

**T.R.**  
**GEBZE TECHNICAL UNIVERSITY**  
**FACULTY OF ENGINEERING**  
**DEPARTMENT OF COMPUTER ENGINEERING**

**UNIFIED FILESYSTEM**

**MERVE HORUZ**

**SUPERVISOR**  
**DR. GÖKHAN KAYA**

**GEBZE**  
**2023**

**T.R.**  
**GEBZE TECHNICAL UNIVERSITY**  
**FACULTY OF ENGINEERING**  
**COMPUTER ENGINEERING DEPARTMENT**

## **UNIFIED FILESYSTEM**

**MERVE HORUZ**

**SUPERVISOR**  
**DR. GÖKHAN KAYA**

**2023**  
**GEBZE**

 <p><b>GEBZE</b> TECHNICAL UNIVERSITY</p>	<p>GRADUATION PROJECT JURY APPROVAL FORM</p>
--	--

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 05/05/2023 by the following jury.

**JURY**

Member

(Supervisor) : Dr. Gökhan KAYA

Member : Prof. Dr. İbrahim SOĞUKPINAR

# ABSTRACT

The aim of this project is to develop a tool called Unified Filesystem, which utilizes the Filesystem in Userspace (FUSE) technology to create a unified virtual drive by combining the storage capacity of different physical devices.

The core technology employed in this project is FUSE, which enables file systems to be implemented in user space rather than within the operating system's kernel. By utilizing FUSE, the Unified Filesystem seamlessly integrates with the operating system, allowing for the creation of a virtual drive that spans multiple physical storage devices. FUSE provides the necessary framework and interface to manage the interactions between the virtual drive and the underlying physical storage, ensuring compatibility across different platforms, including Windows, macOS, and Linux.

Unified Filesystem offers a range of features designed to enhance data management and security. Users can define a virtual drive that encompasses the storage capacity of multiple devices, such as hard drives and flash drives. For instance, if a user has two 2TB hard drives and a 64GB flash drive, Unified Filesystem, built on top of FUSE, combines the available space into a single virtual drive with a total capacity of 4TB + 64GB. This virtual drive is then mounted by the operating system, enabling users to access and manage it through the file explorer, just like any regular drive.

Furthermore, Unified Filesystem incorporates encryption capabilities to protect sensitive data stored on the virtual drive. Users have the option to encrypt their files and folders, ensuring the confidentiality and security of their information. This encryption functionality, integrated with FUSE, guarantees that the data remains protected even if the physical storage devices are compromised.

By leveraging FUSE, Unified Filesystem provides users with a unified storage solution that seamlessly integrates with the operating system. The project aims to simplify data management by abstracting away the complexities of managing multiple devices, allowing users to perform standard file operations, such as creating, deleting, and moving files, with ease.

In conclusion, the Unified Filesystem project, built on the foundation of FUSE, offers a powerful and versatile tool for consolidating storage resources and simplifying data management across different physical devices. By utilizing FUSE's user-space file system capabilities, Unified Filesystem ensures seamless integration, cross-platform compatibility, and enhanced data security through encryption. This tool empowers users to efficiently utilize their available storage space while maintaining the convenience and familiarity of a regular drive within their operating system's file explorer.

**Keywords:** FUSE, file system, encryption.

# ÖZET

Bu proje, FUSE (Filesystem in Userspace) teknolojisini kullanan bir araç olan Birleştirilmiş Dosya Sistemi'ni geliştirmeyi amaçlamaktadır. Bu aracın amacı, farklı fiziksel depolama cihazlarının depolama kapasitelerini birleştirerek sanal bir sürücü oluşturmaktır.

Bu projede kullanılan temel teknoloji FUSE'dir, bu teknoloji sayesinde dosya sistemleri işletim sistemi çekirdeği yerine kullanıcı alanında uygulanabilir. FUSE'nin kullanılmasıyla Birleştirilmiş Dosya Sistemi, işletim sistemiyle sorunsuz bir şekilde entegre olur ve birden fazla fiziksel depolama cihazının kapasitesini kapsayan bir sanal sürücü oluşturur. FUSE, farklı platformlar (Windows, macOS, Linux dahil) arasında uyumluluğu sağlamak için gerekli çerçeve ve arayüzü sunar.

