FreyaFS

A virtual file system with Mix&Slice support

Michele Beretta

A.A. 2019/2020

Contents

1. Introduction

This thesis work describes the realisation of a virtual file system for GNU/Linux that supports the *Mix&Slice* encryption algorithm in a transparent way.

This new file system will take care of creation, modification and deletion of files and folders.

1.1 Context

The main reason for the realisation of this encryption-supporting file system is data privacy. There are often files (or, more generally, information) that, because of confidentiality, we want to make accessible only to few selected people. This is simple to realise thanks to encryption, which allows to protect the desired information by making them available only to people who have a specific key.

However, the usage and the access to this data should be as simple as possible, i.e., it should be like accessing non-encrypted files. This functionality is realised by a virtual file system that encrypts and decrypts files on demand, hiding these operations from the user.

1.2 The file system

1.2.1 Description

One of the most important functions of an operating system is taking care of information management on mass memory, which is any peripheral device other than central memory. The file system is the operating system module that implements this. In particular, the file system must [1]:

- provide a unique and permanent identification mechanism for *files*, so that they can exist and be accessible for a long time;
- provide *access methods* that allow addressing, reading and writing elementary blocks of information contained in the files;
- mask the physical characteristics of the storage units;
- implement access control mechanisms on files, both to ensure confidentiality of information and to avoid concurrency issues;
- guarantee the permanence and consistency of information even in the case of hardware or software malfunctions.

The main data structures a file system works with are file *descriptors*. Descriptors contain various properties of the file, the main ones being its name and its address, i.e., its position in mass memory. A set of descriptors is called *directory* (or folder). Since directories also have to be stored on disk permanently, they are handled by Unix and Unix-like systems like files: more precisely, in the Unix system a directory is a file that contains the list of names and addresses of the files contained within.

1.2.2 File operations

In order to access a file, its descriptor must be obtained from mass memory. However, this operation is computationally heavy in terms of the number of disk accesses. Hence, to solve this problem, the *open* primitive is introduced: before any operation on the file, its descriptor is retrieved and its content transferred to main memory. This descriptor is then used to read and write the file in an efficient way.

When no further operation needs to be performed, the inverse *close* primitive is executed: the descriptor image contained in main memory is copied to disk, so to update the information contained in the file.

We describe the opening of a file as the action of searching and copying in memory the descriptor of a file (and possibly its content), and the closing of a file as the release of the memory dedicated to the descriptor, resulting in the transfer of its content to disk.

1.2.3 Virtual file systems

A virtual file system is a software component that allows the operating system kernel to access the file system through the use of standard functions, independent from the actual file system (or the device used for storage).

Access to file stored on other computers via the network, or access to encrypted files without the user noticing the encryption, are example where the virtual file system comes into play and simplifies the interaction between the user and the computer.

1.2.4 FUSE - File system in Userspace

FUSE is a software system that allows the creation of virtual file systems on the Linux kernel [2]. The FUSE project consists of two parts:

1. the kernel module for communicating with the Linux kernel;

2. the *libfuse* user library that implements the communication between the virtual file system and the kernel module.

Therefore, FUSE allows the user to adopt directly the virtual file system mechanism on Linux and, in general, on Unix and Unix-like systems (including FreeBSD and macOS) thanks to a porting and rewriting process.

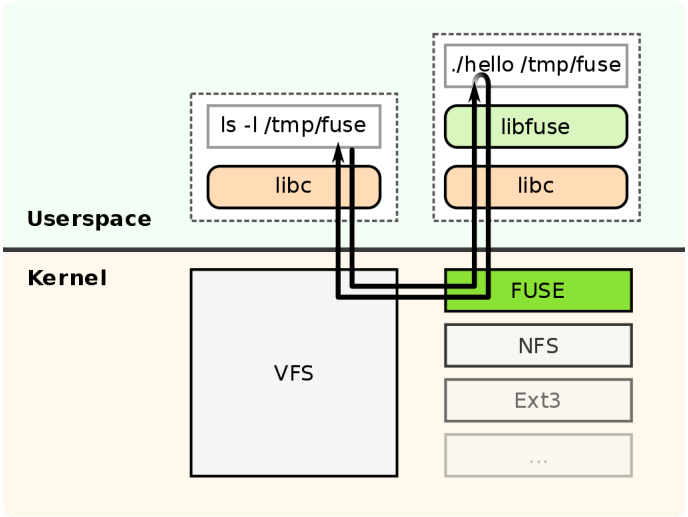

Figure 1.1: FUSE architecture

Figure 1.1 shows how FUSE interacts with the Linux kernel.

Every request made from user space (`ls -l /tmp/fuse`) is intercepted by the VFS kernel module[1] and is redirected to the FUSE kernel module. The request is then passed to the *libfuse* userspace library, which runs a specific program configured to handle a request (`./hello`). The handler response is then sent back to FUSE, and finally forwarded to the initial program in userspace.

[1] *Virtual File System* [3], or *Virtual File System Switch*, a software layer of the Linux kernel that manages the interaction between userspace and the "real" file system.

1.3 Cryptography

1.3.1 Description and usage

Cryptography is an algorithmic technique that allows to represent a message in a obscured way, in order to make it incomprehensible to anyone who is not permitted to read its content. The encrypted message is called *cryptogram* and the used methods are called *encryption techniques*.

Cryptography is based on an algorithm and an encryption key. The algorithm is often public, while the encryption key is the information that guarantee the confidentiality of the message: without the key, it is not possible to access the encrypted information.

