**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**

[Table of Contents 3](#_Toc20295)

[List of Figures 4](#_Toc163)

[List of Tables 6](#_Toc23347)

[1. Encryption of Encryption Algorithm 8](#_Toc17354)

[2. Functional & Non-Functional & Security Features 10](#_Toc5283)

[3. Design of Software 13](#_Toc26606)

[4. Implementation of the Software 17](#_Toc11127)

[5. Testing Plan for Validating Your Software 31](#_Toc15308)

[6. Description of Integration 42](#_Toc114)

[7. Integration Test 45](#_Toc32325)

[8. Limitations & Failures &Difficulties 48](#_Toc23403)

[9. Conclusion & Future Work 50](#_Toc21066)

[10. Improvements &Reflections 52](#_Toc19491)

[11. References 54](#_Toc26346)

**List of Figures**

[Figure 1 : AES Encryption Details 8](#_Toc16769)

[Figure 2 : Inner Design of Process File module 13](#_Toc27452)

[Figure 3 : Main module 14](#_Toc21854)

[Figure 4 : Headers, Variables, and Struct Definition 17](#_Toc3075)

[Figure 5 : Methods implicitly defined 18](#_Toc26459)

[Figure 6 : Linked List Basic Methods Definition 19](#_Toc26102)

[Figure 7 : Key Generation and Binary-Hexadecimal Transfer 20](#_Toc9800)

[Figure 8 : Key Storage and Loading 21](#_Toc24958)

[Figure 9 : Source file and Database File Reading, Writing, and Updating 22](#_Toc6623)

[Figure 10 : Duplicate File Verification, File Deletion, PKCS7 Padding and Cutting. 23](#_Toc333)

[Figure 11 : Encryption Module 24](#_Toc157)

[Figure 12 : Decryption Module 25](#_Toc16550)

[Figure 13 : File Process Module 26](#_Toc17715)

[Figure 14 : Main Module Part 1 27](#_Toc23334)

[Figure 15 : Main Module Part 2 28](#_Toc21549)

[Figure 16 : Main Module Part 3 29](#_Toc14107)

[Figure 17 :Random Key and IV Generation Test Results 31](#_Toc18250)

[Figure 18 : Locked Database, Forbidden from Modification with in OS 32](#_Toc14321)

[Figure 19 : The text file mounted onto Ubuntu Share-Folder (SFS in my case) 32](#_Toc29454)

[Figure 20 : After Encryption 33](#_Toc22753)

[Figure 21 : If compulsorily opened in Windows Share-folder with notepad 33](#_Toc17422)

[Figure 22 : After the Decryption 34](#_Toc16337)

[Figure 23 : Image File Encryption (Part 1) 34](#_Toc2295)

[Figure 24 : Image Encryption (Part 2) 35](#_Toc30581)

[Figure 25 : Encrypted File Compulsorily Opened in Windows 35](#_Toc7633)

[Figure 26 : Image Decryption 35](#_Toc5307)

[Figure 27 : Original MP4 File 36](#_Toc23685)

[Figure 28 : Encrypted File 36](#_Toc19302)

[Figure 29 : Decrypted File 37](#_Toc30365)

[Figure 30 : Multiple Same Type Files Encryption and Decryption (Three Text Files) 37](#_Toc13816)

[Figure 31 : Text file, and Image files and Video File Decryption and Encryption 38](#_Toc7244)

[Figure 32 : File with No Extension 38](#_Toc21718)

[Figure 33 : Empty Test 39](#_Toc5294)

[Figure 34 : Input File Non-existence 39](#_Toc2215)

[Figure 35 : Modified File will be Reported and Stop that Process 39](#_Toc11088)

[Figure 36 : Performance Test Results(0.4s, 0.9s, 0.5s, 0.3s for each video) 40](#_Toc3069)

[Figure 37 : Share Folder(SFS) of Ubuntu with Windows 42](#_Toc5875)

[Figure 38 : Integration of the Header File 42](#_Toc12343)

[Figure 39 : Compiling to Binary File and Integrating into OS 42](#_Toc15757)

[Figure 40 : Creating the folder and the DB file 43](#_Toc24998)

[Figure 41 : Changing the Accessibility 43](#_Toc31121)

[Figure 42 : Algorithm being Called in Any Path 45](#_Toc14062)

[Figure 43 : AES Algorithm in Process of OS 45](#_Toc17520)

[Figure 44 : Multi-thread Decryption 46](#_Toc4015)

[Figure 45 : Performance during Encryption Files 46](#_Toc27232)

**List of Tables**

[Table 1 : Functional Requirements, Non-Functional Requirements and Security Features 10](#_Toc7371)

[Table 2 : Basic Technology and Environment 31](#_Toc31912)

[Table 3 : Key Storage Test Requirements 31](#_Toc11914)

[Table 4 : Text Encryption Requirements 32](#_Toc24517)

[Table 5 : Image Encryption Test Requirements 34](#_Toc13841)

[Table 6 : Video File Encryption Test Requirements 36](#_Toc8243)

[Table 7 : Multiple File Encryption Test Requirements 37](#_Toc6409)

[Table 8 : Encrypting File with No Extension Requirements 38](#_Toc22534)

[Table 9 : Empty File Encryption Requirements 38](#_Toc593)

[Table 10 : Non-existence File Input Test Requirements 39](#_Toc18987)

[Table 11 : Unauthorized Modified Error Test Requirements 39](#_Toc5709)

[Table 12 : Performance Test Requirements 39](#_Toc22870)

