FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Milan Veselý

Virtual file system in user space

Department of Software Engineering

Supervisor of the bachelor thesis:	RNDr. Jakub Yaghob, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

To my family, thank you for feeding me and keeping me alive during this thesis writing process. To my supervisor, RNDr. Jakub Yaghob, Ph.D., thank you for guidance, patience, and expertise throughout the entire thesis process. And to my roommates, thank you for not driving me insane with your noisy shenanigans while I was trying to work. This thesis is dedicated to all of you for making this experience more bearable and even enjoyable at times.

Title: Virtual file system in user space

Author: Milan Veselý

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering

Abstract: This thesis presents a custom Virtual File System (VFS) built with C++, using a custom wrapper for the FUSE library. The VFS is modular in design, facilitating easy extension with new features. Two prototype modules for versioning and encryption are also included, each accompanied by a command-line interface tool for control, such as restoring versions or encrypting files. Thanks to its password/key protection and encryption capabilities, the VFS enhances file management, enabling secure storage and retrieval of sensitive data. In addition, built-in versioning functionality allows users to access and restore previous file versions with ease. Once the VFS is mounted, it can be accessed in the same standard way as any other file system, making it user-friendly and accessible.

Keywords: VFS user space versioning encryption

Contents

Introduction

As the significance of data continues to grow, many users require advanced features such as versioning or file-level encryption for their data storage. The main goal of this thesis is thus to create a modular and easily extensible virtual file system (VFS) in user space as a comprehensive solution for these advanced storage needs. Created with FUSE (Filesystem in Userspace) using C++, the VFS should be able to be mounted irrespective of the underlying file system.

The created VFS should also provide prototypes of modules for encryption and versioning, seamlessly integrating them into the core functionality with minimal user effort. Specifically, the proposed VFS prototypes should enable users to create snapshots for later rollbacks and let them easily encrypt individual files or entire directories with a password or key, providing temporary decryption as needed.

Currently, there is no straightforward method for adding, let alone layering, these features onto an existing file system, and users who desire such additional functionality must rely on separate external programs. Even though some suitable solutions exist, they are usually not provided on file system-level, and the few rare exceptions that do exist are often outdated or difficult to use. On the other hand, if a user does not need benefits of file system-level integration, there are many programs that can be used to achieve similar results. The primary issue with these programs, which this thesis aims to address, is the necessity of accessing files through specialized software.

In contrast, the proposed VFS acts as an intermediate layer between the operating system and specific file systems, delivering desired features in a more streamlined and user-friendly manner. Not to mention the fact that the VFS does not have some limitations of such programs, as it can be mounted on top of any file system, including even network file systems. Furthermore, these features could also be easily layered on top of each other as opposed to using separate programs, where, for example, versioning and encryption would be rather problematic. However, this VFS is not intended to be a replacement for existing solutions, but rather a proof of concept that demonstrates the feasibility of creating a modular and extensible VFS.

The thesis is structured as follows: Chapter 1 provides a theoretical overview of file systems, virtual file systems, encryption, and versioning. Chapter 2 delves into the analysis of alternatives, the rationale behind the project's choices, and the explanations of FUSE and other libraries employed for encryption and testing.

After this theoretical foundation, Chapter 3 describes the necessary steps needed to start the development of the VFS such as cross-platform build system or the testing framework; and In Chapter 4, the design and architecture of the VFS solution are described, including the adaptation of FUSE for usage in C++ and the modular architecture that allows adding various VFS features. Chapter 5 details the implementation of essential FUSE operations, prototyping encryption, and versioning in the VFS and discussing the methods and strategies employed. Finally, in Chapter 6, the usability, reliability, security, and performance of the VFS are evaluated, providing insights into its overall effectiveness compared to existing solutions. The conclusion summarizes the findings and discusses future work for system enhancement, while Appendix A and B offers a guide on installing and using the custom VFS.

Chapter 1

Theoretical overview

The development of a custom Virtual File System (VFS) with modules for versioning and encryption capabilities requires an understanding of several key concepts and technologies. This chapter aims to provide the necessary background and context to better understand the VFS implementation discussed in later chapters. Even for those already familiar with the subject, a brief refresher might be helpful.

1.1 Understanding File Systems

Let us start with what a file system is and what it does, as it is important for understanding the VFS itself.

A file system is a critical component of any operating system, with the task of managing and organizing data stored on a device. As described by Oja et al.[1], a file system represents the methods and data structures used by an operating system to maintain files on a disk or partition, that is, how files are organized on the disk. This consists of multiple operations such as creation, reading, modifying, and deleting files or directories. This is done in order to allow users and applications to interact seamlessly with the underlying storage medium.

It is important to distinguish between a disk or partition and the file system it contains. While some programs operate directly on raw disk sectors or partitions, which could potentially damage or corrupt existing file systems, most programs interact with a file system.

There are various types of file systems, each tailored for specific operating systems and purposes. Notable file systems include NTFS for Windows, HFS+ for macOS, and ext4 for Linux. Cross-platform compatibility can be attained using universal file systems like FAT32 or exFAT, which are compatible with multiple operating systems.

File systems employ diverse techniques to organize data, including partitioning, allocation units, and indexing. Additionally, file systems manage metadata, which is crucial for organizing and accessing stored data. Metadata encompasses information such as file names, creation and modification timestamps, file sizes, and access permissions.

1.2 Virtual File Systems

A VFS is an essential abstraction layer in modern operating systems, designed to ease the interaction between various file systems and user applications. The VFS functions as an intermediary, enabling applications to access various file systems through a unified interface, regardless of the underlying file system's specific characteristics or structure.

According to a book by David A. Rusling[2], the VFS manages various mounted file systems by maintaining data structures that describe the entire virtual file system and the real mounted file systems. The VFS uses superblocks and inodes, similar to the EXT2 file system, to describe files and directories within the system. Each file system register with the VFS during operating system initialization. File system modules are loaded as needed, which allows VFS to read their superblocks. The VFS maintains a list of mounted file systems and their VFS superblocks, with each superblock containing pointers to specific file system functions.

The VFS inodes are traversed as the system's processes access directories and files. Frequently accessed inodes are kept in the inode cache for quicker access. Additionally, Linux VFS uses a common buffer cache, independent of the file systems, to cache data buffers from underlying devices.

VFS also plays a critical role in operating system design as it not only offers file system independence, but also provides benefits such as improved security or better performance. In this thesis, we will also demonstrate that the extensibility of the VFS is a valuable advantage, enabling the incorporation of advanced features without requiring kernel modifications.

1.2.1 VFS Operations

VFS provides a range of operations for interacting with files and directories. These operations are implemented as system calls, such as `open(2)`, `stat(2)`, `read(2)`, `write(2)`, and `chmod(2)`, which are called from a process context.

