Principles of Secure Operating Systems - Coursework

Sino-British Collaborative Programme, Oxford Brookes University & Chengdu University of Technology

Semester 2, Date, 2024

成 都 理 工 大 学 中 英 合 作 项 目

CHENGDU UNIVERSITY OF TECHNOLOGY SINO-BRITISH COLLABORATIONS

A coursework submitted in partial fulfillment of the requirements for the module – Principles of Secure Operating Systems

Title: **Understanding and Modifying An OS -File Encryption**

Name:Albert

Major:	BSc (Hons) Computer Science
Student Number:	202018010212
Name of Course:	Principles of Secure Operating System
Name of Lecturer:	Dr. Chiagoziem Chima Ukwuoma/Dr. Joojo
University:	Oxford Brookes University

Table of Contents

List of Figures

List of Tables

Section 1

1. Encryption of Encryption Algorithm

This work focused on encrypting files using Advanced Encryption Standard Cipher Block Chaining (AES-CBC), an encryption algorithm is popular among all file encryption [1]. Specifically, uses binary key at length of 256 bits to encrypt file therefore having a high level of security [2].

The encryption process starts with the generation of a random encryption key and initialization vector (IV) upon reading the input file. This randomness ensures the system's overall security. The CBC mode of AES encryption is used, requiring an initial vector for each plaintext block [3]. Data is read, applied PKCS#7 padding, aligned to the required block size, and encrypted. The encrypted data is stored in an output file with the suffix automatically modified, while the key and IV are securely recorded in a database file for future decryption. This approach maintains data integrity and confidentiality throughout transmission and storage.

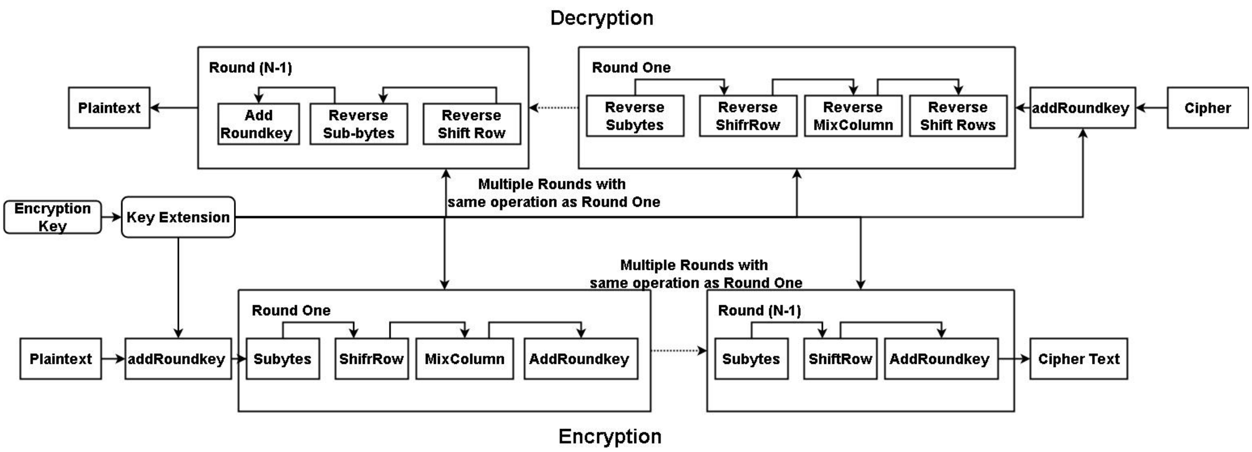


Figure 1: AES Encryption Details

Section 2

2. Functional & Non-Functional & Security Features

To ensure the successful development of the encryption software patch, Functional, Non-Functional requirements and Security Features are considered. Table 1 outlines the detailed specifications, categorizing the requirements and providing a structured framework for the design and implementation phases.

Table 1: Functional Requirements, Non-Functional Requirements and Security Features

Functional Requirements	
Key & IV Generation	The system must generates a unique encryption key and initial vector for each encryption session and store in the database file in secure location
File Encryption	The software must take the input of the correct file path, be able to open and read the contents of the file, and therefore, using AES-256-CBC mode to encrypt the file content
File Suffix Change	Once the file is encrypted, the suffix should be changed into names ".en" that will not be opened normally. "en" is short for "Encrypted"
File Decryption	Software must take a input file path of encrypted files and search the corresponding key to each file input, and restore the encrypted file to original files
Database Management	During the encryption, encryption key and IV will be safely put into a file in the OS
File integrity	The file contents must be the exact same as before encryption
Various Types of File	The encryption algorithm should be able to operate on different types of files
Multi-thread	The software should implement multi-thread mechanism to make each encryption or decryption procedure parallel to each other , thereby enhancing performance, increase the speed of the process
Non-Functional Requirements	
Safety Issues	The key management file normally should not be change by unauthorized users

Performance	The encryption and decryption should complete within a certain amount of time
Stability	The software should handle most of the common error, such as user error input, or handling multiple files with large contents
User Friendly	The command line should be able to tell what users have to do clearly and show the message of error occurred
Multi-thread Monitor	The software should display the start of each thread to monitor whether the multi-thread process is activated or not, thereby ensuring performance
Maintainable	The code should be separated into modules which is convenient for maintenance
Expandable	The code should be expandable for future complex function on, such as integration of more encryption mechanisms
Security Features	
File Access Control	Certain file must be restricted to unauthorized access, such as database and encryption files
Input Safety	Robust input validation and error handling towards user input must be implemented
Multi-thread Security	Particular mechanism should be implemented to avoid concurrent access issues
System Integration Security	The integration into the OS must avoid critical system file modification and safely incorporate with existing software

Section 3

3. Design of Software

The encryption algorithm is built with multiple modules, with each connecting to corresponding ones. The module design and interaction with OS is displayed in Figure 1and Figure 2, where Figure 1 represents the internal part of the algorithm that operates within the thread, responsible for particular file encryption and decryption, whereas Figure 2 contains the process file module and provides multi-threading through the control of OS.

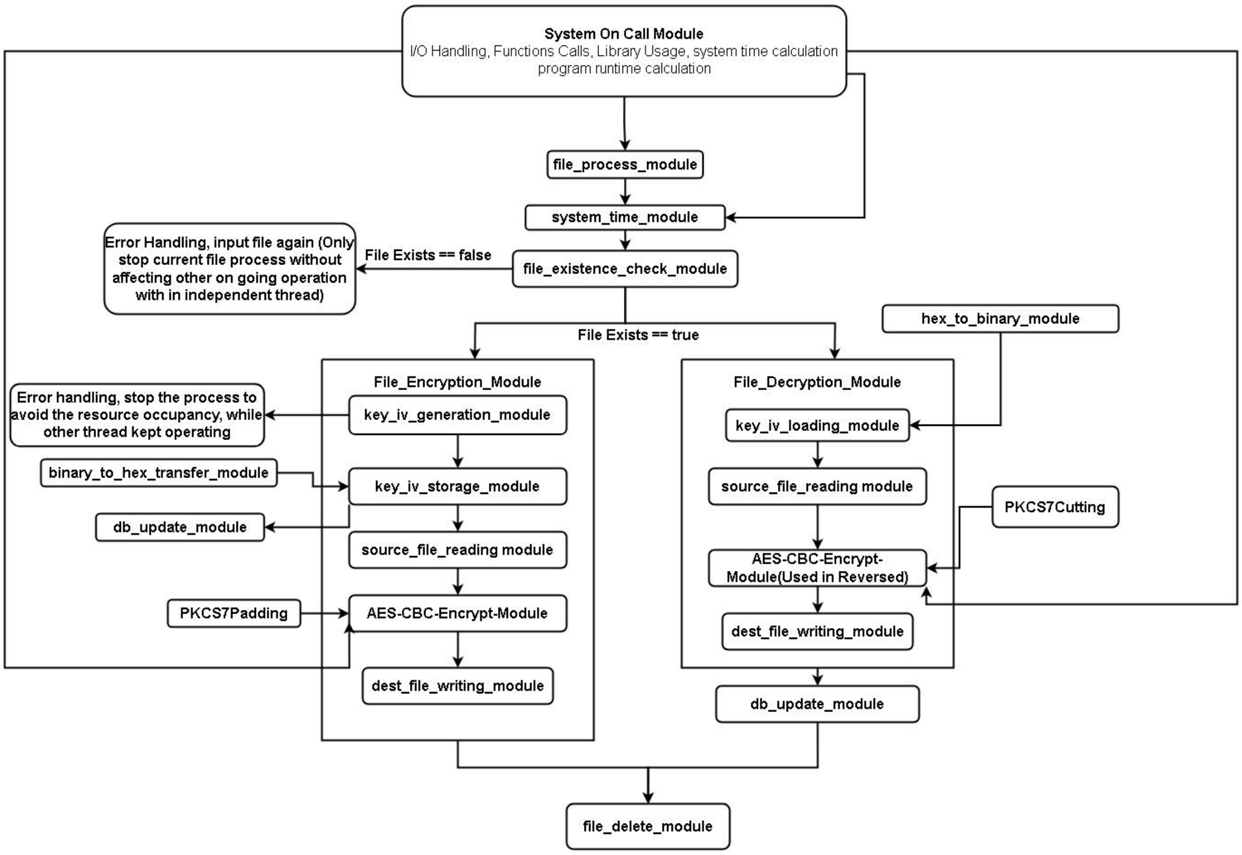


Figure 2: Inner Design of Process File module

The process-file module is the main part of the file encryption system, handling both encryption and decryption tasks. It verifies the input file contents existence and validity using system calls like fopen() or fclose(). It generates a random encryption key and initialization vector using the OpenSSL library, which utilizes the OS's sources. These keys and IVs are saved and queued for database storage through secure file I/O operations.

The module applies AES encryption or decryption based on the user's choice, leveraging OS memory management and OpenSSL functions. The resulting data is written to an output file using system calls such as fwrite(), and the original file is securely deleted with remove() to maintain data confidentiality. System time is captured with gettimeofday() to measure process duration, ensuring efficient performance tracking. This extensive OS interaction ensures secure, efficient, and reliable file handling throughout the encryption and decryption processes.