This technique is used in many IT fields for various purposes, such as to guarantee privacy in a communication between two or more computers on the internet. It can also be used on files containing sensitive data or that must be kept private for any reason.

1.3.2 Symmetric and asymmetric encryption techniques

Encryption algorithms can be symmetric or asymmetric.

Symmetric algorithms make use of a single key, used for both encryption and decryption. Although being fast, these algorithms have some security problems that regards mainly the presence itself of a single key: this key has to be shared among anyone that would access the encrypted information, and if another person obtains the key then he or she is able to read and edit the messages' content.

Asymmetric algorithms, on the other hand, employ two different keys, one to encrypt and one to decrypt. For example, a key can be made public and used to encrypt messages that only the person with the other key can read. The main drawback of this method is its slow speed. For this reason, if the data to encrypt

is large, it is usually best to use both encryption methods: firstly, asymmetric encryption is used to share a key, which can then be used with a symmetric algorithm to reduce computational overhead.

1.4 Encrypted file systems

An encrypted file system is a special type of file system capable of handling encrypted files. In order for a user to access files, it is necessary to provide the file system the key (or keys) used for encryption. The benefits of an encrypted file system are essentially the following:

- *confidentiality*, as the files are encrypted and not accessible by users who do not have access to the encryption key;
- *transparency*, as the encryption and decryption operations are not visible to the authorised user, who browses the through files as he or she would in a non-encrypted file system.

While accessing the stored data, the file system encrypts and decrypts files as necessary, keeping only the encrypted data on mass memory. In this way, an adversary who gets access to the physical storage medium would not be able to read the content of the files.

1.5 State of art

1.5.1 EncFS

EncFS (*Encrypted File System*) [4] is an open-source library released under the LGPL license, which implements a virtual file system with FUSE. EncFS encrypts the files individually and translates all requests for the virtual file system to equivalent calls to the operating system.

In particular, it has some features that distinguish it from other virtual file systems:

- it has a "reversed mode", which means it can provide an encrypted view of directories that are not natively encrypted;
- is relatively fast on traditional hard disks;
- it also works with network file systems.

1.5.2 AONT - All-Or-Nothing Transform

Also known as *all-or-nothing-protocol*, AONT [5] is an encryption mode that allows to read the content of a file only if the whole file is accessible. Hence, if even a single part of the encrypted file is missing, the whole file cannot be accessed.

In particular, a transformation f is called *all-or-nothing transform* if:

1. it is reversible, i.e., it is possible to obtain the input if the corresponding output is available;
2. both the transformation itself and its inverse are computationally efficient;
3. it is not feasible to obtain any part of the input without even a single part of the output.

1.5.3 Mix&Slice

Mix&Slice [6] is an approach that enables a user to enforce, manage and revoke access rights to shared encrypted resources. In a similar way to what happens with an AONT, *Mix&Slice* makes impossible to obtain a resource if even a single part of it is not available. However, AONT techniques cannot be applied in situations where the users that have their access revoked still know the encryption keys, and may even have a local copy of the encrypted resource.

The *Mix&Slice* approach requires to partition the resource in various *macro-blocks*, all of the same size, and then to apply its two basic techniques:

1. *Mixing*: the content of every macro-block is processed by different cycles that "mix" the block's bit. In this way, at the end of the process, every output bit depends from every input bit.

2. *Slicing*: the macro-blocks are hence divided and grouped into *fragments*. These fragments are at the heart of the entire process, since obtaining the original resource is impossible if any of them goes missing.

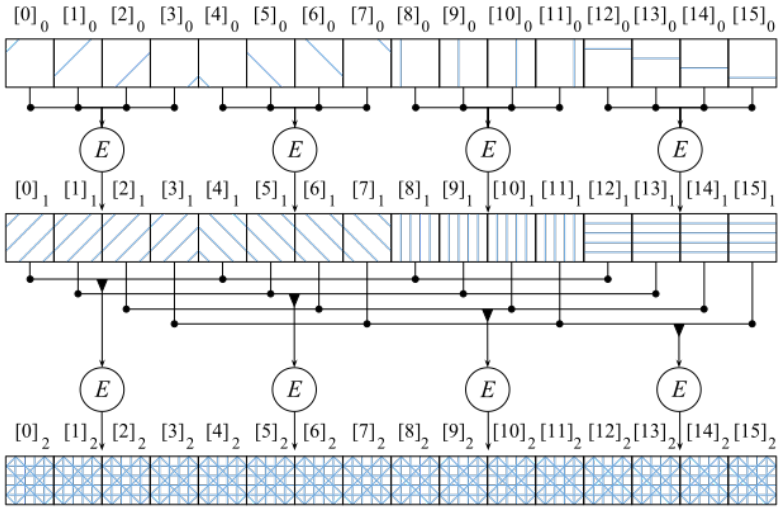

Figure 1.2: A *Mix&Slice* example

2. Project development

2.1 Design and tools

The virtual file system was implemented with the Python language for the Ubuntu operating system. Among the libraries used in the project, the two most important are `fusepy` and `aesmix`.

The `fusepy` [7] is a Python module that offers bindings to the C implementation of FUSE for Linux and macOS systems.

The `aesmix` [8] library provides the implementation of the *Mix&Slice* algorithm to encrypt and decrypt the files used during development. This library offers the Python `mixslice` wrapper, which permits to encrypt and decrypt a file using *Mix&Slice*.

