UNIVERSITÀ DEGLI STUDI DI BERGAMO

SCUOLA DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica

Classe n. L–8

FreyaFS

Filesystem virtuale con supporto a Mix&Slice

Relatore: Chiar.mo Prof. Paraboschi Stefano

Prova finale di Michele Beretta

Matricola n. 1054365

ANNO ACCADEMICO

2019 / 2020

FreyaFS

Filesystem virtuale con supporto a Mix&Slice

Michele Beretta

A.A. 2019/2020

Indice

1. Introduzione

Questa tesi tratta la realizzazione di un filesystem virtuale per GNU/Linux che supporti l'algoritmo di cifratura *Mix&Slice* in modo trasparente.

Il filesystem realizzato si occuperà di gestire la creazione, la modifica e l'eliminazione di file e cartelle, sia da interfaccia grafica sia da riga di comando.

1.1 Contesto

Il motivo principale per il quale si rende necessaria la realizzazione di un filesystem che supporti la cifratura è la privacy dei dati. Spesso si hanno dei file (o, più in generale, delle informazioni) che vogliamo rendere accessibili solamente a poche persone per via della loro confidenzialità. Ciò è garantito dalla cifratura, che permette di proteggere le informazioni rendendole accessibili solamente a chi possiede una chiave.

Si vuole però che l'utilizzo e l'accesso a questi dati sia il più semplice possibile, ovvero non sia in alcun modo diverso dall'accesso a file non criptati. Questa funzionalità viene realizzata dal filesystem virtuale, che cifra e decifra i file all'occorrenza, nascondendo le operazioni all'utente.

1.2 Il filesystem

1.2.1 Descrizione

Una delle funzioni principali di un sistema operativo è la gestione delle informazioni su memoria di massa, ovvero una qualsiasi unità periferica che non sia la memoria centrale. Il filesystem è il modulo del sistema operativo che si occupa di questo. Esso, in particolare, deve [1]:

- fornire un meccanismo di identificazione univoco e permanente dei *file*, affinché essi possano esistere ed essere accessibili per tempi lunghi;
- fornire dei *metodi di accesso* che consentano di indirizzare, leggere e scrivere blocchi elementari di informazione contenuti nei file;
- mascherare le caratteristiche fisiche delle unità di memorizzazione;
- realizzare meccanismi di controllo di accesso sui file, sia per garantire riservatezza delle informazioni sia per evitare problemi di concorrenza;
- garantire la permanenza e la consistenza delle informazioni anche in caso di malfunzionamenti hardware o software.

Le principali strutture dati con cui un filesystem lavora sono i *descrittori* dei file. I descrittori contengono varie proprietà del file, tra cui le principali sono il nome del file ed il suo indirizzo, ovvero la sua posizione in memoria di massa. Un insieme di descrittori è detto *directory* (cartella). Dal momento che anche le cartelle devono essere memorizzate su disco permanentemente, esse vengono trattate dai sistemi Unix e Unix-Like al pari dei file: più precisamente, nel sistema Unix una cartella è un file che contiene la lista di nomi e indirizzi dei file contenuti al suo interno.

1.2.2 Operazioni sui file

Per poter accedere ad un file bisogna reperirne il descrittore dalla memoria di massa. Questa operazione però è computazionalmente pesante in termini di numero di accessi al disco. Di conseguenza, per risolvere questo problema, si introduce il concetto di *apertura* del file: prima di una qualsiasi operazione sul file si ricerca il suo descrittore e se ne trasferisce il contenuto nella memoria centrale. Questo descrittore sarà poi adoperato per leggere e scrivere sul file in modo efficiente.

Al termine dell'utilizzo di un file, segue l'operazione duale di *chiusura*: l'immagine del descrittore contenuta in memoria centrale viene copiata su disco in modo da aggiornare le informazioni contenute nel file.

Descriviamo l'apertura di un file come l'azione di ricerca e copia in memoria del descrittore di un file (ed eventualmente del suo contenuto) e la chiusura di un file come l'atto del rilascio della memoria dedicata al descrittore, con conseguente trasferimento del contenuto su disco.

1.2.3 Filesystem virtuale

Un filesystem virtuale è un componente software che permette al kernel di un sistema operativo di accedere al filesystem tramite l'uso di funzioni standard e indipendenti dal filesystem reale (o dal dispositivo usato per la memorizzazione).

L'accesso a file memorizzati su altri computer tramite la rete, o ancora l'accesso a file cifrati senza che l'utente si accorga della cifratura, sono esempi in cui il filesystem virtuale entra in gioco e semplifica l'interazione tra l'utente ed il computer.

1.2.4 FUSE - Filesystem in Userspace

FUSE è un'interfaccia software che consente la realizzazione di filesystem virtuali su kernel Linux [2]. Il progetto FUSE consta di due parti:

1. il modulo kernel per la comunicazione con il kernel Linux;

2. la libreria utente `libfuse` che implementa la comunicazione tra il filesystem virtuale ed il modulo kernel;

FUSE permette quindi all'utente di servirsi direttamente del meccanismo del filesystem virtuale su Linux e, in generale, sui sistemi Unix e Unix-like (tra cui FreeBSD e macOS) grazie ad un processo di porting e di riscrittura.

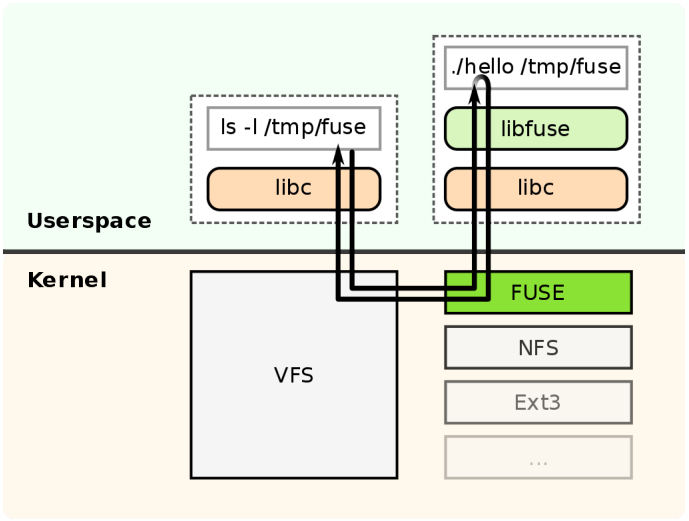

Figura 1.1: Funzionamento di FUSE

Nella figura 1.1 si può vedere un diagramma che rappresenta il funzionamento di FUSE con il kernel Linux.