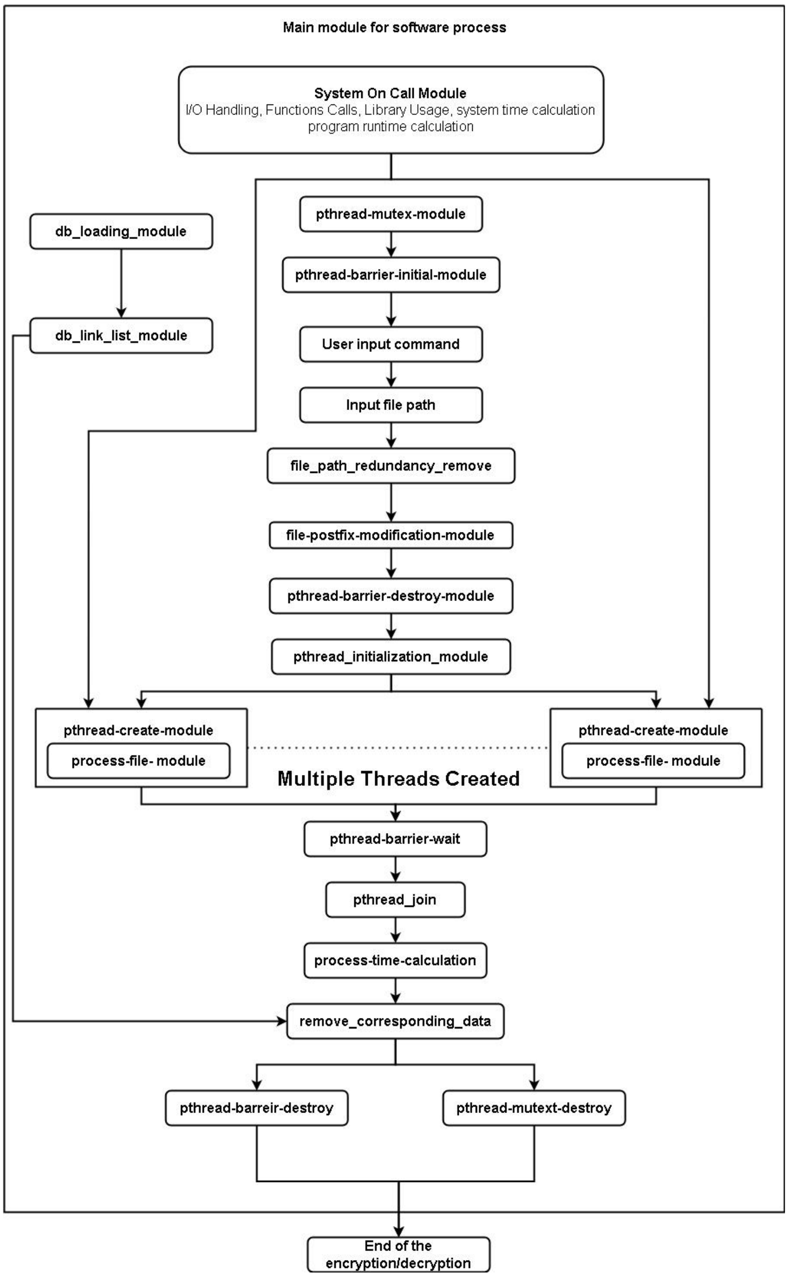


Figure 3: Main module

The main module provides the direct interaction with user, it manages user inputs and multi-threaded programs in the file encryption system. It initializes pthread mutex and barrier for efficient thread synchronization and resource sharing, leveraging the threading ability of OS [4]. The module prompts input command of user, adjusts file extensions, and restructure the pthread barrier. The core part is the loop where it creates multiple threads dynamically, managed by OS based on input file number. Program run time can be calculated through system time stamp. These kinds of extensive interaction ensure the algorithm runs efficiently and safely.

Section 4

4. Implementation of the Software

This section displays the implementation of C code. Figure 4 represents the setting part for the whole program, where the headers, variables, and linked list are defined.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <dirent.h>
#include <sys/stat.h>
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <pthread.h>
#include <sys/time.h>  // For gettimeofday
#include <unistd.h>    // For sleep
// #include <PKCS7.h>
/*This can be excluded since the main code has define the attributes
This is only used for further expandability for the exncryption code
*/
#define MESSAGE_PIECE_LEN   16
#define MAX_FILES 10
// Define a structure for list node
typedef struct ListNode {
    char filename[256];  // Assuming maximum filename length is 255 characters
    unsigned char key[32];
    unsigned char iv[16];
    struct ListNode *next;
} ListNode;

// Define a structure for list
typedef struct {
    ListNode *head;
} List;

typedef struct {
    char inputFilename[256];
    char outputFilename[256];
    char dbFilename[256];
    struct timeval start_time, end_time;
    char operation;
    List* list;
} ThreadData;

typedef struct {
    unsigned char key[32];
    unsigned char iv[16];
    char outputFilename[256];  // to link the encryption key with the file
} KeyIV;
```

Figure 4: Headers, Variables, and Struct Definition

Figure 5 shows where methods are implicitly pre-defined. The reason is that C language operates sequentially, error would occur if the method is called but without declaring ahead.

```c
// global list for key and IV storage
KeyIV key_iv_list[MAX_FILES];
int key_iv_count = 0;  // to keep track of the elements within the list


pthread_barrier_t start_barrier;
pthread_mutex_t db_mutex;

//initialize an Linked list to store the information from database(A secured located file)
void init_list(List *list);
//adding information into the linked list
void insert_list_entry(List *list, const char *filename, const unsigned char *key, const unsigned char *iv);
//remove the corresponding files to update the database
void remove_list_entry(List *list, const char *filename);
//free the allocated memory
void free_list(List *list);
//the main part of file processing, including encyrptiona and decryption
void* process_file(void* arg);
//the specific method for encryption
void encrypt_single_file(const char *inputFilename, const char *outputFilename, const char *db_filename);
//the specific method for decryption
int decrypt_single_file(const char *inputFilename, const char *outputFilename, const char *db_filename);
// check if there are ducplicated files with in the output file when decryptin or encryptin
int check_file_duplicate(const char * filename, const char *db_filename);
//update the database
void update_db_file(const char *db_filename, List *list);
//delete specific files
int delete_file(const char *filename);
//Save each unique key and vector of each file
void save_key_iv(const char *db_filename, const char *output_filename, const unsigned char *key, const unsigned char *iv, int key_len, int iv_len);
//load the key store before for decrypting
int load_key_iv(const char *db_filename, const char *filename, unsigned char *key, unsigned char *iv, int key_len, int iv_len);
void print_list(const List *list);
```

Figure 5: Methods implicitly defined

Figure 6 defines the basic methods for operation on linked list which will then use for file record removal within the database file where key and iv are also stored.

```c
// Function to insert a filename, key, and IV into the list
void insert_list_entry(List *list,
const char *filename, const unsigned char *key, const unsigned char *iv) {
    ListNode *new_node = (ListNode *)malloc(sizeof(ListNode));
    if (new_node == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    strcpy(new_node->filename, filename);
    memcpy(new_node->key, key, sizeof(new_node->key));
    memcpy(new_node->iv, iv, sizeof(new_node->iv));
    new_node->next = list->head;
    list->head = new_node;
}

/*Remove the file in the list, for example when decrypting the file,
*it should remove specific content related to the this file
*/
void remove_list_entry(List *list, const char *filename) {
    ListNode *current = list->head;
    ListNode *prev = NULL;
    while (current != NULL) {
        if (strcmp(current->filename, filename) == 0) {
            if (prev == NULL) {
                list->head = current->next;
            } else {
                prev->next = current->next;
            }
            free(current);
            return;
        }
        prev = current;
        current = current->next;
    }
    printf("File not found in list: %s\n", filename);
}

// Function to free the memory occupied by the list
void free_list(List *list) {
    ListNode *current = list->head;
    while (current != NULL) {
        ListNode *temp = current;
        current = current->next;
        free(temp);
    }
    list->head = NULL;
}
```

Figure 6: Linked List Basic Methods Definition

Figure 7 displays the hexadecimal and binary transfer for key storage, filed reading, database updating. Generate_key_and_iv() randomly produces encryption keys and IVs.

```c
char *bin2hex(const unsigned char *bin, int len) {
    char *hex = malloc(len * 2 + 1);
    for (int i = 0; i < len; i++) {
        sprintf(hex + i * 2, "%02X", bin[i]);
    }
    hex[len * 2] = '\0';
    return hex;
}

// Function to convert hex string to binary data
void hex2bin(const char *hex, unsigned char *bin, int len) {
    for (int i = 0; i < len; i++) {
        sscanf(hex + i * 2, "%02hhX", &bin[i]);
    }
}

/*
Generate random AES key and IV
Randomly generates the AES encryption
key and Initial vector based on the library of openssl
*/
int generate_key_and_iv(unsigned char *key,
unsigned char *iv, int key_size, int iv_size) {
    if (!RAND_bytes(key, key_size)) {// using random bytes to encrypt
        fprintf(stderr, "Error generating random key.\n");
        return 0;
    }
    if (!RAND_bytes(iv, iv_size)) {//if
        fprintf(stderr, "Error generating random IV.\n");
        return 0;
    }
    return 1;
}
```

Figure 7: Key Generation and Binary-Hexadecimal Transfer

Figure 8 shows the storage and loading of the key using binary-hexadecimal transfer module mentioned above. Key, IV storage and loading happen in encryption and decryption respectively.

```c
/*Saving the key to the database file called (key_iv_db.ent)*/
void save_key_iv(const char *db_filename, const char *output_filename,
const unsigned char *key, const unsigned char *iv, int key_len, int iv_len) {
    char *hex_key = bin2hex(key, key_len);
    char *hex_iv = bin2hex(iv, iv_len);

    pthread_mutex_lock(&db_mutex); // protect the file using mutex
    FILE *db_file = fopen(db_filename, "a");
    if (db_file) {
        fprintf(db_file, "%s,%s,%s\n", output_filename, hex_key, hex_iv);
        fclose(db_file);
    } else {
        fprintf(stderr, "Failed to open database file for writing.\n");
    }
    pthread_mutex_unlock(&db_mutex); // unlock the mutex

    free(hex_key);
    free(hex_iv);
}


// Load key and IV from a database file
int load_key_iv(const char *db_filename, const char *filename,
unsigned char *key, unsigned char *iv, int key_len, int iv_len) {
    FILE *db_file = fopen(db_filename, "r");
    if (!db_file) {
        fprintf(stderr, "Failed to open database file for reading.\n");
        return 0;
    }
    char line[256];
    while (fgets(line, sizeof(line), db_file)) {
        //this here helps loading the correct structure of data in database file
        char *saved_filename = strtok(line, ",");
        char *saved_key = strtok(NULL, ",");
        char *saved_iv = strtok(NULL, ",");
        if (strcmp(saved_filename, filename) == 0) {
            hex2bin(saved_key, key, key_len);
            hex2bin(saved_iv, iv, iv_len);
            fclose(db_file);
            return 1;
        }
    }
    fclose(db_file);
    return 0;
}
```

Figure 8: Key Storage and Loading

Figure 9 displays the file reading and writing functions, which operates on the file before and after encryption or decryption. Database function includes reading the database data structure with the linked list, where each nodes represents a specific file path following with IV and key. The update database function is used to overwrite the database based on the linked contents, thereby achieving updating the database efficiently. File existence check assists handling error input of file name, construct the robustness of program.