The way this is implemented in Linux VFS is described by an article by Richard Gooch and Pekka Enberg[3]. During a system call, the VFS translates a pathname into a directory entry (dentry) by searching through the dentry cache. However, if the cache is too small to fit all dentries in RAM, the VFS may have to create dentries and load inodes to resolve the pathname. Each dentry typically has a pointer to an inode, which represents a file system object that may be on disk or in memory.

When a file is opened, a file structure is allocated and initialized with a pointer to the dentry and file operation member functions taken from the inode data. The `open()` file method is then called to enable the file system implementation to perform its work. Finally, the file structure is placed into the file descriptor table for the process.

To read, write, and close files, the user-space file descriptor is used to call the appropriate file structure method. As long as the file is open, the corresponding dentry and VFS inode remain in use.

1.3 Encryption

Another concept that has to be briefly discussed is encryption. It is simply a process of applying cryptographic algorithms to information in such a way that only authorized parties can access it. In this thesis, the encryption will be used as a mean to transform the contents of files and directories into an unreadable format, which can only be deciphered with the appropriate decryption key. Utilizing encryption enables users to effectively safeguard their data against unauthorized access, data breaches, and other malicious activities.

For file and directory encryption, symmetric key algorithms are commonly utilized due to their computational efficiency and robust security properties. The Advanced Encryption Standard (AES) is a widely adopted option; however, in this thesis, the more modern and faster XChaCha20-Poly1305 algorithm has been chosen as the primary encryption method. To derive an encryption key from a user-provided password, password-based key derivation functions will be implemented.

1.3.1 XChaCha20-Poly1305

XChaCha20-Poly1305 is an advanced symmetric encryption algorithm that combines the XChaCha20 stream cipher with the Poly1305 message authentication code (MAC). This modern encryption algorithm is considered faster than AES, especially in software implementations, making it a suitable choice for this thesis [4].

The XChaCha20 stream cipher operates with a 256-bit secret key and a 192-bit nonce. The increased nonce size in XChaCha20 significantly improves its resistance to nonce reuse attacks compared to the original ChaCha20 cipher. The algorithm employs a series of simple operations, such as addition, rotation, and

XOR, to generate a keystream, which is subsequently combined with the plaintext to produce the ciphertext.

To encrypt data using XChaCha20-Poly1305, a user first generates a secret key, which is a random sequence of 256 bits. Then, the algorithm uses the secret key to transform the plaintext into ciphertext. The ciphertext can only be decrypted back to its original plaintext form using the same secret key.

1.4 Versioning

Last but not least, versioning is a technique used to maintain multiple copies or states of a file or directory, allowing users to access and restore previous versions as needed. It enables tracking of changes made to files over time, facilitating collaboration and recovery from accidental data loss or corruption.

Usually, versioning is implemented by storing multiple copies of a file or directory, each representing a different version. This approach is straightforward and easy to implement, but it can be inefficient in terms of storage space and performance. To address these issues, versioning can be implemented using a diff-based approach, which stores only the differences between versions, rather than storing full copies of every version.

Chapter 2

Analysis

In order to determine the feasibility of the proposed VFS, it is crucial to analyse existing alternatives. In this chapter, we will delve into a detailed examination of various alternative solutions, highlighting their drawbacks and limitations. Additionally, we will explore the essential libraries and tools required for the successful implementation of the proposed solution. By thoroughly analyzing existing alternatives and exploring necessary components, we can better determine the viability of the proposed VFS.

2.1 Analysis of Alternatives

The proposed solution presents a unique approach, as there is currently no existing virtual file system specifically designed to layer additional features atop pre-existing file systems. Nevertheless, FUSE, the very library which was employed in developing this VFS, could be viewed as its competition, given that it can be utilized to enhance existing file systems with added features. On the contrary, FUSE demands considerable effort to build a custom file system from scratch, and even if a developer undertakes the challenge, they would essentially be replicating a simplified version of this thesis' objective.

However, using VFS is not the only way to achieve the desired functionality of adding features, and in order to judge the potential benefits of our VFS, it is essential to individually examine alternative solutions that simply provide custom features on top of existing file systems. This section will analyse existing non-virtual and virtual file systems with versioning and encryption features, as well as higher-level applications offering similar functionality.

Let us start with versioning; non-virtual versioning file systems is a well-documented concept, and there are several implementations available. Notable non-virtual versioning file systems include OpenVMS, a versioning file system from Digital Equipment Corporation that automatically creates new instances of files with version numbers appended, and NILFS, a Linux-based log-structured file system that supports versioning and continuous snapshotting of the entire file system. And even though these solutions would be viable options, they are not what someone might consider user-friendly. They also offer limited configurability and extensibility, which is a significant drawback for a user who wants to customize the versioning process.

Still, several virtual file systems with versioning capabilities have been developed:

- User space file systems implemented with FUSE:
  - A simple versioning file system for Linux using FUSE[5], written in Go.
  - Wayback: A User-level Versioning File System for Linux[6], developed in Perl for the USENIX 2004 Annual Technical Conference using an older version of FUSE.
- Copy-on-Write Version Support for VFS under Linux by Stephan Müller and Sven Widmer[7], implemented as a kernel patch.
- A versioning virtual filesystem by Steve Huntley[8], written in Tcl. However, this solution primarily serves as a language demonstration rather than a practical implementation.

These existing solutions are also not without various constraints, such as being discontinued or being platform-specific.

The third option for versioning is to use higher-level applications. These applications usually offer a wide range of features while still allowing extendability. Some of the most popular applications include Git, Subversion, Time Machine, and Dropbox. And to be honest, using these applications would be a practical solution, as they are well-maintained and offer a wide range of features.

Regarding encryption, while such focused VFSs are basically nonexistent, there are several non-virtual file systems that offer encryption capabilities, such as the EFS which provides cryptographic protection of individual files on NTFS file system volumes using a public-key system. And surely enough, such solutions are viable options, but they from my personal experience lack the configurability and flexibility that a virtual file system can provide, and once more the extensibility is limited. For instance, as a dual-boot user, if I would like to have my files encrypted on the common partition, it would be not possible or at least very difficult to achieve with existing solutions.

Moving back to the VFS domain, the closest example to the desired functionality is rvault[9], which focuses on encrypting small files (passwords, keys, and secrets) and makes them accessible through one-time password authentication. However, this does differ from the intended functionality of the proposed VFS and could not be considered as an alternative, rather it may serve as a source of inspiration. Again, the competition is mainly higher-level applications, such as Folder Lock, Gpg, or Encrypto.

2.1.1 Drawbacks of higher-level applications

Since we now established that the main competition is higher-level applications, let us examine why developing proposed VFS is still a reasonable idea even though one might label it as "overkill". While we are not saying that higher-level applications are not a viable option, they are not meant for the same user base. Depending on specific solution, negative aspects could include problematic layering of additional features, limited multi-platform support and leaking implementation details to the user. And even more importantly, users are limited to using that particular application, while with VFS they can use any application they want. Yet, for the usage of VFS to be convenient, the user's application of choice would need to be slightly modified.