Ogni richiesta fatta dallo spazio utente (`ls -l /tmp/fuse`) viene intercettata dal modulo kernel VFS[1] e viene reindirizzata al modulo kernel FUSE. La richiesta viene poi passata alla libreria utente `libfuse` che esegue un particolare programma configurato per gestire una specifica richiesta (`./hello`). La risposta del gestore viene quindi ceduta nuovamente a FUSE, per poi essere indirizzata infine al programma iniziale nello spazio utente.

[1]Il *Virtual Filesystem* [3], o *Virtual Filesystem Switch*, è un layer software del kernel Linux che fornisce l'interfacciamento tra il filesystem reale e lo spazio utente.

1.3 La crittografia

1.3.1 Descrizione ed utilità

La crittografia è una tecnica algoritmica che permette di rappresentare un messaggio in maniera offuscata, in modo da renderlo non comprensibile a persone non autorizzate a leggerne il contenuto. Il messaggio cifrato è detto *crittogramma* ed i metodi usati sono detti *tecniche di cifratura*.

La crittografia si basa su un algoritmo ed una chiave di cifratura. L'algoritmo è spesso pubblico, mentre la chiave di cifratura è l'informazione che effettivamente garantisce la confidenzialità del messaggio: senza la chiave, non è possibile accedere alle informazioni cifrate.

Le tecnologie informatiche ne fanno ampio utilizzo, ad esempio per garantire la privacy in un comunicazione tra due o più computer nella rete internet. Essa può essere anche usata su file contenenti dati riservati, sensibili o semplicemente che non si desidera vengano resi disponibili pubblicamente.

1.3.2 Tipologie

Gli algoritmi di cifratura possono essere simmetrici o asimmetrici.

Negli algoritmi *simmetrici* si fa uso di un'unica chiave sia per cifrare sia per decifrare il messaggio. Pur essendo veloci, questi algoritmi hanno dei problemi di sicurezza che riguardano principalmente proprio la presenza di un'unica chiave: questa dev'essere condivisa fra chiunque debba accedere alle informazioni cifrate, e se una terza persona ne viene in possesso può sia leggere il contenuto dei messaggi criptati sia modificarli.

Negli algoritmi *asimmetrici* invece si fa uso di due chiavi differenti, una per cifrare ed una per decifrare. Ad esempio, una chiave può essere resa pubblica ed essere usata per cifrare dei messaggi che solamente la persona in possesso dell'altra

chiave potrà leggere. Lo svantaggio che comporta la crittografia asimmetrica è la lentezza. Infatti, se i dati da trasmettere sono tanti, è buona norma servirsi opportunamente di entrambi i metodi di cifratura: in primis si usa la crittografia asimmetrica per condividere una chiave, per poi utilizzare la crittografia simmetrica, meno computazionalmente pesante.

1.4 Filesystem criptati

Un filesystem criptato è un tipo particolare di filesystem in grado di trattare dei file cifrati. Affinché svolga le sue funzionalità è necessario fornirgli una chiave (o una coppia di chiavi) usata per la cifratura. I vantaggi di un filesystem criptato sono essenzialmente i seguenti:

- la *privacy*, in quanto i file sono cifrati e non sono accessibili a persone che non conoscano la chiave di cifratura;
- la *trasparenza*, in quanto le operazioni di cifratura e decifratura non sono visibili all'utilizzatore, che naviga tra i file come farebbe in un filesystem non cifrato;

Durante l'accesso alle informazioni, il filesystem cifra e decifra i file all'occorrenza, mantenendo su memoria di massa solamente i dati criptati. In questo modo, anche se si riuscisse ad avere accesso al supporto fisico di memorizzazione, non sarebbe possibile leggere il contenuto dei file.

1.5 Stato dell'arte

1.5.1 EncFS

EncFS (*Encrypted Filesystem*) [4] è una libreria open source rilasciata sotto licenza LGPL che implementa un filesystem virtuale con FUSE. EncFS cifra i file indi-

vidualmente e traduce tutte le richieste per il filesystem virtuale nelle equivalenti chiamate al sistema operativo.

In particolare, ha alcune funzionalità che lo distinguono da altri filesystem virtuali:

- ha una "modalità inversa", ossia fornisce una visione criptata di cartelle che non lo sono;
- è relativamente veloce su hard disk tradizionali;
- funziona anche con filesystem di rete;

1.5.2 AONT - All-Or-Nothing Transform

Anche conosciuta come *all-or-nothing-protocol*, AONT [5] è una modalità di cifratura che permette di recuperare il contenuto del file solo se l'intero contenuto è accessibile. Quindi, se anche solo una parte del file criptato viene a mancare, l'intero file non è recuperabile.

In particolare, una trasformazione f è detta *all-or-nothing transform* se:

1. è reversibile, ovvero è possibile ottenere il suo input se si ha a disposizione l'output;
2. sia essa sia la sua inversa sono computabili in modo efficiente;
3. non è computazionalmente possibile ricavare una qualsiasi parte dell'input se viene a mancare anche una sola parte dell'output;

1.5.3 Mix&Slice

Mix&Slice [6] è un approccio che permette di imporre e gestire revoche di accesso a risorse criptate condivise. In modo simile a quanto si verifica con AONT, *Mix&Slice* rende impossibile ottenere la risorsa se non si ha accesso anche ad una parte della

risorsa stessa criptata. Le tecniche di AONT non sono però applicabili in casi in cui gli utenti a cui è applicata la revoca sono a conoscenza delle chiavi di cifratura e potrebbero anche aver mantenuto una copia locale della risorsa criptata.

L'approccio proposto da *Mix&Slice* richiede di partizionare la risorsa in tanti *macro blocchi*, tutti della stessa dimensione, per poi applicare le due tecniche sulle quali esso si fonda:

1. *Mixing*: il contenuto di ogni macro blocco è processato da diversi cicli di cifratura che "mischiano" i bit del blocco. In questo modo, alla fine del processo, ogni bit di input ha un effetto su ogni bit di output.

2. *Slicing*: i macro blocchi sono quindi spezzati e raggruppati in *frammenti*. Questi frammenti sono al centro dell'intero approccio, in quanto la mancanza di un qualsiasi frammento rende impossibile ricostruire la risorsa originale.