```c
/*reading the file (Notice,
*this is reading the file thats about to be encrypt or decrypt)
*/
unsigned char *read_file(const char *filename, int *length) {
    FILE *file = fopen(filename, "rb");
    if (!file) {
        perror("Failed to open file for reading");
        return NULL;
    }

    fseek(file, 0, SEEK_END);
    *length = ftell(file);
    fseek(file, 0, SEEK_SET);

    unsigned char *data = malloc(*length);
    if (data) {
        fread(data, 1, *length, file);
    }
    fclose(file);
    return data;
}


/*Function to read data from the database file and build the list
read the database file only
*/
void read_db_file(const char *db_filename, List *list) {
    FILE *file = fopen(db_filename, "r");
    if (!file) {
        fprintf(stderr, "Failed to open file: %s\n", db_filename);
        return;
    }
    char line[256];
    while (fgets(line, sizeof(line), file)) {
        char *filename = strtok(line, ",");
        char *hex_key = strtok(NULL, ",");
        char *hex_iv = strtok(NULL, ",");
        unsigned char key[32], iv[16];
        hex2bin(hex_key, key, sizeof(key));
        hex2bin(hex_iv, iv, sizeof(iv));
        insert_list_entry(list, filename, key, iv);
    }
    fclose(file);
}
```

```c
// Function to update the database file after decryption
void update_db_file(const char *db_filename, List *list) {
    // Open file in write mode to overwrite
    FILE *db_file = fopen(db_filename, "w");
    if (!db_file) {
        fprintf(stderr, "Failed to open database file for writing.\n");
        return;
    }
    ListNode *current = list->head;
    while (current != NULL) {
        char *hex_key = bin2hex(current->key, sizeof(current->key));
        char *hex_iv = bin2hex(current->iv, sizeof(current->iv));
        fprintf(db_file, "%s,%s,%s\n", current->filename, hex_key, hex_iv);
        free(hex_key);
        free(hex_iv);
        current = current->next;
    }
    fclose(db_file);
}


//Write the content into the desired file,
//such as encrypted----> decrypt / decrypt--->encrypt
void write_file(const char *filename,
const unsigned char *data, int length) {
    FILE *file = fopen(filename, "wb");
    if (!file) {
        perror("Failed to open file for writing");
        return;
    }
    fwrite(data, 1, length, file);
    fclose(file);
}

//check whether whether the input file exists
int check_file_exists(const char *filename) {
    FILE *file = fopen(filename, "rb");
    if (file) {
        fclose(file);
        return 1; // file exist
    }
    return 0; // file does not exist
}
```

Figure 9: Source File Reading and Destination File Writing, Database Reading and Modifying, and File Existence Verification

Figure10 illustrates how algorithm checks duplication of the output the file, and the way to delete file, for example, it should delete source files after decryption and encryption.

PKCS7 padding is one of the paramount points of maintaining the correct of encryption when input is not at required length which happens constantly, whereas cutting operates in decryption, eliminating the padding part given from encryption process which delete the unreadable characters following the plaintext, offers the same content before encryption.

```c
//chekcing the duplication
int check_file_duplicate(const char * filename, const char *db_filename){
    FILE *file = fopen(db_filename, "r");
    if (!file) {
        fprintf(stderr, "Failed to open database file.\n");
        return 0;
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
        // each line contains filename, encryption key, initial vector
        char *comma = strchr(line, ',');
        if (comma) {
            *comma = '\0';  // cut the string, only the file name remains
        }
        if (strcmp(line, filename) == 0) {
            fclose(file);
            return 1;  // duplicate found
        }
    }

    fclose(file);
    return 0;
}

//delete the source file after encryptin / decrypting
int delete_file(const char *filename){
    if(remove(filename) == 0){
        return 1;

    }else {
        perror("Failed to delete file");
        return 0;
    }
}
```

```c
// Give the input proper padding
unsigned int PKCS7Padding(char *p, unsigned int plen)
{
    unsigned int padding_len = 0;
    unsigned char padding_value = 0;

    if(0 < plen){
        if(0 ==(plen % MESSAGE_PIECE_LEN)){
            padding_value = MESSAGE_PIECE_LEN;
        }
        else{
            padding_value = MESSAGE_PIECE_LEN - (plen % MESSAGE_PIECE_LEN);
        }

        padding_len = (plen / MESSAGE_PIECE_LEN + 1) * MESSAGE_PIECE_LEN;

        for( ; plen < padding_len; plen++){
            p[plen] = padding_value;
        }
    }
    return padding_len;
}


unsigned int PKCS7Cutting(char *p, unsigned int plen, unsigned int max_len) {
    if (plen == 0 || plen > max_len) {
        return 0;
    }

    unsigned char lastByte = p[plen - 1];
    if (lastByte > MESSAGE_PIECE_LEN || lastByte == 0 || lastByte > plen) {
        return 0;
    }

    unsigned int padding_len = lastByte;

    for (unsigned int i = 0; i < padding_len; i++) {
        if (p[plen - 1 - i] != lastByte) {
            return 0;
        }
    }
    return plen - padding_len;
}
```

Figure 10: Duplicate File Verification, File Deletion, PKCS7 Padding and Cutting.

Figure 11 demonstrates the encryption module of the algorithm. It integrates with the former functions such as PKCS7 padding, file existence and duplicate verification, key storage etc. It also uses the core of the algorithm, “AES_cbc_encrypt’ functions which completes corresponding calculation procedure in Figure 1.

```c
//encrypting the file with specific procedure
void encrypt_single_file(const char *inputFilename,
const char *outputFilename, const char *db_filename) {
    unsigned char key[32], iv[16];
    AES_KEY aes_key;

    // check the existance of file
    if (!check_file_exists(inputFilename)) {
        fprintf(stderr, "Input file does not exist.
        Please check the file name and try again.\n");
        return;
    }

    // key and IV generation
    if (!generate_key_and_iv(key, iv, sizeof(key), sizeof(iv))
     || AES_set_encrypt_key(key, 256, &aes_key) < 0) {
        fprintf(stderr, "Failed to set encryption key.\n");
        return;
    }
    save_key_iv(db_filename, outputFilename,
    key, iv, sizeof(key), sizeof(iv));

    // read the file
    FILE *file = fopen(inputFilename, "rb");
    if (!file) {
        perror("Failed to open file for reading");
        return;
    }

    fseek(file, 0, SEEK_END);
    int data_len = ftell(file);
    rewind(file);

    // allocates the memory for AES block
    int size = data_len + AES_BLOCK_SIZE;
    unsigned char *file_data = (unsigned char *)malloc(size);
    if (!file_data) {
        fprintf(stderr, "Failed to allocate memory for file data.\n");
        fclose(file);
        return;
    }

    // read the file into the allocated memory
    fread(file_data, 1, data_len, file);
    fclose(file);

    // Padding for input data
    unsigned int padded_data_len
    = PKCS7Padding((char*)file_data, data_len);
    if (padded_data_len == 0) {
        fprintf(stderr, "Padding failed.\n");
        free(file_data);
        return;
    }

    // the length after encryption is based on the padding
    int out_len = ((padded_data_len + AES_BLOCK_SIZE - 1)
    / AES_BLOCK_SIZE) * AES_BLOCK_SIZE;
    unsigned char *out_data = (unsigned char*)malloc(out_len);
    if (!out_data) {
        fprintf(stderr, "Failed to allocate memory for encrypted data.\n");
        free(file_data);
        return;
    }

    // AES encryptin method, called from the library
    AES_cbc_encrypt(file_data, out_data,
    padded_data_len, &aes_key, iv, AES_ENCRYPT);
    write_file(outputFilename, out_data, out_len);

    // release the memory
    free(file_data);
    free(out_data);
    printf("Encryption completed. Output file is %s\n", outputFilename);
}
```

Figure 11: Encryption Module

Figure 12 illustrates the decryption module where key and IV loading is applied for unique encryption key and IV mapping. “AES_cbc_encrypt” function here, is utilized in the reversed way, furthermore, PKCS7 cutting is applied as well.

```c
int decrypt_single_file(const char *inputFilename, const char *outputFilename, const char *db_filename) {
    unsigned char key[32], iv[16];
    AES_KEY aes_key;

    if (!check_file_exists(inputFilename)) {
        fprintf(stderr, "Input file does not exist for decryption. Please check the file name and try again.\n");
        return 0;
    }

    if (!load_key_iv(db_filename, inputFilename, key, iv, sizeof(key), sizeof(iv)) || AES_set_decrypt_key(key, 256, &aes_key) < 0) {
        fprintf(stderr, "Failed to set decryption key or no key/IV pair found.\n");
        return 0;
    }

    int data_len;
    unsigned char *encrypted_data = read_file(inputFilename, &data_len);
    if (!encrypted_data) return 0;

    int out_len = data_len;
    unsigned char *decrypted_data = malloc(out_len);
    if (!decrypted_data) {
        free(encrypted_data);
        return 0;
    }

    AES_cbc_encrypt(encrypted_data, decrypted_data, data_len, &aes_key, iv, AES_DECRYPT);

    unsigned int actual_data_len = PKCS7Cutting((char*)decrypted_data, out_len, out_len);
    if (actual_data_len == 0) {
        fprintf(stderr, "Failed to remove PKCS7 padding.\n");
        free(encrypted_data);
        free(decrypted_data);
        return 0;
    }

    write_file(outputFilename, decrypted_data, actual_data_len);

    free(encrypted_data);
    free(decrypted_data);
    printf("Decryption completed. Output file is %s\n", outputFilename);
    return 1; // indicate success
}
```

Figure 12: Decryption Module

Process file within Figure 13 represents one of the most important parts of the algorithm logic. It operates inside the thread that OS offers, which may encounter resource rivalry, memory competition, for instance, accessing the same the same file when checking for duplication in database, or deleting file from same folder. And therefore, a mutex locker is provided to prevent similar situation from happening, guaranteeing the robustness, efficiency and security on system resource level, especially in multi-threaded system.

```c
//processing the file with multi-thread
void* process_file(void* arg) {//with in this, it contains operation of multi thread
    ThreadData* data = (ThreadData*)arg;

    pthread_barrier_wait(&start_barrier);// starting the barrier
    gettimeofday(&data->start_time, NULL);
     // print the start time of the thread
    printf("Thread processing %s started at %ld seconds and %ld microseconds.\n",
           data->inputFilename, data->start_time.tv_sec, data->start_time.tv_usec);


    pthread_mutex_lock(&db_mutex);//the process is protected by mutex lock
    int is_duplicate = check_file_duplicate(data->outputFilename, data->dbFilename);
    pthread_mutex_unlock(&db_mutex);

    if (is_duplicate) {
        fprintf(stderr, "Output filename %s already exists in the database. Please choose a different name.\n", data->outputFilename);
        return NULL;
    }
    int success = 0;
    if (data->operation == 'e') {//encryption based on the command of user
        encrypt_single_file(data->inputFilename, data->outputFilename, data->dbFilename);
        success = 1;
    } else if (data->operation == 'd') {//decryption based on the command of user
        success = decrypt_single_file(data->inputFilename, data->outputFilename, data->dbFilename);
    }

    // Attempt to delete the input file after encryption/decryption
    if (!success) {
        fprintf(stderr, "Failed to process the file: %s.\n", data->inputFilename);
    } else {
        if (!delete_file(data->inputFilename)) {
            fprintf(stderr, "Failed to delete the file %s.\n", data->inputFilename);
        }
    }

    gettimeofday(&data->end_time, NULL);
    printf("Thread processing %s ended.\n", data->inputFilename);
    return (void*)(intptr_t)success;
}
```

Figure 13: File Process Module

Figure 14 to Figure 15 illustrates the full process combining all former functions. And it is where multi-threaded operation takes places. It first loads the database into the linked list with corresponding nodes representing unique file's path, reason is that, multiple accessing may work sequentially, however the multiple access within multi-thread which is parallel to each other, even with mutex protection, unknown error could occur as well. Therefore, an individual place for storing the content from database is reasonable and direct.