**Section 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.

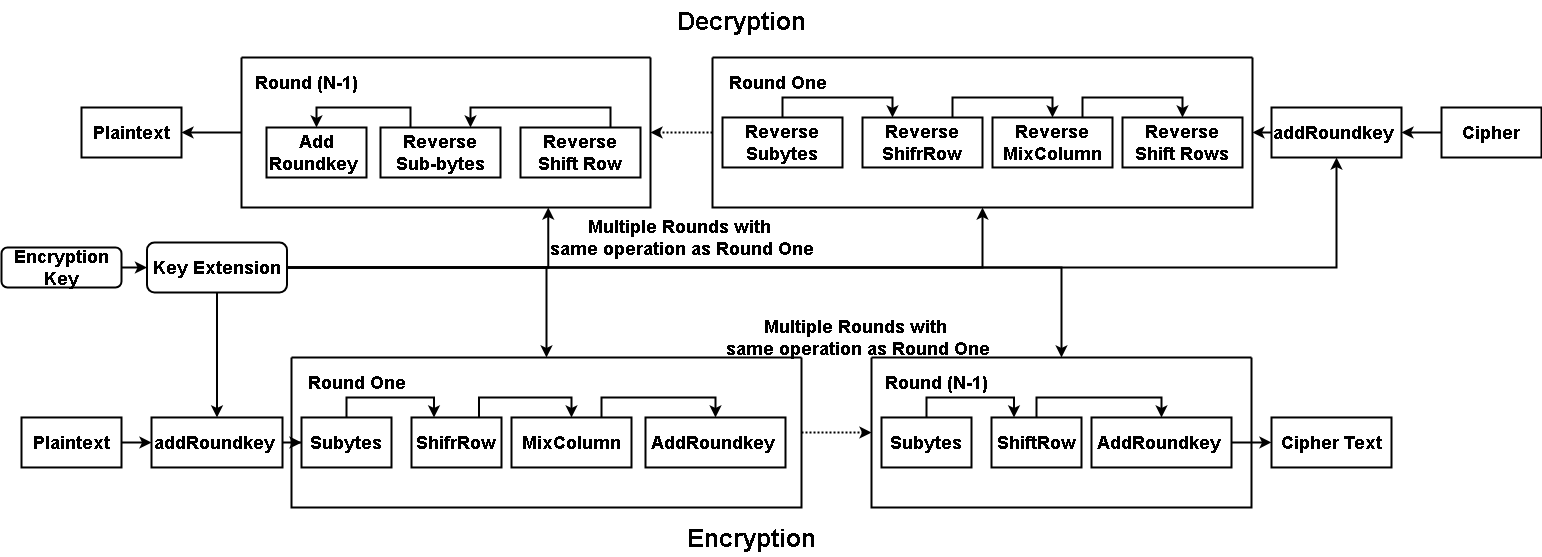


Figure 1: AES Encryption Details

**Section 2**

# **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**

# **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.

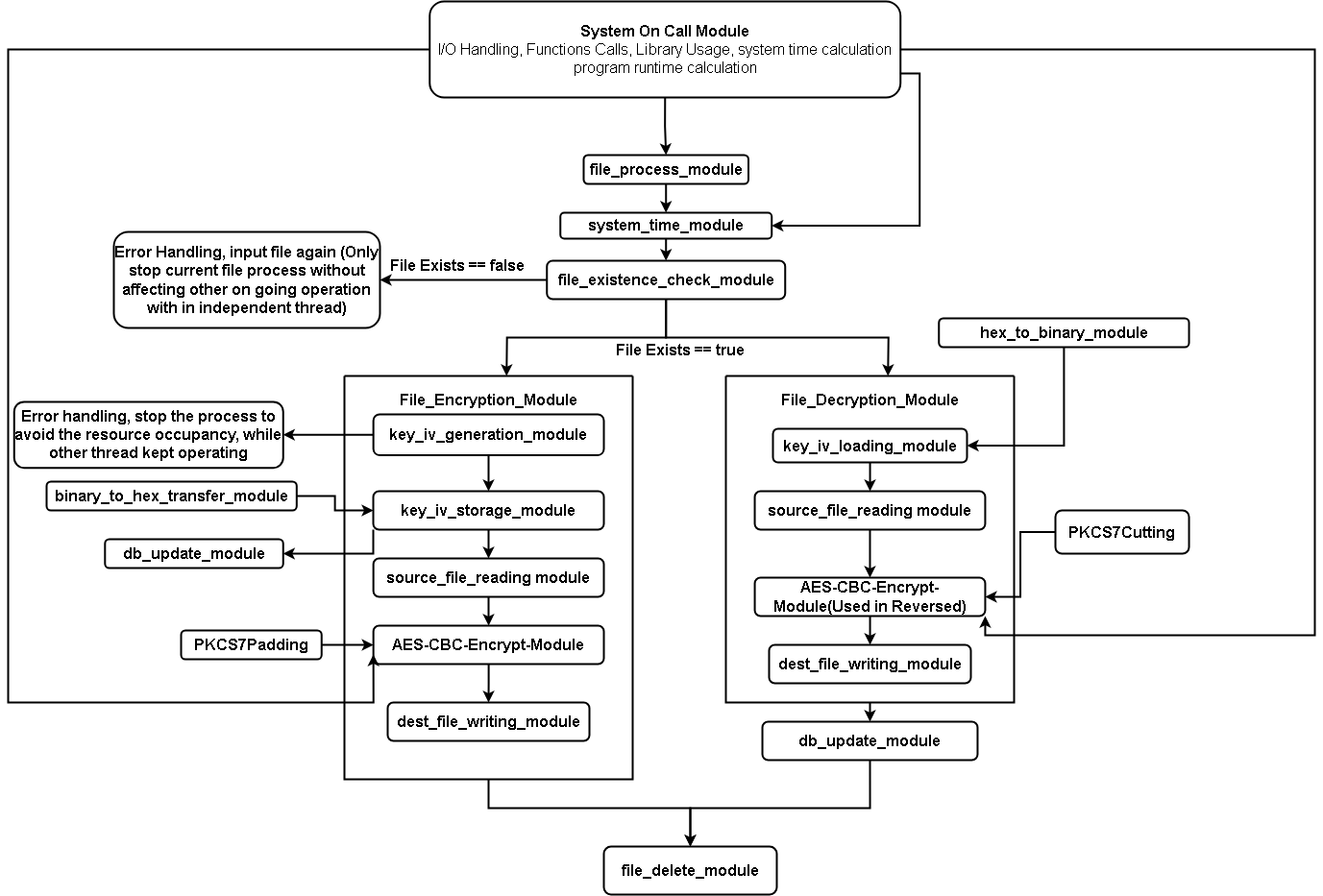


Figure 2: Inner Design of Process File module

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.

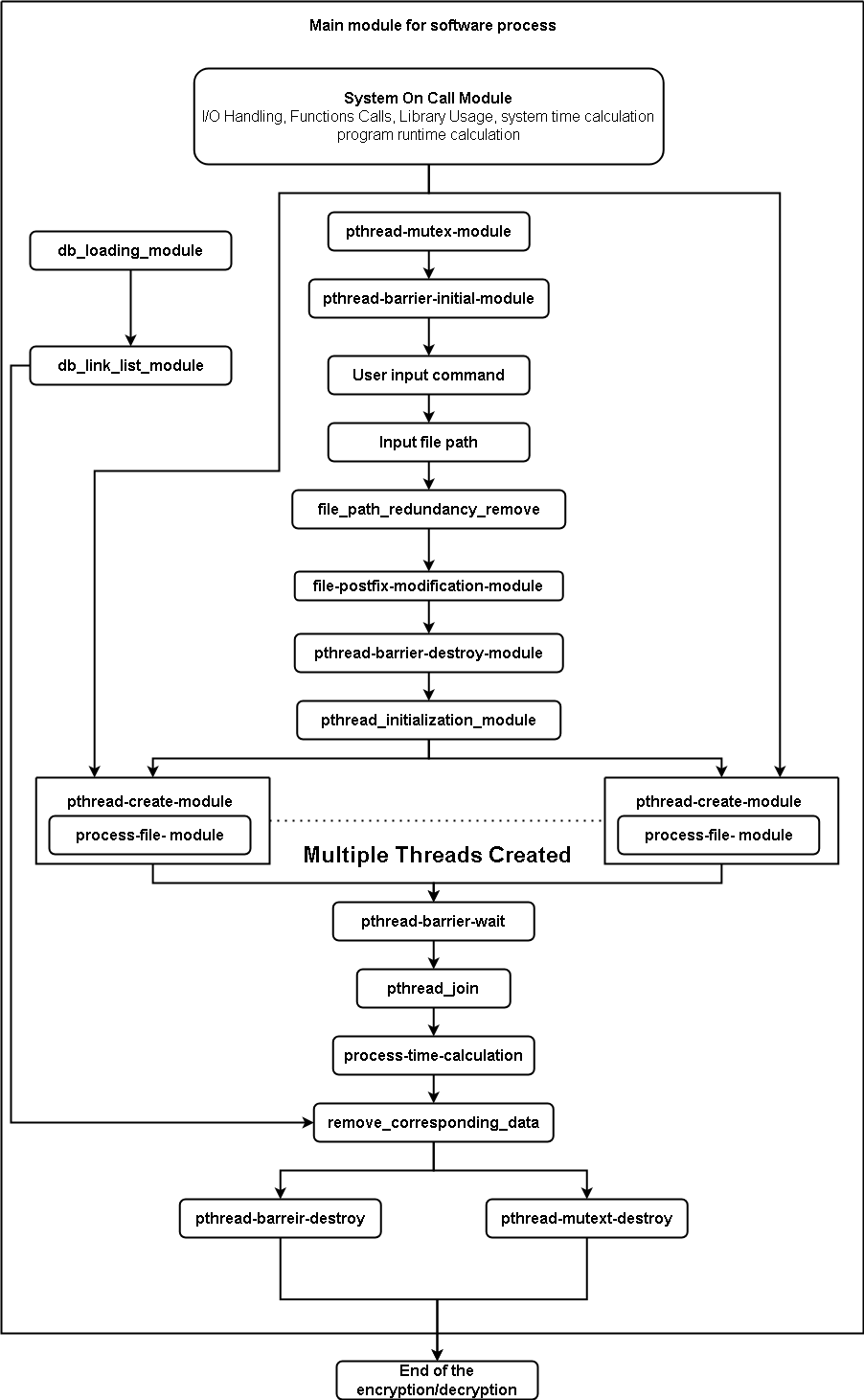


Figure 3: Main module

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**

# **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.

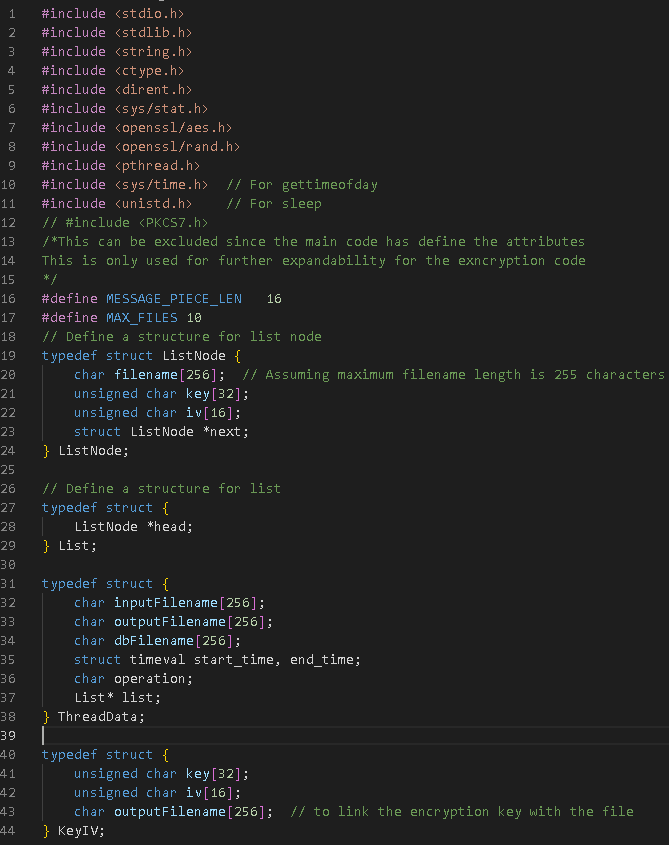


Figure 4: Headers, Variables, and Struct Definition

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.