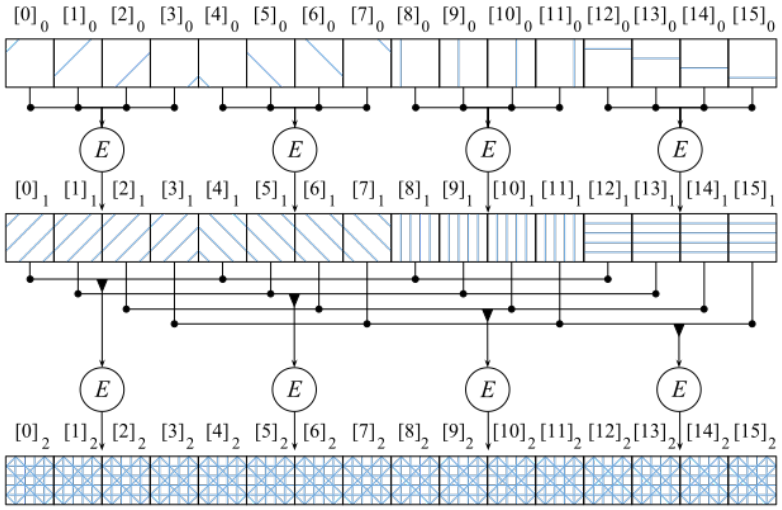

Figura 1.2: Un esempio di *Mix&Slice*

2. Sviluppo del progetto

2.1 Ambiente e strumenti

Il filesystem virtuale è stato implementato con il linguaggio Python per il sistema operativo Ubuntu. Tra le librerie usate nel progetto sono presenti `fusepy` e `aesmix`.

La libreria `fusepy` [7] è un modulo Python che offre dei bindings all'implementazione in C di FUSE per i sistemi Linux e macOS.

L'utilità `aesmix` [8] fornisce invece l'implementazione dell'algoritmo *Mix&Slice* per poter cifrare i file da utilizzare come test durante lo sviluppo. Questa mette a disposizione il wrapper Python `mixslice`, che permette di cifrare e decifrare un file utilizzando, appunto, *Mix&Slice*.

Il wrapper Python accetta come input un file e scrive in output tre elementi:

- una cartella contenente i vari frammenti cifrati dati da *Mix&Slice*;
- un file di metadati `.public`, che contiene la chiave pubblica;
- un file di metadati `.private`, che contiene la chiave privata.

I metodi principali di `aesmix` usati appartengono tutti alla classe `MixSlice` e sono i seguenti:

- `load_from_file`, che carica in memoria i file necessari alla decifratura;

- `save_to_files`, che salva i vari frammenti ed i file di metadati su disco;
- `encrypt`, per cifrare un file;
- `decrypt`, per decifrare un file.

2.2 Logica e funzionamento

2.2.1 Mountpoint, cartella di dati e cartella di metadati

Il filesystem virtuale necessita di tre cartelle per poter essere montato ed utilizzato:

1. una che indica il mountpoint, ovvero il punto dove verrà montato il filesystem virtuale;
2. una contenente i vari dati cifrati, eventualmente organizzati in sottocartelle;
3. una che fornisce tutti i file di metadati (`.public`, `.private` e `.finfo`).

Le cartelle dei file cifrati e dei file di metadati sono diverse tra loro per poter consentire una generalità nell'utilizzo del filesystem. È necessario che la struttura di entrambe sia la stessa per garantire univocità nell'associazione tra file cifrato e file di metadati corrispondente, poiché possono esistere file con nomi uguali in diversi punti del filesystem.

Ad esempio, se un file cifrato ha il percorso `dir/immagine.png`, allora i suoi file di metadati dovranno essere `dir/immagine.png.public`, `dir/immagine.png.private` e `dir/immagine.png.finfo`.

Dal momento che l'utilità *aesmix* dà in output una cartella che contiene i frammenti, per distinguere tra cartella e file cifrato il filesystem fa riferimento all'esistenza o meno dei file di metadati: se per un dato percorso sono presenti i relativi file di metadati allora esso viene trattato come se fosse un file cifrato, altrimenti la gestione viene lasciata al sistema operativo.

2.2.2 Gestione di file criptati

Il filesystem deve mostrare le cartelle contenenti i frammenti cifrati come dei file. Per fare in modo che ciò avvenga è necessario cambiare le informazioni che si trovano nel descrittore del file visto dal sistema. Dal momento che le cartelle sono dei file, anche per esse viene recuperato il descrittore corrispondente in modo del tutto analogo a come avviene per i file (vedi sezione 1.2).

Alla richiesta del descrittore da parte del sistema, vengono quindi modificate le seguenti proprietà [9]:

- `st_mode`: contiene i permessi ed il tipo di file. Per far apparire le cartelle cifrate come dei file si impone il flag `S_IFREG`, che identifica un file, e si rimuove il flag `S_IFDIR`, che identifica una cartella. Gli altri permessi rimangono invariati.
- `st_nlink`: il numero di hard link del file. Questo è pari a 2 per una cartella, quindi si impone a 1 per far apparire le cartelle cifrate come file.
- `st_size`: la dimensione in Byte del file.

In particolare, per trovare la dimensione reale del file, è stato necessario decifrare il file stesso poiché il sistema operativo utilizza il campo `st_size` per sapere quanti Byte leggere da disco[1]. Infatti, in un primo momento era stata mantenuta la dimensione totale della cartella (ovvero la somma delle dimensioni dei frammenti in essa contenuti) e questo poteva causare dei problemi alla lettura del file, soprattutto da interfaccia grafica. All'apertura del file, il sistema poteva leggere meno Byte di quanti ne contenesse il file, considerando quindi il file corrotto, oppure leggere più Byte del dovuto, andando in crash e richiedendo una chiusura forzata dell'applicazione.

Decifrare un file ogni volta che viene fatta una richiesta del descrittore può però causare rallentamenti. Per questo motivo è stato introdotto un nuovo tipo di file

[1] Questo non è un comportamento comune a tutto il sistema, infatti è adottato dall'interfaccia grafica ma non da linea di comando. Da terminale, anche con un valore di `st_size` sbagliato, è possibile leggere il contenuto di un file con `cat` e scriverlo tramite la combinazione di `echo` e `>`.

di metadati, con estensione `.finfo`, che contiene informazioni sulla dimensione dei file nel filsystem virtuale. Questa dimensione viene aggiornata ad ogni scrittura e troncatura sul file, solamente se è effettivamente cambiata. In questo modo, seppur con un overhead iniziale dovuto alla scrittura su file, il filesystem risulta decisamente più reattivo nel suo utilizzo normale.

Quanto descritto in questa sezione può essere fatto solamente se è possibile aprire e decifrare il file e se esistono dei file di metadati associati ad esso. Le cartelle che non contengono i frammenti cifrati vengono quindi trattate direttamente dal sistema, che le gestisce senza modifiche da parte del filesystem virtuale.

2.2.3 Lettura, scrittura e creazione di file

Per poter leggere e scrivere un file è necessario aprirlo, ovvero decifrarlo e trasferire il suo contenuto in memoria.

