FFS: A CRYPTOGRAPHIC CLOUD-BASED STEGANOGRAPHIC FILESYSTEM THROUGH EXPLOITATION OF ONLINE WEB SERVICES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Glenn Olsson

June 2022

COMMITTEE MEMBERSHIP

TITLE:	FFS: A cryptographic cloud-based steganographic filesystem through exploitation of online web services
AUTHOR:	Glenn Olsson
DATE SUBMITTED:	June 2022
COMMITTEE CHAIR:	Zachary Peterson, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science

ABSTRACT

FFS: A cryptographic cloud-based steganographic filesystem through exploitation of online web services

Glenn Olsson

Today there are free online services that can be used to store files of arbitrary types and sizes, such as Google Drive. However, these services are often limited by a certain total storage size. The goal of this thesis is to create a filesystem that can store arbitrary amount and types of data, i.e. without any real limit to the total storage size. This is to be achieved by taking advantage of online webpages, such as Flickr, where text and files can be posted on free accounts with no apparent limit on storage size. The aim is to have a filesystem that behaves similar to any other filesystem but where the actual data is stored for free on various websites.

ACKNOWLEDGMENTS

Thanks to my mom, dad, and the rest of my family for their constant support

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

To keep files and data secure we often use encrypted filesystems. However, while these filesystems hide the content of the data, they often do not conceal the existence of data. For instance, using snapshots of the filesystems from different moments in time, it could be possible to notice a difference in the data stored and therefore that data exists and where it is located. Snapshots could even reveal user passwords [**hanMultiuserSteganographicFile2010**].

Deniable filesystems are intended to make the data deniable, meaning that the user is supposed to be able to plausibly deny the existence of data. This is often accomplished through the use of digital steganography. There are many reasons why this is important. For instance, in 2011, a Syrian man recorded videos of attacks on civilians carried out by Syrian security forces, which he wanted to share with the world [**westheadHowSyrianRefugee2012**]. By cutting his arm, he was able to hide a memory card inside the wound and smuggled it out of the country. However, if he would have used methods such as an encrypted deniable filesystem, the border control may not have been able to discover even the existence of data, even if they would have found the memory card. By only encrypting the data, the border control would have been able see that he was trying to hide data and make him reveal the decryption key, either by legal measures or by force, which is why he smuggled it out.

There exists multiple deniable filesystems that are designed to combat this problem on physical devices, such as memory cards. However, even just carrying a memory card might subject you to suspicion of hiding data, no matter how the filesystem is designed. Another solution to hiding the data is therefore to hide it somewhere else, for instance online through the use of cloud-based filesystem service, such as Google Drive. Someone searching your body and devices, at for instance an airport or border control, might not realize that you are using a cloud-based filesystem service to hide your data. Although, more thorough investigations of a person might reveal user

accounts used on the service, leading to legal processes where the service is forced to disclose your data. Even if you encrypt the data you upload to such a service, you can still be forced to reveal the decryption keys. What we want to achieve is a combination of a deniable filesystem and a cloud-based filesystem, where the data is stored using digital cryptographic and steganographic methods but without any company or person other than the user controlling the actual data. To accomplish this, we can store the data on online social media platforms.

Social media platforms such as Twitter and Flickr have many millions of daily users that post texts and images (for example, of their cats or funny videos). According to Henna Kermani at Twitter, they processed 200 GB of image data every second in 2016 [**MobileScaleLondona**]. The photos posted on Twitter, as opposed to the ones stored on cloud services such as Google Drive, are stored for free on the service for the user, for what seems to be an indefinite period. There is also no specified limit of how many images or tweets one can make. Although, as stated in their terms of service, such limits can be imposed on specific users whenever Twitter wishes and tweets can be removed at any point in time [**twitterTwitterTermsService2021**].

This project intends to create a cryptographic and deniable cloud-based filesystem called the *Fejk FileSystem* () which takes advantage of free online web services, such as Twitter and Flickr, for the actual storage. The idea is to save the user's files by posting an encrypted version of the file as images and text posts these web services. The intention is not to create a revolutionary fast and usable filesystem but instead to explore how well it is possible to utilize the storage that Twitter and similar services provide their users for free, as a cryptographic and deniable cloud-based filesystem. Additionally, the performance and limits of this filesystem will be analyzed and compared to alternative filesystems, such as Google Drive, to compare the advantages and disadvantages of the developed filesystem compared to professional filesystems. The security of the filesystem will also be discussed, as well as an analysis of the steganographic capability of the developed filesystem.