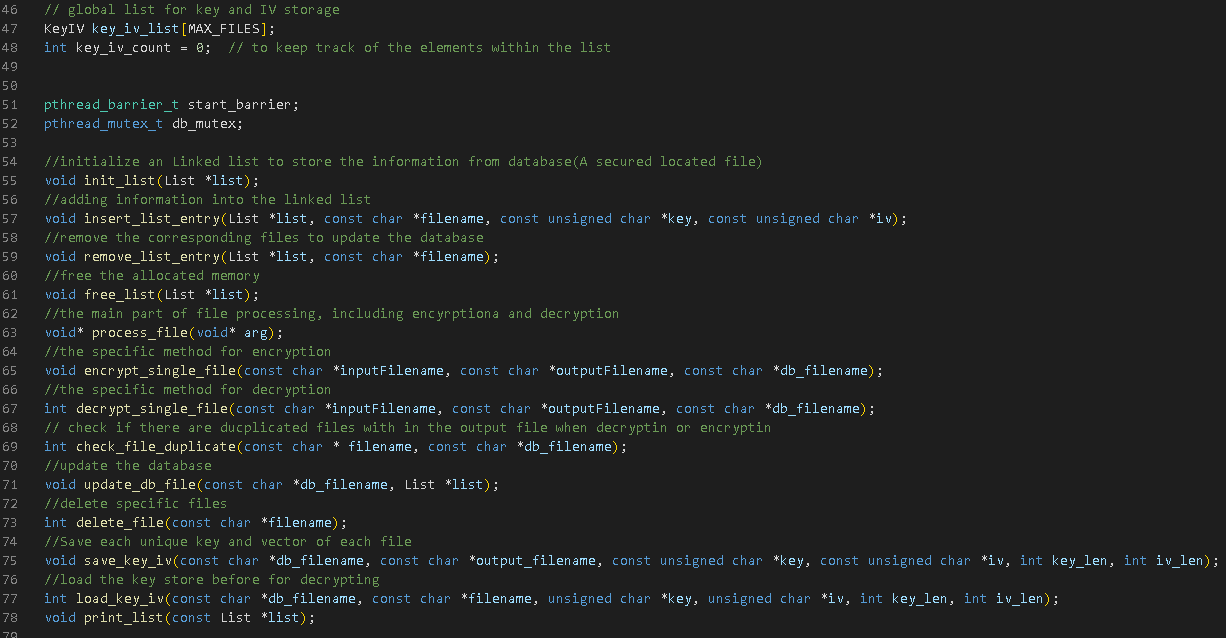


Figure 5: Methods implicitly defined

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.

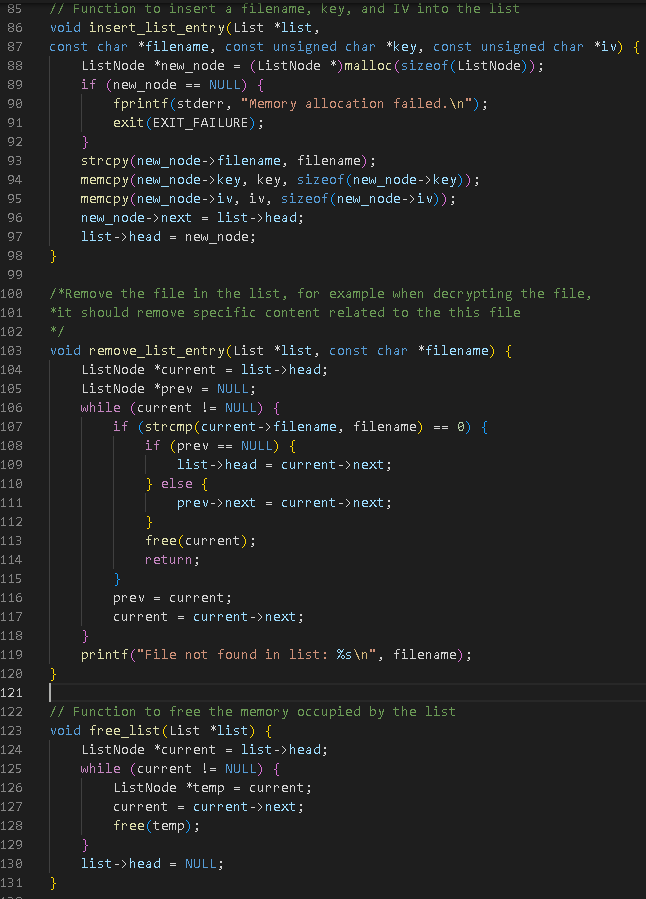


Figure 6: Linked List Basic Methods Definition

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.

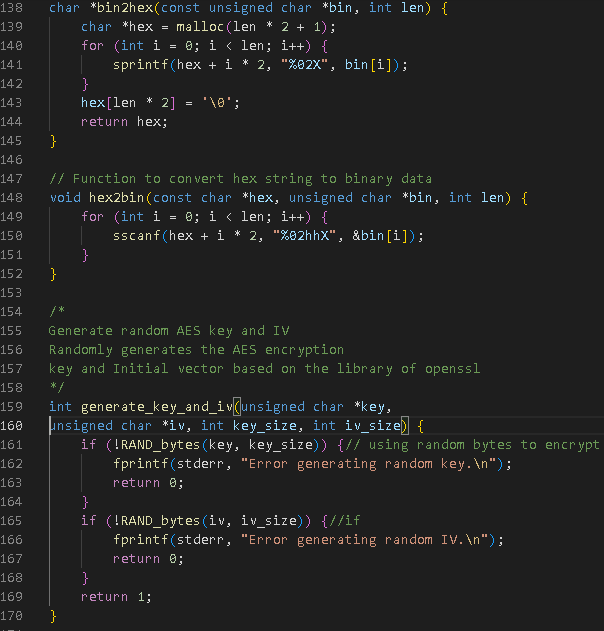


Figure 7: Key Generation and Binary-Hexadecimal Transfer

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.

:

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.

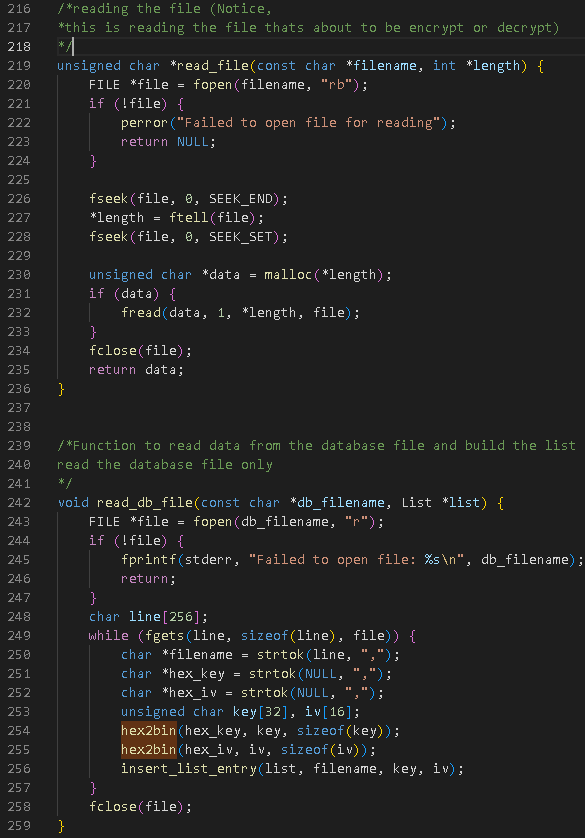


Figure 9: Source File Reading and Destination File Writing, Database Reading and Modifying, and File Existence Verification

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.

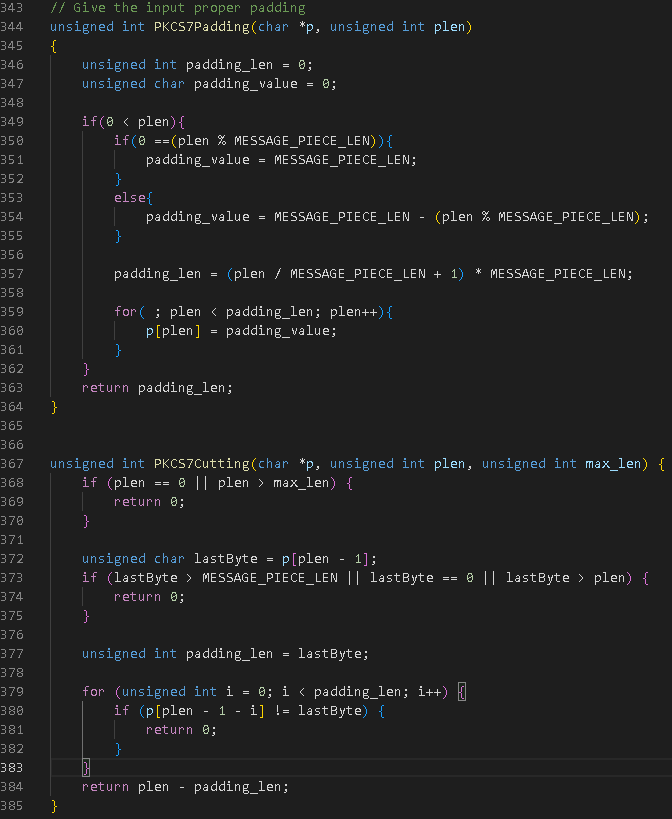
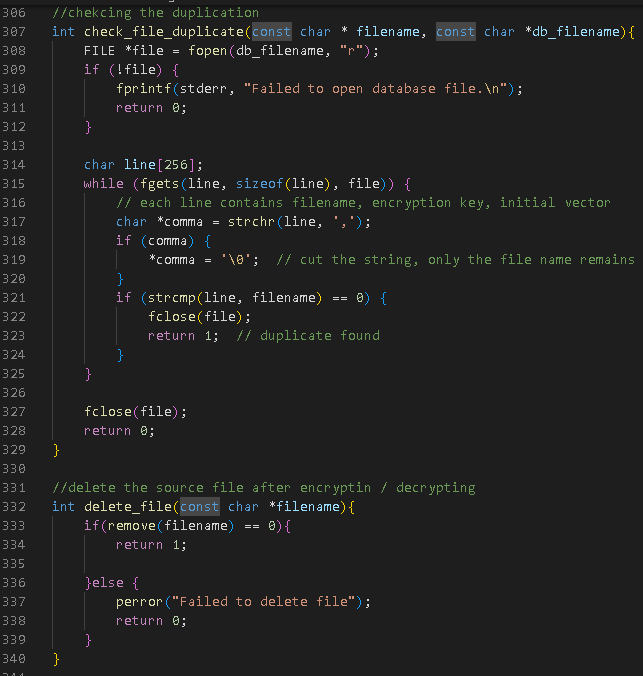


Figure 10: Duplicate File Verification, File Deletion, PKCS7 Padding and Cutting.

