypt.o samples/sample_1k.txt

----------Benchmarking AES_128_CBC----------
Average encryption time: 0.0038 ms  (min 0.0016 ms  max 8.4563 ms)
Average decryption time: 0.0020 ms  (min 0.0010 ms  max 0.0456 ms)

----------Benchmarking CAMELLIA_128_CBC----------
Average encryption time: 0.0068 ms  (min 0.0053 ms  max 0.0766 ms)
Average decryption time: 0.0068 ms  (min 0.0052 ms  max 0.2795 ms)

----------Benchmarking ARIA_128_CBC----------
Average encryption time: 0.0086 ms  (min 0.0074 ms  max 0.0778 ms)
Average decryption time: 0.0083 ms  (min 0.0071 ms  max 0.0566 ms)
```

Here the test on the 10kb text file.

```
$ HW02 ./encrypt.o samples/sample_10k.txt

----------Benchmarking AES_128_CBC----------
Average encryption time: 0.0123 ms  (min 0.0080 ms  max 8.9995 ms)
Average decryption time: 0.0037 ms  (min 0.0024 ms  max 0.0456 ms)

----------Benchmarking CAMELLIA_128_CBC----------
Average encryption time: 0.0497 ms  (min 0.0435 ms  max 0.1677 ms)
Average decryption time: 0.0496 ms  (min 0.0434 ms  max 0.1620 ms)

----------Benchmarking ARIA_128_CBC----------
Average encryption time: 0.0723 ms  (min 0.0667 ms  max 1.5631 ms)
Average decryption time: 0.0695 ms  (min 0.0642 ms  max 2.4402 ms)
```

Lastly, the test on the 10mb binary file.

```
$ $HW02 ./encrypt.o samples/sample_10m.bin

----------Benchmarking AES_128_CBC----------
Average encryption time: 0.0029 ms  (min 0.0009 ms  max 8.3851 ms)
Average decryption time: 0.0020 ms  (min 0.0009 ms  max 0.0340 ms)

----------Benchmarking CAMELLIA_128_CBC----------
Average encryption time: 0.0029 ms  (min 0.0016 ms  max 0.1076 ms)
Average decryption time: 0.0029 ms  (min 0.0017 ms  max 0.0655 ms)

----------Benchmarking ARIA_128_CBC----------
Average encryption time: 0.0023 ms  (min 0.0016 ms  max 0.0423 ms)
Average decryption time: 0.0025 ms  (min 0.0017 ms  max 0.0375 ms)
```

4 Conclusions

The results above highlight several key observations. Throughout all tests conducted, AES vastly supersedes both Aria and Camellia in encryption and decryption speed, having 3 to 8 times theirs. The difference is especially noticeable while comparing AES to Aria, probably also because of the higly specialized hardware now common inside all CPUs.

Another noteworthy result is the noticeable ratio between encryption and decryption time in AES that do not present in the other two cases. Given that the transformations (SubBytes, ShiftRows, MixColumns and AddRoundKey) are fully invertable with the same computation complexity, may suggest that there are some software/hardware optimizations that make it slightly asymmetric.

Finally, as we already hinted at before, the higher entropy of the binary file made encrypting it way faster comparing to even the 1kb sample. To add to this, Text files have more predictable data, leading to higher padding and initialization overhead, which slows down encryption even more.

5 Source code

5.1 Key code