The Python wrapper accepts a file as input and outputs three elements:

- a directory containing the encrypted fragments produced by *Mix&Slice*;
- a `.public` metadata file, which contains the public key;
- a `.private` metadata file, which contains the private key.

The main methods of the `aesmix` library that were used for this work belong to the `MixSlice` class and are the following:

- `load_from_file`, which loads the files needed for decryption into memory;

- `save_to_files`, which saves the various fragments and metadata files to disk;
- `encrypt`, to encrypt a file;
- `decrypt`, to decrypt a file.

2.2 Business logic and usage

2.2.1 Mountpoint, data folder and metadata folder

The virtual file system needs three directories to be mounted and used:

1. a directory indicating the mountpoint, that is, the point where the virtual file system will be mounted;
2. a directory containing the encrypted data, possibly organised in subdirectories;
3. a directory that contains all the metadata files (`.public`, `.private`, and `.finfo`).

The directories for the encrypted files and metadata files are separate in order to allow generality in the use of the file system. The structure of the two must be the same to ensure uniqueness in the association between encrypted files and the corresponding metadata, as files with the same names can exist in several points of the file system.

For example, if an encrypted file has the path `dir/image.png`, then its metadata must be `dir/image.png.public`, `dir/image.png.private`, and `dir/image.png.finfo`.

Since the `aesmix` utility produces a directory containing the fragments, to distinguish between directories and encrypted files, the file system refers to the existence of a metadata file: for a given path, if the corresponding metadata file exists, then the directory is treated as if it were an encrypted file, otherwise its management is left to the operating system.

2.2.2 Management of encrypted files

The file system must show the directories containing the encrypted fragments as files. To make this happen, it is necessary to change the information found in the file descriptor seen by the system. Since directories are files, their descriptor is retrieved in an analogous way as it is done for files (see Section 1.2).

At the request of the descriptor by the system, the following properties are modified [9]:

- `st_mode`: contains permissions and file type. To let encrypted directories appear as files, the `S_IFREG` flag (which identifies a file) is set, and the `S_IFDIR` flag (which identifies a directory) is removed. The other permissions remain unchanged.
- `st_nlink`: the number of hard links of the file. This is equal to 2 for a directory, hence it is set to 1 to make encrypted directories appear as files.
- `st_size`: the size of the file in bytes.

In particular, in order to find the real size of a file, it is necessary to decrypt it since the operating system uses the `st_size` field to know how many bytes to read from disk[1]. In a first iteration of the project, the total size of the directory was used (i.e., the sum of the dimensions of the single fragments contained in it), but this caused problems when accessing the file content, especially when using a GUI. When the file was opened, the operating system could read fewer bytes than necessary, hence considering the file as corrupt, or read more bytes than necessary, leading to crashed and forced closures of applications.

However, decrypting a file every time a descriptor request is made to determine its actual size can have a significant performance impact. For this reason, we introduced

[1]This behaviour is not common throughout the whole system: it is used by the GUI but not by the command line. When using a terminal, even with a wrong `st_size`, it is possible to read the content of a file with `cat` and to write it with a combination of `echo` and `>`.

a new metadata file type, with the extension `.finfo`, which contains information about the size of the file in the virtual file system. This size is updated at each write operation, only if it has actually changed. In this way, although with an initial overhead when writing to files, the file system is significantly more responsive in its normal use.

What is described in this Section can only be done if it is possible to open and decrypt the file and if there are any metadata files associated with it. The directories that do not contain the encrypted fragments are therefore processed directly by the system, which manages them without any intervention by the virtual file system.

2.2.3 Reading, writing, and creating files

In order to read and write a file, it must be first opened. That is, it must be decrypted and its content transferred in memory.

The *reading* takes place in blocks on the content in memory, through an offset and a total number of bytes to read. *Writing*, on the other hand, takes place via a buffer containing the data to be saved and an offset indicating the position of the byte to start writing from. It is also possible to *truncate* a file, or to limit its length to a certain value. When opening a file, a new value must be assigned to `st_atime`, which indicates the time instant of the last access to the file [9].

Writing has the particularity of not being carried out directly on mass memory, but it is done on the representation of the file in main memory. The file is encrypted on disk at a later time, when the file is flushed. For the sake of efficiency, flushing is done only if changes have been made to the file content, that is, if at least one write operation has been executed. In this way, if a file has not been modified since it was opened, no unnecessary encryptions are made, improving the efficiency of read-only operations. Writing also requires to change the value of `st_mtime`, which indicates the instant (Unix timestamp) of the last modification [9] of the file content.

Creating a file on disk corresponds to creating an area in memory with the

necessary content, even empty[2], and "force flushing" the file. Moreover, file creation enables the ability to copy and duplicate files, as copying is essentially a creation followed by one or more write operations.

2.2.4 Renaming and deleting files

Renaming an encrypted file also renames the associated metadata files. If done from a graphical interface, the operation does not allow a user to rename to an already existing file, while from the command line the renaming is indistinguishable from moving a file. The implementation of a correct renaming operation therefore enables both the ability to move files and the ability to overwrite files in the virtual file system.

Permanent *deletion* of a file also involves deleting the metadata files. Moving a file to the trash bin, on the other hand, is actually a move operation, hence supported with renaming. If on the virtual volume created by FUSE there does not exist a directory used as a trash bin, this will be created. Even the files in the trash bin will be encrypted.

Neither the deletion nor the renaming operations need to open and close files, therefore they are particularly efficient due to the absence of the encryption and decryption steps. Both the operations, however, need to change the information kept in memory regarding various file characteristics, to avoid referring to data which is no longer valid (see the management of `st_size` in Section 2.2.2 and the implementation of `EncFilesManager` in Section 2.3.3). It can also happen that a file is renamed while it is open.