1.1 Problem

Current cryptographic filesystems are mainly based on local-disk solutions, and while services such Google Drive might encrypt your data, it can be considered unsafe storage as they might give out your data. A cryptographic and deniable decentralized cloud-based filesystem where the data is not controlled by any entity other than the user can be of importance, for instance for journalists in unsafe countries. Social media services often provide free storage which makes it a potentially good host of the data in such a filesystem as they would not be able to access the unencrypted data nor have any idea how the posts are connected, and it might even go unnoticed due to their constant heavy load of data from regular users of the services. Is it possible to exploit the storage on various social media services to create a cryptographic and deniable filesystem where the data is stored on these online web services through the use of free user accounts? What are the drawbacks of such a filesystem compared to similar filesystem solutions with regards to write and read speed, storage capacity, and reliability? Are there advantages to such a filesystem in regards to security and deniability?

1.2 Purpose and motivation

The purpose of this research is to explore the possibility to create a secure, steganographic cloud-based filesystem that stores data on online services and to compare the performance, benefits, and disadvantages of such a filesystem to existing steganographic filesystems and distributed filesystem services. A distributed filesystem service, such as Google Drive, provide data storage for users which can be both free and cost money. Even though Google Drive encrypts the user's data, they control the encryption and decryption keys, and the method of encryption [**johnsonGoogleDriveSecure2021**]. This means that they can give out the user's files and data if faced with legal actions such as subpoenas. It also opens up the possibility of hackers gaining access to the files without the user having any way to control them.

The idea behind FFS is to have a decentralized cloud-based filesystem were only the user is able to control the unencrypted data. By encrypting and decrypting the files locally before uploading and after downloading them to these services (end-to-end encryption), it is possible to make sure that the user is the only one who has access to the encryption and decryption keys and therefore the unencrypted data. Even if the web service would look at the data uploaded by the user, it is not readable without the decryption key. An interesting aspect of this is that online web services, such as social media, provide users with essentially an infinite amount of storage for free. Anyone can create any number of accounts on Twitter and Facebook without cost, and with enough accounts, one could potentially store all their data using such a filesystem. We aim to exploit the storage web services give their users for free. As the file data is only accessible by the user, and as the filesystem can be unmounted to hide its existence, it is steganographic.

There are several steganographic filesystems available but these lack certain aspects that FFS aims to solve. Some filesystems are based on the local disk of the device in use, such as the physical storage device on a computer or phone, or an external storage device connected to a computer or phone. While these filesystems have advantages compared to cloud-based solutions, such as latency, they lack accessibility as you need to have the device to access the content on it. It also means that when you want to share or transport the data, you must physically move the device which can mean problems as it could for instance be taken from you or be destroyed. Cloud-based solutions counter this by being available from any location that has internet access to the services used. However, existing cloud-based solutions introduce other disadvantages. One example is CovertFS [**baliga2007web**] where data is stored in images posted on web services. The images are actual images representing something, meaning that there's a limit on how much steganographic data can be stored. CovertFS limit this to 4 kB which means that such a filesystem with a lot of data will require many images which could lead to suspicion from the owners of the web services. More examples of filesystems similar to the idea will be presented in Chapter **??**.

1.3 Goals

The project aims to create a secure, deniable filesystem that stores its data on online web services by taking advantage of the storage provided to its users. This can be split into the following subgoals:

1. to create a mountable filesystem where files and directories can be stored, read, and deleted,
2. for the filesystem to store all the data on online web services rather than on a local disks,
3. for the system to be secure in the sense that even with access to the uploaded files and the software, the data is not readable without the correct decryption key,
4. to provide the user of the filesystem with plausible deniability of its data in the sense that it is not possible to associate the user with FFS if the filesystem is not mounted,
5. to analyze the write and read speed, storage capacity, and reliability of the filesystem and compare it to commercial cloud-based filesystems and local filesystems, and,
6. to analyze and discuss environmental and ethical aspects of the filesystem.

1.4 Research Methodology

The filesystem created through this thesis will be developed on a Macbook laptop running macOS Monterey, version 12.3.1. It will be written in C++20 and use the Filesystem in Userspace (FUSE) MacOS library [**HomeMacFUSE**] which enables the writing of a filesystem in userspace rather than in kernel space. FUSE is available on other platforms too, such as Linux, but the filesystem will be developed on a Macbook laptop thus macFUSE is chosen. C++ is chosen because the FUSE API is available in C, and C++ version 20 is well established and used. Further details about the development environment will be found in Section **??**.