On that note, it is also worth noting that some operating systems, specifically macOS, provide built-in support for versioning and limited encryption features. For example, macOS's Time Machine offers versioning the way that it is possible to revert to previous versions of files or restore deleted data. However, these features are not available on other platforms, and they may not provide the desired level of control over the process.

So, while there are several existing solutions that offer versioning and encryption features, they are usually platform-specific, lack the desired level of control or extensibility. Additionally, there are no existing virtual file systems that provide the desired functionality.

2.2 Necessary libraries

In the development of a virtual file system, there are two primary approaches to consider: writing kernel drivers or utilizing a user space library. Writing kernel drivers offers low-level access to the operating system but requires extensive knowledge of the kernel and platform-specific implementations. This method can be time-consuming, error-prone, and challenging to maintain. Furthermore, this would result in limited extensibility, as the kernel driver would be responsible for all the functionality of the file system. Not to mention the fact that kernel drivers are platform-specific, and they must be re-written for each platform. However, despite the tedious process, it may result in a more efficient and robust solution.

An alternative to writing kernel drivers is using a user space file system library, such as FUSE. This type of library allows developers to focus on the functionality and logic of their custom file system, rather than the intricate details of kernel programming. This approach is more portable and easier to maintain, but it may not be as efficient as a kernel driver. Nonetheless, the performance difference between the two approaches is negligible in most cases, and the benefits of using a user space library heavily outweigh the drawbacks in this particular scenario.

2.2.1 FUSE

FUSE (named after Filesystem in Userspace) is an open-source software interface that allows developers to create custom file systems in user space without the need for kernel modifications. It operates by providing a bridge between the kernel's VFS layer and user-space file system implementations. The kernel communicates with the user-space file system using a well-defined API, allowing developers to create file systems without the need for kernel-level programming. This abstraction greatly simplifies the development process and reduces the risks associated with kernel modifications.

FUSE has gained widespread adoption due to its ease of use and modularity. Many popular file systems have been implemented using FUSE, such as SSHFS, GlusterFS, S3FS and NTFS-3G, among others. These implementations demonstrate the versatility and robustness of the FUSE framework in handling a wide range of file system requirements. Yet, FUSE is not without its limitations. It is arguable whether it is the best choice for a C++ implementation, as it is written in C, but more on that in a later section 3.1. Another complication is its portability. Fortunately, even though FUSE was initially designed for Linux, variants are available for other platforms, ensuring cross-platform compatibility:

- **Linux**: libfuse - The reference implementation of FUSE [10].
- **macOS**: macFUSE - A macOS port of FUSE [11].
- **Windows**: WinFsp - A Windows File System Proxy that provides FUSE-compatible functionality [12].

To develop a new file system using FUSE, a handler program connected to the provided libfuse library must be created. The primary objective of this handler program is to define the file system's behavior in response to read, write, and stat requests. Additionally, the handler program is responsible for mounting the new file system. Upon mounting, the handler is registered with the kernel.

When a user initiates read, write, or stat requests for the newly mounted file system, the kernel forwards these I/O requests to the handler, which processes them accordingly. The handler's response is then relayed back to the user by the kernel, ensuring a seamless interaction between the custom file system and the operating system.

The FUSE flow-chart diagram created by Wikipedia user named Sven[13] depicted in Figure 2.1 effectively illustrates the process involved in handling a command like 'ls', from its initial submission to the virtual file system layer, through the FUSE kernel module, to a handler program in user space and back.

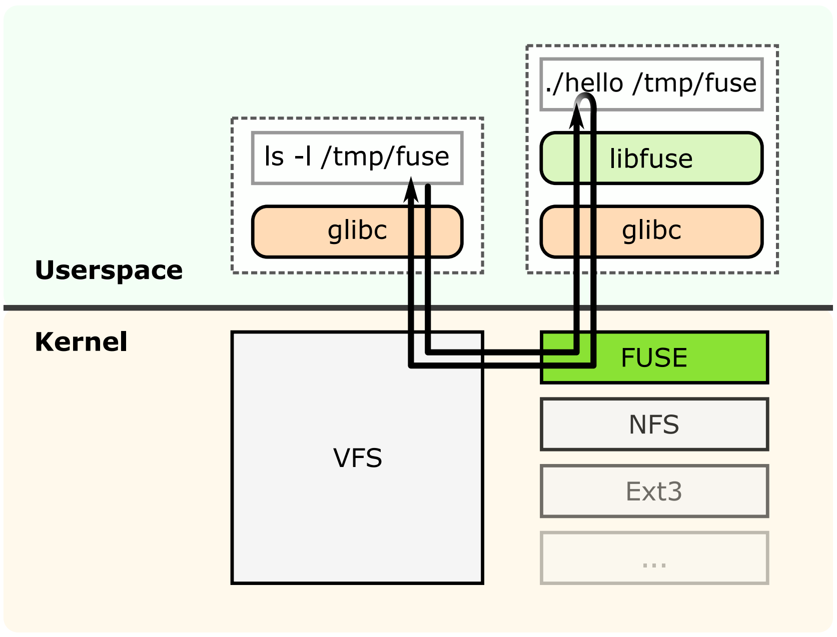

Figure 2.1 FUSE flow-chart diagram

Considering the advantages of user space development and the availability of FUSE for multiple platforms, FUSE is my preferred choice for implementing the proposed VFS. This approach enables the development of a cross-platform VFS with versioning and encryption capabilities while avoiding the complexity of kernel driver development. Still, several other libraries will need to be employed to address various aspects of the project.

2.2.2 Encryption

Initially, Crypto++[14] was considered for integration into the custom VFS, given its comprehensive open-source C++ class library covering a wide array of cryptographic schemes, such as encryption, hashing, and authentication algorithms.

Utilizing Crypto++ would provide file-level encryption, effectively safeguarding the security and privacy of the data stored within the VFS.

However, the implementation of Crypto++ led to various issues, primarily complicating the build process. As a result, the decision was made to switch to libsodium[15], which is a more user-friendly alternative. It offers a robust selection of encryption, decryption, and cryptographic functions while maintaining a focus on simplicity and efficiency. With that, the custom VFS can achieve the desired level of data protection without incurring the complexities associated with Crypto++.

2.2.3 Testing

Google Test[16], commonly referred to as gtest, is a widely-used C++ testing framework developed by Google. This library enables testing of individual components and the overall functionality of the custom VFS. By employing gtest, the quality, reliability, and performance of the custom VFS can be evaluated, guaranteeing that it fulfills the requirements and expectations outlined in this thesis.