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.

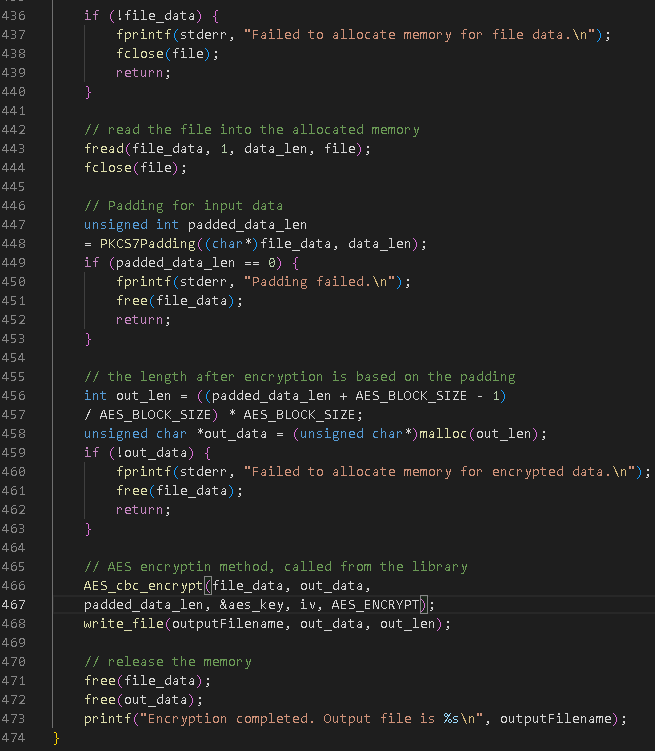
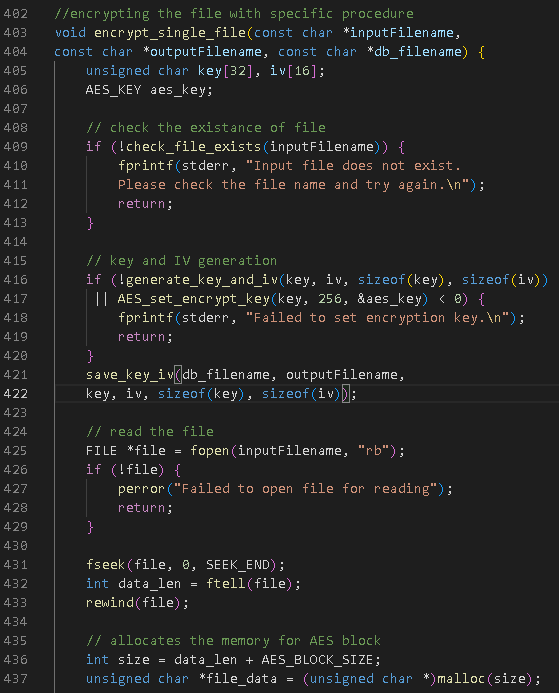


Figure 11: Encryption Module

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.

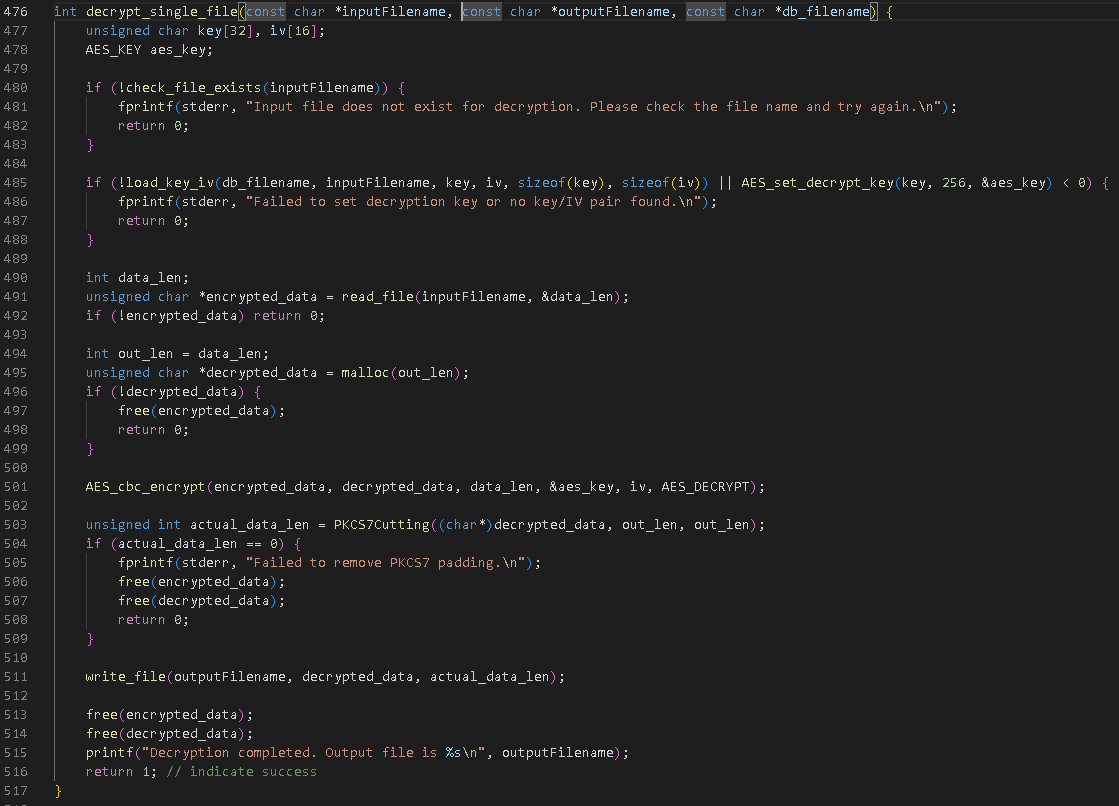


Figure 12: Decryption Module

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.

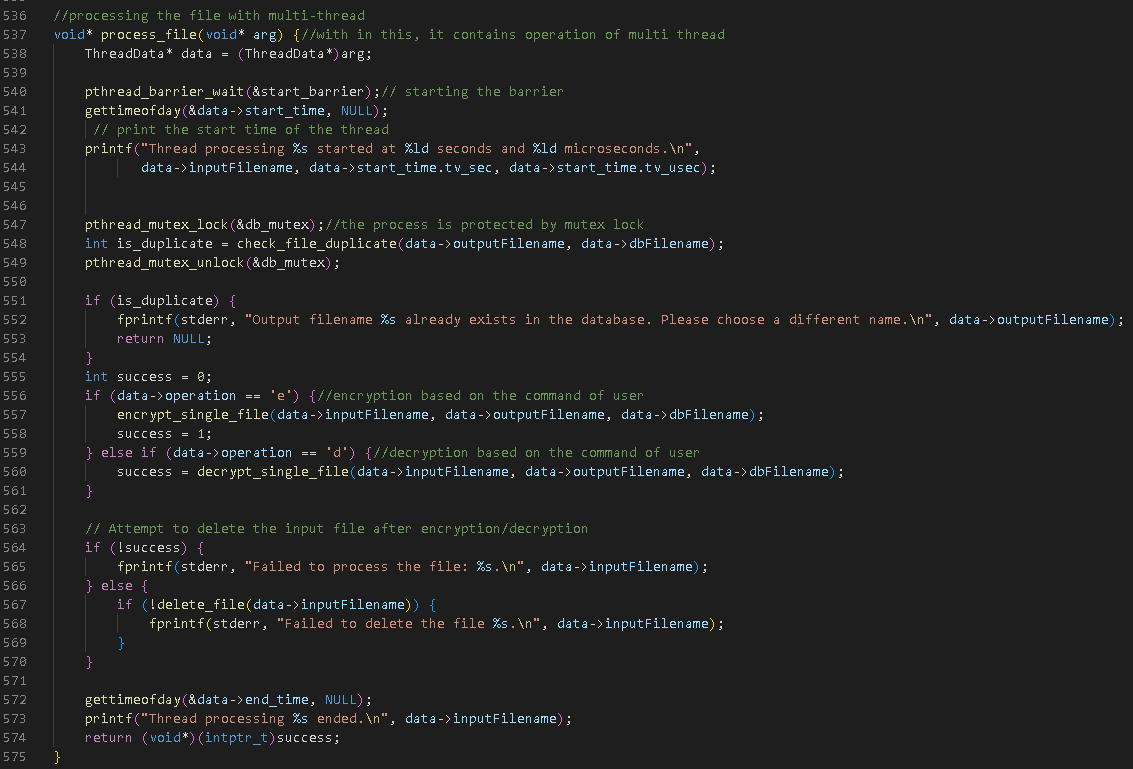


Figure 13: File Process Module

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.