The resulting filesystem will be evaluated against other filesystems, both commercial distributed systems, such as Google drive, and an instance of Apple File System (APFS) [**appleinc.AppleFileSystem**] on the Macbook laptop referenced above. Quantitative data will be gathered from the different filesystems through the use of experiments with the filesystem benchmarking software IOzone [**IozoneFilesystemBenchmark**]. IOzone was chosen because it is, compared to tools such as Fio and Bonnie++, simpler to use while still powerful [**agarwalComparingIOBenchmarks2018**]. We will look at attributes such as the differences in read and write speeds between different filesystems, as well as the speed of random read and random write. However, according to **tarasovBenchmarkingFileSystem2011**, benchmarking filesystems using benchmarking tools is difficult to perform in a standardized way [**tarasovBenchmarkingFileSystem201** which will be taken into consideration during the evaluation and when concluding the thesis. Further discussion about this will be found in Section **??**.

1.5 Delimitations

Due to limitations in time and as the system is only a prototype for a working filesystem and not a production filesystem, some features found in other filesystems are not going to be implemented in FFS. The focus will be to implement a subset of the POSIX standard functions, containing only crucial functions for a simple filesystem, specifically, the FUSE functions *open*, *read*, *write*, *mkdir*, *rmdir*, *readdir*, and *rename*. However, file access control is not a necessity and will therefore not be implemented, thus functions such as *chown* and *chmod* are not going to be implemented. The reason is that the goal is to present and evaluate the possibility of creating a secure steganographic filesystem with a storage medium based on online web services and thus FFS will only aim to implement a minimal filesystem.

There is also an argument that could be made that FFS should support multiple users so that anyone can mount the filesystem but only browse their files. However, as this project is only a proof-of-concept of the filesystem, this will not be implemented. Instead, FFS will be built for single-user support where only a password will

unlock everything FFS is storing. This means that anyone who mounts FFS with the password will access everything that other users might have stored.

1.6 Structure of the thesis

Chapter **??** presents theoretical background information of filesystems and the basis of FFS while Chapter **??** mentions and analyzes related work. Chapter **??** describes the implementation and the design choices made for the system, along with the analysis methodology. Chapter **??** presents the results of the analysis and Chapter **??** discusses the findings and other aspects of the work. Lastly, Chapter **??** will finalize the conclusion of the thesis and discuss potential future work.

Chapter 2

BACKGROUND

This chapter presents concepts and information that is relevant for understanding, implementing, and evaluating FFS. We first present the idea of inode-based filesystems and how data is stored in a filesystem. Following is the introduction of Filesystem in Userspace () which will be used to implement the filesystem. Later sections present background information about Twitter and the potential threat adversaries of the filesystem.

2.1 Filesystems and data storage

This section presents how certain filesystems used today are structured. We present the idea of inode-based filesystems and distributed filesystems. Following, we describe how data is stored in a storage system and how this information can be used in FFS.

2.1.1 Unix filesystems

A Unix filesystem uses a data structure called an *inode*. The inodes are found in an inode table and each inode keeps track of the size, blocks used for the file's data, and metadata for the files in the filesystem. A directory simply contains the filenames and each file or directory's inode id. The system can with an inode id find information about the file or directory using the inode table. Each inode can contain any metadata that might be relevant for the system, such as creation time and last update time.

Figure **??** shows an example inode filesystem and how it can be visualized. The blocks of an inode entry are where in the storage device the data is stored, each block is often defined as a certain amount of bytes. Listing **??** describes a simple implementation of an inode, an inode table, and directory entries.