2.2.4 Options parsing

The Boost Program Options library[17] is a C++ library used for option parsing, which makes it easy to parse command-line options and arguments. This library has been utilized for the custom VFS to handle different command-line options effectively, including setting the mount point, enabling or disabling debugging mode, and so on. Besides that, it was also used for the supporting tools.

The reason why this library was chosen in particular is because of its ease of use, popularity, and variety of features.

2.2.5 Space for improvement

While the libraries mentioned above have been used for the custom VFS, there are still some areas for improvement. Mainly, some libraries could potentially be used instead of "reinventing the wheel" such as source code for handling of paths, configuration files, and logging. But it is hard to say how severe this issue is, as all of these implementations are elementary and straightforward. Furthermore, all of these implementations are written as such that they could be easily replaced with a library if needed.

Chapter 3

Design and Architecture

This chapter will outline the architecture and design decisions made during the development of the custom and extendable VFS. The first section will describe complications that arose from the use of FUSE in C++, while the second section will focus on the architecture of the VFS itself.

3.1 Using FUSE in C++

Properly incorporating FUSE in C++ presents many design challenges, primarily due to its interface design, which necessitates the use of a static wrapper for operations. The issue originates from FUSE expecting a C struct with pointers to functions, which poses a problem for non-static C++ methods. To solve this issue, two solutions can be considered: either writing C++ code without utilizing objects, rendering the use of C++ instead of C rather pointless, or implementing a singleton wrapper with only static methods.

As is evident, the latter solution was chosen. Fortunately, there was no need to start from scratch, as existing repositories already provide a solution to this problem. However, none of them were complete and up-to-date, and it was necessary to write or update portions of the code.

The foundation for this project was built upon the fusexx[18] repository, which was found most suitable for the needs of this thesis. But as was said, significant changes were made to the existing code, as it was not written in a modern C++ style and did not adhere to the project's design goals. The first step was therefore to refactor the code to use modern C++ features and resolve numerous clang-tidy warnings. Moreover, the repository was not updated for several years, and it was necessary to change some of the code's version-specific parts. And as a final touch, some methods were completely rewritten to improve readability or efficiency.

Needless to say, there was also a third, rather “hacky” solution, more like a sub-solution of the second one, which involved beside other implementation details using a data field in the FUSE context to store a pointer to the C++ object. Something similar is done in the fusepp[19] repository. But this approach did not seem to be the most elegant solution, and it was decided to use the singleton wrapper instead. Furthermore, the fusepp repository was not updated for even longer and would be harder to be used as a base for the project.

3.2 Architecture

Once the foundation was laid, the next step was to design a core VFS. It is important to note that this implementation was not the primary focus of this thesis as it was rather focused on providing modularity and extensibility. The result of this is that the entire core VFS is built on top of already present filesystem in the operating system. It is based on simply using a backing directory to store all the files and directories. However, such implementation details can be easily replaced, allowing for future improvements.

Still, there were some design decisions to be made regarding the object-oriented design of the VFS. And multiple approaches were contemplated: mainly one involving a single VFS class and another comprising separate File, Directory, and VFS classes. Although the latter approach adheres more closely to object-oriented principles, We have decided to go with the former, as it is better suited for use with FUSE. The reason for that is that it would still be needed to somehow bound all those classes into one, meaning at least the wrapper would have to be more complicated. Another concern regarding the second approach is that it would be harder to make the VFS extensible, but more on that in the following sections.

3.2.1 Modularity

The VFS is designed to be modular, allowing for the addition of new features without the need to modify the core implementation. To achieve this, the decorator pattern was selected as the primary design pattern for this project.

The decorator pattern is a structural design pattern that involves a set of decorator classes used to wrap concrete components. Decorator classes mirror the types of the components they decorate, sharing the same interface but adding or overriding behavior.

The benefit of this pattern is that it allows for the dynamic addition of new features to an object without modifying its implementation. Also, it is possible to combine multiple decorators to add multiple features to the same object at the same time. These properties make the decorator pattern a perfect fit for this project.

The decorator pattern is thus used to implement the VFS prototypes, such as encryption and versioning. For example, an `EncryptionVfs` class is implemented as a decorator, inherited from the `VfsDecorator` class. The `VfsDecorator` class, in turn, inherits from the base `CustomVfs` class. The decorator class wraps an instance of the base VFS class, allowing it to perform additional encryption-related tasks when reading or writing files.

Following code snippet 3.1 showcase the standard implementation of decorator pattern:

Listing 3.1 Decorator pattern implementation

```cpp
class VfsDecorator : public CustomVfs {
public:
    explicit VfsDecorator(CustomVfs& wrapped_vfs);
protected:
    CustomVfs& wrapped_vfs_;
};

class EncryptionVfs : public VfsDecorator {
public:
    explicit EncryptionVfs(CustomVfs& wrapped_vfs);
    int read(/*...*/) override;
    // ...
};
```

It is important to note that currently both inheritance and composition are used in the decorator pattern. The reason behind that is that it allows sparing some boilerplate code and make the code more readable as the CustomVfs is used more like a value type (if there was such thing in C++), meaning its behaviour is defined not by its reference but the values inside it. This allows me to simply copy construct the parent `CustomVfs` object with passed decorator.

The use of this pattern is then illustrated in the following code snippet 3.2, which shows how the VFS is initialized and used in the `main` function:

Listing 3.2 VFS initialization and usage

```cpp
void main(int argc, char* argv[]) {
    CustomVfs custom_vfs();
    VersioningVfs versioned_vfs(custom_vfs);
    EncryptionVfs enc_vers_vfs(versioned_vfs);
    enc_vers_vfs.main(argc, argv);
}
```

Modularity caveats

Unfortunate thing about this modularity design is complications with accessing the files other ways than using the parent decorator. To demonstrate the point, We will explore the complication with combing versioning and encryption. With that, you have to consider what do you want to perform first. In case of first versioning and then encryption, you have to encrypt all the concerned files. In case of first encrypting and then versioning, you lose the ability to version the diff-based way.

This order decision has to be done by the person creating the result vfs, but it is my responsibility to provide sufficient tools. For the described example, it means simply that each decorator has to be able to ask its parent for all concerned files.

3.2.2 Access to VFS features

Another crucial aspects of the architecture is determining the method by which users can access the added VFS features. In this implementation, hooks have been designed to integrate seamlessly with standard filesystem operations. By 'hook', we are referring to a file with a specific name that triggers a specific action when accessed. Additionally, a simple command-line interface (CLI) has been implemented to facilitate creating hooks for the VFS features.

The filesystem captures the mentioned operation, performs the requested action, and writes the result of the operation to the hook file. This approach allows users to interact with the VFS features without deviating from standard filesystem operations.

The CLI consists of two primary components: one for encryption and another for versioning. Users can execute commands related to these features via the CLI, making it a user-friendly and accessible interface for interacting with the VFS. Commands for encryption may include encryption key management and selecting encryption algorithms, while versioning commands may involve creating new versions, listing available versions, and reverting to a previous version.