[2]When creating an already existing or an already opened file, its content must be kept and not overwritten.

2.2.5 Concurrency in opening a file

A file can be opened at the same time by multiple applications. When closing any one of these, the file system prevents a file from being closed if it is still in use by other programs, in order to avoid anomalies or data corruption.

This translates into the need to maintain a counter for each open file in memory, which is increased at each opening and decreased at each closure of the file. The file is eventually closed, releasing all the dedicated memory areas, when this counter reaches the value 0. It remains possible to flush the file multiple times to update its content on disk.

The various applications take care themselves of updating the content of the file if this has changed on disk, or they offer the user the option to reload it. This happens thanks to the update of `st_mtime` during write operations.

2.2.6 Directory management

To keep the exact same structure between the directory with the encrypted files and the directory with metadata files, the virtual file system must take care of replicating all directory operations (create, edit and delete) between the two locations on disk.

This management does not present particular difficulties, except for the rename operation, as it is carried on directories by the operating system through the same primitive used for files, `rename`. This is different from other operations: for example, to remove a file the `unlink` method is used, whereas to remove a directory the `rmdir` operation is used. In the case of renaming, it is therefore necessary to differentiate between an encrypted file and a directory through the `st_mode` property of the appropriately modified descriptor, as explained in Section 2.2.2.

2.2.7 Multithreading

The `libfuse` Linux library permits to mount virtual file systems in two ways:

1. *single-thread*, where the entire file system runs on only one thread inside one process;

2. *multi-thread*, in which the various operations (writing, reading, etc.) are assigned each time to different threads.

The single-thread mode is less powerful from a performance point of view, but it is simpler and more suitable for development, since it requires no checking of concurrency conflicts.

The multi-thread mode instead offers better performance, as it permits to parallelize file operations whenever possible, such as in the case of reading a large amount of data. Specifically, the `libfuse` library protects the critical sections of the code from concurrent access through semaphores, implemented according to the POSIX standard, through the `pthread.h` library.

FreyaFS maintains these two paradigms, operating by default on a single thread. Although `libfuse` already takes care of the protection of critical sections at a low level, it is still necessary to protect access to shared memory areas, mainly represented by the content of open files and counters of applications that have opened a given file.

The file system handles the action of writing and reading of open files differently. More readings can take place in parallel in order to increase performance, while writes must not interfere with each other and must have exclusive access to the resource. This behaviour is accomplished through the use of *read locks* and *write locks*. The virtual file system keeps for each open file a counter of how many threads are reading the resource: this is incremented every time a thread starts the read operation and decremented when it ends. When a thread requires write access to the file, it is put on hold if there are any readers present and is notified when there are no more. If instead the resource is free, exclusive access is guaranteed through a semaphore.

2.3 Implementation

The core of the project is made up of five files:

- `main.py`, which handles command line arguments and mounts the file system;
- `freyafs.py`, which manages the operations of the file system in general and communicates with the operating system;
- `encfilesmanager.py`, which implements the various input/output operations on files;
- `encfilesinfo.py`, which manages the `.finfo` metadata files;
- `filebytecontent.py`, which manages concurrent accesses to files.

Figure 2.1 shows the static structure of the code and how modules are related to each other. An arrow from module A to module B indicates that B uses module A. In the "external" modules we have the `aesmix` library, which implements *Mix&Slice*, and `fusepy`, which provides the implementation of FUSE for the Python language. The "internal" modules are those managing the encrypted files and realise the logic of the virtual file system. Finally, `main` only handles the interface with the user from the command line and invokes `fusepy` to mount the file system.

2.3.1 User interaction

User interaction is managed by the `main.py` module, which is responsible to process the parameters passed on the command line and to mount the file system. The available parameters are shown in Table 2.1.

2.3.2 FreyaFS

The `freyafs` module implements FUSE's methods for managing the virtual file system. In particular, the most important part of the module is the `FreyaFS` class,

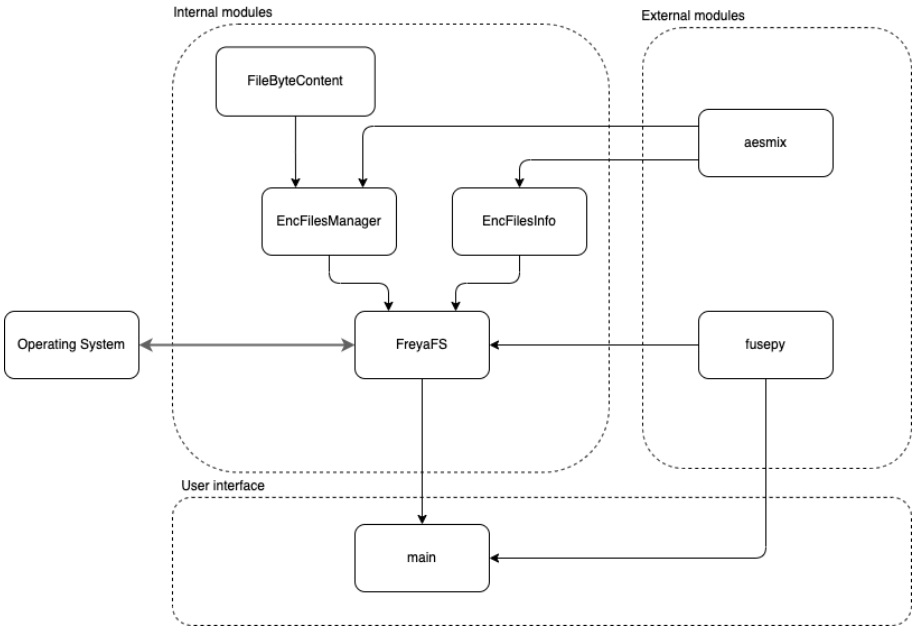

Figure 2.1: Code structure

Parameter	Description
`MOUNT`	The mountpoint, the path where the file system will be mounted
`--data DATA`	The path to the directory containing the encrypted files
`--metadata METADATA`	The path to the directory containing the metadata files
`--multithread`	Flag indicating whether to mount the file system in multi-thread mode (default to `False`)