Birleştirilmiş Dosya Sistemi, veri yönetimini ve güvenliğini artırmak için çeşitli özellikler sunar. Kullanıcılar, sabit diskler ve flash sürücüler gibi birden çok cihazın depolama kapasitesini içeren bir sanal sürücü tanımlayabilirler. Örneğin, kullanıcının iki adet 2TB sabit diski ve bir adet 64GB flash sürücüsü varsa, FUSE üzerine inşa edilen Birleştirilmiş Dosya Sistemi, bu cihazların birleşik alanını 4TB + 64GB kapasiteli bir sanal sürücü olarak oluşturur. Bu sanal sürücü, işletim sistemi tarafından bağlanır ve kullanıcılar ona, herhangi bir normal sürücü gibi, dosya gezgininden erişebilir ve yönetebilirler.

Ayrıca, Birleştirilmiş Dosya Sistemi, sanal sürücüde depolanan hassas verileri korumak için şifreleme yetenekleri içerir. Kullanıcılar dosyalarını ve klasörlerini şifrelemeyi tercih edebilirler, böylece bilgilerinin gizliliği ve güvenliği sağlanmış olur. Bu şifreleme özelliği, FUSE ile entegre bir şekilde çalışarak, fiziksel depolama cihazları tehlikeye düşse bile verilerin korunduğunu garanti altına alır.

FUSE'dan faydalanan Birleştirilmiş Dosya Sistemi, işletim sistemiyle sorunsuz bir şekilde entegre olur. Proje, birden çok cihazın yönetimiyle ilgili karmaşıklıkları soyutlayarak kullanıcıların standart dosya işlemlerini (dosya oluşturma, silme, taşıma vb.) kolaylıkla gerçekleştirebilmelerini amaçlar.

Sonuç olarak, FUSE temelinde inşa edilen Birleştirilmiş Dosya Sistemi projesi, farklı fiziksel cihazlar arasında depolama kaynaklarını birleştiren ve veri yönetimini basitleştiren güçlü ve esnek bir araç sunar. FUSE'nin kullanıcı alanında dosya sistemi yeteneklerini kullanarak Birleştirilmiş Dosya Sistemi, sorunsuz entegrasyon, çapraz platform uyumluluğu ve şifreleme aracılığıyla veri güvenliğini sağlar. Bu araç, kullanıcıların mevcut depolama alanlarını etkin bir şekilde kullanmalarını ve işletim sistemi dosya gezginindeki normal bir sürücü gibi kullanmalarını sağlayarak kullanıcı dostu bir deneyim sunar.

**Anahtar Kelimeler:** FUSE, dosya sistemi, şifreleme.

# ACKNOWLEDGEMENT

I would like to express my deepest gratitude and appreciation to my family and friends for their unwavering support throughout this project. Their encouragement, understanding, and belief in me have been invaluable, providing the motivation and strength to overcome challenges and pursue my goals. I am grateful for their patience, understanding, and love, which have sustained me during the ups and downs of this journey.

I am especially thankful to my supervisor, Gökhan KAYA, for his guidance, expertise, and continuous support throughout the development of this project. His invaluable insights, encouragement, and constructive feedback have significantly contributed to the success of this endeavor. I am truly fortunate to have had the opportunity to work under his mentorship and learn from his vast knowledge and experience.

Furthermore, I would like to extend my appreciation to all my teachers and mentors who have played a crucial role in my academic and personal growth. Their dedication, passion, and commitment to education have been instrumental in shaping my abilities and fostering my curiosity. I am indebted to them for imparting knowledge, inspiring me to think critically, and nurturing my intellectual development.