La *lettura* avviene a blocchi sul contenuto in memoria, tramite un offset e un numero totale di Byte da leggere. La *scrittura*, invece, avviene tramite un buffer con i dati da salvare ed un offset che indica il Byte da cui iniziare a scrivere. È anche possibile *troncare* un file, ovvero limitarne la lunghezza ad un certo valore, eliminando qualsiasi informazione in eccesso. All'apertura viene impostato anche il nuovo valore da dare a `st_atime`, che indica l'istante temporale dell'ultimo accesso al file [9].

La *scrittura* ha la particolarità di non essere effettuata su disco direttamente, ma viene effettuata sul contenuto del file in memoria. Il file viene cifrato su disco in un momento successivo, al flush del file. Per motivi di efficienza, il flush viene fatto solamente se sono state eseguite delle modifiche al contenuto del file, ovvero se è stata fatta almeno una scrittura. In questo modo, se un file non è stato modificato da quando è stato aperto non si effettuano cifrature non necessarie, rendendo più veloci le operazioni di sola lettura. La scrittura inoltre si occupa di cambiare il valore di `st_mtime`, che indica l'istante (Unix timestamp) di ultima modifica [9] al contenuto del file.

La *creazione* di un file su disco corrisponde alla creazione in memoria di un'area con il dovuto contenuto, anche vuoto[2], ed al flush forzato del file. Inoltre, la creazione di un file abilita alla possibilità di copiare e duplicare i file, in quanto la copia è sostanzialmente una creazione seguita da una o più scritture.

2.2.4 Rinomina ed eliminazione di file

La *rinomina* di un file cifrato comporta anche la rinomina dei file di metadati associati. Se fatta da interfaccia grafica, l'operazione non permette la rinomina in un file già esistente, mentre da linea di comando la rinomina è indistinguibile dallo spostamento di un file. L'implementazione di una corretta rinomina abilita quindi sia la possibilità di spostare i file sia la possibilità di sovrascrivere i file nel filesystem virtuale.

L'*eliminazione* definitiva di un file comporta anche l'eliminazione dei file di metadati. Lo spostamento nel cestino, invece, è a tutti gli effetti uno spostamento, quindi supportato con la rinomina. Se sul volume virtuale creato da FUSE non esiste una cartella adibita a cestino, questa verrà creata. Anche i file nel cestino saranno cifrati.

Né l'eliminazione né la rinomina necessitano di apertura e chiusura dei file, quindi risultano particolarmente performanti per l'assenza di cifrature e decifrature. Entrambe le operazioni, però, necessitano di modificare le informazioni mantenute in memoria riguardanti le varie caratteristiche dei file, per evitare di fare riferimento a dati non più validi (vedi la gestione di `st_size` alla sezione 2.2.2 e l'implementazione di `EncFilesManager` a 2.3.3). Può anche capitare che un file venga rinominato mentre è aperto.

[2]Se si tenta di creare un file già esistente o già aperto, il contenuto di questo dev'essere mantenuto e non dev'essere sovrascritto.

2.2.5 Aperture multiple dello stesso file

Un file può essere aperto contemporaneamente da più applicazioni. Alla chiusura di una di queste, il filesystem impedisce la chiusura di un file se è ancora in uso da altri programmi, per evitare comportamenti indesiderati o corruzione dei dati.

Questo si traduce operativamente nel mantenere un contatore per ogni file aperto in memoria, che è incrementato ad ogni apertura e decrementato ad ogni chiusura del file in questione. Il file viene infine chiuso, rilasciando le aree di memoria dedicate, quando questo contatore arriva al valore 0. Resta possibile effettuare più volte il flush di un file per aggiornarne il contenuto su disco.

Le applicazioni si occupano da sé di aggiornare il contenuto del file se questo è cambiato su disco, oppure propongono all'utente di ricaricarlo. Questo avviene grazie all'aggiornamento di `st_mtime` durante le operazioni di scrittura.

2.2.6 Gestione di cartelle

Per mantenere la stessa identica struttura tra la cartella con i file cifrati e la cartella con i file di metadati, il filesystem virtuale deve occuparsi di replicare tutte le operazioni sulle cartelle (creazione, modifica ed eliminazione) tra le due posizioni su disco.

La gestione non presenta particolari difficoltà, fatta eccezione per l'operazione di rinomina, in quanto essa è effettuata dal sistema operativo tramite la stessa primitiva usata per i file, `rename`, a differenza di altre operazioni: ad esempio, per rimuovere un file si usa il metodo `unlink`, mentre per rimuovere una cartella si usa il metodo `rmdir`. Nel caso della rinomina è quindi necessario differenziare tra file cifrato e cartella tramite la proprietà `st_mode` del descrittore opportunamente modificato, come spiegato nella sezione 2.2.2.

2.2.7 Multithreading

La libreria `libfuse` di Linux permette di montare i filesystem virtuali in due modalità:

1. *single-thread*, in cui l'intero filesystem viene eseguito su un solo thread all'interno di un processo;
2. *multi-thread*, in cui le varie operazioni (scrittura, lettura, etc.) sono assegnate ogni volta a thread diversi.

La modalità single-thread è meno performante dal punto di vista prestazionale ma è più semplice e adatta allo sviluppo poiché non richiede nessun controllo di concorrenza.

La modalità multi-thread offre invece performance migliori in quanto consente di parallelizzare le operazioni sui file quando possibile, come per esempio nel caso di una lettura di una grande quantità di dati. Nello specifico, la libreria `libfuse` protegge le sezioni critiche del codice da accessi concorrenti tramite semafori implementati secondo lo standard POSIX tramite la libreria `pthread.h`.

FreyaFS mantiene questi due paradigmi, operando di default su un singolo thread. Sebbene `libfuse` si occupi già a basso livello della protezione delle sezioni critiche, resta comunque necessario proteggere l'accesso alle zone di memoria condivise, rappresentate principalmente dal contenuto dei file aperti ed i contatori delle applicazioni che hanno aperto un dato file.

Il filesystem gestisce in modo diverso le azioni di scrittura e di lettura del contenuto dei file aperti. Più letture possono avvenire in parallelo al fine di aumentare le prestazioni, mentre le scritture non devono interferire le une con le altre e devono avere l'accesso esclusivo alla risorsa. Questo comportamento è realizzato tramite l'utilizzo di *read lock* e *write lock*. Il filesystem virtuale mantiene per ogni file aperto un contatore di quanti thread stanno leggendo la risorsa: questo viene incrementato ogni volta che ad un thread inizia l'operazione di lettura e decrementato quando ter-

mina. Quando un thread richiede l'accesso in scrittura al file viene messo in attesa se sono presenti dei lettori e viene notificato quando non ce ne sono più. Se invece la risorsa è libera viene garantito l'accesso in modo esclusivo tramite un semaforo.