Table 2.1: `main.py` parameters

that extends the FUSE `Operations` class and overrides some methods. A lot of the methods of the class simply call the operating system, as they do not require encryption support (such as the test to access a file). Other methods are redefined to support the file encryption and decryption, performed by the `encfilesmanager` module.

What follows is an extract of the `getattr` method, used to obtain information

about files and folders. The code visible here focuses on the modification of the descriptor's information about the folders containing the encrypted fragments.

```
1   if full_path not in self.enc_info:
2     public_metadata, _, finfo = self._metadata_names(path)
3     self.enc_info[full_path] = EncFilesInfo(full_path, public_metadata,
          finfo)
4
5   return {
6       'st_mode': stat.S_IFREG | (st.st_mode & ~stat.S_IFDIR),
7       'st_nlink': 1,
8       'st_atime': st.st_atime,
9       'st_ctime': st.st_ctime,
10      'st_gid': st.st_gid,
11      'st_mtime': st.st_mtime,
12      'st_size': self.enc_info[full_path].size,
13      'st_uid': st.st_uid
14  }
```

The main duties of this code snippet are two: to calculate the size of the encrypted file, and to change the flags so that the folder containing the fragments appears as a single file.

The size calculation is done through instantiating the `EncFilesInfo` class and only if not already done before (in order to improve performance, rows from 1 to 3). The `self.enc_info` property is a dictionary containing the various `EncFilesInfo` instances, one for every encrypted file found in the system.

The flag change is done so that the same access rights are kept, as visible in row 6, represented by `st.st_mode`, which is the set of permissions of the original folder. The `stat.S_IFREG` is set to 1 through a bit-by-bit `OR` operation, while to remove the `stat.S_IFDIR` it is necessary to do a bit-by-bit `AND` with the bit-by-bit negation of `stat.S_IFDIR`.

The following snippet shows an extract of the `unlink` method, used to remove files.

```
1    full_path = self._full_path(path)
2    public_metadata, private_metadata, finfo = self._metadata_names(path)
3
4    os.unlink(public_metadata)
5    os.unlink(private_metadata)
6    if os.path.isfile(finfo):
7        os.unlink(finfo)
8
9    if full_path in self.enc_info:
10       del self.enc_info[full_path]
11
12   shutil.rmtree(full_path)
```

In this code there are only calls to the operating system primitives, without opening nor closing any file. The metadata files are also deleted, and the associated memory areas are freed (row 10). It is also necessary to check whether the `.finfo` file exists (row 6) because, unlike `.public` and `.private` files, it may not exists as it is created on the first `getattr` call.

The `rename` methods manages the renaming of a file, which is more complex, since it is the same operation for both files and folders. Moreover, due to the encrypted files being actually folders, if one wants to rename a file to another already existing file[3] it is necessary to delete the latter (rows 6-7), in order to prevent the system from recognising it as a folder and moving the former "into" the latter. Lastly, the in-memory paths relative to the file and its metadata files are renamed (rows 19-25).

When dealing with "regular" folders, i.e., those not containing encrypted fragments, the rename is left to the operating system and replicated both in the encrypted files path and in the metadata files path (the `else` branch).

[3]This is prohibited when using a GUI, but allowed through the command line `mv` command.

```
1    full_old_path = self._full_path(old)
2    full_new_path = self._full_path(new)
3
4    if self._is_file(old):
5        if self._is_file(new):
6            self.unlink(new)
7
8        old_public_metadata, old_private_metadata, old_finfo = self.
             _metadata_names(old)
9        new_public_metadata, new_private_metadata, new_finfo = self.
             _metadata_names(new)
10
11       os.rename(old_public_metadata, new_public_metadata)
12       os.rename(old_private_metadata, new_private_metadata)
13       if os.path.isfile(old_finfo):
14           os.rename(old_finfo, new_finfo)
15
16       os.rename(full_old_path, full_new_path)
17
18       if full_old_path in self.enc_files:
19           self.enc_files.rename(full_old_path, full_new_path)
20
21       if full_old_path in self.enc_info:
22           self.enc_info[full_old_path].rename(full_new_path,
                 new_public_metadata, new_finfo)
23           self.enc_info[full_new_path] = self.enc_info[full_old_path]
24           del self.enc_info[full_old_path]
25   else:
26       old_metadata_path = self._metadata_full_path(old)
27       new_metadata_path = self._metadata_full_path(new)
28       os.rename(old_metadata_path, new_metadata_path)
29       os.rename(full_old_path, full_new_path)
```

All methods that act on file input and output all act as a proxy to the `EncFilesManager` methods and, if necessary, update the relative size managed by `EncFilesInfo`. As an example, the following snippet shows the `write` method.

```
1    def write(self, path, buf, offset, fh):
2        full_path = self._full_path(path)
3        if full_path in self.enc_files:
4            bytes_written = self.enc_files.write_bytes(full_path, buf,
                 offset)
5            self._update_enc_file_size(full_path)
6            return bytes_written
7
8        os.lseek(fh, offset, os.SEEK_SET)
9        return os.write(fh, buf)
```

In this two extracts, `self.enc_files` is an instance of the `EncFilesManager` class, which contains all the contents of the currently opened files.