**Merve Horuz**

# CONTENTS

<b>Abstract</b>	<b>iv</b>
<b>Özet</b>	<b>v</b>
<b>Acknowledgement</b>	<b>vi</b>
<b>Contents</b>	<b>vii</b>
<b>1 INTRODUCTION</b>	<b>1</b>
<b>2 SYSTEM DETAILS</b>	<b>3</b>
2.1 BACK-END . . . . .	4
2.1.1 Why FUSE and C Programming Language? . . . . .	4
2.1.2 Back-End Structure of the Project . . . . .	5
2.1.2.1 Structs . . . . .	5
2.1.2.2 Get Attribute Callback . . . . .	7
2.1.2.3 Open Callback . . . . .	8
2.1.2.4 Read Callback . . . . .	8
2.1.2.5 Read Directory Callback . . . . .	8
2.1.2.6 Write Callback . . . . .	9
2.1.2.7 Create Callback . . . . .	10
2.1.2.8 Unlink Callback . . . . .	10
2.1.2.9 Truncate Callback . . . . .	10
2.1.2.10 Release Callback . . . . .	10
2.1.2.11 Make directory Callback . . . . .	10
2.1.2.12 Release Directory Callback . . . . .	11
2.1.2.13 Access and Modification Times Update Callback . .	11
2.1.2.14 Change Owner Callback . . . . .	11
2.1.2.15 Flush Data and User-Data Callback . . . . .	11
2.1.2.16 Rename Callback . . . . .	11
<b>3 Conclusions</b>	<b>12</b>
<b>Bibliography</b>	<b>13</b>

# 1. INTRODUCTION

The rapid advancement of technology has led to an exponential growth in data generation, resulting in an increased need for efficient storage solutions. As individuals and organizations accumulate vast amounts of digital content, managing and accessing data across multiple storage devices has become a complex task. The emergence of a unified filesystem presents a promising solution to this challenge, offering a seamless and user-friendly approach to consolidating storage resources. 1.1

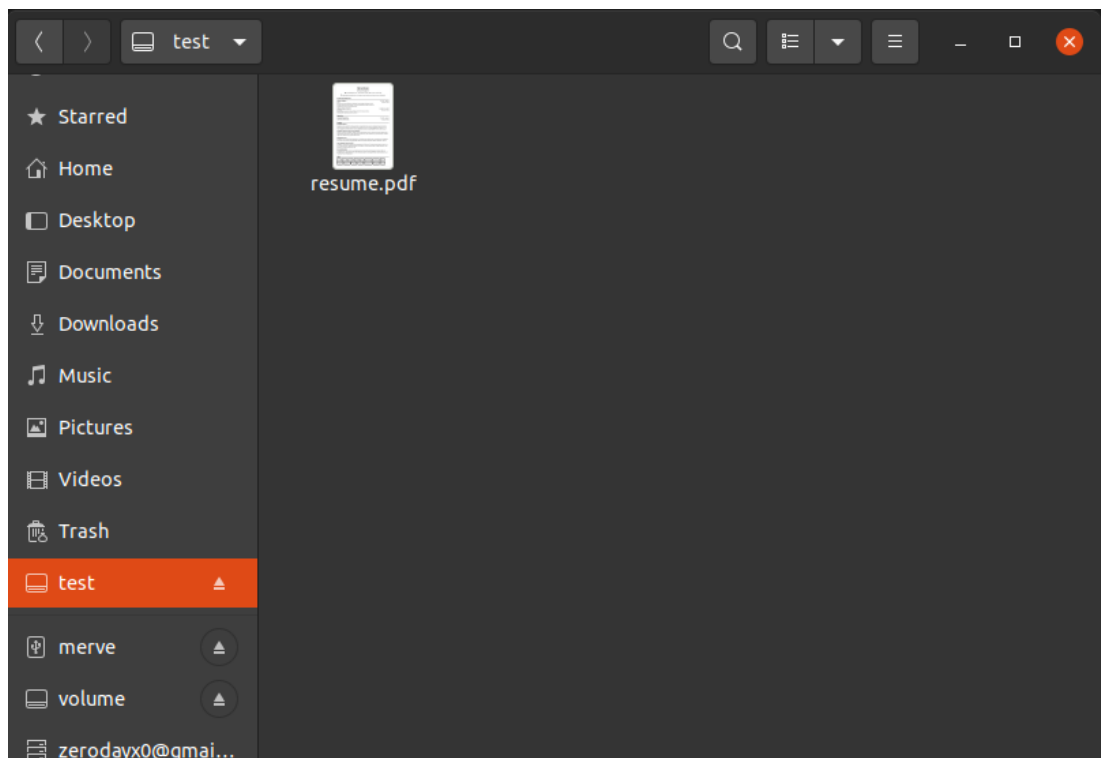


Figure 1.1: Sample view of virtual drive and hard drives.