2.3 Implementazione

Il progetto è organizzato in cinque file principali:

- `main.py`, che si occupa della gestione degli argomenti da linea di comando e del montaggio del filesystem;
- `freyafs.py`, che gestisce il funzionamento del filesystem in generale e comunica con il sistema operativo;
- `encfilesmanager.py`, che implementa le varie operazioni di input/output sui file;
- `encfilesinfo.py`, che gestisce i file di metadati `.finfo`;
- `filebytecontent.py`, che si occupa della gestione della concorrenza sul contenuto dei file in memoria.

Nella figura 2.1 si vede la struttura statica del codice, in particolare come i moduli sono in relazione tra loro. Una freccia da un modulo A ad un modulo B indica che B usa il modulo A. Nei moduli "esterni" sono rappresentati `aesmix`, che realizza l'implementazione di *Mix&Slice*, e `fusepy`, che fornisce l'implementazione di FUSE per il linguaggio Python. I moduli "interni" sono quelli che si occupano di gestire i file cifrati e che compongono a tutti gli effetti la logica del filesystem virtuale. Infine, `main` gestisce solamente l'interfacciamento con l'utente da riga di comando e fa uso di `fusepy` per montare il filesystem.

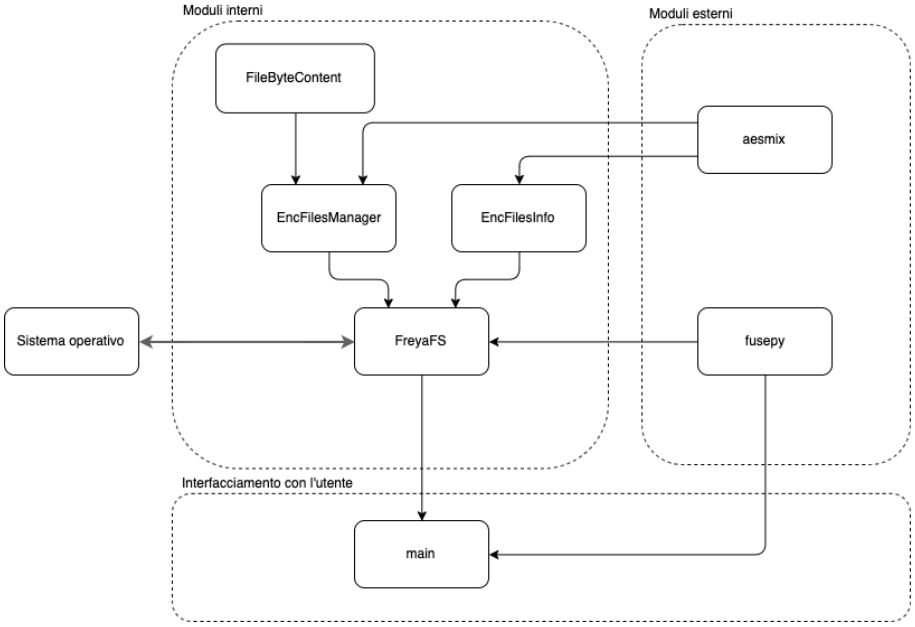

Figura 2.1: Struttura del codice

2.3.1 Interfacciamento con l'utente

L'interfacciamento con l'utente è gestito dal modulo `main.py`, che si occupa di organizzare gli argomenti passati da riga di comando e di montare il filesystem. I parametri che è possibile impostare sono presenti nella tabella 2.1.

Parametro	Utilizzo
`MOUNT`	Il mountpoint, ovvero il percorso dove verrà montato il filesystem
`--data DATA`	Il percorso della cartella contenente i file cifrati
`--metadata METADATA`	Il percorso della cartella dei file di metadati
`--multithread`	Flag che indica se montare il filesystem in modalità multi-thread (di default a `False`)

Tabella 2.1: Parametri di `main.py`

2.3.2 FreyaFS

Il modulo `freyafs` implementa i metodi di FUSE per la gestione del filesystem virtuale. In particolare, la parte più importante del modulo è la classe `FreyaFS`, che estende la classe `Operations` di FUSE e ne ridefinisce alcuni metodi, facendone l'overloading. Molti dei metodi della classe richiamano semplicemente il sistema operativo, in quanto non necessitano del supporto alla cifratura (come ad esempio il test di accesso ad un file). Altri metodi, invece, sono ridefiniti per supportare la cifratura e decifratura dei file, realizzata dal modulo `encfilesmanager`.

Di seguito un estratto del metodo `getattr`, usato per ricavare informazioni del descrittore di file e cartelle. Il codice qui presente riguarda la modifica delle informazioni del descrittore delle cartelle contenenti i frammenti cifrati.

```
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  }
```

Questo frammento di codice ha due compiti principali: il calcolo della dimensione del file cifrato e il cambio di flag per far apparire la cartella coi frammenti come un unico file.

Il calcolo della dimensione è fatto dall'istanziamento della classe `EncFilesInfo`, solo se non già stato effettuato in precedenza (al fine di migliorare le prestazioni evi-

tando decifrature sovrabbondanti, righe dalla 1 alla 3). La proprietà `self.enc_info` è un dizionario che contiene le varie istanze di `EncFilesInfo` per ogni file cifrato trovato nel sistema.

Il cambio di flag viene effettuato in modo da mantenere gli stessi permessi, come si evince dall'espressione alla riga 6, dove `st.st_mode` sono i permessi originali della cartella con i frammenti. Il flag `stat.S_IFREG` è alzato tramite un'operazione di `OR` bit a bit, mentre per abbassare il flag `stat.S_IFDIR` è necessario fare un `AND` bit a bit con la negazione bit a bit di `stat.S_IFDIR`, ovvero `st.st_mode & ~stat.S_IFDIR`.

Segue un estratto del metodo `unlink`, che consente la rimozione dei file.

```
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)
```

Nel codice vengono solamente chiamate le primitive del sistema operativo, senza aprire o chiudere nessun file. Vengono anche eliminati i file di metadati e le zone di memoria dedicate alle informazioni dei file (riga 10). È necessario controllare l'esistenza del file `.finfo` (riga 6) perché, a differenza di `.public` e `.private`, può non esistere in quanto viene creato alla prima chiamata di `getattr` per il file in questione.