Although the current implementation utilizes a command-line interface, the hooks can potentially be integrated into a graphical user interface (GUI) in the future. Incorporating the VFS features into a GUI would provide users with an even more intuitive and visually appealing interface for managing encryption and versioning. However, the development of a GUI was beyond the scope of this thesis.

Benefits of this approach include the ability to easily integrate the VFS features into existing applications, such as file managers. This is because the VFS features are accessed through standard filesystem operations, which are already supported by most applications. Additionally, the hooks allow for the seamless integration of new features, such as encryption and versioning, without the need to modify the core VFS implementation as can be seen in the following code example:

Listing 3.3 Example of hook implementation

```
int Vfs::write(const std::string &ppath, char *buffer,
               size_t size, off_t offset) {
    if (is_hook(path)) {
        return handle_hook(path, buffer, size, offset);
    }
    // Do normal write operation...
}
```

Each module has then its own implementation. It is also necessary to mention that all hooks share the same format, which is also the same as the format of hidden files. For this reason, the module needs to be able to ignore hooks that are not meant for it.

This format is `#<module_name>(<args>)#<affected_file>` and creation and parsing of this format is handled by the `PrefixParser` class.

3.2.3 Architectural overview

We will conclude this chapter by providing an architectural overview of the VFS. The architecture of our system is designed using object-oriented principles to ensure modularity, extensibility, and maintainability. The core of the system is built around the FuseWrapper class, which provides a base implementation for interacting with the FUSE library. By deriving from this base class, we can create custom virtual file systems with various features and functionality.

The `FuseWrapper` class, which is a parent class of `CustomVfs`, serves as a foundation for more specialized virtual file systems. By utilizing the decorator pattern, we implement additional functionality in separate classes, such as EncryptionVfs and VersioningVfs, which both inherit from the VfsDecorator class. This approach allows us to easily extend the system with new features while minimizing the impact on existing code.

The key components of the architecture, along with their relationships and key methods, are illustrated in Figure 3.1. The diagram shows the main classes, their inheritance relationships, and some of the methods that define their behavior.

Still, there are a lot of helper classes that are not shown in the diagram, but they are not important for the understanding of the architecture. These classes would typically include logging, handling paths or custom tool for prefixing file names with custom metadata.

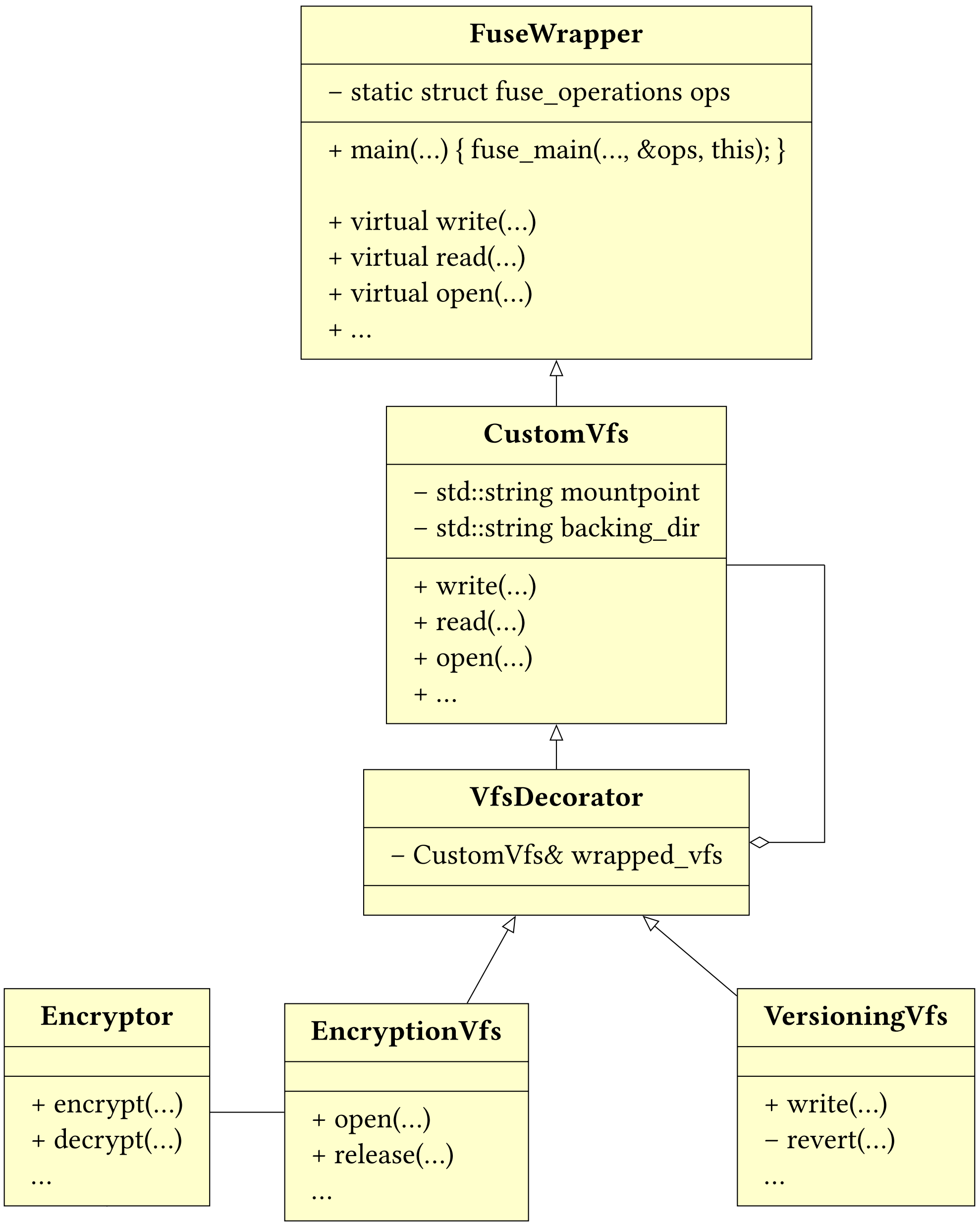

Figure 3.1 UML diagram of the system architecture.

Chapter 4

Preparation

With the architecture and design decisions established, the focus now shifts to the implementation of the custom and extendable VFS. This concise chapter will outline the tools and technologies employed throughout the development process, ensuring a solid foundation for the implementation phase.

4.1 Cross-platform build System

CMake[20], an open-source build system, has been selected for this project due to its cross-platform compatibility, user-friendliness, and extensive support for various platforms. CMake generates native build environments, such as Makefiles or project files for integrated development environments (IDEs), streamlining the development process.