```
unsigned char KEY[] = {
    0x97, 0xd5, 0x00, 0x14, 0xbc, 0x66, 0xed, 0x5d, 0x30, 0x06, 0
        xce, 0x15, 0x5c, 0xa9, 0xe9, 0x39
};
unsigned char IV[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0
        x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
};
```

5.2 Benchmarking code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <openssl/camellia.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include "key.h"
#include <time.h>

#define ITERATIONS 10000

typedef enum {
    AES_128_CBC,
    CAMELLIA_128_CBC,
    ARIA_128_CBC
} cipher_t;

unsigned char* encrypt(cipher_t mode, const unsigned char* key,
    const unsigned char* iv, const char* plaintext, int* out_len) {
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        perror("EVP_CIPHER_CTX_new");
        exit(1);
    }

    EVP_CIPHER* cipher = NULL;
    int BLOCK_SIZE = 0;

    switch(mode) {
        case AES_128_CBC:
            cipher = EVP_aes_128_cbc();
            BLOCK_SIZE = AES_BLOCK_SIZE;
            break;
        case CAMELLIA_128_CBC:
            cipher = EVP_camellia_128_cbc();
```

```
            BLOCK_SIZE = CAMELLIA_BLOCK_SIZE;
            break;
        case ARIA_128_CBC:
            cipher = EVP_aria_128_cbc();
            BLOCK_SIZE = AES_BLOCK_SIZE;
            break;
    }
    if (EVP_EncryptInit_ex(ctx, cipher, NULL, key, iv) != 1) {
        perror("EVP_EncryptInit_ex");
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }

    int plaintext_len = strlen(plaintext);
    int ciphertext_len = plaintext_len + BLOCK_SIZE;
    unsigned char* ciphertext = (unsigned char*)malloc(
        ciphertext_len);
    if (!ciphertext) {
        perror("malloc");
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }

    int len;
    if (EVP_EncryptUpdate(ctx, ciphertext, &len, (unsigned char*)
        plaintext, plaintext_len) != 1) {
        perror("EVP_EncryptUpdate");
        free(ciphertext);
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }
    ciphertext_len = len;

    if (EVP_EncryptFinal_ex(ctx, ciphertext + len, &len) != 1) {
        perror("EVP_EncryptFinal_ex");
        free(ciphertext);
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }
    ciphertext_len += len;

    EVP_CIPHER_CTX_free(ctx);

    *out_len = ciphertext_len;
    return ciphertext;
}

unsigned char* decrypt(cipher_t mode, const unsigned char* key,
    const unsigned char* iv, const unsigned char* ciphertext, int
    ciphertext_len, int* out_len) {
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        perror("EVP_CIPHER_CTX_new");
        exit(1);
    }

    EVP_CIPHER* cipher = NULL;
```

```
    int BLOCK_SIZE = 0;
    switch(mode) {
        case AES_128_CBC:
            cipher = EVP_aes_128_cbc();
            BLOCK_SIZE = AES_BLOCK_SIZE;
            break;
        case CAMELLIA_128_CBC:
            cipher = EVP_camellia_128_cbc();
            BLOCK_SIZE = CAMELLIA_BLOCK_SIZE;
            break;
        case ARIA_128_CBC:
            cipher = EVP_aria_128_cbc();
            BLOCK_SIZE = AES_BLOCK_SIZE;
            break;
    }

    if (EVP_DecryptInit_ex(ctx, cipher, NULL, key, iv) != 1) {
        perror("EVP_DecryptInit_ex");
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }

    int plaintext_len = ciphertext_len;
    unsigned char* plaintext = (unsigned char*)malloc(plaintext_len
        + BLOCK_SIZE);
    if (!plaintext) {
        perror("malloc");
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }

    int len;
    if (EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext,
        ciphertext_len) != 1) {
        perror("EVP_DecryptUpdate");
        free(plaintext);
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }
    plaintext_len = len;

    if (EVP_DecryptFinal_ex(ctx, plaintext + len, &len) != 1) {
        perror("EVP_DecryptFinal_ex");
        free(plaintext);
        EVP_CIPHER_CTX_free(ctx);
        exit(1);
    }
    plaintext_len += len;

    EVP_CIPHER_CTX_free(ctx);

    *out_len = plaintext_len;
    return plaintext;
}

int open_file(char* filename) {
    int fd = open(filename, O_RDONLY);
```

```
    if (fd == -1) {
        perror("open");
        exit(1);
    }
    return fd;
}

// Memory map the file so that we have all of the text on a char
    array, removing the overhead of reading the file while
    benchmarking.

void* map_file(int fd) {
    struct stat st;
    if (fstat(fd, &st) == -1) {
        perror("fstat");
        exit(1);
    }
    void* memfile = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE,
        fd, 0);
    if (memfile == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    return memfile;
}

void benchmark_encryption_decryption(cipher_t mode, const char*
    plaintext, int plaintext_len, const unsigned char* key, const
    unsigned char* iv) {
    int ciphertext_len;
    double total_encryption_time = 0, total_decryption_time = 0;
    double max_encryption_time = 0, max_decryption_time = 0;
    double min_encryption_time = 1e9, min_decryption_time = 1e9;

    for (int i = 0; i < ITERATIONS; i++) {
        struct timespec start, end;

        clock_gettime(CLOCK_MONOTONIC, &start);
        unsigned char* ciphertext = encrypt(mode, key, iv,
            plaintext, &ciphertext_len);
        clock_gettime(CLOCK_MONOTONIC, &end);

        double encryption_time = (end.tv_sec - start.tv_sec) * 1e3
            + (end.tv_nsec - start.tv_nsec) / 1e6;
        total_encryption_time += encryption_time;
        if (encryption_time > max_encryption_time)
            max_encryption_time = encryption_time;
        if (encryption_time < min_encryption_time)
            min_encryption_time = encryption_time;

        int decrypted_len;

        clock_gettime(CLOCK_MONOTONIC, &start);
        unsigned char* decrypted = decrypt(mode, key, iv,
            ciphertext, ciphertext_len, &decrypted_len);
        clock_gettime(CLOCK_MONOTONIC, &end);
```

```
        double decryption_time = (end.tv_sec - start.tv_sec) * 1e3
            + (end.tv_nsec - start.tv_nsec) / 1e6;
        total_decryption_time += decryption_time;
        if (decryption_time > max_decryption_time)
            max_decryption_time = decryption_time;
        if (decryption_time < min_decryption_time)
            min_decryption_time = decryption_time;

        decrypted[decrypted_len] = '\0';

        if (strcmp(plaintext, (char*)decrypted) != 0) {
            fprintf(stderr, "Decrypted plaintext does not match
                original plaintext\n");
        }
        free(ciphertext);
        free(decrypted);
    }

    printf("Average encryption time: %f milliseconds\t(min %f ms\
        tmax %f ms)\n", total_encryption_time / ITERATIONS,
        min_encryption_time, max_encryption_time);
    printf("Average decryption time: %f milliseconds\t(min %f ms\
        tmax %f ms)\n", total_decryption_time / ITERATIONS,
        min_decryption_time, max_decryption_time);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    int fd = open_file(argv[1]);
    void* memfile = map_file(fd);

    char* plaintext = (char*)memfile;
    int plaintext_len = strlen(plaintext);

    printf("\n---------Benchmarking AES_128_CBC---------\n");
    benchmark_encryption_decryption(AES_128_CBC, plaintext,
        plaintext_len, KEY, IV);
    printf("\n---------Benchmarking CAMELLIA_128_CBC---------\n");
    benchmark_encryption_decryption(CAMELLIA_128_CBC, plaintext,
        plaintext_len, KEY, IV);
    printf("\n---------Benchmarking ARIA_128_CBC---------\n");
    benchmark_encryption_decryption(ARIA_128_CBC, plaintext,
        plaintext_len, KEY, IV);

    struct stat st;
    if (fstat(fd, &st) == -1) {
        perror("fstat");
        exit(1);
    }

    if (munmap(memfile, st.st_size)) {
        perror("munmap");
        exit(1);
```

```
    }
    close(fd);
    return 0;
}
```