La rinomina di un file, gestita dal metodo `rename`, risulta invece più complessa, in quanto è la stessa operazione usata per lo spostamento sia di file sia di cartelle. Inoltre, siccome i file cifrati sono cartelle, se si vuole rinominare un file con un nome

di un altro file già esistente[3] bisogna prima eliminare quest'ultimo (righe 6-7), per evitare che il sistema lo riconosca come cartella e tenti di spostare il primo "dentro" il secondo. Infine, sono anche rinominati i percorsi presenti in memoria relativi al file ed ai suoi metadati (righe 19-25).

Per le cartelle "normali", ovvero non contenenti frammenti cifrati, la rinomina è lasciata al sistema operativo e replicata sia nel percorso dei dati cifrati sia nel percorso dei metadati (caso `else`).

```
1   full_old_path = self._full_path(old)
2   full_new_path = self._full_path(new)
3
4   if self._is_file(old):
5       # Rinomino un file
6       if self._is_file(new):
7           self.unlink(new)
8
9       old_public_metadata, old_private_metadata, old_finfo = self.
            _metadata_names(old)
10      new_public_metadata, new_private_metadata, new_finfo = self.
            _metadata_names(new)
11
12      os.rename(old_public_metadata, new_public_metadata)
13      os.rename(old_private_metadata, new_private_metadata)
14      if os.path.isfile(old_finfo):
15          os.rename(old_finfo, new_finfo)
16
17      os.rename(full_old_path, full_new_path)
18
19      if full_old_path in self.enc_files:
20          self.enc_files.rename(full_old_path, full_new_path)
21
22      if full_old_path in self.enc_info:
23          self.enc_info[full_old_path].rename(full_new_path,
                new_public_metadata, new_finfo)
```

[3]Questo comportamento è impedito da interfaccia grafica, ma consentito da linea di comando con il comando `mv`.

```
24              self.enc_info[full_new_path] = self.enc_info[full_old_path]
25              del self.enc_info[full_old_path]
26      else:
27          # Rinomino una cartella
28          old_metadata_path = self._metadata_full_path(old)
29          new_metadata_path = self._metadata_full_path(new)
30          os.rename(old_metadata_path, new_metadata_path)
31          os.rename(full_old_path, full_new_path)
```

Tutti i metodi che agiscono su input ed output di file, invece, richiamano i dovuti metodi della classe `EncFilesManager` e, se necessario, aggiornare la lunghezza corrente gestita da `EncFilesInfo`. A scopo esemplificativo, di seguito viene riportato il metodo `write`.

```
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 questi due estratti, `self.enc_files` è un'istanza della classe `EncFilesManager` che contiene tutti i contenuti dei file attualmente aperti, in modo da poterli leggere e scrivere.

I metodi di `open`, `create`, `truncate`, `read`, `flush` e `release` hanno tutti una struttura similare. Se un file non risulta aperto tra quelli cifrati, allora il file viene gestito dal sistema operativo come farebbe di consueto.

In conclusione, la tabella 2.2 riassume alcuni metodi usati da *FreyaFS* che è stato necessario modificare rispetto al caso considerato "base", ovvero il caso in cui per ogni metodo vengono semplicemente richiamate le primitive del sistema operativo corrispondenti.

Metodo	Utilizzo
`getattr`	Ottiene le informazioni di un file cifrato o cartella
`readdir`	Fornisce il contenuto di una data cartella, includendo `.` e `..` ed ignorando file di metadati
`mkdir`	Crea una cartella, sia nel percorso dei dati cifrati sia nel percorso dei metadati
`rmdir`	Rimuove una cartella, sia nel percorso dei dati cifrati sia nel percorso dei metadati
`unlink`	Rimuove un file cifrato, eliminando anche i corrispondenti file di metadati
`rename`	Rinomina un file cifrato (ed i corrispondenti file di metadati) o una cartella
`utimens`	Aggiorna gli istanti di ultimo aggiornamento e di ultima modifica per file cifrati e per cartelle
`open`	Apre un file cifrato
`create`	Crea un file cifrato
`read`	Legge un certo numero di Byte da un file cifrato aperto
`write`	Scrive un certo numero di Byte in un file cifrato aperto
`truncate`	Tronca il contenuto di un file cifrato aperto ad una data lunghezza
`flush`	Forza le scritture fatte su in file cifrato dalla memoria al disco
`release`	Rilascia un file cifrato aperto

Tabella 2.2: Metodi di *FreyaFS*

2.3.3 EncFilesManager

Questo modulo definisce la classe omonima per l'input ed output sui file cifrati. In particolare, la classe viene istanziata una sola volta e contiene le informazioni di tutti i file (cifrati con *Mix&Slice*) al momento aperti, identificabili tramite il loro percorso nel filesystem reale. Si serve di un contatore per ogni file aperto che

indica quante applicazioni lo stanno usando al momento: in questo modo si evita di chiudere "accidentalmente" il file se solamente alcune di queste applicazioni lo rilasciano mentre altre devono mantenerlo ancora aperto.

I metodi definiti sono i seguenti:

- `open`: apre un file, decifrandone il contenuto e mantenendolo in memoria;
- `create`: crea un file vuoto, ovvero riserva dello spazio vuoto in memoria per il contenuto del file (solo se non già aperto) e ne impone il flush;
- `read_bytes`: legge un certo numero di Byte da un file aperto;
- `write_bytes`: sovrascrive una parte del contenuto di un file aperto con dei dati contenuti in buffer;
- `truncate_bytes`: tronca il contenuto di un file aperto ad una certa lunghezza;
- `flush`: scrive il contenuto di un file aperto su disco, cifrandolo;
- `release`: rilascia le aree di memoria dedicate ad un file, chiudendolo;
- `cur_size`: ottiene la lunghezza attuale del contenuto di un file aperto;
- `rename`: rinomina i vari percorsi riferiti ad un file aperto.

È fondamentale avere un metodo `rename` perché un file può essere rinominato anche se aperto: in questo caso, per evitare di fare riferimento a file non più esistenti e causare un comportamento errato del filesystem, bisogna aggiornare tutti i percorsi relativi al file in questione.

La classe `EncFilesManager` tiene anche traccia dello stato di modifiche di un file tramite dei flag booleani. In questo modo, se viene richiesto un `flush` ma non sono presenti modifiche al file, è possibile evitare l'operazione di cifratura per migliorare le prestazioni del filesystem ed evitare tempi morti essenzialmente inutili.

Infine, la classe gestisce anche la mutua esclusione per la modalità multi-thread del filesystem virtuale, tramite un semaforo di tipo `Lock` del modulo `threading` di

Python (vedere la sezione 2.3.6). Questo semaforo garantisce l'accesso esclusivo alle strutture che contengono il contenuto dei file aperti ed i contatori dei file aperti.

Il codice qui riportato è l'implementazione del metodo `open`:

```
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
```

Il metodo si occupa anche di aggiornare il timestamp di ultimo accesso al file, tramite `self.atimes`, ed il timestamp di ultima modifica al file, tramite `self.mtimes`. Tra i due, solamente il secondo viene aggiornato durante le scritture sul contenuto del file ed entrambi sono sempre scritti su disco al flush del file, anche in caso di assenza di modifiche.

2.3.4 EncFilesInfo

Il modulo `encfilesinfo` definisce la classe `EncFilesInfo` che si occupa di gestire varie informazioni dei file cifrati che non sono contenute nei file di metadati `.private` e `.public`. Il suo compito principale è interfacciare `freyafs` con i file `.finfo`, che contengono le varie informazioni serializzate come JSON. La classe, a differenza di `EncFilesManager`, non è a conoscenza di tutti i file trovati dal filesystem virtuale, ma gestisce solamente le informazioni di un singolo file specificato all'inizializzazione.

Al momento il modulo supporta solamente la gestione della lunghezza in Byte del contenuto del file cifrato, che si può ottenere ed impostare tramite la proprietà `size`.

All'istanziamento della classe, viene calcolata la dimensione del file cifrato desiderato e viene memorizzata nella proprietà `size` e anche nel file `.finfo` (che, se non presente, viene creato). Quando si vuole modificare questa proprietà, viene anche aggiornato il valore contenuto nel file di metadati solamente se il valore è diverso da quello precedente. In questo modo si evita di aprire un file, effettuare il parsing del JSON in esso contenuto, aggiornare la proprietà per poi serializzare nuovamente il JSON prodotto nel caso in cui queste operazioni non siano strettamente necessarie.

```
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()
```

La classe mette anche a disposizione un metodo `rename`, usato per rinominare i percorsi del file di cui mantiene le informazioni.

2.3.5 FileByteContent

Questo modulo si occupa unicamente di effettuare letture e scritture sul contenuto in memoria di un file aperto in modo mutualmente esclusivo. Per garantire ciò, la classe `FileByteContent` ha le seguenti proprietà:

- `_text`, che rappresenta il contenuto del file come stringa di Byte;
- `_readers`, che indica il numero di lettori presenti in un dato momento;
- `_cond`, di tipo `threading.Condition` che viene usata come semaforo.

La classe `FileByteContent` consente di avere un semaforo per ogni file aperto. In questo modo, sono possibili letture e scritture parallele su file diversi, ma non sullo stesso file.

A scopo esemplificativo, di seguito alcuni metodi della classe.

```
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 questo estratto di codice `_r_acquire()` e `_r_release()` si occupano rispettivamente di acquisire e rilasciare il lock in lettura, condivisibile tra più thread, mentre `_w_acquire()` e `_w_release()` sono l'equivalente per il lock in scrittura, che garantisce l'accesso esclusivo al contenuto del file.

L'acquisizione di un lock per la lettura incrementa la variabile `_readers`, che viene decrementata con il rilascio. Quando si raggiunge lo 0, si notificano eventuali thread in attesa di scrittura.

Alla richiesta di un lock in scrittura, il thread rimane in attesa fintantoché sono presenti dei lettori, ovvero se `_readers` è positiva. Quando i lettori liberano la risorsa, viene acquisito il lock e viene rilasciato al termine dell'operazione.

I metodi per l'acquisizione ed il rilascio dei lock di lettura e scrittura sono implementati come segue. Per dettagli più specifici sulla proprietà `_cond` fare riferimento alla sezione 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 La libreria `threading`

La libreria `threading` [10] è una libreria integrata in Python che permette la gestione di thread ad alto livello e che fornisce strumenti per la creazione e l'utilizzo di semafori.

Le classi utilizzate nel progetto per i *read lock* ed i *write lock* di `FileByteContent` (sezione 2.3.5) sono:

- `threading.Lock`, che offre l'implementazione del concetto di mutex, un semaforo mutualmente esclusivo;
- `threading.Condition`, che rappresenta una condizione associata ad una particolare istanza di `Lock`.

Nel codice della classe `FileByteContent` è presente la proprietà `_cond` di tipo `Condition`, la quale fornisce i seguenti metodi:

- `acquire`: acquisisce il lock sottostante;
- `release`: rilascia il lock sottostante;
- `wait`: sospende il thread chiamante e rilascia il lock sottostante, che dev'essere stato già acquisito dal thread;
- `notify_all`: risveglia tutti i thread in coda, che escono dalla condizione di attesa solo quando ottengono nuovamente il controllo sul lock.

I semafori forniti da `Lock` sono degli oggetti che possono trovarsi in uno stato "bloccato" oppure "libero". Quando il lock è nello stato libero, esso può essere acquisito tramite il metodo `acquire` che lo porta nello stato bloccato. Se invece `acquire` viene invocato mentre il semaforo è bloccato, il thread chiamante si sospende fino a quando una chiamata a `release` non libera il mutex.

Gli oggetti istanziati da `Condition` sono più flessibili rispetto ai semplici mutex, ma sono comunque sempre associati ad un semaforo. Tramite questi oggetti è pos-

sibile sospendere un thread con il metodo `wait`, ponendolo in una coda di attesa e causando il rilascio del lock sottostante, e risvegliarlo con `notify` o `notify_all`. All'uscita dalla coda di attesa, il thread acquisisce di nuovo il lock per poi continuare nella sua esecuzione. In questo modo è possibile evitare le *busy wait*, ovvero delle attese in cui il thread controlla continuamente lo stato del semaforo, occupando risorse computazionali che potrebbero essere usate da altri thread non sospesi.

Infine, entrambe le classi supportano il *context management protocol*, ovvero l'utilizzo del costrutto `with` di Python. Quando il processo o thread entra in un blocco delimitato da `with` viene richiamato il metodo `acquire` del lock o della condizione, mentre all'uscita è invocato il metodo `release`. Questo costrutto è visibile nell'implementazione della classe `EncFilesManager` alla sezione 2.3.3.

3. Testing ed analisi

3.1 Sistema

Il filesystem virtuale è stato sviluppato e testato sul seguente sistema:

- sistema operativo Ubuntu 20.04 LTS Focal Fossa;
- processore Intel Core i5-7200U quad-core da 2.5 GHz;
- RAM da 8 GB;
- disco a stato solido da 120 GB.