```c
int main() {// the main logic of the program
    ThreadData data[MAX_FILES];
    // the location of the database file
    char dbFilename[] = "/mnt/hgfs/SFS/TestingCode/key_iv_db.ent";
    // char dbFilename[] = "/var/lib/EncryptionDB/key_iv_db.ent";
    char option;
    int fileCount = 0;
    // Initialize the list for each option selection

    List list;
    free_list(&list);
    init_list(&list);
    read_db_file(dbFilename, &list); // Read database file once

    pthread_mutex_init(&db_mutex, NULL);// Start the mutex
    // initialize the barrier as the mas file number
    pthread_barrier_init(&start_barrier, NULL, MAX_FILES);

    struct timeval program_start, program_end;
    gettimeofday(&program_start, NULL);

    while (1) {
        char optionInput[10];  // Reasonable length for user input
        option = 0;
        printf("Do you want to (E)ncrypt, (D)ecrypt, or (Q)uit? Enter 'E', 'D', or 'Q': ");
        fgets(optionInput, sizeof(optionInput), stdin);
        if (strchr(optionInput, '\n') == NULL) {
            while (getchar() != '\n');  // clear the buffer
        }
        for (int i = 0; optionInput[i]; i++) {
            optionInput[i] = tolower(optionInput[i]);
        }
        if (strcmp(optionInput, "e\n") == 0) {
            option = 'e';
        } else if (strcmp(optionInput, "d\n") == 0) {
            option = 'd';
        } else if (strcmp(optionInput, "q\n") == 0) {
            option = 'q';
        } else {
            printf("Invalid option. Please enter 'E', 'D', or 'Q'.\n");
            continue;
        }

        if (option == 'q') {
            printf("Exiting program.\n");
            break;
        }
```

Figure 14: Main Module Part 1

Figure 15 shows the process of the basic modification on input file path, removing the redundances of file path. Furthermore, it changes the suffix into “.en” which will normally not be recognizable and opened on the system, thus protecting encrypted data.

```c
    printf("Enter the file paths to encrypt/decrypt, separated by spaces: ");
    char buffer[1024];
    fgets(buffer, sizeof(buffer), stdin);

    buffer[strcspn(buffer, "\n")] = 0; // Remove newline
    remove_apostrophes(buffer);
    char *token = strtok(buffer, " ");
    fileCount = 0;

    // change the suffix of each file
    while (token && fileCount < MAX_FILES) {
        strcpy(data[fileCount].inputFilename, token);
        strcpy(data[fileCount].outputFilename, token);
        strcpy(data[fileCount].dbFilename, dbFilename);
        data[fileCount].list = &list;  // Pass the list pointer
        char *dot = strrchr(data[fileCount].outputFilename, '.');
        if (option == 'e') {
            if (dot) {
                strcpy(dot, ".en");  // Change extension for encryption
                data[fileCount].operation = 'e';
            } else {
                fprintf(stderr, "Error: File does not have an extension.
                Please enter a valid file with extension.\n");
                break; // Skip this file and continue with the next token
            }
        } else if (option == 'd') {
            if (dot && strcmp(dot, ".en") == 0) {
                //removing the specific head file in the database
                // remove_list_entry(&list, data[fileCount].inputFilename);
                strcpy(dot, ".txt");  // Change extension for decryption
                data[fileCount].operation = 'd';
            } else {
                fprintf(stderr, "Error: File for
                decryption does not have the expected .en extension.\n");
                break; // Skip this file and continue with the next token
            }
        }
        fileCount++;
        token = strtok(NULL, " ");
    }

    // Initialize barrier with the actual number of files + 1 for main thread
    pthread_barrier_destroy(&start_barrier);
    pthread_barrier_init(&start_barrier, NULL, fileCount + 1);
```

Figure 15: Main Module Part 2

Figure 16 shows how multi-threading is used. First, the barrier is reinitialized with “fileCount + 1” to guarantee that the main thread waits for all worker threads to be ready before proceeding. This synchronization ensures that no thread processes anything before it. Threads are generated to process file encryption and decryption concurrently, which dramatically improves performance. The mutex “db_mutex” protects shared resources, such as the database list, while providing thread-safe operations. For example, during decryption, the mutex locks the database list to prevent concurrent alterations, hence ensuring data integrity and preventing race circumstances [4]. This approach allows for efficient and safe multi-threaded file processing.

```c
        for (int i = 0; i < fileCount; i++) {
        }
        //wait for each thread to be over
        pthread_barrier_wait(&start_barrier);

        for (int i = 0; i < fileCount; i++) {
            void* status;
            //multi-thread stops and joined together
            pthread_join(threads[i], NULL);
             // return the void pointer and interpret the status of decryption
            int success = (int)(intptr_t)status;

            long microseconds = (data[i].end_time.tv_sec - data[i].start_time.tv_sec)
            * 1000000L + (data[i].end_time.tv_usec - data[i].start_time.tv_usec);
            printf("Thread processing %s ran for %ld
             microseconds.\n", data[i].inputFilename, microseconds);