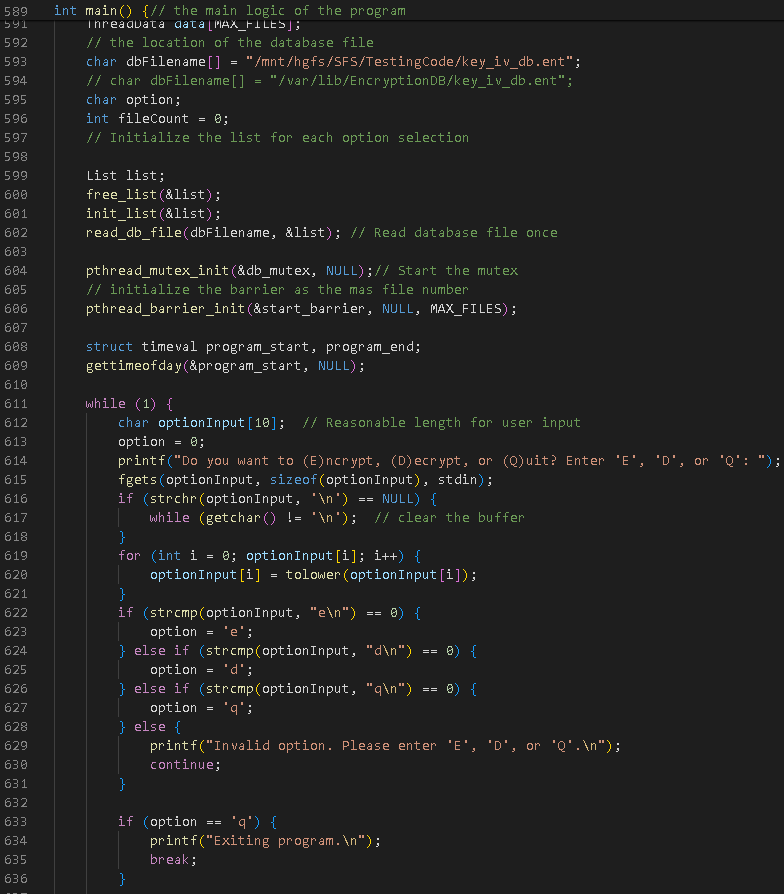


Figure 14: Main Module Part 1

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.



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.

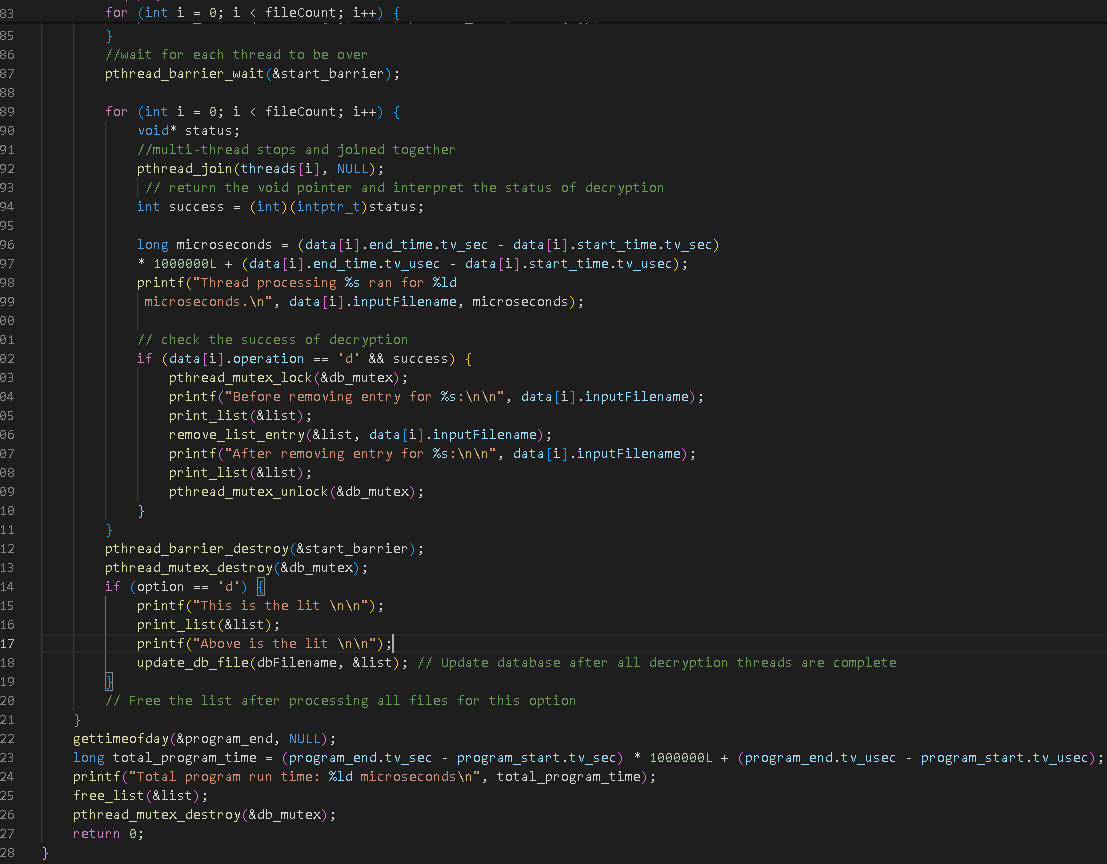


Figure 16: Main Module Part 3

**Section 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 |
| 1. The Key and IV must store within a database file |
| 1. The structure should be file path, IV Key |

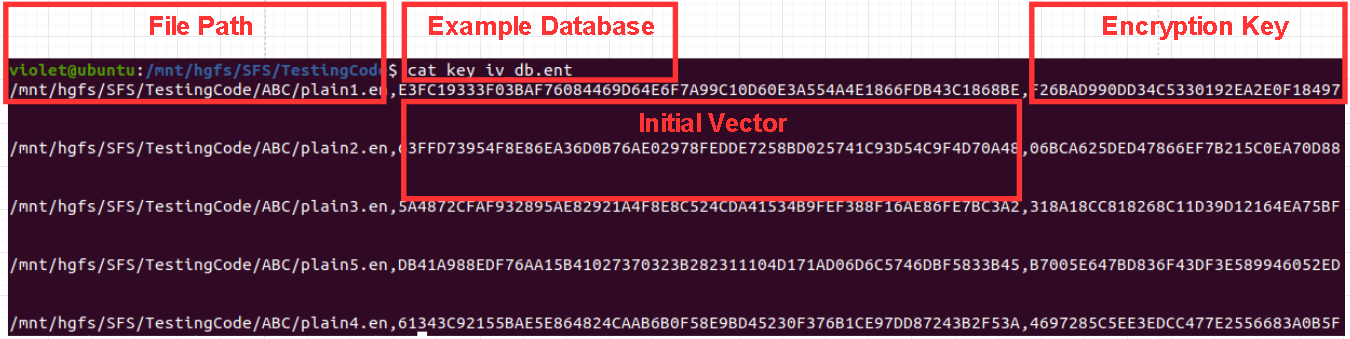


Figure 17:Random Key and IV Generation Test Results

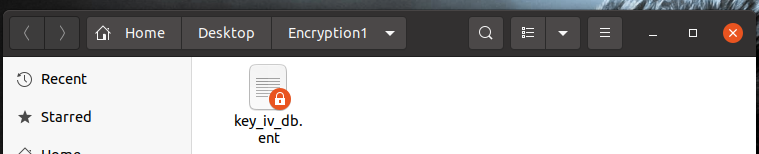


Figure 18: Locked Database, Forbidden from Modification with in OS

1. Text File Encryption

Table 4: Text Encryption Requirements

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

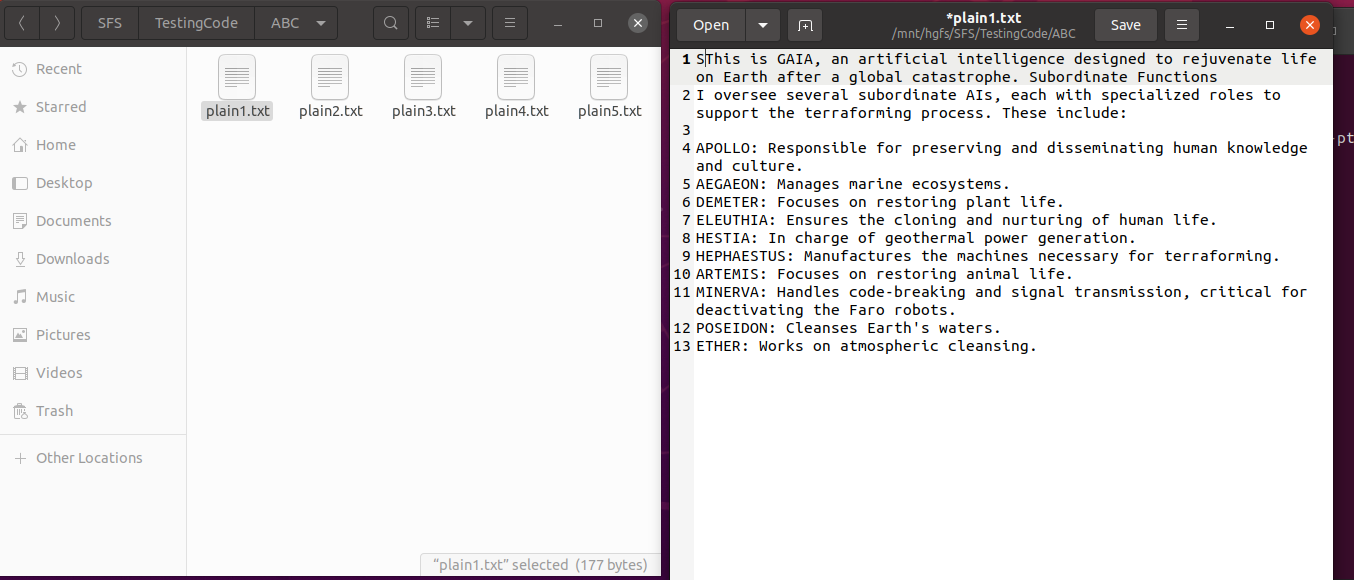


Figure 19: The text file mounted onto Ubuntu Share-Folder (SFS in my case)