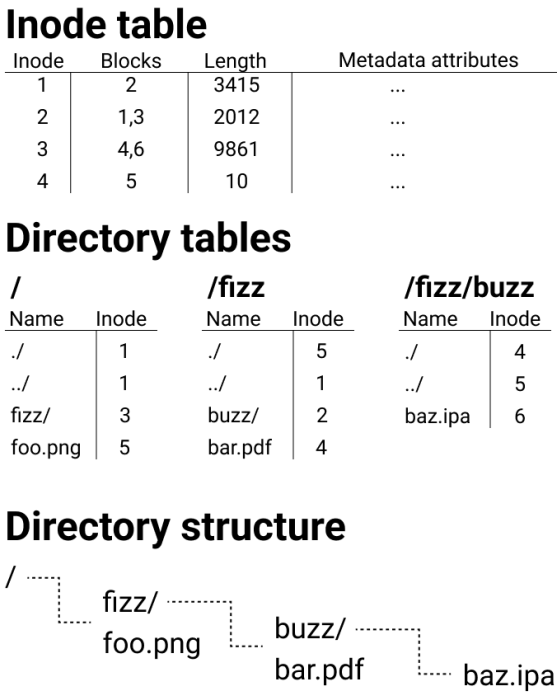

Figure 2.1: Basic structure of inode-based filesystem

Listing 2.1: Pseudocode of a minimalistic inode filesystem structure

```
struct inode_entry {
        int     length
        int[]   blocks
        // Metadata attributes are defined here
}

struct directory_entry {
        char*   filename
        int     inode
}

// Maps inode_id to a inode_entry
map<int, inode_entry> inode_table
```

Different filesystems provide different features and limitations. The Extended Filesystem (ext) exists in four different versions: ext, ext2, ext3, and ext4. This filesystem is often used on Unix systems. Each iteration brings new features and changes the limitations. For instance, comparing the two latest iterations, ext3 and ext4, ext4 can theoretically store files up to 16 TiB while ext3 can store files up to 2 TiB [**salterUnderstandingLinuxI**
Additionally, ext4 supports timestamps in units of nanoseconds while et3 only supports timestamps with a resolution of one second. Additionally. ext4 natively supports encryption at the directory level through the use of the fscrypt API [**FscryptArchWiki**].

The Apple Filesystem () is a modern filesystem that is used on iPhones and Mac and can store files with a size up to 9 EB [**igotofferAPFSAppleFile2017**]. It supports timestamps in units of nanoseconds and is built to be used on solid-state drives () [**nelsonWhatAPFSDoes**]. It also supports modern features that its predecessor Mac OS Extended (HFS+) does not support, such as Snapshots and Space Sharing. APFS natively supports encryption of the filesystem volume [**appleinc.FileSystemFormats**].

2.1.2 Distributed filesystems

Filesystems are used to store data, for instance locally on a hard drive of a computer, or in the cloud. Google Drive is an example of a filesystem that enables users to save their data online with up to 15 GB for free [**CloudStorageWork**] using Google's clusters of distributed storage devices, meaning that the data is saved on Google's servers which can be located wherever they have data centers [**DistributedStorageWhat**]. Paying customers can have a greater amount of storage using the service. Apple's iCloud and Microsoft's OneDrive are two additional examples of distributed filesystems where users have the option of free-tier and paid-tier storage.

Cloud-based filesystems, as opposed to a filesystem on a physical disk, are accessible from multiple computers and devices without requiring the user to connect a physical disk to the computer. Instead, as the filesystem is accessible through the internet, it can be accessed regardless of the user's location and on multiple devices, as long as a connection to the filesystem can be established. Thus, even if the user would lose their computer or if it would malfunction, the data on the cloud-based

filesystem can still be accessed which means that the data could still be recovered. These filesystems are often owned by companies, such as Google Drive and Apple's iCloud, as they are big companies that can provide reliable storage. This also means that they have their own agenda and policies, and as they are hosting the data they have the possibility of accessing your data. The data is often encrypted, but in the case of Google Drive they have access and control of the encryption and decryption keys which in turn means that they have access and control of the data stored [**johnsonGoogleDriveSecure2021**]. While they mention in their Terms of Service that the user retains ownership of the data [**googleGoogleDriveTerms**], they also mention that they can disclose your data for legal reasons and that they retain the right to review the content uploaded by users [**googleGoogleTermsService**]. By them controlling the encryption and decryption keys, it also enables the possibility of hackers gaining access to your data by attacking Google. iCloud uses end-to-end encryption for some parts of the service, but not for the whole suit [**appleinc.ICloudSecurityOvervie**
For instance, backup data and iCloud drive is not end-to-end encrypted while the Keychain and Memoji data is.