            // check the success of decryption
            if (data[i].operation == 'd' && success) {
                pthread_mutex_lock(&db_mutex);
                printf("Before removing entry for %s:\n\n", data[i].inputFilename);
                print_list(&list);
                remove_list_entry(&list, data[i].inputFilename);
                printf("After removing entry for %s:\n\n", data[i].inputFilename);
                print_list(&list);
                pthread_mutex_unlock(&db_mutex);
            }
        }
        pthread_barrier_destroy(&start_barrier);
        pthread_mutex_destroy(&db_mutex);
        if (option == 'd') {
            printf("This is the lit \n\n");
            print_list(&list);
            printf("Above is the lit \n\n");
            update_db_file(dbFilename, &list); // Update database after all decryption threads are complete
        }
        // Free the list after processing all files for this option
    }
    gettimeofday(&program_end, NULL);
    long total_program_time = (program_end.tv_sec - program_start.tv_sec) * 1000000L + (program_end.tv_usec - program_start.tv_usec);
    printf("Total program run time: %ld microseconds\n", total_program_time);
    free_list(&list);
    pthread_mutex_destroy(&db_mutex);
    return 0;
}
```

Figure 16: Main Module Part 3

Section 5

5. Testing Plan for Validating Your Software

Before integrating into the Operating system, the program is operated in the share folder which only operates with the Linux based environment. The test process is displayed as follow.

Table 2: Basic Technology and Environment

Operating System	Ubuntu 20.0.04
Compiling Editor	GCC
Library	OpenSSL, SYSTEM (Only significant ones that are include the in the test is written over here, most of others could be seen in section 4 in the header file include)
Extra Header	PKCS#7.h (Written manually, not a standard header, which is then integrated into /usr/include/ of Ubuntu OS)
Testing Environment	Ubuntu (Linux Based)

Test Case 1: Basic Function Test

1. Key & IV Generation and Storage Test

Table 3: Key Storage Test Requirements

Key & IV Generation & Storage
1) The Key and IV should be generated randomly, unique for each file
2) The Key and IV must store within a database file
3) The structure should be file path, IV Key

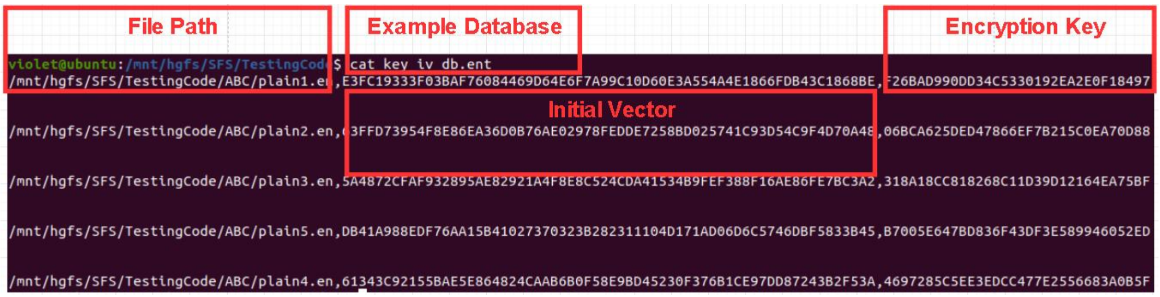


Figure 17:Random Key and IV Generation Test Results

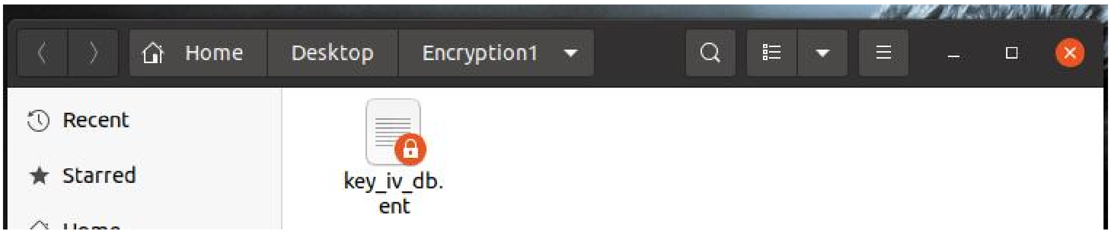


Figure 18: Locked Database, Forbidden from Modification with in OS

2. Text File Encryption

Table 4: Text Encryption Requirements

Text File Encryption Test
1) Any text files, should be encrypted or decrypted
2) File suffix should be changed into “.en” when encryption, “.txt” after decryption
3) The content should be human-unreadable after encryption

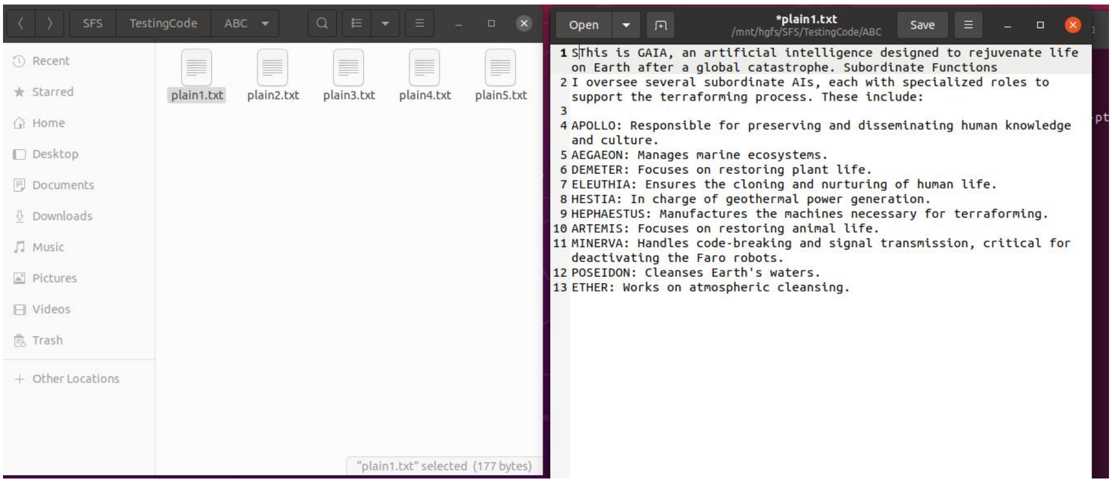


Figure 19: The text file mounted onto Ubuntu Share-Folder (SFS in my case)

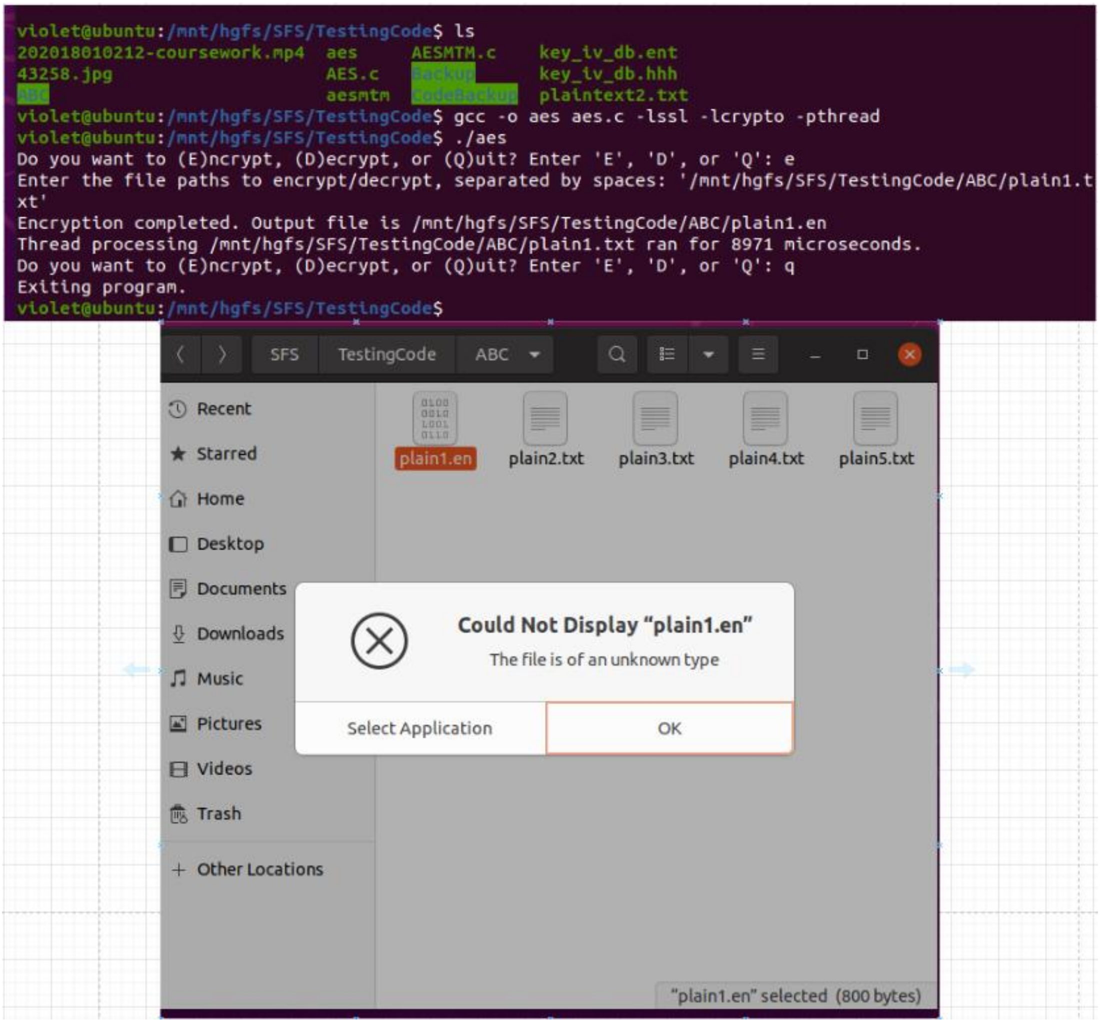


Figure 20: After Encryption

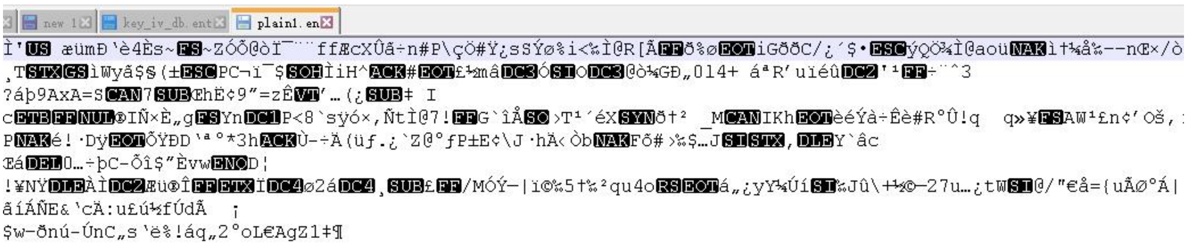


Figure 21: If compulsorily opened in Windows Share-folder with notepad

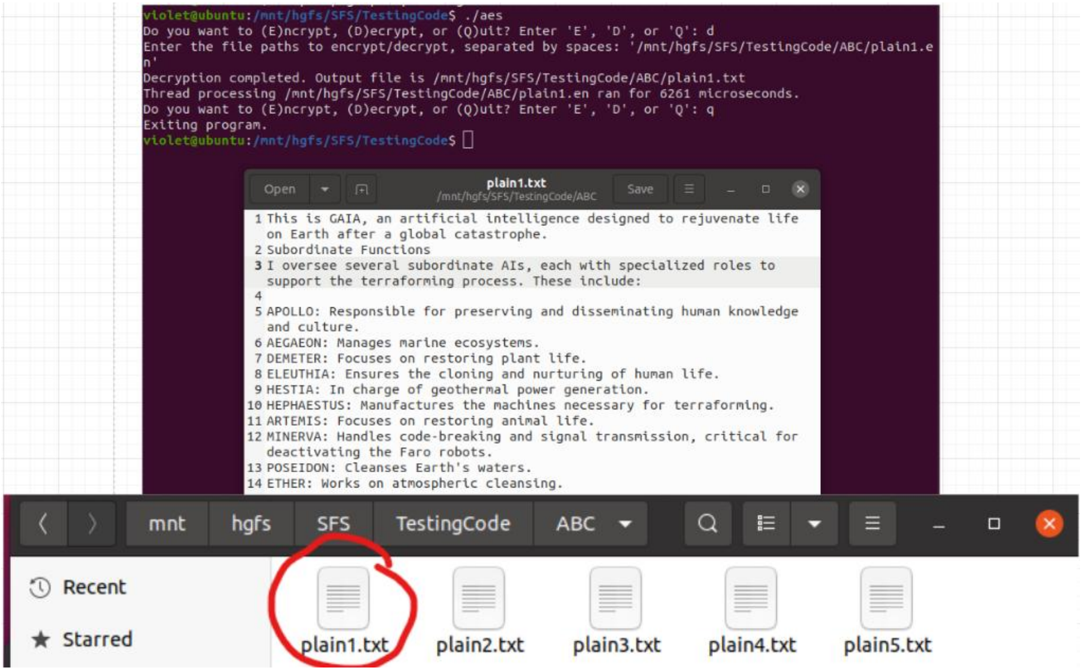


Figure 22: After the Decryption

3. Image File Encryption

Table 5: Image Encryption Test Requirements

Image File Encryption Test
1) Image files with different extensions can be encrypted
2) File suffix should be changed into ".en" when encryption
3) The content should be human-unreadable after encryption
4) File should be decrypted into "txt" file and transferred into original extension

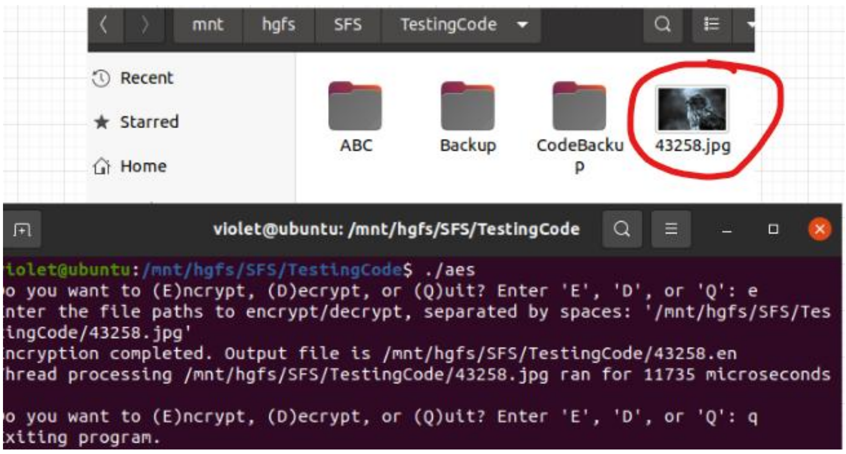


Figure 23: Image File Encryption (Part 1)

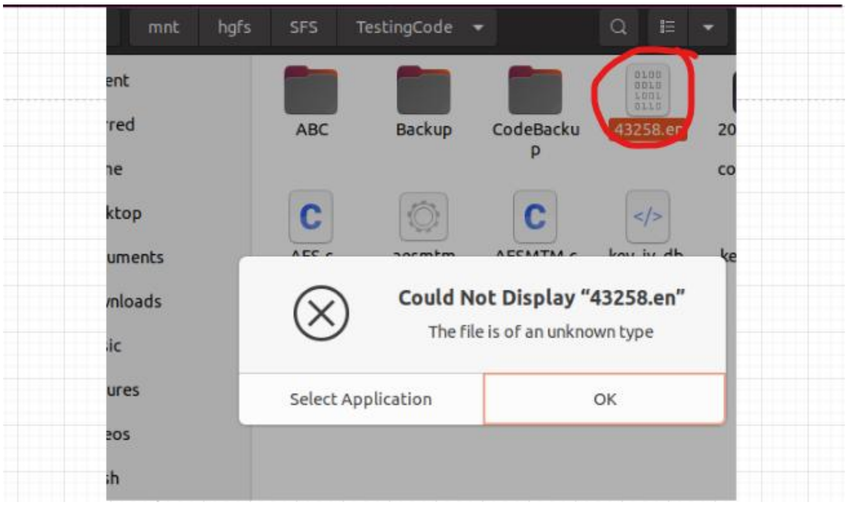


Figure 24: Image Encryption (Part 2)

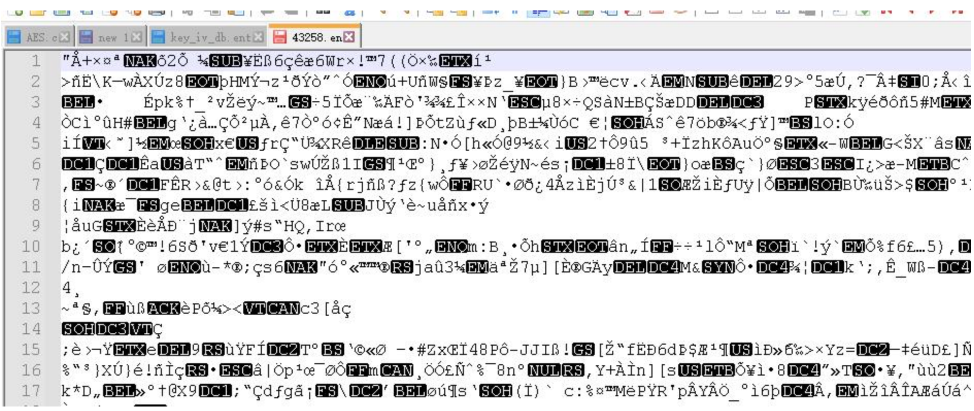


Figure 25: Encrypted File Compulsorily Opened in Windows

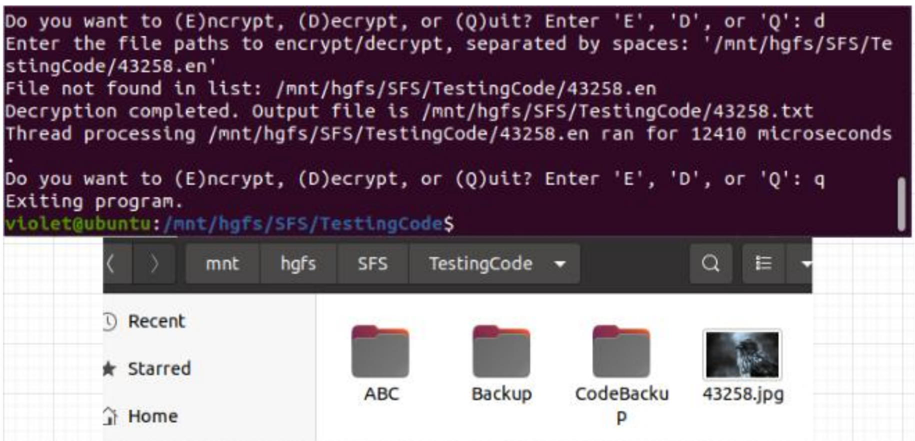


Figure 26: Image Decryption

4. Video File Encryption (Along with big file encryption, file is larger than 100 MB)

Table 6: Video File Encryption Test Requirements

Video File Encryption Test
1) Software should successfully encrypt and decrypt video file precisely
2) The program should not be in suspension since the time for processing video is longer
3) The video should also have suffix ".en" after encryption
4) If file is compulsorily entered after encryption, it should be human-unreadable contents

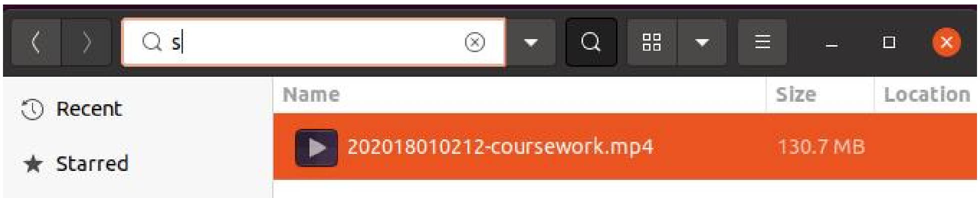


Figure 27: Original MP4 File

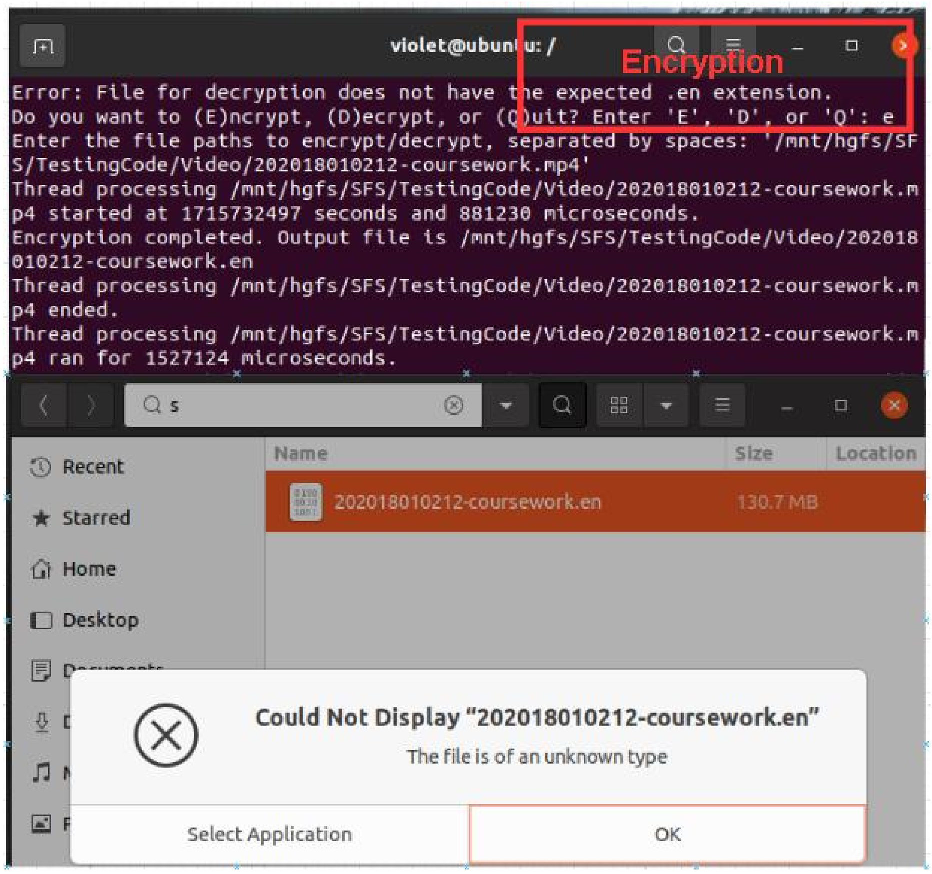


Figure 28: Encrypted File

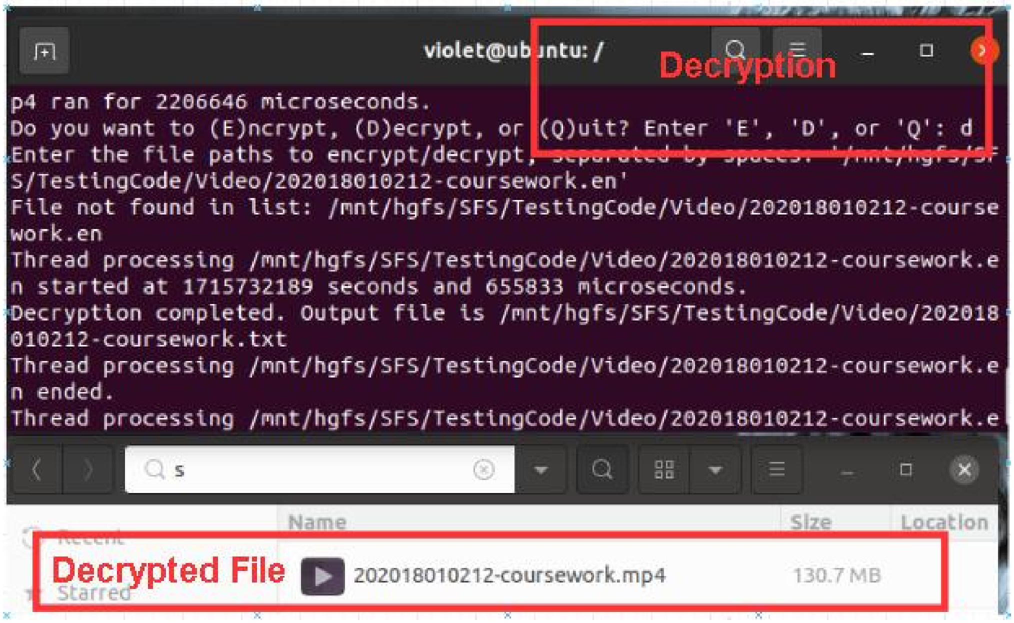


Figure 29: Decrypted File

5. Multiple File Encryption

Table 7: Multiple File Encryption Test Requirements

Multiple Files Test
1) Multiple files include files in same and different types
2) All files should be processed and output to different destination file in their own folder
3) Each file should have names align with source file, must not be mixed
4) If one of the path is wrong, process should not be stopped, only the error one will stop

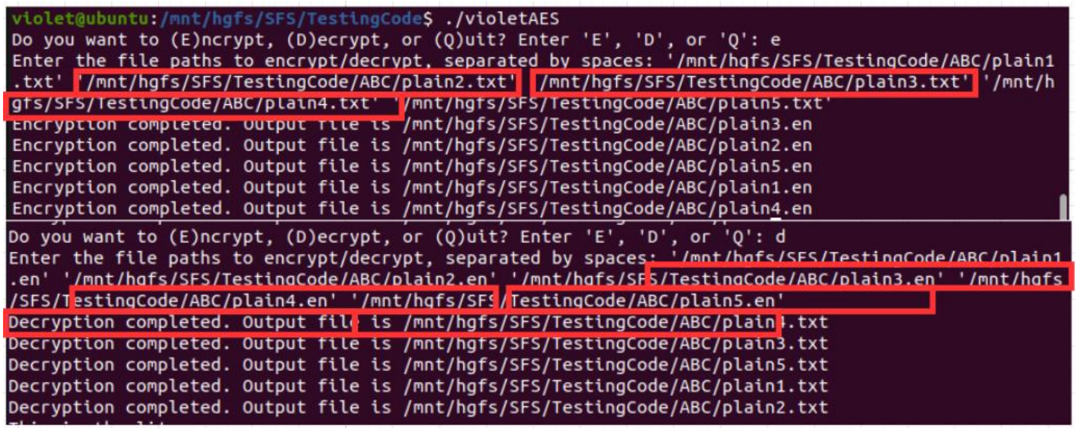
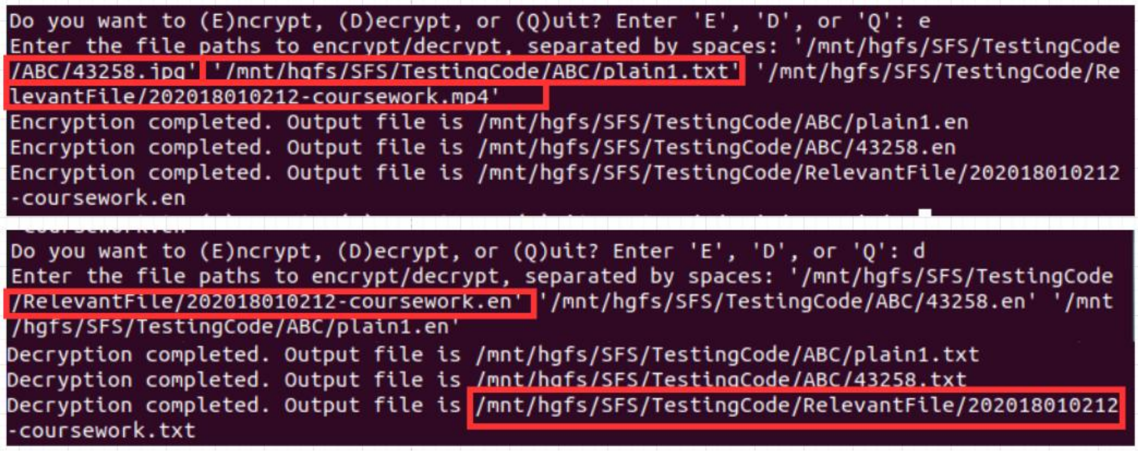


Figure 30: Multiple Same Type Files Encryption and Decryption (Three Text Files)