4.1.1 Platform-Specific Challenges

Despite the cross-platform advantages provided by CMake, several platform-specific challenges arose during development. For instance, running applications on an M1 Mac proved difficult due to the lack of compatible binaries for the hardware. While building the applications from source was an option, there were no compelling reasons to do so for the purposes of this thesis. Furthermore, the osxfuse port had not received an update in five years, exacerbating the situation. On Windows, WinFsp[12] is required to ensure FUSE compatibility, introducing additional hurdles for seamless cross-platform development.

Unfortunately, the current implementation of the encryption module is not compatible with Apple Silicon. This was found out during the final stages of development, as the project was being tested on an M1 Mac. The reason for this is that libsodium does not properly support Apple Silicon yet. However, the rest of the project should work just fine, and the part of encryption module could be easily replaced with a different implementation.

4.2 Gitlab CI

The MFF faculty GitLab instance was utilized for this project[21]. GitLab CI/CD pipelines were employed to automate various tasks, such as checking code formatting with clang-format, building the project, running Google Test unit tests, and providing the ability to download build artifacts. This approach facilitated a more efficient and organized development process.

Regarding the clang-format check, the project was configured to use the Google C++ style guide[22] with a few minor modifications. The Google style guide was also used regarding the naming convention, although that unfortunately is not enforced by the clang-format tool.

4.2.1 Testing

For the testing purposes, Google Test was integrated into the GitLab CI/CD pipelines. This allowed for the execution of simple tests on every commit, such as creating directories and files. Apart from testing the added VFS functionality, the tests also check helper classes, such as the `Encryptor` class for encryption and decryption, or a custom library to allow prefixing file names with additional information.

To optimize the testing process, the tests were run on a shared mount-point that was cleared every time, although this approach is not ideal as it can make it harder to determine the root cause of a failure. Nonetheless, the primary purpose of these tests was to ensure that basic functionality was not broken with each commit, rather than to diagnose the cause of any failures that occurred.

Originally, the tests were supposed to mount and unmount the VFS, but this proved to be problematic. Debugging the tests was difficult in particular, since the `gdb` did not properly handle detached processes and would not stop on VFS breakpoints. As a result of that, mounting of the VFS is now done manually before running the tests and the mount-point needs to be passed as an argument.

Another fact that proved to be problematic was that FUSE requires a kernel module, as mentioned in section 4.3. Because of that, the tests needed to run on a custom runner with privileged access, but as it would be a complication to let them run all the time, CI/CD pipeline was limited to only run helper method tests.

In the end, the tests were not as extensive as could be, but they still provided a good baseline for the project.

4.3 Docker

Docker[23] was also incorporated into the project because of several reasons. First, it allowed for the creation of a consistent and reproducible development environment, ensuring that the project could be built and run across different platforms without issues.

Additionally, Docker provided two methods for debugging or even demoing the VFS: one that displays debug output in a terminal, and another that runs the VFS in the background. By integrating Docker into the project, the development and deployment process was simplified, allowing for a more efficient and effective implementation of the custom and extendable VFS.

To create the Docker environment for the project, a base Ubuntu 20.04 image was used. Of course, some dependencies included FUSE, GTest, Boost Program Option, and Libsodium were also installed. After that, the source code is copied into the container and the project is built.

Sadly, the Docker container in this project has one major drawback, as FUSE requires a kernel module, which Docker is not able to provide. Meaning that the Docker would still need access to the fuse device, complicating the process of running the container across different platforms. Besides that, it also needs to run with privileged access, which is not ideal.

Overall, the incorporation of Docker into the project provided a consistent and reproducible development environment, simplified the deployment process, and allowed for easier debugging and testing of the custom VFS.

Chapter 5

Implementation

5.1 Essential FUSE Operations

Initially, it was necessary to identify and incorporate various FUSE operations that need to be implemented and passed to `FuseWrapper` class. This section provides a brief overview of some of the essential operations, incorporating information from the Facile Engineering tutorial and IBM Developer article[24, 25]:

- **getattr**: Retrieves the metadata of a given path. This operation is always called before any other operation made on the filesystem. It is responsible for reading the file or directory attributes, such as size, access permissions, and timestamps.
- **readdir**: Lists the contents of a directory, filling a buffer with the structure of the accessed directory.
- **mknod**: Creates a non-directory node, such as a file.
- **unlink**: Deletes a file.
- **open**: Called when the system requests a file to be opened.
- **read**: Called when FUSE is reading data from an opened file, while filling the buffer with the content of those bytes.
- **write**: Writes data to an open file.
- **mkdir**: Creates a new directory.
- **rmdir**: Deletes an existing directory.
- **rename**: Renames a file or directory.

Besides these operations a full-featured filesystem might need operations such as `truncate`, `symlink`, `link`, `chmod`, `chown`, `utime`, `statfs`, `flush`, `release`, `fsync`, `setxattr`, `getxattr`, `listxattr`, and `removexattr`.

5.1.1 Operations for the Base VFS

The core of the implementation is provided by `CustomVfs` class, which is responsible for handling all file system operations. This particular class mainly redirects the operations to the appropriate system calls, which are then handled by the operating system. The exact way this is done is using a backing directory, which is mounted to the `.customvfs-mount` directory that is created when the filesystem is mounted. This particular directory is then used to store all the files and directories that are created by the user by simply forwarding the operations to the backing directory. Consequently, the debugging process is simplified as the user can easily access the backing directory and verify that the operations are being handled correctly.

Using the base VFS is also then as simple as creating an object and invoking the `main` method, as demonstrated in the following code snippet 5.1.

Listing 5.1 Main method of the `CustomVfs` class

```
void main(int argc, char* argv[]) {
    CustomVfs custom_vfs();
    custom_vfs.main(argc, argv);
}
```

The arguments passed to the `main` method are then simply forwarded to the wrapper and consequently to the FUSE library. This allows the user to specify the mount point and other options such as the debug mode or ignoring non-empty mount point.

5.2 Encryption

To implement the prototype module of encryption, the XChaCha20-Poly1305, described in the initial chapter 1.3.1, was chosen. Particularly, was used the implementation from libsodium library as discussed in 2.2.2.

There are two main approaches how to use the added encryption functionality. The first approach is to use a user-defined password, which is then employed to generate the encryption key. This method is not very practical, as the user has to manually lock and unlock files, since there is no mechanism to request the password when accessing a file.

The second approach involves using a key file. This method is more convenient, as users can store the key file in a secure location and access their files without additional steps. However, this brings up the question of when to encrypt and decrypt the files.

Encryption and decryption of files are easily performed upon opening and closing the file, respectively. In contrast, handling directories is more complicated, as there is no way to identify when the directory is being accessed. As a result, there is no encryption of the file names; instead solely the individual file contents are decrypted when the file is opened. This approach reduces unnecessary encryption and decryption operations and maintains relatively fast performance.

The encryption itself was done by asking the parent vfs for the proper file path and then using the libsodium library to encrypt or decrypt the file. Encryption of directories was then done using stack instead of recursion.