2.1.3 Data storage and encoding

Different file types have different protocols and definitions of how they should be encoded and decoded, for instance, a JPEG and a PNG file can be used to display similar content but the data they store is different. At the lowest level, storage devices often represent files as a string of binary digits no matter the file type (however there are non-binary storage devices [**MultistateDataStorage2020**], but this is outside the scope of this thesis). If one would represent an arbitrary file of X bytes, each byte (0x00 - 0xFF) can be represented as a character such as the Extended ASCII (EASCII) keyset and we can therefore decode this file as X different characters. Using the same set of characters for encoding and decoding we can get a symmetric relation for representing a file as a string of characters. EASCII is only one example of such a set of characters, any set of strings with 256 unique symbols can be used to create such a symmetric relation, for instance, 256 different emojis or a list of 256 different words. However, if we are using a set of words we would also have to introduce a

unique separator so that the words can be distinguished. If we would use a single space character as the separator, we could make the encoded text look like a text document; however, random words one after another lead to a high probability of creating an unstructured text document. Further, if punctuation is introduced, for instance as part of some words, the text document could look like it contains random and unstructured sentences.

This string of X bytes can also be used as the data in an image. An image can be abstracted as a $h * w$ matrix, where each element is a pixel of a certain color. In an image with 16-bit Red-Green-Blue () color depth, each pixel consists of three 16-bit values, i.e. three pairs of bytes. One can therefore imagine that we can use this string of X bytes to assign colors in this pixel matrix by assigning the first two bytes as the first pixel's red color, the next two bytes as the same pixel's color green color, and so forth. The seventh and eight byte would represent the second pixels red color. This means that X bytes of data can be represented as

$$ceil(\frac{X}{2 * 3})$$

pixels, where *ceil* rounds a float to the closest larger integer. For a file of 1 MB, i.e. $X = 1\,000\,000$ we need 166 667 pixels in an image with 16-bit RGB color depth. The values of h and w are arbitrary but if we for instance want a square image we can set $h = w = 409$ which means that there will be 167 281 pixels in total, and the remaining 614 pixels will just be fillers to make the image a reasonable size. Using filler pixels requires us to keep track of the number of bytes that we store in the image so that we do not read the filler bytes when the image is decoded. However, we could choose $h = 1$ and $w = 166\,667$ which would mean a very wide image but would not require filler pixels. The string of bytes X is referred to as the Pixel Color Data ().

This means that we can represent any file as a string of bytes which can then be encoded into text or as an image, which can be posted on for instance social media. However, there is a possibility that the social media services compress the images uploaded which could lead to data loss in the image, which would mean that the decoded data would be different from the encoded data. In this case, we would not

be able to retrieve the original data that was stored unless we would use methods such as error correcting codes.

2.2 FUSE

Filesystem in Userspace (FUSE) is a library that provides an interface to create filesystems in userspace rather than in kernel space which is otherwise often considered the standard when writing commercial filesystems [**Libfuse2021**]. The reason to implement a filesystem in kernel space is that it leads to faster system calls than when writing a filesystem in userspace. However, while filesystems written with FUSE are generally slower than a kernel-based filesystem, using FUSE simplifies the process of creating filesystems. macFUSE is a port of FUSE that operates on Apple's macOS operating system and it extends the FUSE API [**HomeMacFUSE**]. macFUSE provides an API for C and Objective C.

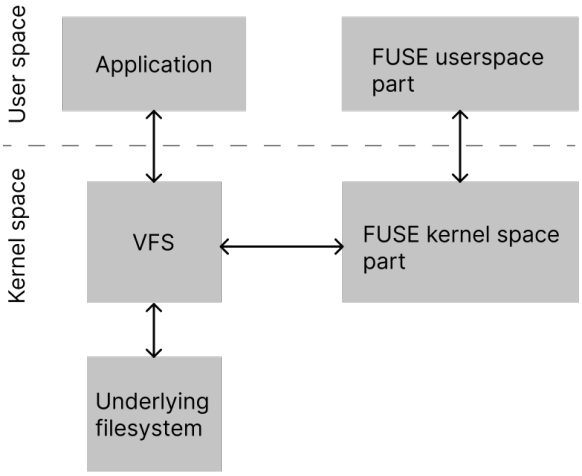

Figure 2.2: Simple visualization of how FUSE operations are executed

Figure **??** shows an overview how FUSE works. FUSE consists of a kernel space part and a userspace part that perform different tasks [**vangoorFUSENotFUSE2017**]. The kernel part of FUSE operates with the Virtual Filesystem () which is a layer in both the Linux kernel and the macOS kernel that exposes a filesystem interface for userspace applications [**goochOverviewLinuxVirtual**, **singhMacOSInternals2006**]. The VFS interface is independent of the underlying filesystem and is an abstraction