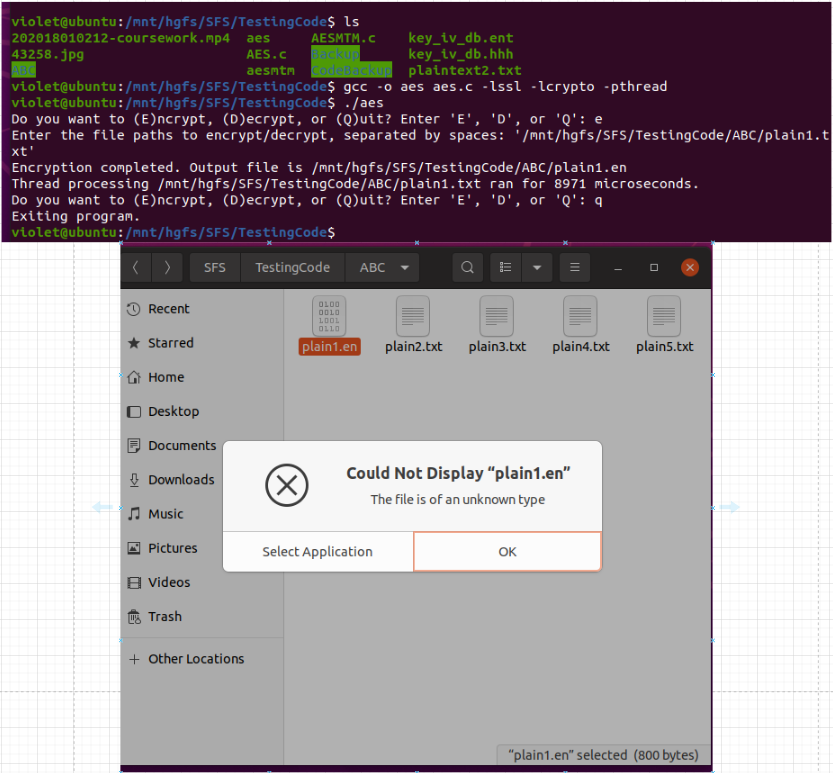


Figure 20: After Encryption

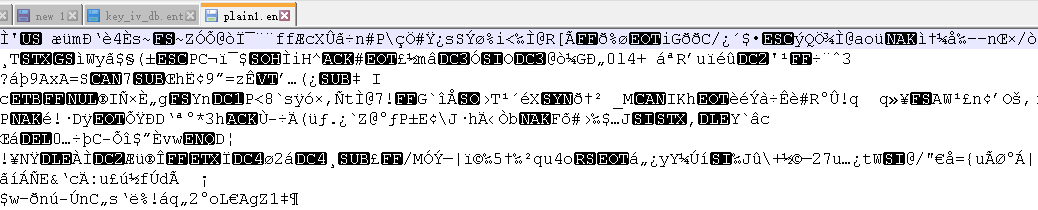


Figure 21: If compulsorily opened in Windows Share-folder with notepad

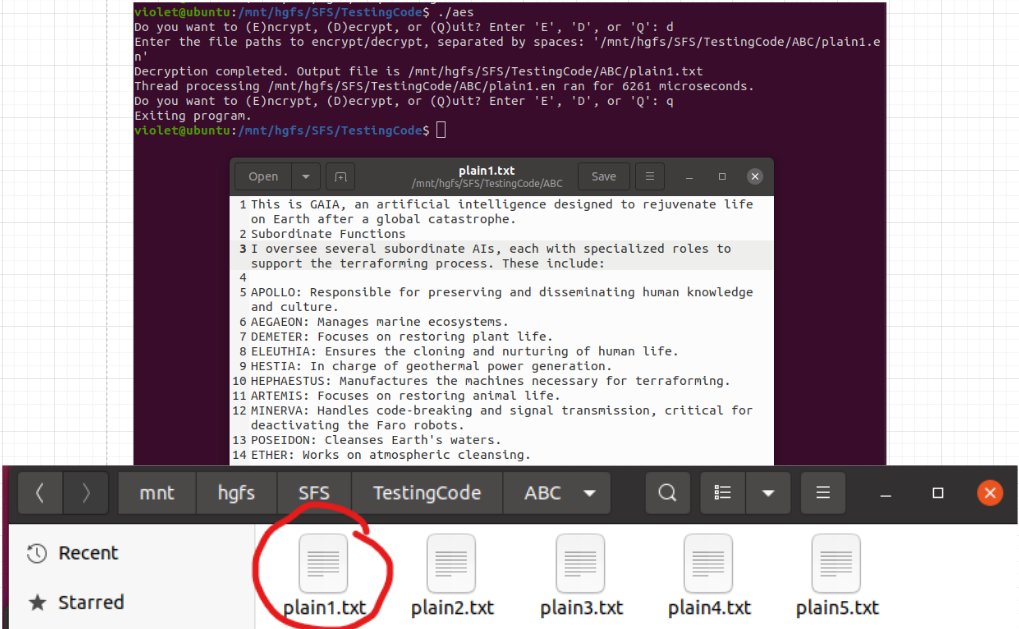


Figure 22: After the Decryption

1. Image File Encryption

Table 5: Image Encryption Test Requirements

|  |
| --- |
| **Image File Encryption Test** |
| 1. Image files with different extensions can be encrypted |
| 1. File suffix should be changed into “.en” when encryption |
| 1. The content should be human-unreadable after encryption |
| 1. File should be decrypted into “txt” file and transferred into original extension |

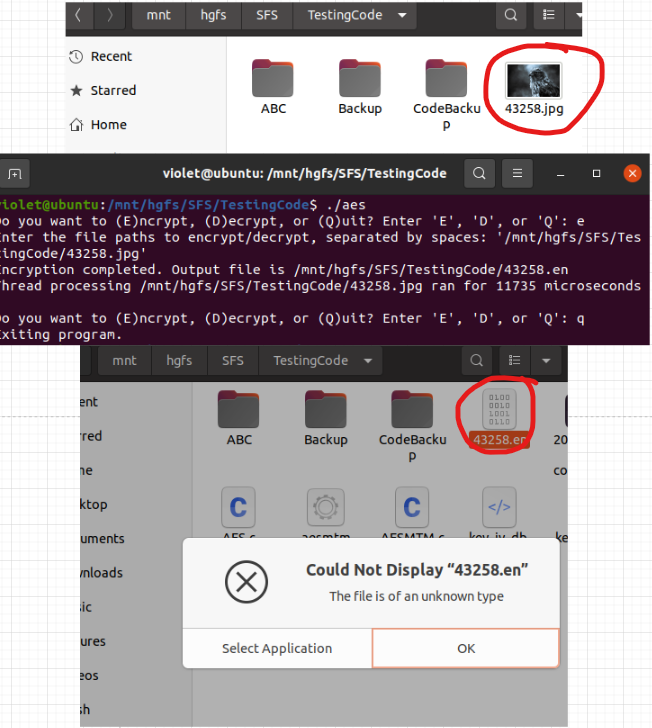


Figure 23: Image File Encryption (Part 1)

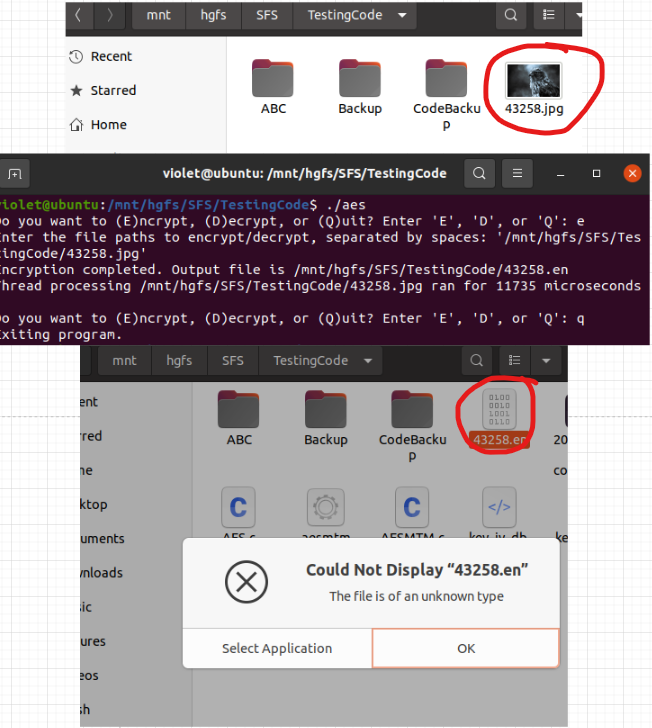


Figure 24: Image Encryption (Part 2)