The purpose of this report is to introduce a tool called Unified Filesystem that aims to revolutionize data management by creating a virtual drive that combines the storage capacity of different physical devices. By leveraging the capabilities of Unified Filesystem, users can define a virtual drive that spans across various storage mediums, such as hard drives, solid-state drives, and flash drives. This unified virtual drive appears to the operating system as a single entity, simplifying data management and providing users with a cohesive and intuitive experience.

The Unified Filesystem abstracts away the complexities of managing multiple storage devices, providing a user-friendly interface that resembles a regular drive within



the operating system's file explorer. Users can perform common file operations such as creating, deleting, moving, and renaming files and folders, thereby streamlining their workflow and enhancing productivity. By leveraging the power of a unified filesystem, users can optimize their available storage space, eliminate the need for manual data organization across multiple drives, and improve overall data management efficiency.

In addition to the consolidation of storage resources, the Unified Filesystem provides advanced features to enhance data security and privacy. Encryption capabilities are integrated into the tool, allowing users to encrypt their data stored on the virtual drive. This ensures that sensitive information remains protected, even if the physical storage devices are compromised. By implementing robust encryption algorithms, the Unified Filesystem ensures the confidentiality and integrity of data, providing peace of mind to users.

In conclusion, the Unified Filesystem project offers a transformative approach to data management by creating a unified virtual drive that consolidates storage resources from various physical devices. With its encryption capabilities, user-friendly interface, and seamless integration with the operating system, the Unified Filesystem empowers users to efficiently organize, access, and secure their data. By simplifying data management and enhancing data security, the Unified Filesystem unlocks new possibilities for optimizing storage space, streamlining workflows, and improving overall productivity in today's data-driven world.

## 2. SYSTEM DETAILS

The Unified Filesystem is developed using FUSE and C on the backend. Its purpose is to develop a tool that combines different physical storage devices into a unified virtual drive, simplifying data management and enhancing storage capacity. By leveraging FUSE and encryption capabilities, the project aims to provide users with a seamless and secure solution for accessing and organizing their data across multiple devices.

The backend system of the Unified Filesystem project is built on the Filesystem in Userspace (FUSE) [1] technology and implemented using the C programming language. FUSE allows the implementation of file systems in userspace, providing greater flexibility and portability across different platforms. By utilizing FUSE, the Unified Filesystem achieves seamless integration with the operating system, enabling the virtual drive to be recognized and accessed like a regular drive within the file system.

The C programming language is chosen for its efficiency and low-level system interactions, enabling optimized data handling and performance. The combination of FUSE and C empowers the Unified Filesystem with the advantages of efficient system-level operations, and the ability to leverage the robust ecosystem of libraries and tools available in the C programming language.2.1 [2]

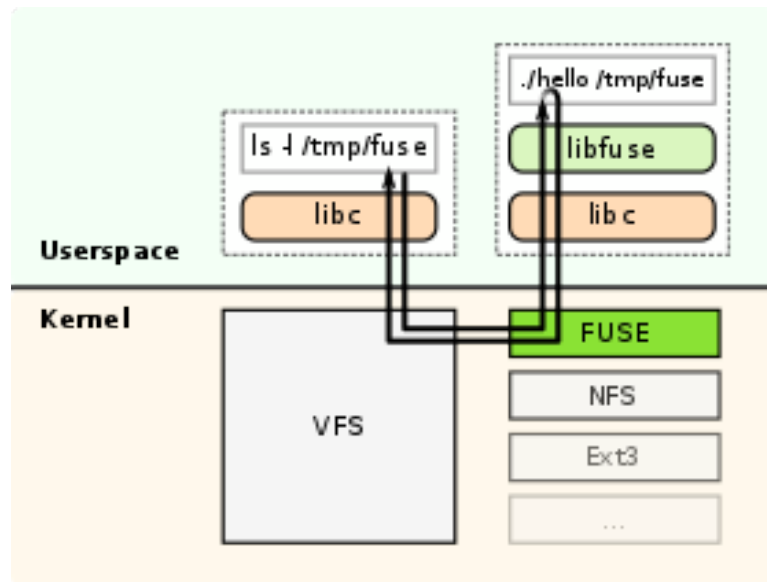


Figure 2.1: Libfuse Structure.