of the underlying filesystem operations which can be used on any filesystem the VFS supports. The userspace part of FUSE communicates with the kernel space part through a block device. Operations on a mounted FUSE filesystem are sent to the VFS from the user application, which is then sent to the kernel part of FUSE. If needed, the operations are transmitted to the userspace part of FUSE where the operation is handled and a response is sent back to the VFS and the user application through the FUSE kernel module. However, some actions can be handled by the FUSE kernel module directly, such as if the file is cached in the kernel part of FUSE [**vangoorFUSENotFUSE2017**]. The response is then sent back to the user application from the kernel module through the VFS.

2.3 Online web services

This section presents two online web services (OWSs), Twitter and Flickr, where one can create free-tier accounts. On both of these OWSs, free-tier accounts can make numerous of posts for free. The OWSs provide free-to-use Application Programming Interfaces (APIs) for non-commercial development.

2.3.1 Twitter

Twitter is a micro-blog online where users can sign up for a free account and create public posts (tweets) using text, images, and videos. Each post has a unique id associated with it [**twitterTwitterIDs**]. Text posts are limited to 280 characters while images can be up to 5 MB and videos up to 512 MB [**MediaBestPractices**]. An post with images can contain up to 4 images in one post. There is also a possibility to send private messages to other accounts, where each message can contain up to 10 000 characters and the same limitations on files. However, direct messages older than 30 days are not possible to retrieve through Twitter's API [**RetrievingOlder302018**]. It is possible to create threads of Twitter posts where multiple tweets can be associated in chronological order.

Twitter's API defines technical limits of how many times certain actions can be executed by a user [**UnderstandingTwitterLimits**]. A maximum of 2 400 tweets can be sent per day, and the limit is further broken down into smaller limits at semi-hourly intervals. Hitting a limit means that the user account no longer can perform the actions that the limit represents until the time period has elapsed.

2.3.2 Flickr

Flickr is a public image and video hosting service, used to store and share photos and videos. Unlike Twitter, a post on Flickr is based on the image or video. The post can, optionally, have a title, a description, or both. However, the post must have exactly one photo or video. Flickr supports multiple image and video formats, including PNG and MP4 [**FlickrUploadRequirements2022**]. Restrictions are set for each post, depending on the media type. Images uploaded to Flickr can be a maximum of 200 MB and a video can be maximum of 1 GB. Further, free-tier accounts can only have total of 1 000 photos or videos on their account. A Flickr Pro account has unlimited storage on Flickr, but is still subject to the per-item limit of 200 MB and 1 GB for images and videos, respectively [**flickrinc.UpgradeEverythingYou**]. Flickr Pro costs between 7.49€ to 5.49€ per month, depending on the subscription time the user signs up for. The description of a post has a limit of 65535 characters according to Shhexy Corin [**FlickrHelpForum2009**]. This has been verified through testing. The title of a post has also been discovered to have a limit of 255 characters through testing. A post can also be tagged with a numerous of custom tags. Users can search for posts with a certain tag using the web interface or the API.

The images and videos uploaded to Flickr is stored in its original form **without any compression**, and can be downloaded by the user as the same file as was uploaded[**flickrinc.DownloadPermissions**]. Flickr also stores other formats of the file, such as thumbnails. User accounts can restrict who, other than themselves, can download the original image. The original video can only be downloaded by the user [**flickrinc.DownloadPermissions**]. Flickr do not state if it will always be possible to download the original versions of the file. Further, Flickr states that it retains the right to remove user content from the service at any time [**flickrinc.FlickrTermsConditions20**

The Flickr API defines a query limit of 3 600 requests per hour, per application, across all API calls [**flickrinc.FlickrFlickrDeveloper**]. However, according to Sam Judson in 2013, this is not a hard limit [**WhatAreAPI2013**]. There is no official information from Flickr of what happens if you break the hourly request limit. The Flickr API states that the API is monitored on other factors as well [**flickrinc.FlickrFlickrDeveloper**]. If abuse is detected, Flickr reserves the right to revoke API keys.