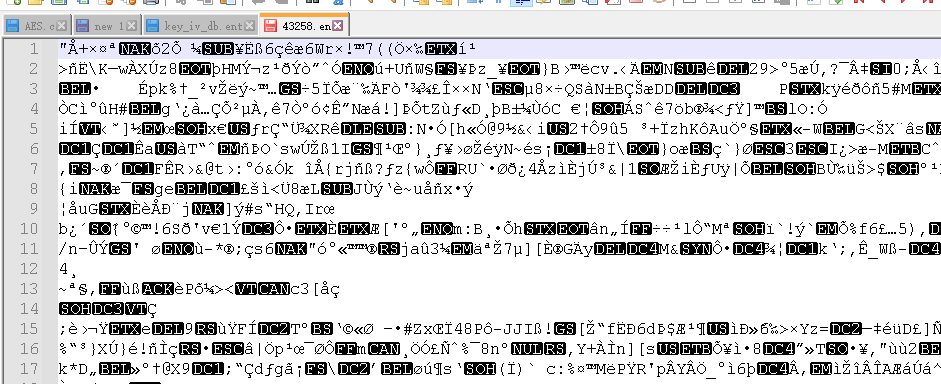


Figure 25: Encrypted File Compulsorily Opened in Windows

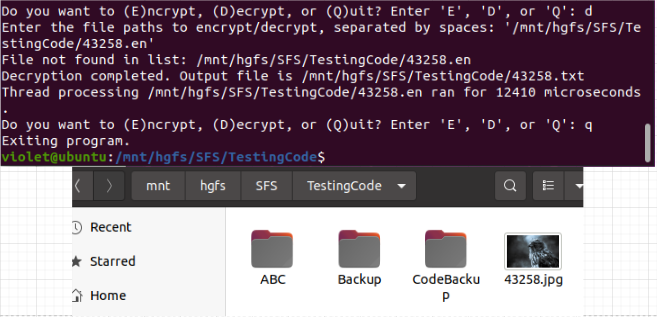


Figure 26: Image Decryption

1. 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 |
| 1. The program should not be in suspension since the time for processing video is   longer |
| 1. The video should also have suffix “.en” after encryption |
| 1. If file is compulsorily entered after encryption, it should be human-unreadable   contents |

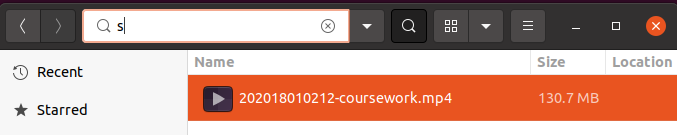


Figure 27: Original MP4 File

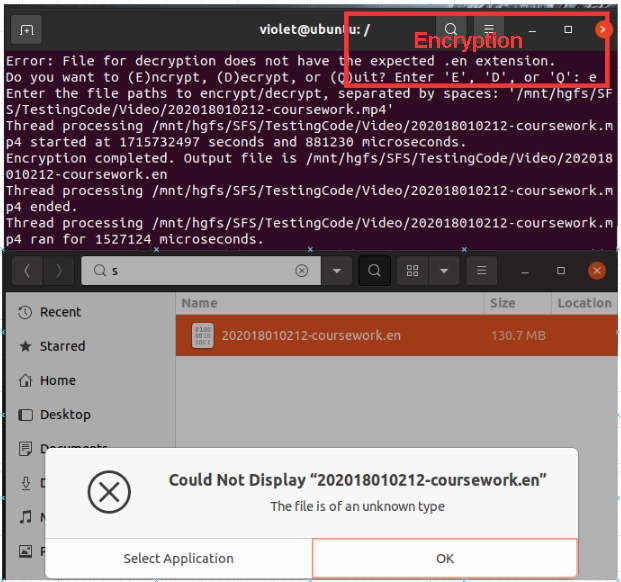


Figure 28: Encrypted File

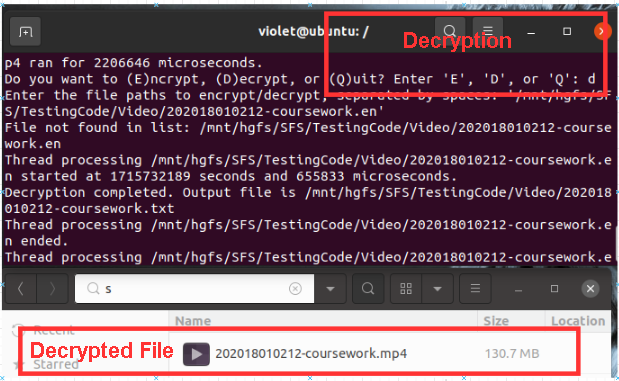


Figure 29: Decrypted File

1. Multiple File Encryption

Table 7: Multiple File Encryption Test Requirements

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

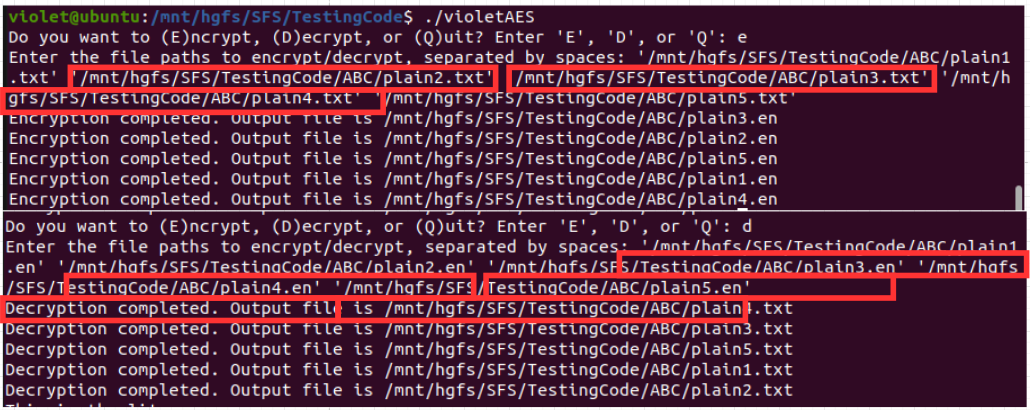


Figure 30: Multiple Same Type Files Encryption and Decryption (Three Text Files)

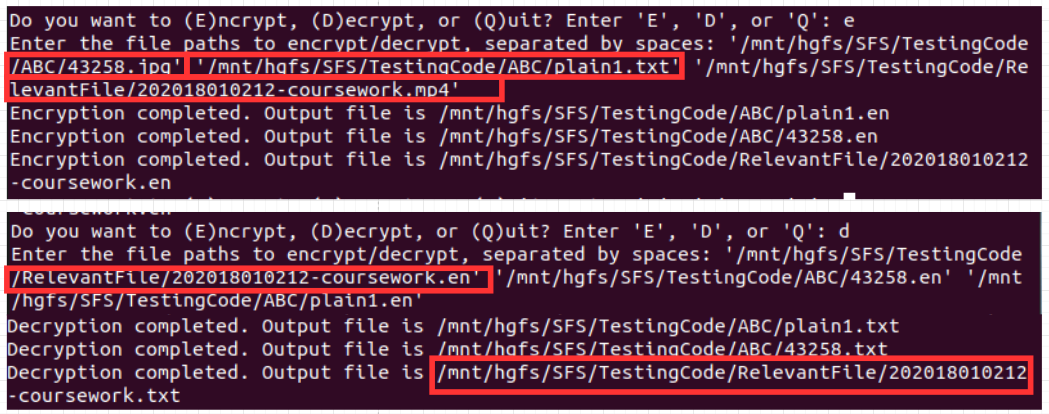


Figure 31: Multiple different types of File Encryption and Decryption (Text file, and Image files and Video File)

**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 |
| 1. Error message should be reported |

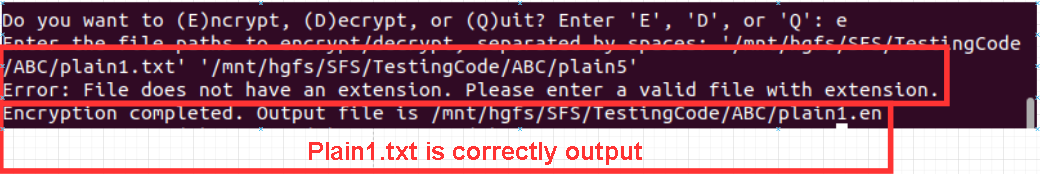


Figure 32: File with No Extension

1. Empty File Encryption

Table 9: Empty File Encryption Requirements

|  |
| --- |
| **Empty File Input** |
| 1. The padding should be in an error state |
| 1. Encryption process should be stopped |
| 1. Error message will display from PKCS7 padding function |

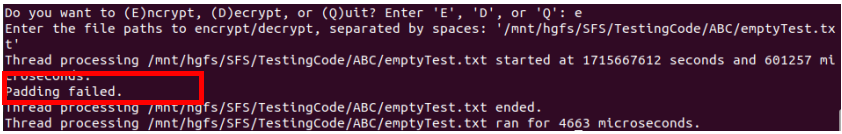


Figure 33: Empty Test

**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 |
| 1. Error message “No such file or directory” should be displayed |

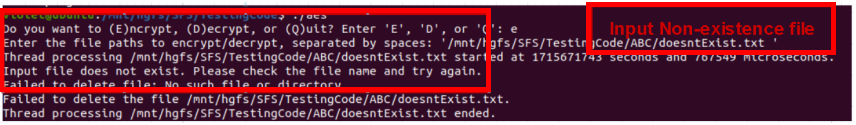


Figure 34: Input File Non-existence

1. 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 |
| 1. Processing files should be stopped then showing the file that cannot be handled |
| 1. 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 |
| 1. The encryption and decryption has recorded the time in UNIX format |
| 1. 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.