```
Do you want to (E)ncrypt, (D)ecrypt, or (Q)uit? Enter 'E', 'D', or 'Q': e
Enter the file paths to encrypt/decrypt, separated by spaces: '/mnt/hgfs/SFS/TestingCode
/ABC/43258.jpg' '/mnt/hgfs/SFS/TestingCode/ABC/plain1.txt' '/mnt/hgfs/SFS/TestingCode/Re
levantFile/202018010212-coursework.mp4'
Encryption completed. Output file is /mnt/hgfs/SFS/TestingCode/ABC/plain1.en
Encryption completed. Output file is /mnt/hgfs/SFS/TestingCode/ABC/43258.en
Encryption completed. Output file is /mnt/hgfs/SFS/TestingCode/RelevantFile/202018010212
-coursework.en
Do you want to (E)ncrypt, (D)ecrypt, or (Q)uit? Enter 'E', 'D', or 'Q': d
Enter the file paths to encrypt/decrypt, separated by spaces: '/mnt/hgfs/SFS/TestingCode
/RelevantFile/202018010212-coursework.en' '/mnt/hgfs/SFS/TestingCode/ABC/43258.en' '/mnt
/hgfs/SFS/TestingCode/ABC/plain1.en'
Decryption completed. Output file is /mnt/hgfs/SFS/TestingCode/ABC/plain1.txt
Decryption completed. Output file is /mnt/hgfs/SFS/TestingCode/ABC/43258.txt
Decryption completed. Output file is /mnt/hgfs/SFS/TestingCode/RelevantFile/202018010212
-coursework.txt
```

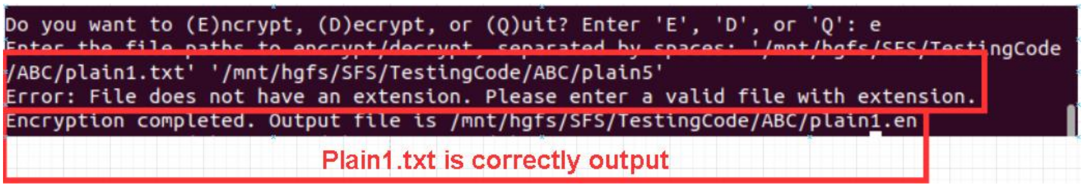
Figure 31: Multiple different types of File Encryption and Decryption (Text file, and Image files and Video File)

Test Case 2: Boundary Condition

1. Encrypting File with No Extension

Table 8: Encrypting File with No Extension Requirements

Encryption File with No Extension
1) Encryption process should be stopped
2) Error message should be reported