The frontend system of the Unified Filesystem project is built using the Qt framework. Qt provides a robust and versatile development environment for creating graph-

ical user interfaces (GUI) that are intuitive and platform-independent. By leveraging Qt, the Unified Filesystem achieves a user-friendly and visually appealing interface for interacting with the virtual drive. Qt's extensive library of UI components and widgets allows for the seamless integration of file explorer functionalities, enabling users to perform standard file operations such as creating, deleting, moving, and renaming files and folders with ease. [3]

## **2.1. BACK-END**

### **2.1.1. Why FUSE and C Programming Language?**

FUSE (Filesystem in Userspace) has been selected as the backend for the Unified Filesystem project due to its unique advantages in implementing file systems. FUSE allows the development of file systems in userspace rather than within the operating system kernel. This decoupling of the file system from the kernel provides several benefits. [4]

Firstly, FUSE offers increased flexibility in developing file systems. By operating in userspace, developers have greater freedom to experiment and innovate without the constraints imposed by kernel-level programming. This flexibility enables the implementation of advanced features and functionalities that may not be readily available in traditional kernel-based file systems. Additionally, FUSE allows for the integration of user-defined file systems, enabling users to customize their storage solutions according to their specific needs. [5]

Secondly, FUSE ensures portability across different platforms and operating systems. The unified interface provided by FUSE abstracts the underlying system-specific details, making the file system compatible with various operating systems, including Windows, macOS, and Linux. This cross-platform compatibility is invaluable in today's diverse computing landscape, enabling users to seamlessly access and manage their data regardless of the operating system they are using. [6]

Furthermore, the use of the C programming language in the backend of the Unified Filesystem project offers distinct advantages. C is a low-level programming language known for its efficiency, performance, and direct access to system resources. These characteristics make it an ideal choice for implementing a file system that requires precise control over storage operations, data handling, and system interactions. By leveraging the power of C, the Unified Filesystem can efficiently manage storage resources, handle file operations, and interact with the underlying hardware, ensuring optimal performance.

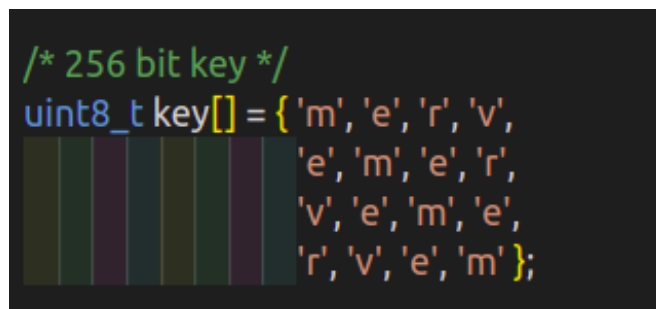
In conclusion, the selection of FUSE as the backend and the utilization of the C

programming language in the Unified Filesystem project demonstrate a strategic choice to harness the flexibility, portability, and efficiency offered by these technologies. The combination of FUSE's userspace file system capabilities and C's low-level system interactions provides a robust foundation for building a feature-rich and high-performance file system that meets the diverse needs of users across different platforms and operating systems.

### 2.1.2. Back-End Structure of the Project

There are 2 parts in the backend structure of the project: encryption and main program.

- Encryption [7]: AES(Advanced Encryption Standard) encryption algorithm is used. It is a symmetric encryption algorithm, meaning the same key is used for both encryption and decryption processes. AES employs a series of mathematical operations, such as substitution, permutation, and bitwise operations, to transform plaintext into ciphertext. During encryption, the AES algorithm applies multiple rounds of these operations to increase the security of the encrypted data. The number of rounds 14 rounds for a 256-bit key.2.2 [8]



```
/* 256 bit key */
uint8_t key[] = {'m','e','r','v',
                 'e','m','e','r',
                 'v','e','m','e',
                 'r','v','e','m'};
```

Figure 2.2: The key is used in AES.