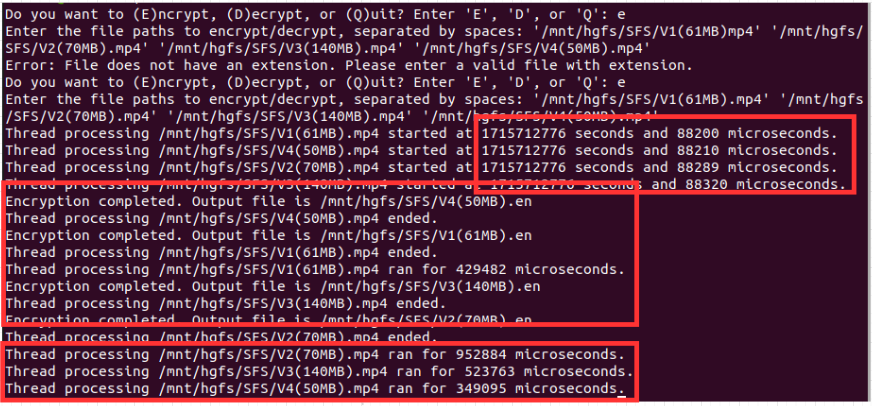


Figure 36: Performance Test Results(0.4s, 0.9s, 0.5s, 0.3s for each video)

**Section 6**

# **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.

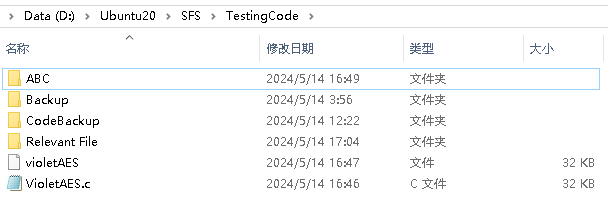


Figure 37: Share Folder(SFS) of Ubuntu with Windows

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.



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.

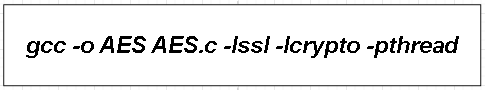


Figure 39: Compiling to Binary File and Integrating into OS

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



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.

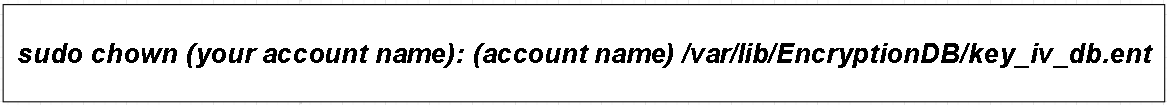




Figure 41: Changing the Accessibility

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**

# **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.

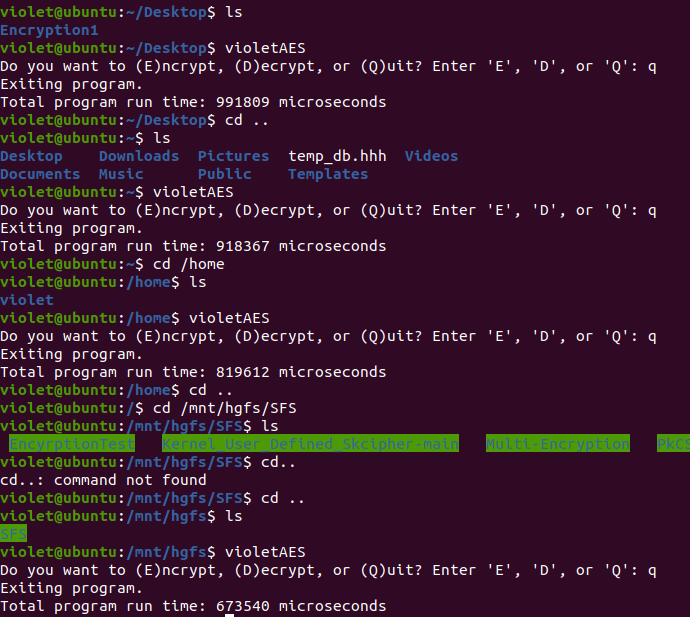


Figure 42: Algorithm being Called in Any Path

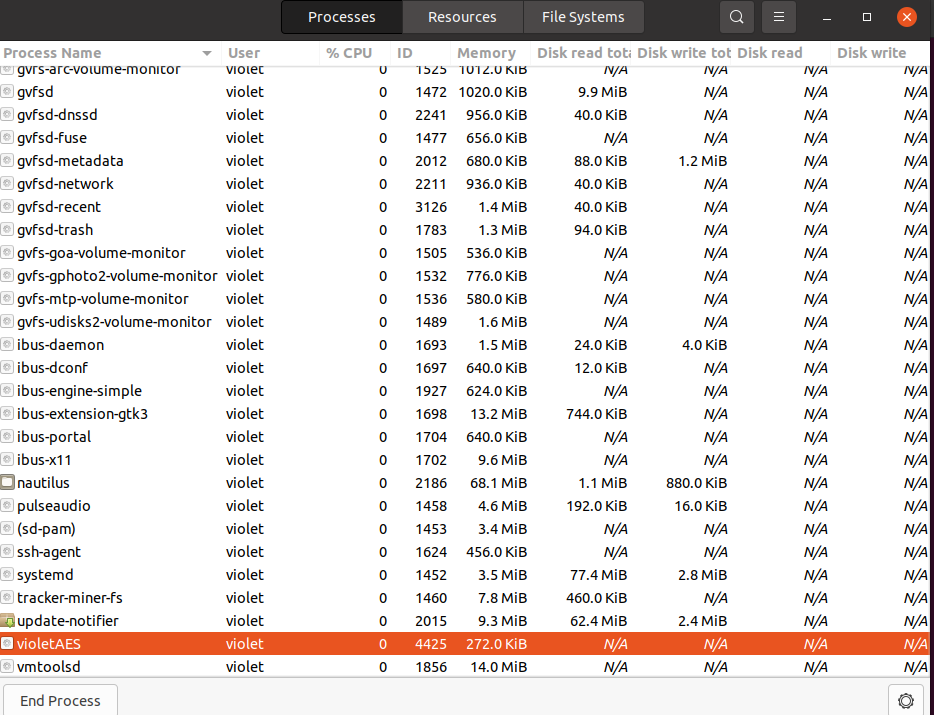


Figure 43: AES Algorithm in Process of OS

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.

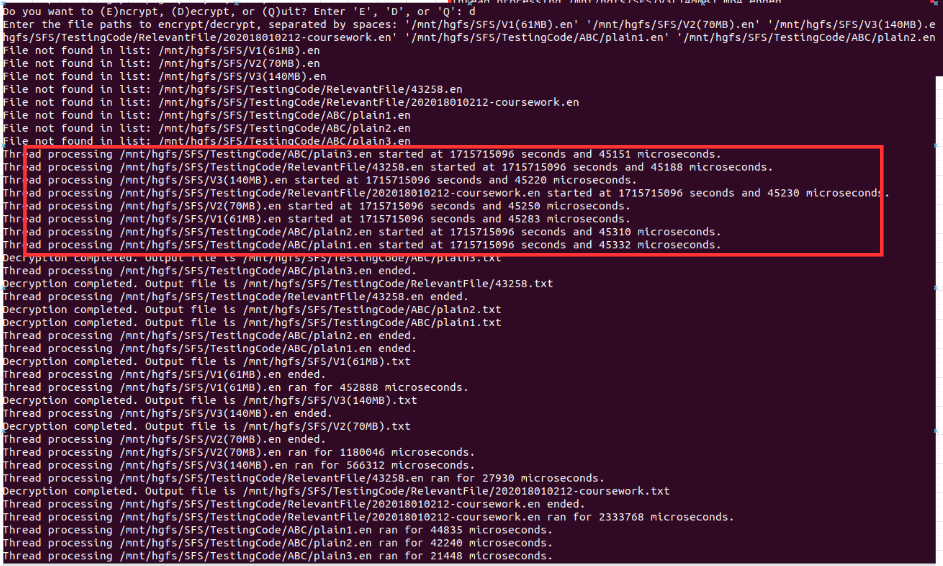


Figure 44: Multi-thread Decryption

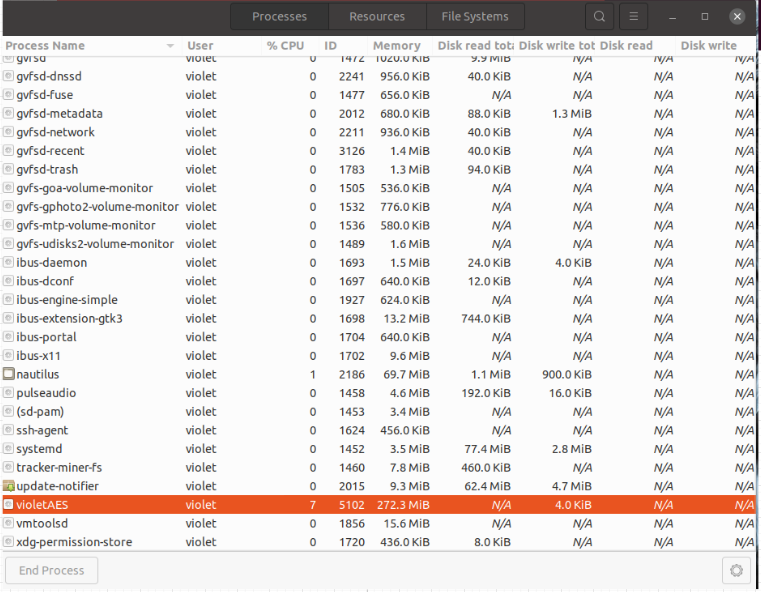


Figure 45: Performance during Encryption Files

**Section 8**

# **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**

# **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**

# **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.

# **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.