3.2 Apertura multipla dello stesso file

L'apertura di uno stesso file con più applicazioni dedicate è stata testata sui seguenti formati di file:

1. file di testo `.txt`, con *gedit* e *Visual Studio Code*;
2. file di testo `.md`, con *gedit* e *Typora*;
3. file binari `.jpg`, con *Image Viewer* e *Firefox*;
4. file binari `.pdf`, con *Document Viewer* e *Xournal*.

Per i file `.txt`, entrambe le applicazioni aprono il file e lo richiudono immediamente dopo averne letto l'intero contenuto. Solamente all'azione di salvataggio il file viene aperto nuovamente per apportare le dovute modifiche. Questo riduce la possibilità di imbattersi in problemi di concorrenza che possono esserci quando due applicazioni mantengono contemporaneamente aperto uno stesso file. La stessa identica situazione avviene con i file `.md` ed i file `.jpg`. Le applicazioni trattano in modo diverso solamente l'aggiornamento del file quando il contenuto su disco è cambiato: ad esempio, *gedit* propone sempre di ricaricare l'intero file, mentre *Typora* e *Visual Studio Code* aggiornano il file automaticamente ed avvisano l'utente solamente se sono presenti modifiche in sospeso.

Le applicazioni usate per testare i file `.pdf` si comportano invece diversamente: sia *Document Viewer* sia *Xournal* mantengono il file aperto e lo rilasciano solamente alla loro chiusura. Nello specifico, *Xournal* legge il contenuto parzialmente man mano che l'utente scorre il documento. Per questo motivo si rende necessario l'utilizzo di contatori per i file aperti discussi nella sezione 2.2.5 per evitare comportamenti indesiderati e problemi di concorrenza.

Nella sezione 2.3.3 si era detto che un file può essere rinominato anche mentre è aperto: questo comportamento è visibile al salvataggio di file di testo con *gedit*. Infatti, l'applicazione crea un file temporaneo nascosto per poi aprirlo, aggiornarne il contenuto e rinominarlo nel file originale per sovrascriverlo.

3.3 Performance

3.3.1 Metodi di misura

Per la stima delle performance sono state effettuate delle misure dei tempi di esecuzione di alcune operazioni comuni, calcolate sulla base di 150 prove.

Al fine di poter automatizzare le misure si è fatto uso di script Python appositi (di seguito). I file necessari per le prove sono file di testo creati randomicamente

tramite il comando `base64 /dev/urandom | head -c BYTES > FILENAME.txt`.

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

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


# Test di scrittura
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()
```

Nel codice Python, `msnow` è una funzione che ritorna il timestamp Unix in millisecondi e la lista `times` è usata per calcolare medie e deviazioni standard.

I tempi di scrittura sono misurati sovrascrivendo interamente il file: per un file grande D Byte, i Byte scritti su disco sono esattamente D.

3.3.2 Tempi di esecuzione

Nelle tabelle 3.1, 3.2 e 3.3 sono riportate media μ e deviazione standard σ dei tempi di esecuzione di alcune operazioni comuni, ottenute tramite gli script della sezione precedente sia con *FreyaFS* in modalità single-thread sia direttamente con il sistema operativo.

Dimensione del file	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

Tabella 3.1: Tempi di apertura

Dimensione del file	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

Tabella 3.2: Tempi di lettura

Dimensione del file	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

Tabella 3.3: Tempi di scrittura

Analizzando i tempi di *FreyaFS*, la lettura risulta essere particolarmente veloce rispetto alla scrittura. Questo comportamento è dovuto al funzionamento del filesystem virtuale: il file viene decifrato all'apertura ed il contenuto viene poi mantenuto in memoria fino alla chiusura. La lettura quindi avviene su dati presenti in RAM e non da disco, rendendone così l'accesso quasi immediato.

Nelle misure del sistema operativo, invece, l'apertura risulta essere con buona approssimazione sempre uguale e molto minore rispetto all'apertura con *FreyaFS*.

Questo avviene perché con *FreyaFS* l'apertura comporta anche la lettura di tutti e 1024 i frammenti e la decifratura del file cifrato, operazioni che rallentano l'esecuzione. Un discorso simile si applica alla scrittura: il dover cifrare nuovamente tutto il contenuto introduce un ulteriore ritardo.

4. Conclusioni

FreyaFS nasce con l'obiettivo di creare un filesystem virtuale per il sistema operativo GNU/Linux che supporti la cifratura con *Mix&Slice*. La realizzazione di questo progetto ha comportato due fasi di studio che sono state affiancate allo sviluppo ed alla stesura del codice: una prima riguardante l'analisi del filesystem ed una seconda atta a gestire la concorrenza in un ambiente con kernel Linux.

In un primo momento è stato infatti necessario studiare il funzionamento della libreria FUSE e del filesystem reale del sistema operativo, al fine di poter permettere la comunicazione tra i due e di rendere il più trasparente possibile l'interazione con l'utente. In particolare, la disamina del comportamento del sistema operativo nella gestione dei file ha avuto una rilevanza non indifferente nel progetto, a più livelli di astrazione: a basso livello l'utilizzo dei descrittori, a più alto livello l'interazione della shell grafica con i file e le cartelle.

In secondo luogo, per permettere un accesso parallelizzato allo stesso file da parte di applicazioni diverse e per sincronizzare l'accesso a risorse condivise tra thread è stato indispensabile ricorrere ai semafori. L'implementazione di read/write lock è servita per proteggere risorse dagli accessi concorrenti di thread diversi.

Il filesystem virtuale sviluppato in questo lavoro di tesi realizza correttamente le funzioni fondamentali di gestione di file e cartelle, ossia creazione, eliminazione, spostamento e modifica. Il vantaggio principale che presenta *FreyaFS* è dato dalla sua natura di filesystem cifrato, in grado di agire sui dati garantendone la privacy grazie a metodi di cifratura sicuri ed innovativi come *Mix&Slice*.

Bibliografia

[1] Francesco Tisato e 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`.

Ringraziamenti

I miei primi ringraziamenti vanno alla mia famiglia per il sostegno datomi in tutti questi anni.

Un grazie in particolare va a tutti i collaboratori del Seclab dell'Università degli Studi di Bergamo, che hanno saputo guidarmi sia nello sviluppo del progetto sia nella stesura di questa tesi.

Ringrazio i miei compagni di corso ed amici Alpha e Bianca, che mi hanno accompagnato in questo percorso di laurea triennale, ed Ilaria per essere sempre stata presente. Infine un pensiero va ad Irene, che con pazienza ed interesse ha seguito lo sviluppo della tesi, dandomi validi consigli ed offrendomi sempre il suo supporto.

Grazie anche a tutte le persone che mi hanno permesso di conseguire degli obiettivi che altrimenti non avrei mai pensato di raggiungere.