- Main program: The main program is detailed below.

#### 2.1.2.1. Structs

The struct named "drives" is used to store information about hard drives, specifically their paths and the number of drives present.2.3

The "file part" struct is used to store information about a part or fragment of a file. This struct is likely used in the Unified Filesystem project to manage files that have been split into smaller parts for storage or transfer purposes. It allows for tracking

```
struct drives {
    char** paths;
    size_t num;
};
```

Figure 2.3: The struct drives.

individual file parts and their associated information, such as timestamps, sizes, and paths. 2.4

```
// used to store file parts
struct file_part {
    unsigned long timestamp;
    size_t part_no;
    char* part_path;
    size_t size;
};
```

Figure 2.4: The struct  $file_{part}$ .

The "file" struct is used to store information about a file. The "file" struct is likely used to organize and manage files within the Unified Filesystem. It provides a comprehensive set of attributes and pointers to handle various file-related operations, such as accessing file metadata, managing file parts, and tracking timestamps. 2.5

```
// used to store file information
struct file {
    char* file_name;
    char* file_path;
    size_t size;
    size_t part_count;
    int mode;
    // access time
    struct timespec atime;
    // modify time
    struct timespec mtime;
    struct file_part* parts;
};
```

Figure 2.5: The struct file.

The "directory" struct is used to store information about a directory (folder). This struct is likely used to build a hierarchical structure for organizing directories and files within the Unified Filesystem project. It allows for tracking relationships between directories, storing file information, and maintaining counts of child directories and files. 2.6

```
// used to store directory information
struct directory {
    struct directory* parent;
    struct directory* children;
    struct file* files;
    size_t num_children;
    size_t num_files;
    char* name;
    int mode;
};
```

Figure 2.6: The struct directory.

### 2.1.2.2. Get Attribute Callback

The "getattr callback" function is a callback function used in the Unified Filesystem project. This function is called to retrieve the attributes of a file or directory specified by the given path.

The function begins by initializing the struct stat object to all zeros. It then checks if the path corresponds to the root directory ("/"). If it does, the function sets the appropriate mode and link count for a directory and returns successfully.

Next, the function extracts the file name and the rest of the path from the given path by splitting it at the last '/' character. It handles the edge case for the root directory by setting the rest of the path to "".

The function then checks if the file name is empty. If it is, it attempts to retrieve the corresponding directory using the rest of the path. If the directory exists, the function sets the mode and link count for a directory and returns successfully. Otherwise, it returns an error indicating that the directory does not exist.

If the file name is not empty, the function checks if the full path corresponds to a directory. If it does, the function sets the appropriate mode and link count for a directory and returns successfully.

If the full path is not a directory, the function attempts to find the file within the given directory. If the file is found, it sets the mode, link count, and size attributes for a regular file and returns successfully.

If the file or directory is not found in the specified path, the function returns an error indicating that the entry does not exist.

In summary, the "getattr callback" function handles the retrieval of file and directory attributes based on the given path, distinguishing between directories and regular files and returning the appropriate attribute values or error codes.

### **2.1.2.3. Open Callback**

Open path. mode is as for the system call open. We do not really have to implement this function. It's useful to start prefetch and to cache open files, and check that the user has permission to read/write the file.

### **2.1.2.4. Read Callback**

The "read callback" function is a callback function used in the Unified Filesystem project. This function is called when reading data from a file specified by the given path.

The function starts by extracting the file name and the remaining path from the given path using the "split path and file name" function. It then retrieves the corresponding directory using the remaining path.

Next, the function searches for the file within the directory based on the extracted file name. Once found, it obtains a pointer to the file.

To read the file data, the function allocates memory for the data buffer. It iterates over each part of the file, considering offsets and sizes to read the appropriate data. It opens each part file, reads the data into a temporary buffer, decrypts the data, and then copies the decrypted data into the data buffer. The process is repeated until the requested size is read or until all parts are processed.

Finally, the function copies the requested portion of the data buffer into the buf parameter, frees allocated memory, and returns the size of the read data.

In summary, the "read callback" function handles the reading of file data by retrieving the file's parts, decrypting the data if necessary, and copying the requested portion of the data into the output buffer.