5.3 Versioning

The versioning module prototype is as of now rather limited since implementing a full-featured versioning system would be a significant undertaking. Namely, instead of using a diff-based approach for versioning, the entire file is stored in the versioning system as it simplifies the implementation. Diff-based approach could be then implemented in the future.

Similar to the encryption implementation, the read and write operations were wrapped to include versioning. The following code snippet shows a high-level overview of how to wrap the write operation with versioning functionality.

Listing 5.2 Write with versioning functionality

```
int VersVfs::write(const std::string &file_path,
                   char *buf, size_t size, off_t off) {
    store_previous_version(file_path);
    if (off > 0) {
        prepare_newfile_for_write(file_path);
    }
    wrapped_vfs_->write(file_path, buf, size, off);
}
```

That means the current version is always stored in a file with the real name, whereas the previous versions are suffixed with special symbol and their version number. With this approach, operations such as restore, delete, or list are relatively straightforward to implement. Restore just creates a new stored version with the current content of the file and renames the restored version to the real name. Delete operation then simply removes the file and all its versions. And listing operation just finds all the files with the real name and the special symbol and returns them.

Chapter 6

Evaluation

Once the implementation of the custom VFS is complete, it is necessary to evaluate the project. The evaluation is performed in terms of its functionality, usability, reliability, security, and performance.

6.1 Usability

The VFS itself is user-friendly, as the user only needs to specify the mount directory on the command line to start using it. The features are currently accessed through a set of CLI tools, which are slightly less user-friendly, but that could be improved over time. Despite the limited effort put into developing the CLI, it remains a functional and accessible interface.

In future iterations, the usability of the custom VFS could be further enhanced by providing a graphical user interface (GUI) to make it even more accessible for users who are not familiar with command-line interfaces. Moreover, more detailed documentation and tutorials could be provided to facilitate user onboarding and promote the adoption of the custom VFS.

6.2 Reliability and Security

The VFS implementation has undergone testing through unit tests using Google Test, ensuring good code coverage and the core features, such as encryption and communication with kernel code, rely on well-tested external libraries.

However, there is still room for improvement in the core VFS implementation to further enhance its reliability. In future iterations, a more extensive suite of tests could be developed, covering not only unit tests but also integration and stress tests. This would help ensure that the VFS behaves correctly under various scenarios and is resilient against potential failures.

In terms of security, the implemented encryption relies on the Libsodium library, which is a well-tested and widely used library for encryption.

6.3 Performance

Performance was not the primary focus of this project, as it is still a prototype. For that reason, there were no performance tests conducted. On the other hand, there are no known performance bottlenecks in the current implementation, as the VFS is built on top of the FUSE library, which is known for its high performance.

Future work could involve conducting performance tests to identify potential areas of improvement and optimize the VFS accordingly. This would be particularly important for applications with high I/O demands or large-scale deployments.

6.4 Feature overview

To showcase the features of the custom VFS; the following table compares created VFS to similar software. I have even included some file systems, as they have some of the same features.

It is important to note that the custom VFS is not a direct competitor to the other software, as it is a prototype and not a fully-fledged product. Besides that, the custom VFS has the ability to combine features easily, which could be used to create a more complete product.

Hopefully, most of the columns' names are self-explanatory, but there is a brief explanation of the more ambiguous ones just in case. The versioning *Dir* and *File* columns simply tell if the VFS can version directories and files. The *Type* means whether it stores full snapshots or only the differences between snapshots, whereas the *Auto* there means that the VFS stores versions automatically, without any user intervention, similarly *Auto unlock* in encryption describes whether the files are automatically unlocked on access.

	Win	**Linux**	**macOS**	**Files**	**Dirs**	**Type**	**Auto**
This VFS	–	✓	✓	✓	–	Full	✓
Btrfs	–	✓	–	✓	✓	Diff	✓
ZFS	–	✓	✓	✓	✓	Diff	✓
Time Machine	–	–	✓	✓	✓	Diff	✓
Shadow Copy	✓	–	–	✓	✓	Full	✓
Git	✓	✓	✓	✓	✓	Diff	–

Table 6.1 Versioning Tools Evaluation

	Win	**Linux**	**macOS**	**Password**	**Key**	**Auto Unlock**
This VFS	–	✓	✓	✓	✓	✓
VeraCrypt	✓	✓	✓	✓	✓	–
BitLocker	✓	–	–	✓	✓	✓
AxCrypt	✓	–	–	✓	–	✓
7-Zip	✓	✓	–	✓	–	–

Table 6.2 Encryption Tools Evaluation

Conclusion and Future Work

This thesis has successfully presented the design and implementation of a modular and easily extensible VFS in user space. Moreover, I have provided prototypes for encryption and versioning modules, which seamlessly integrate with the core functionality and require minimal user effort. The proposed VFS prototypes enable users to create snapshots for potential rollbacks and effortlessly encrypt individual files or entire directories using a password or key, offering temporary decryption when necessary. Importantly, the VFS transcends the limitations of similar programs, as it can be mounted on any file system, including network file systems, and its features can be easily layered on top of one another.

As we look ahead, there are several opportunities for future development and research to enhance the results of this thesis:

- **Optimization**: The current VFS implementation can be optimized for improved performance and storage efficiency.
- **GUI Integration**: Integrating the VFS features into a graphical user interface or even existing file managers will provide users with a more intuitive and visually appealing experience.
- **Additional Features**: Developers can create and incorporate new features or modules into the VFS, such as versioning based on diff, compression, deduplication, or support for various storage backends.
- **Real-world Deployment**: Refining and testing the VFS for real-world deployment will enable the evaluation of its performance, reliability, and usability in more diverse and demanding environments.
- **Windows Support**: The VFS can be ported to Windows, allowing users to access its features on a wider range of operating systems.

In conclusion, this thesis has substantiated the feasibility of creating an extensible Virtual File System with support for encryption and versioning. Future work can expand upon this foundation, developing a more robust, efficient, and feature-rich VFS suitable for a wide range of applications.

Bibliography

[1] Joanna Oja et al. “The Linux System Administrator’s Guide”. In: Linux Documentation Project, 2000. Chap. 5.10. URL: `https://tldp.org/LDP/sag/html/filesystems.html`.

[2] David A Rusling. “The Virtual File System”. In: *The Linux Kernel*. Internet: Self-published, 1997. URL: `http://www.science.unitn.it/~fiorella/guidelinux/tlk/node102.html#SECTION00112000000000000000`.

[3] Richard Gooch and Pekka Enberg. *Overview of the Linux Virtual File System*. URL: `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`.

[4] *XChaCha20Poly1305*. URL: `https://www.cryptopp.com/wiki/XChaCha20Poly1305`.

[5] FooSoft. *A simple versioning file system for Linux using FUSE*. URL: `https://github.com/FooSoft/vfs`.