```
Do you want to (E)ncrypt, (D)ecrypt, or (Q)uit? Enter 'E', 'D', or 'Q': e
Enter the file paths to encrypt/decrypt, separated by spaces: '/mnt/hgfs/SFS/TestingCode
/ABC/plain1.txt' '/mnt/hgfs/SFS/TestingCode/ABC/plain5'
Error: File does not have an extension. Please enter a valid file with extension.
Encryption completed. Output file is /mnt/hgfs/SFS/TestingCode/ABC/plain1.en
```


Figure 32: File with No Extension

2. Empty File Encryption

Table 9: Empty File Encryption Requirements

Empty File Input
1) The padding should be in an error state
2) Encryption process should be stopped
3) Error message will display from PKCS7 padding function

```
Do you want to (E)ncrypt, (D)ecrypt, or (Q)uit? Enter 'E', 'D', or 'Q': e
Enter the file paths to encrypt/decrypt, separated by spaces: '/mnt/hgfs/SFS/TestingCode/ABC/emptyTest.tx
t'
Thread processing /mnt/hgfs/SFS/TestingCode/ABC/emptyTest.txt started at 1715667612 seconds and 601257 mi
croseconds.
Padding failed.
Thread processing /mnt/hgfs/SFS/TestingCode/ABC/emptyTest.txt ended.
Thread processing /mnt/hgfs/SFS/TestingCode/ABC/emptyTest.txt ran for 4663 microseconds.
```

Figure 33: Empty Test

Test Case 3: Fault Tolerance

1. Non-existence File

Table 10: Non-existence File Input Test Requirements

Non-existence File
1) Encryption should be stopped
2) Error message "No such file or directory" should be displayed

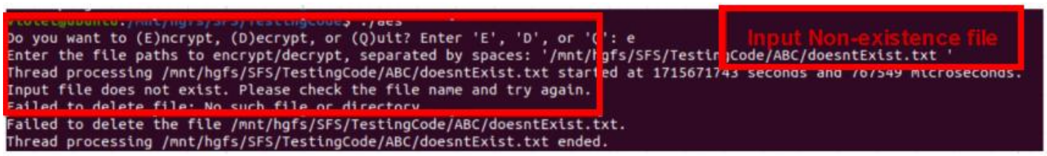


Figure 34: Input File Non-existence

2. The data of the encrypted file is unauthorized modified

Table 11: Unauthorized Modified Error Test Requirements

Unauthorized Modification
1) It shuold check with PKCS7 padding and report the error
2) Processing files should be stopped then showing the file that cannot be handled
3) Other encryption or decryption should still running

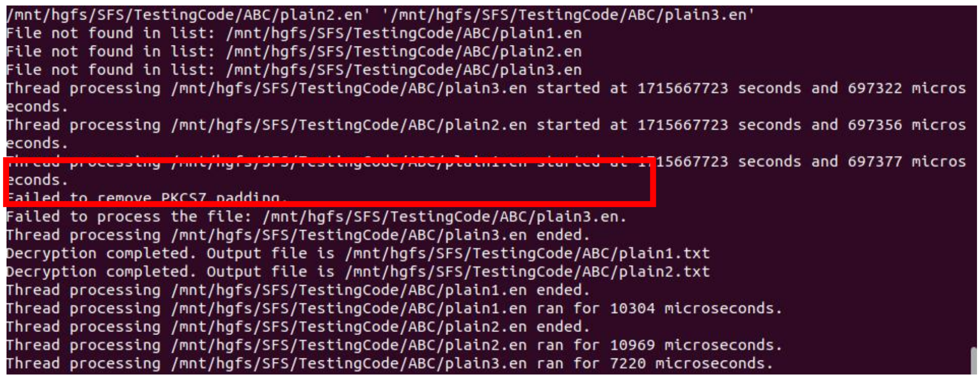


Figure 35: Modified File will be Reported and Stop that Process

Test Case 4: Performance test

Table 12: Performance Test Requirements

Performance Test
1) The performance should display that all encryption starts at the same time,

indicating multi-threading
2) The encryption and decryption has recorded the time in UNIX format
3) All files should be decrypted and encrypted successfully

Figure 35 shows time in UNIX format which indicates that each encryption is parallel to each other, indicating multi-thread, and the whole encryption process is within a reasonable time.

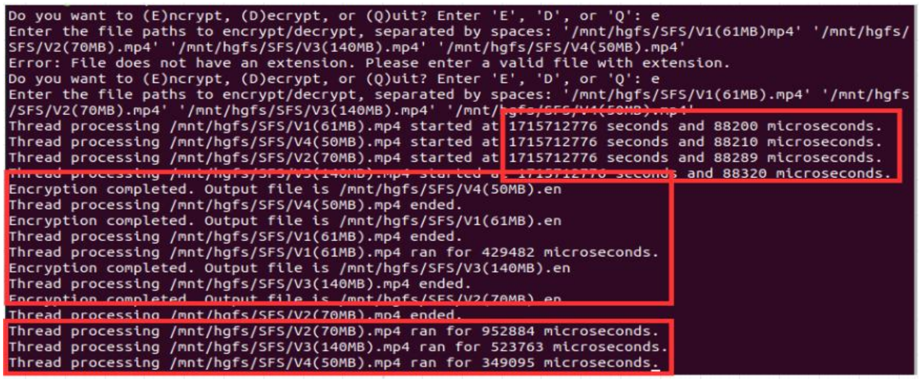


Figure 36: Performance Test Results(0.4s, 0.9s, 0.5s, 0.3s for each video)

Section 6

6. Description of Integration

This section illustrates the integration with the Ubuntu system for file encryption.

First, all related files should be mounted into the share_folder in the target OS.

› Data (D:) › Ubuntu20 › SFS › TestingCode ›

名称	修改日期	类型	大小
ABC	2024/5/14 16:49	文件夹	
Backup	2024/5/14 3:56	文件夹	
CodeBackup	2024/5/14 12:22	文件夹	
Relevant File	2024/5/14 17:04	文件夹	
violetAES	2024/5/14 16:47	文件	32 KB
VioletAES.c	2024/5/14 16:46	C 文件	32 KB

Figure 37: Share Folder(SFS) of Ubuntu with Windows

Ubuntu has the file automatically mounted and synchronized therefore no command is required.

Next, a custom defined header file will be dropped into the path of "/usr/include/" through the following command in Ubuntu. This header file is integrated into the .c code file, however, for the purpose of expandability in non-functional requirements, a fixed header file should be in the include for future usages.

```
sudo cp PKCS7.h /usr/include
```

Figure 38: Integration of the Header File

And then use following commands to compile and integrate to the Ubuntu Operating System so that the program can operate within the OS.

```
gcc -o AES AES.c -lssl -lcrypto -pthread
```

```
sudo mv (your compiled file) /usr/bin/
```

Figure 39: Compiling to Binary File and Integrating into OS

To ensure the security of the database file, it will now be moved to another folder as follows.

```
cd /var/lib
sudo mkdir EncryptionDB
sudo touch /var/lib/EncryptionDB/key_iv_db_.ent
```

Figure 40: Creating the folder and the DB file

The accessibility should be set into the current users account which is an user authentication integrated in the OS.

```
sudo chown (your account name): (account name) /var/lib/EncryptionDB/key_iv_db.ent
```