### **2.1.2.5. Read Directory Callback**

The "readdir callback" function is a callback used in the Unified Filesystem project to read and retrieve the contents of a directory specified by the given path.

To begin, the function calls the "get directory" function to obtain the corresponding directory structure based on the provided path. Once the directory structure is obtained, the function utilizes a loop to iterate over the files and child directories within the current directory. Using the filler function, the function adds the names of the current directory (".") and the parent directory ("..") to the buffer.

Subsequently, the function traverses through the files in the directory, adding their names to the buffer using the filler function. Similarly, it iterates through the child

directories and includes their names in the buffer as well.

Finally, the function successfully returns, signifying the completion of the directory reading operation.

#### **2.1.2.6. Write Callback**

The "write callback" function is a callback used in the Unified Filesystem project to handle the write operation for a file specified by the given path.

Upon receiving the write request, the function retrieves the file name and the remaining path by utilizing the "split path and file name" function. It then obtains the corresponding directory structure by calling the "get directory" function on the remaining path.

Next, the function searches for the file within the directory based on the retrieved file name. If the file is not found, it returns an error indicating that the file does not exist.

Once the file is located, the function invokes the "write to drives" function, passing the buffer (buf), size, file information, and the remaining path. This function handles the actual writing of the data to the appropriate drives.

After the write operation is completed, the function frees the allocated memory for the file name and remaining path and returns the size of the data written.

In this callback, called "write to drives" function. The "write to drives" function is responsible for writing the data of a file to the available drives in the Unified Filesystem project.

First, the function splits the input data into smaller parts using the "split file" function, resulting in a collection of parts and the total count of parts. It then initializes variables to keep track of the number of bytes written, the file descriptor for the part file, and the path of the part file.

In a loop, the function determines the drive with the most available free space by iterating over the available drives and retrieving their information using the "get drive" info function. It selects the drive with the maximum free space.

The function then constructs the path for the part file by combining the drive path, the given path, a timestamp, and the part count. It opens the part file and performs error checking.

Next, it determines the size of the part to write, encrypts the part data using the encrypt function, and adjusts the part size to ensure it is a multiple of 16 bytes. It writes the encrypted part data to the part file and updates the file's metadata, including the timestamp, part path, size, and part number.

After completing the loop for all parts, the function closes the part file, frees allocated memory, and updates the file size. The write operation to the drives is now



complete.

#### **2.1.2.7. Create Callback**

The "create callback" function is responsible for handling the creation of a new file in the Unified Filesystem project.

When called, the function first extracts the file name and the remaining path from the given path using the "split path and file name" function. It then retrieves the corresponding directory using the "get directory" function.

Next, the function checks if the directory already contains files. If so, it increments the file count and reallocates memory for the updated file array. Otherwise, it initializes the file count and allocates memory for the file array.

The function then assigns the file name, size, mode, part count, and parts array to the newly created file in the directory's file array. Memory is allocated for the file name and copied from the extracted file name string.

#### **2.1.2.8. Unlink Callback**

Remove a file.

#### **2.1.2.9. Truncate Callback**

Shrink or enlarge a file.

#### **2.1.2.10. Release Callback**

The inverse of open.

#### **2.1.2.11. Make directory Callback**

The "mkdir callback" function handles the creation of a new directory in the Unified Filesystem project. Upon invocation, the function extracts the directory name and the remaining path from the given path using the "split path and file name" function. It then retrieves the corresponding directory using the "get directory" function.

Next, the function checks if the directory exists. If not, it returns -1 to indicate an error. Otherwise, it proceeds to create a new child directory within the current directory. If the directory already contains children, it increments the child count and reallocates memory for the updated child directory array. Otherwise, it initializes the

child count and allocates memory for the child directory array.

The function assigns the directory name, number of children, child directory array, number of files, file array, and mode to the newly created child directory. Memory is allocated for the directory name and copied from the extracted directory name string.

#### **2.1.2.12. Release Directory Callback**

The inverse of open.

#### **2.1.2.13. Access and Modification Times Update Callback**

The "utimens callback" function is responsible for updating the access and modification times of a file in the Unified Filesystem project. Upon invocation, the function extracts the file name and the remaining path from the given path.