The `open`, `create`, `truncate`, `read`, `flush`, and `release` methods all have a similar structure. If an open file is not among those encrypted, then the file is managed by the operating system as it normally would.

In conclusion, Table 2.2 summarises the methods used by *FreyaFS* that were changed with respect to the basic case, where each method simply invokes the corresponding primitives of the operating system.

2.3.3 EncFilesManager

This module defines the class with the same name managing the input and output on encrypted files. Specifically, the class is instantiated only once and contains information on all files (encrypted with *Mix&Slice*) currently open, identifiable by their path in the real file system. It makes use of a counter for each open file that indicates how many applications are currently using it: this way, the class avoids to accidentally close a file if only some of these applications have released it while others have it still open.

The offered methods are the following:

- `open`: opens a file, decrypting its content and keeping it in memory;

Method	Goal
`getattr`	Retrieve the properties of an encrypted file or directory
`readdir`	Retrieve the content of a given directory, including the `.` and `..` entries, and ignoring metadata files
`mkdir`	Create a directory, both in the encrypted data path and in the metadata path
`rmdir`	Remove a directory, both in the encrypted data path and in the metadata path
`unlink`	Remove an encrypted file, also deleting the corresponding metadata files
`rename`	Rename an encrypted file (and corresponding metadata files) or a directory
`utimens`	Update the instants of the last update and the last editing for encrypted files and directories
`open`	Open an encrypted file
`create`	Create an encrypted file
`read`	Read a certain number of bytes from an encrypted file
`write`	Write a certain number of bytes from an encrypted file
`truncate`	Truncate the content of an open encrypted file at a given length
`flush`	Force writes made to an encrypted file from memory to disk
`release`	Release an open encrypted file

Table 2.2: *FreyaFS*'s methods

- `create`: creates an empty file, i.e., it reserves empty space in memory for the content of the file (only if not already open) and flushes it;

- `read_bytes`: reads a certain number of bytes from an open file;

- `write_bytes`: overwrites part of the content of an open file with data contained in a buffer;

- `truncate_bytes`: truncates the content of an open file to a certain length;
- `flush`: writes the content of an open file to disk, encrypting it;
- `release`: releases the memory areas dedicated to a file, closing it;
- `cur_size`: gets the current length of the content of an open file;
- `rename`: renames the various paths referring to an open file.

It is essential to have a `rename` method, because a file can be renamed even if open: in this case, to avoid referencing files that no longer exist and cause the file system to behave incorrectly, it is necessary to update all the paths related to the file in question.

The `EncFilesManager` class also tracks the change status of a file using Boolean flags. That way, if a `flush` is requested but there are no changes to the file, the encryption operation can be skipped, improving performance and avoiding wasted time.

Finally, the class also handles mutual exclusion for the multi-thread mode of the virtual file system, via a `Lock` semaphore of the Python `threading` module (see Section 2.3.6). This semaphore guarantees exclusive access to structures containing the content of open files and open file counters.

The following code snippet shows the implementation of the `open` method:

```
1   def open(self, path, public_metafile_path, private_metafile_path,
        mtime):
2       with LOCK:
3           if path in self.open_files:
4               self.open_counters[path] += 1
5               return
6
7           self.public_metafiles[path] = public_metafile_path
8           self.private_metafiles[path] = private_metafile_path
9
10          self.open_files[path] = FileByteContent(self._decrypt(path))
```

```
11              self.open_counters[path] = 1
12
13          self.touched_files[path] = False
14          self.atimes[path] = int(time())
15          self.mtimes[path] = mtime
```

The method also takes care of updating the "last accessed" timestamp for a file via `self.atimes`, and the "last changed" timestamp via `self.mtimes`. Between the two, only the latter is updated when writing the file, while both of them are always updated when flushing, even without any change to the file content.

2.3.4 EncFilesInfo

The `encfilesinfo` module defines the `EncFilesInfo` class, which is responsible of the management of information about the encrypted files that is not contained in the `.private` and `.public` metadata files. Its main job is to interface `freyafs` with `.finfo` files, which contain information serialised in the JSON format. Unlike the `EncFilesManager` class, the `EncFilesInfo` class is not aware of all the files managed by the virtual file system, but it only manages the information of a single file specified at initialisation.

Specifically, the module supports the management of the length in bytes of the content of the encrypted file, which can be obtained and set using the property `size`.

When the class is instantiated, the size of the desired encrypted file is calculated and is stored in the `size` property and also in the `.finfo` file (which, if not present, is created). When this property is updated, the value contained in the metadata file is also updated, but only if the value is different from the previous one. This avoids opening a file, parsing the JSON contained in it, updating the property and then serialising the JSON produced, in case these operations are not strictly necessary.

```
1   @property
2   def size(self):
3       if self._size is None:
4           self._size = size_decrypt(self._path, self._public_metadata)
5           self._update_finfo()
6       return self._size
7
8   @size.setter
9   def size(self, value):
10      if self._size == value:
11          return
12      self._size = value
13      self._update_finfo()
```

The class also provides a `rename` method, used to rename the path to the file containing this information.