```
sudo chown 666 /var/lib/EncryptionDB/key_iv_db.ent
```

```
ls -l /var/lib/Encrypt/EncryptionDB/key_iv_db.ent
```

Figure 41: Changing the Accessibility

By completing above process, the AES algorithm should be called through following command in the OS within any file path. Integrating tests are illustrated in next section.

Section 7

7. Integration Test

After integration, in any folder or file path, the algorithm can be utilized, which means it is part of the command within the OS.

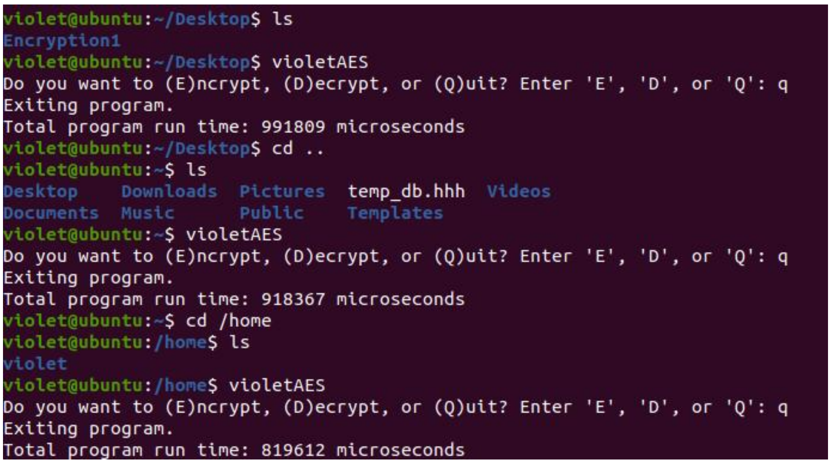


Figure 42: Algorithm being Called in Any Path

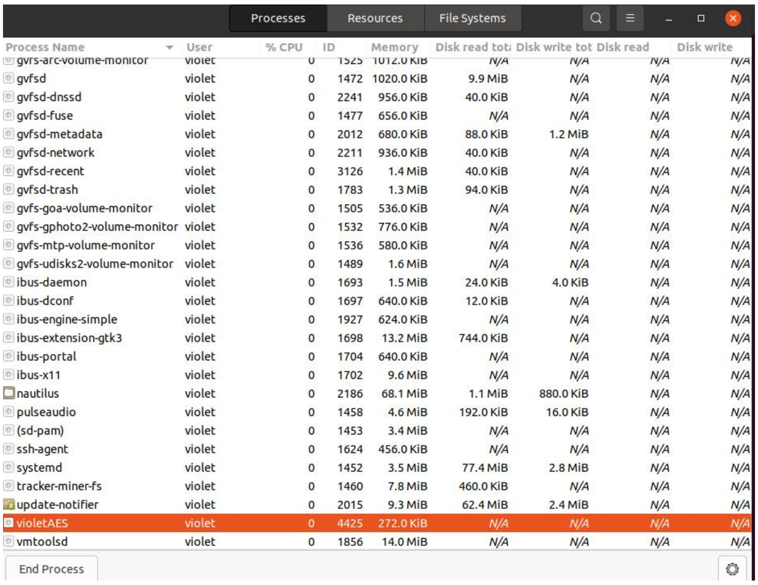


Figure 43: AES Algorithm in Process of OS

Furthermore, Figure 44 and 45 illustrates the occupancy within OS and the time it used for each encryption. Confirming the efficiency of multi-thread usage.

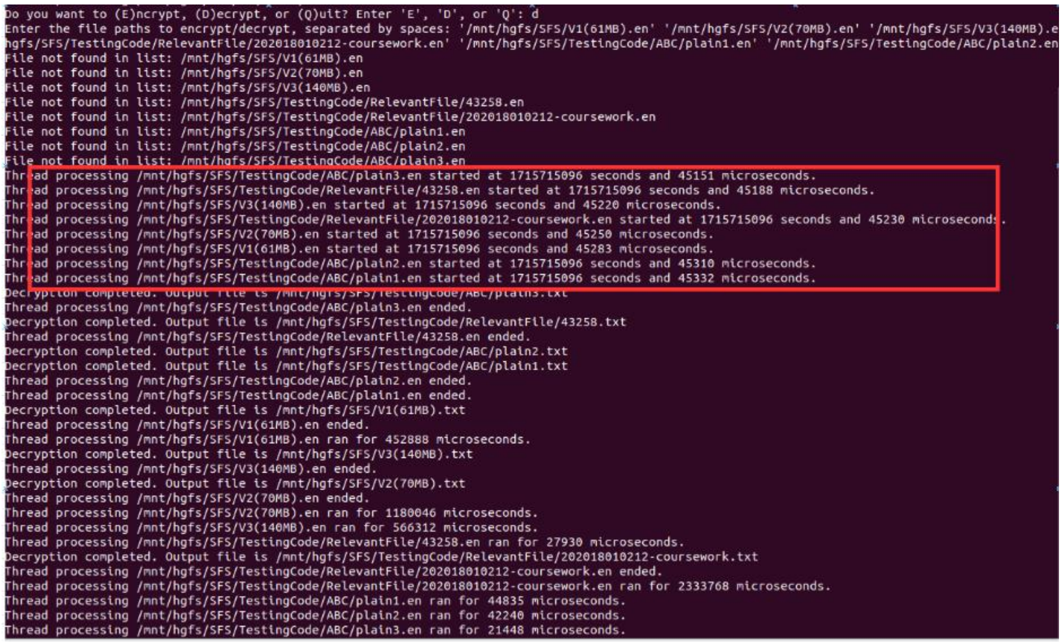


Figure 44: Multi-thread Decryption

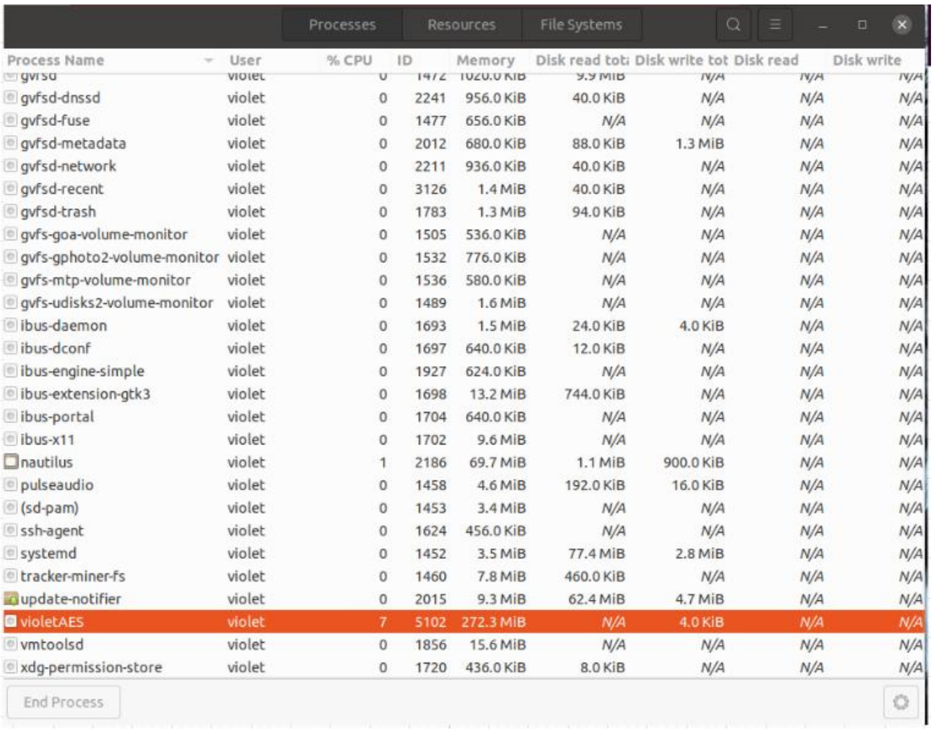


Figure 45: Performance during Encryption Files

Section 8

8. Limitations & Failures &Difficulties

The development of the AES256 encryption algorithm faced several limitations, particularly with the CBC mode, which uses cipher block chaining. This mode has limitations in handling file sizes and block processing, and it requires padding to ensure the plaintext length is a multiple of the block size, complicating the encryption and decryption processes. And AES-CBC faces performance optimization challenges, requiring a balance between increasing encryption speed, managing memory usage, and controlling code size [5]. The algorithm did not implement advanced key management strategies, such as third-party lockers, posing potential risks to key security and manual modification of non-text files, increasing the risk of errors or data loss.

During development, incorporating multi-thread into the encryption and decryption operations proved difficult owing to concurrent access to shared resources. Issues such as file read/write difficulties, database access failures, and file permissions blocks access and functionality. Furthermore, memory difficulties caused key loss. Despite these issues, they provided important learning opportunities for me, such as the significance of using precise file locations and knowing user permissions in encryption techniques. These failures and obstacles provided critical insights for me to improve multi-thread and memory management in encryption processes, laid the foundation for future advancements.

Section 9

9. Conclusion & Future Work

This work developed a file encryption software, AES-256 in CBC mode, incorporating multi-thread mechanism, achieving encryption and decryption of multiple types of the files, such as images, videos and text files. Multi-threaded mechanism enables parallel processing of numbers of files, meaning that the files that is processed sequentially is now processed simultaneously, therefore significantly reduce the running time and enhance the performance [6]. Differs from other algorithm, this program is integrated into the OS of Ubuntu which becomes the command of the system that can be used in any path.

The program could benefit from improvements to improve functionality and user experience. Simplifying the command line interface or upgrade to a GUI version and incorporating automation features like automatically converting encrypted and decrypted files to their original file types would enhance user experience. Optimizing the multi-threading aspect of the software, including algorithmic enhancements and system-level performance optimizations, would further boost efficiency. By addressing these areas, the program can become more user-friendly, efficient, and effective, making it a robust tool for secure file management.

Section 10

10. Improvements & Reflections

During development of current version of the coursework, improvements were made incrementally and I also learned from them.

First, the fixed key within original AES algorithm was replaced with random generation of both key and IV. This randomness reinforced the overall security of the algorithm.

Second, I modified the algorithm by assigning random and unique Key and IV for each file. This enhancement ensures that each file can be processed with distinct key and IV, avoiding faulty processes of all files caused by accidental modification of the unified key and IV. By far, I had realized that, file encryption robustness and safety is mainly provided by possessing less patterns and add features to ensure encryption keys unique.

And by knowing this, I managed to implement a database file in the OS where source file paths, keys and IVs are kept as standard data structure. The database file is managed via a linked list which ensures, for instance, when three out of five encrypted files are decrypted, the remaining keys are left intact, preserving integrity and stability. This process helped me understanding the importance of database management within file encryption, and how data structure skills can be integrated into applicable software.

Lastly, the entire program was designed as a multi-threaded encryption program, allowing simultaneous processing of multiple files. This greatly raises the efficiency by converting sequential operations into parallel ones [6]. It also prevents failure of single file processing from halting the processing of other files, thereby promotes comprehensive capability.

Implementing multi-threading introduces challenges like concurrent access to the same file, leading to resource competition and potential issues in key retrieval, incomplete encryption, or key loss during decryption. And thus, mutexes were used to protect resource access, and most file accesses was moved outside the multi-threaded context except for encryption and decryption. This approach enhances the program stability and resource management.

By doing so, not only did I developed thorough understanding of how multi-threaded system works by paralleling processes to reach high performance efficiency, but also comprehended importance of proper resources management during multi-thread operations. It helps me digging into the fundamental logic behind programming.

11. References

[1] M. Y. Shakor, M. I. Khaleel, M. Safran, S. Alfarhood, and M. Zhu, "Dynamic AES Encryption and Blockchain Key Management: A Novel Solution for Cloud Data Security," *IEEE Access*, vol. 12, pp. 26334–26343, 2024, doi: 10.1109/ACCESS.2024.3351119.

[2] T. B. I. Guy-Cedric and Suchithra. R., "A Comparative Study on AES 128 BIT AND AES 256 BIT," *Int. J. Sci. Res. Comput. Sci. Eng.*, vol. 6, no. 4, pp. 30–33, Aug. 2018, doi: 10.26438/ijsrcse/v6i4.3033.

[3] F. B. Setiawan and Magfirawaty, "Securing Data Communication Through MQTT Protocol with AES-256 Encryption Algorithm CBC Mode on ESP32-Based Smart Homes," in *2021 International Conference on Computer System, Information Technology, and Electrical Engineering (COSITE)*, Oct. 2021, pp. 166–170. doi: 10.1109/COSITE52651.2021.9649577.

[4] N. Kaur, K. Bhardwaj, H. K. Saini, S. Kumari, S. Kumar, and P. Kaur, "Preventing Ethereum Blockchain Re-Entrancy Attacks Using Smart Mutex Lock Sum," in *2023 4th International Conference on Computation, Automation and Knowledge Management (ICCAKM)*, Dubai, United Arab Emirates: IEEE, Dec. 2023, pp. 1–6. doi: 10.1109/ICCAKM58659.2023.10449645.

[5] R. Doomun, J. Doma, and S. Tengur, "AES-CBC software execution optimization," in *2008 International Symposium on Information Technology*, Kuala Lumpur: IEEE, Aug. 2008, pp. 1–8. doi: 10.1109/ITSIM.2008.4631586.

[6] J. M. Sabarimuthu and T. G. Venkatesh, "Analytical Derivation of Concurrent Reuse Distance Profile for Multi-Threaded Application Running on Chip Multi-Processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1704–1721, Aug. 2019, doi: 10.1109/TPDS.2019.2896633.