The function iterates through the files within the directory to find the file that matches the extracted file name. Once a matching file is found, the function updates its access time (atime) and modification time (mtime) using the provided ts array, which contains the new timestamps.

#### **2.1.2.14. Change Owner Callback**

Change the owner of a file or directory.

#### **2.1.2.15. Flush Data and User-Data Callback**

Flush data and user-data to disk.

#### **2.1.2.16. Rename Callback**

The "rename callback" function is responsible for renaming a file within the Unified Filesystem project. Upon invocation, the function extracts the file name and the remaining path from the given path and "new path". The function first checks if a file with the same name already exists at the destination path (new path). If such a file exists, it is deleted by freeing its memory and shifting the remaining files to the left to close the gap.

Next, the function searches for the file to be renamed within the source directory (dir). Once found, it frees the old file name and allocates memory for the new file name. It then copies the new file name (new file name) into the allocated memory.

### 3. CONCLUSIONS

In conclusion, the Unified Filesystem project has successfully developed a tool that revolutionizes data management by creating a unified virtual drive from multiple physical storage devices. By leveraging the power of FUSE technology and the C programming language, the project has achieved its goal of providing users with a seamless and secure solution for accessing and organizing their data.

The implementation of FUSE as the backend for the Unified Filesystem has proven to be a strategic choice. FUSE's userspace file system capabilities offer flexibility, portability, and compatibility across different platforms and operating systems. This allows the virtual drive to be seamlessly integrated into the operating system's file system, presenting it as a regular drive and enabling users to perform standard file operations without any additional complexity. The decoupling of the file system from the kernel also provides developers with the freedom to innovate and implement advanced features to enhance data management.

The use of the C programming language in the backend further strengthens the Unified Filesystem project. C's efficiency, performance, and low-level system interactions have enabled optimized data handling, efficient storage management, and seamless hardware interactions. This ensures that the Unified Filesystem operates smoothly, even with large data volumes and complex file operations.

The integration of AES encryption within the Unified Filesystem project adds an important layer of security to protect sensitive data. AES, being a widely adopted encryption algorithm, ensures that the encrypted data remains confidential and resistant to cryptographic attacks. By incorporating AES encryption, users can have peace of mind knowing that their data stored on the virtual drive is safeguarded from unauthorized access.

Overall, the Unified Filesystem project has successfully addressed the challenges of data management across multiple storage devices. By providing a unified virtual drive, seamless integration with the operating system, and advanced encryption capabilities, the project offers users a user-friendly and secure solution for organizing, accessing, and protecting their data. The combination of FUSE, C programming language, and AES encryption demonstrates the project's commitment to flexibility, efficiency, and security, making the Unified Filesystem a valuable tool for optimizing storage space and enhancing data management in today's data-driven world.

# BIBLIOGRAPHY

- [1] “FUSE (Filesystem in Userspace) Official Website.” (), [Online]. Available: <https://github.com/libfuse/libfuse> (visited on 06/18/2023).
- [2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, 1988.
- [3] T. Q. Company. “Qt Framework.” (), [Online]. Available: <https://www.qt.io/> (visited on 06/18/2023).
- [4] M. Szeredi, “FUSE: Implementing a Fully Functional File System in User Space,” *Proceedings of the Ottawa Linux Symposium*, pp. 111–124, 2001.
- [5] M. Szeredi and R. Hinds, “FUSE: Filesystem in Userspace - Design and Implementation,” in *Proceedings of the Linux Symposium*, 2005, pp. 223–238.
- [6] D. Powell, *Practical File System Design with the FUSE Library*. No Starch Press, 2019.
- [7] K. S. Kumar and D. P. R. Kumar, “Data Encryption and Decryption Using Advanced Encryption Standard (AES) Algorithm,” *International Journal of Engineering and Computer Science*, vol. 4, no. 9, pp. 14 474–14 478, 2015.
- [8] J. Daemen and V. Rijmen, “The Design of Rijndael: AES - The Advanced Encryption Standard,” *Springer-Verlag*, vol. 31, no. 3, pp. 97–139, 2002.