[6] Cornell. “Wayback: A User-level Versioning File System for Linux”. In: *USENIX 2004 Annual Technical Conference*. URL: `https://www.usenix.org/legacy/events/usenix04/tech/freenix/cornell.html`.

[7] Stephan Müller and Sven Widmer. *Copy-on-Write Version Support for VFS under Linux*. URL: `https://osm.hpi.de/vvfs/`.

[8] Steve Huntley. *A versioning virtual filesystem*. URL: `https://wiki.tcl-lang.org/page/A+versioning+virtual+filesystem`.

[9] rmind. *Rvault repository*. URL: `https://github.com/rmind/rvault`.

[10] *FUSE repository*. URL: `https://github.com/libfuse/libfuse`.

[11] *FUSE for macOS*. URL: `https://osxfuse.github.io/`.

[12] *WinFsp repository*. URL: `https://github.com/billziss-gh/winfsp`.

[13] Sven. *FUSE flow-chart diagram*. 2007. URL: `https://commons.wikimedia.org/wiki/File:FUSE_structure.svg`.

[14] *Crypto++*. URL: `https://www.cryptopp.com/`.

[15] *libsodium*. URL: https://doc.libsodium.org/.

[16] *Google Test repository*. URL: https://github.com/google/googletest.

[17] *Boost Program Options*. URL: https://www.boost.org/doc/libs/1_82_0/doc/html/program_options.html.

[18] *Fuse++ repository*. URL: https://github.com/xloem/fusexx.

[19] *Fusepp repository*. URL: https://github.com/jachappell/Fusepp.

[20] *CMake*. URL: https://cmake.org/.

[21] Milan Vesely. *Thesis Repository*. 2023. URL: https://gitlab.mff.cuni.cz/teaching/theses/yaghob/vesely-milan/.

[22] Google. *Google C++ Style Guide*. URL: https://google.github.io/styleguide/cppguide.html.

[23] *Docker*. URL: https://www.docker.com/.

[24] *Develop your own filesystem with FUSE*. URL: https://developer.ibm.com/articles/l-fuse/.

[25] Lorenzo Fontana. *Writing a file system with FUSE*. URL: https://engineering.facile.it/blog/eng/write-filesystem-fuse/.

[26] *System and kernel extensions in macOS*. URL: https://support.apple.com/en-gb/guide/deployment/depa5fb8376f/web.

Appendix A

Installation

Dependencies

Before we even start, we need to clone the repository.

```
git clone git@gitlab.mff.cuni.cz:teaching/\
theses/yaghob/vesely-milan/source-code.git
```

...or in case you do not have access to MFF GitLab:

```
git clone https://github.com/vesmil/thesis-source
```

After that we need to install the dependencies.

Linux

On Debian-based Linux systems (such as Ubuntu), you may install these dependencies with apt:

```
sudo apt-get update && sudo apt-get install -y \
    clang cmake make g++ fuse libfuse-dev \
    libgtest-dev libsodium-dev \
    pkg-config libboost-program-options-dev
```

Other linux distributions may have different package names for the dependencies. But overall, it should be similar.

Docker

Even though there is a functioning Docker file, its main purpose was for testing. The Reason behind that is that FUSE requires kernel module, and Docker would have to reuse the host's kernel.

macOS

For macOS, you have to use Homebrew:

```
brew install --cask macfuse && \
brew install libsodium boost googletest cmake llvm
```

Even though macs are supported and tested for both Intel and Apple Silicon; the setup does include some extra steps. Namely, you need to enable support for kernel extensions. This is done through recovery mode, but I will not go into details as there are multiple guides on the internet on how to do that[26]. I may also note that in case you skip this step, the program will prompt you to do so upon running.

Unfortunately, the current implementation of encryption module is not compatible with Apple Silicon. This is due to the fact that libsodium does not support it yet. However, the rest of the project should work just fine.

Windows

The project is not supported on Windows as both WinFSP and Dokany have different API than FUSE. However, it could be easily ported to Windows by providing a different implementation of `FuseWrapper`. All the other used libraries should be platform independent. Nonetheless, it could be theoretically run on Windows using WSL.

Build

Once the environment is set up, you can build the project easily using CMake. From the root of the repository follow these steps:

```
mkdir build
cd build
```

Configure the project using CMake:

```
cmake ..
make
```

Optionally, you can save yourself some typing by adding the executables to your `PATH` variable or creating aliases such as:

```
  alias cvfs_encrypt="path/to/cvfs_encrypt"
  alias cvfs_version="path/to/cvfs_version"
  alias custom_vfs="path/to/customvfs_exec"
```

Appendix B

Usage

Mount the VFS

Mounting the VFS is rather simple, you just need to run the following command:

```
.\customvfs_exec /path/to/mountpoint
```

In case you want to use already existing directory in this VFS, you can use the `-b` or `--backing` option with a specified path to the directory. Note that the directory will be altered and used for helper files, hooks, and other stuff.

Another option you might use is `-f` or `--fuse-args` which will pass the following argument to the FUSE library. In case of multiple arguments, you have to use quotes.

Example

The following command will mount the file system to `/path/to/mountpoint` with backing directory set to `/path/to/backing/directory`. Besides that, it will also pass `-o nonempty -f -d` to the FUSE main, meaning it will run in foreground, print debug messages and allow mounting to non-empty directory.

```
.\customvfs_exec -b /path/to/backing/directory \
                 -f "-o nonempty -f -d" \
                 /path/to/mountpoint
```

Unmount

That is even simpler as you just have to run:

```
fusermount -u /path/to/mountpoint
# or in case of macOS - umount /path/to/mountpoint
```

Commands

Now when you are all set up, you can start using the file system. All standard operations stay the same as you are used to. But if you wish to use the custom features, you need to use one of the two provided tools from the `tools` directory.

Encryption

The `cvfs_encrypt` tool is used for encrypting files or directories. The following commands can be utilized:

- **Encrypt a file using a password (or a custom key):**
  `cvfs_encrypt --lock <file> (--key <key>)`
- **Encrypt a file with a default key:**
  `cvfs_encrypt --default-lock <file>`
- **Decrypt a file using a password (or a custom key):**
  `cvfs_encrypt --unlock <file> (--key <key>)`
- **Generate a new encryption key:**
  `cvfs_encrypt --generate <file>`
- **Set a default path for the encryption key:**
  `cvfs_encrypt --set-key-path <vfs> <file>`

Versioning

The `cvfs_version` tool is used for file versioning, and available are the following commands:

- **List all versions of a file:**
  `cvfs_version --list <file>`
- **Restore a file to a specific version:**
  `cvfs_version --restore <version> <file>`
- **Delete a specific version of a file:**
  `cvfs_version --delete <version> <file>`
- **Delete all versions of a file:**
  `cvfs_version --delete-all <file>`