2.3.5 FileByteContent

This module deals with reading and writing the content of a file opened in a mutually exclusive way. To ensure this, the `FileByteContent` class has the following properties:

- `_text`, which represents the content of the file as a string of bytes;
- `_readers`, which indicates the number of current readers in a given moment;
- `_cond`, of type `threading.Condition`, which is used as a semaphore.

The `FileByteContent` class uses a semaphore for each open file In this way, parallel reads and writes to different files are possible, but not on the same file.

As an example, the following are some of the methods of the class.

```
1   def read_bytes(self, offset, length):
2       self._r_acquire()
3       text = self._text[offset:offset + length]
```

```
4         self._r_release()
5         return text
6
7     def write_bytes(self, buf, offset):
8         self._w_acquire()
9         bytes_written = len(buf)
10        new_text = self._text[:offset] + buf + self._text[offset+
              bytes_written:]
11        self._text = new_text
12        self._w_release()
13        return bytes_written
14
15    def truncate(self, length):
16        self._w_acquire()
17        self._text = self._text[:length]
18        self._w_release()
```

In this snippet, methods `_r_acquire()` and `_r_release()` permit respectively to acquire and release the read lock, which can be shared between multiple threads, while methods `_w_acquire()` and `_w_release()` are the equivalent for the write lock, which guarantees exclusive access to the file content.

The acquisition of a lock for reading increases the `_readers` variable, which is decremented at every release. When value 0 is reached, any thread waiting for a write lock is notified.

When asking for a write lock on a file, the thread waits as long as there are readers, i.e., if `_readers` is positive. When all the readers have released the resource, the lock is acquired and then released at the end of the write operation.

The methods for the acquisition and release of read and write locks are as follows. For more details on the `_cond` property, see Section 2.3.6.

```
1   def _r_acquire(self):
2       self._cond.acquire()
3       try:
4           self._readers += 1
5       finally:
6           self._cond.release()
7
8   def _r_release(self):
9       self._cond.acquire()
10      try:
11          self._readers -= 1
12          if self._readers == 0:
13              self._cond.notify_all()
14      finally:
15          self._cond.release()
16
17  def _w_acquire(self):
18      self._cond.acquire()
19      while self._readers > 0:
20          self._cond.wait()
21
22  def _w_release(self):
23      self._cond.release()
```

2.3.6 The `threading` library

The `threading` library [10] is a built-in Python library that supports the management of high-level threads and provides tools for creating and using semaphores.

The classes used for *read locks* and *write locks* in the `FileByteContent` class (Section 2.3.5) are:

- `threading.Lock`, which offers the implementation of the concept of mutex, a mutually exclusive semaphore;
- `threading.Condition`, which represents a condition associated with a partic-

ular `Lock` instance.

In the code of the `FileByteContent` class there is the `_cond` property of type `Condition`, which provides the following methods:

- `acquire`: acquires the underlying lock;
- `release`: releases the underlying lock;
- `wait`: suspends the calling thread and releases the underlying lock, which must have previously been acquired by the thread;
- `notify_all`: wakes up all queued threads, which leave the wait queue only when they gain control over the lock.

The semaphores provided by `Lock` are objects that can be in a state "locked" or "free". When the lock is in the free state, it can be acquired through the `acquire` method, which puts it in the locked state. Instead, if `acquire` is invoked while the semaphore is locked, the calling thread pauses until a `release` call frees the mutex.

Objects instantiated by `Condition` are more flexible than simple mutexes, but they are always associated with a semaphore. Through these objects it is possible to suspend a thread with the `wait` method, placing it in a wait queue and causing the underlying lock to be released, and awaken it with `notify` or `notify_all`. Upon exiting the queue, the thread acquires the lock again and then continues in its execution. In this way, it is possible to avoid *busy waits*, that is, situations in which the thread continuously checks the state of the semaphore, occupying computational resources that could be used by other unblocked threads.

Finally, both classes support the *context management protocol*, that is, the use of the Python `with` construct. When the process or thread enters a block delimited by `with`, the `acquire` method of the lock or condition is called, while at the exit the `release` method is invoked. This construct is visible in the implementation of the `EncFilesManager` class in Section 2.3.3.

3. Testing and analysis

3.1 System

The virtual file system was developed and tested on the following system:

- Ubuntu 20.04 LTS Focal Fossa as the operating system;
- a Intel Core i5-7200U quad-core 2.5 GHz CPU;
- 8 GB RAM;
- 120 GB solid state drive.

3.2 Multiple opening of the same file

The opening of the same file with multiple applications has been tested on the following file formats:

1. a `.txt` text file, with *gedit* and *Visual Studio Code*;
2. a `.md` text file, with *gedit* and *Typora*;
3. a `.jpg` binary file, with *Image Viewer* and *Firefox*;
4. a `.pdf` binary file, with *Document Viewer* and *Xournal*.

For `.txt` files, both the applications open the file and immediately close it again after reading the entire content. Only when the save operation is invoked, the file is opened again to make the necessary updates. This reduces the possibility of running into concurrency problems that may arise when two applications keep the same file open at the same time. The same exact situation occurs with `.md` files and `.jpg` files. Applications react differently to changes to the content on disk of an opened file: for example, *gedit* always proposes to the user to reload the entire file, while *Typora* and *Visual Studio Code* transparently update the file and notify the user only if there are pending changes created by the user.

The applications used to test `.pdf` files behave differently: both *Document Viewer* and *Xournal* keep the file open and only release it when it is closed. Specifically, *Xournal* reads the content partially as the user scrolls through the document. For this reason, it is necessary to use the counters for open files discussed earlier in Section 2.2.5, to avoid anomalies and concurrency problems.

In Section 2.3.3 it was noted that a file could be renamed while opened: this behaviour is visible when saving a file with *gedit*: the application creates a temporary hidden file, opens it, updates its content, and finally renames it into the original file so to overwrite it.

3.3 Performance

3.3.1 Measuring methods

The execution times were measured to estimate the performance of some common operations, computed on the basis of 150 tests.

To automate the measurements, we used dedicated Python scripts. The files needed for testing are text files created randomly using the command `base64 /dev/urandom | command head -c BYTES > FILENAME.txt`.

```
times = []
for i in range(150):
    start = msnow()
    with open(filename) as f:
        end = msnow()
        times.append(start - end)

times = []
for i in range(150):
    with open(filename) as f:
        start = msnow()
        data = f.read()
        end = msnow()
        times.append(start - end)


times = []
f = open(filename, 'r+')
content = f.read()
for i in range(150):
    f.seek(0)
    start = msnow()
    f.write(content)
    end = msnow()
    f.truncate()
    times.append(start - end)
f.close()
```

In the Python snippet, `msnow` is a function that returns the Unix timestamp in milliseconds, while `times` is a list used to calculate expected values and standard deviations.

Write times are measured by overwriting the entire file: for one file of size D bytes, the bytes written to disk are exactly D.

3.3.2 Execution times

Tables 3.1, 3.2, and 3.3 show the mean μ and standard deviation σ of the execution times of some common operations, obtained through the scripts described in the previous section, comparing *FreyaFS* in single-thread mode with the direct use of the operating system.

File size	μ_{FreyaFS} [ms]	σ_{FreyaFS} [ms]	μ_{OS} [ms]	σ_{OS} [ms]
4 KB	13.57	0.69	0.01	0.11
1 MB	13.60	0.52	0.01	0.11
10 MB	71.05	2.04	0.01	0.11

Table 3.1: Opening times

File size	μ_{FreyaFS} [ms]	σ_{FreyaFS} [ms]	μ_{OS} [ms]	σ_{OS} [ms]
4 KB	0.11	0.35	0.01	0.11
1 MB	1.22	0.54	0.38	0.79
10 MB	8.85	0.84	5.21	2.38

Table 3.2: Reading times

File size	μ_{FreyaFS} [ms]	σ_{FreyaFS} [ms]	μ_{OS} [ms]	σ_{OS} [ms]
4 KB	0.14	0.30	0.01	0.08
1 MB	69.08	8.97	0.26	0.44
10 MB	12203	95	4.54	0.62

Table 3.3: Writing times

Looking at the times of *FreyaFS*, reading turns out to be quite faster than writing. This behaviour is due to how the virtual file system works: a file is decrypted upon opening and its content is then kept in memory until each file is closed. The reading then takes place on data present in RAM and not from disk, thus offering almost immediate access.

In the measurements of the operating system, however, the opening appears to be with good approximation always the same and much less than the opening with

FreyaFS. This happens because with FreyaFS opening also involves reading each of the fragments and decrypting them, operations that slow down execution. A similar note can be done about the write operation: the encryption of all of the new content introduces more delays.

4. Conclusions

FreyaFS is meant to create a virtual file system for the GNU/Linux operating system that supports encryption through *Mix&Slice*. This project has been divided into two phases that were always side by side with development and code production: an analysis of the file system, and the management of concurrency in a Linux kernel.

Firstly, it was necessary to study how the FUSE library and how a real file system work, in order to allow communication between the two and make as transparent as possible the interaction with the user. Specifically, the analysis of the operating system's file management was a substantial part of the project at multiple levels of abstractions: how to handle file descriptors at a low-level, and how to interface with a graphical shell at a high-level.

Secondly, semaphores were crucial in order to allow concurrent accesses to the same file by different applications and to sync accesses to resources that are shared between threads. The read/write lock implementation proved useful to protect resources from concurrency problems.

The virtual file system developed in this thesis work realises correctly the fundamental functions of managing files and folders, i.e., creating, deleting, moving, and modifying them. The main advantage that *FreyaFS* brings is its *encrypted* nature, since it manages data while guaranteeing privacy thanks to innovative and secure encryption methods such as *Mix&Slice*.

Bibliography

[1] Francesco Tisato and Roberto Zicari. *Sistemi Operativi, Architettura e progetto*. clup, 1985.

[2] *libfuse*. URL: https://github.com/libfuse/libfuse.

[3] *Overview of the Linux Virtual Filesystem*. URL: https://www.kernel.org/doc/html/latest/filesystems/vfs.html.

[4] Valient Gough. *EncFS - an Encrypted Filesystem*. URL: https://github.com/vgough/encfs.

[5] Ronald L. Rivest. *All-Or-Nothing Encrypytion and The Package Transform*. 1997.

[6] S. Paraboschi et al. *Mix&Slice: Efficient Access Revocation in the Cloud*. 2016.

[7] *fusepy*. URL: https://github.com/fusepy/fusepy.

[8] UniBG Seclab. *aesmix*. URL: https://github.com/unibg-seclab/aesmix.

[9] *stat(2) - Linux manual page*. URL: https://man7.org/linux/man-pages/man2/stat.2.html.

[10] *Thread-based parallelism*. URL: https://docs.python.org/3/library/threading.html.