2.4 Cryptography

The Advanced Encryption Standard () is a encryption standard established by the U.S. National Institute of Standards and Technology (NIST), more specificity specifying the Rijndael block cipher [**kumarvermaPerformanceAnalysisRC62012**]. AES is a symmetrical cipher, meaning that the same key is used for encryption and decryption. AES is used to make the data confidential, so that no one except the person with the key can access the unencrypted data. AES produces 128-bit encrypted cipher blocks, and supports key sizes of 128 bits, 192 bits, or 256 bits. The security of AES has been heavily researched since its introduction in the early 2000s, and literature has found it is well resistant to quantum attacks as well [**bonnetainQuantumSecurityAnalysis2019**].

While AES is a good standard for the confidentiality of the data, confidentiality is often not enough to secure the data [**rosswallrabensteinWhenItComes2021**]. Importance of ensuring the authenticity of the data is also high. This means that we want to know that the data has not been modified since it was encrypted. This problem can be solved by using authenticated encryption [**khovratovichAnswerWhyShould2013**]. The Galois/counter mode () is a block cipher mode of operation which provides authenticated encryption [**mcgrewGaloisCounterMode2004**]. GCM can be used with AES to provide secure, authenticated encryption of data. To encrypt using GCM, the encryption function requires a key, a randomized Initialization Vector (IV) and the data to encrypt. The output is the encrypted cipher text and an authentication tag. The decryption function of GCM requires the same key and IV as was used as input in the encryption function, as well as the authentication tag and the

cipher text received as output by the encrypting function. Further, both the encryption function and the decryption support Additional authentication data (ADD) to be provided. ADD is data that should be authenticated, but not encrypted. If ADD is provided to the encryption function, it must also be provided to the decryption function.

The key used when encrypting using AES is often derived from a password that the user provides. Password-Based Key Derivation Functions (PBKDFs) are functions that can be used to derive a key used for, for instance, AES. The input to a PBKDF is a secret, such as a password [**kodwaniSecurityKeyDerivation2021**]. An example of a PBKDF schema is the hashed message authentication code () based key derivation functions () presented by **krawczykCryptographicExtractionKey2010** [**krawczykCryptograp** which utilizes a hashing algorithm that provide a pseudo-random key. HKDF supports multiple hashing algorithms. The security of HKDF is partially dependent on the security of the hashing algorithm used. A well-defined suit of hashing algorithms is the Secure Hash Algorithms (), which covers, among other hash functions, SHA-256 [**hansenUSSecureHash2011**]. SHA-256 is a cryptographic hash function which outputs a 256-bit pseudo-random cipher from its input, which can, for instance, be a password. Further, HKDF uses a salt to improve the security of the provided secret. The salt is random data used to further diffuse the produced key, making two keys with the same secret but different salts, different [**ariasAddingSaltHashing2021**]. The salt does not have to be secret, and is sometimes stored with the produced cipher so that the decryption function easily can re-use the salt when deriving the decryption key. If the key used for encryption and the key used for decryption are derived using different salts, the keys will differ and the cipher cannot be decrypted.

2.5 Threats

To consider a filesystem secure it is important to imagine different potential adversaries who might attack the system. Considering that FFS has no real control of the data stored on the different services, all the data must be considered to be stored in an insecure system. Even if we could hide the posts made on for in-

stance Twitter by making the profile private, we must still consider that Twitter themselves could be an adversary or that they could potentially give out information, such as tweets or direct messages, to entities such as the police. Twitter's privacy policy mentions that they may share, disclose, and preserve personal information and content posted on the service, even after account deletion for up to 18 months [**TwitterPrivacyPolicy**]. Therefore, to achieve security the data stored must always be encrypted. We assume that an adversary has access to all knowledge about FFS, including how the data is converted, encrypted, and posted. We also assume they know which websites and accounts could post data from the filesystem - but we assume they do **not** have the decryption key. There are multiple secure ways of encrypting data, including AES which is one of the faster and more secure encryption algorithms [**mahajanStudyEncryptionAlgorithms2013**]. However, even though the data is encrypted, other properties such as your IP address can be compromised which can expose the user's identity. The problem of these other sources of information external to FFS is not addressed in FFS but remains for future work.

Other than adversaries for FFS, we might also imagine that the underlying services might face attacks that can potentially harm the security of the system or even cause the service to go offline, potentially indefinitely. One solution is to use redundancy - by duplicating the data over multiple services, we can more confidently believe that our data will be accessible as the probability of all services going offline at the same time is lower.

The deniability of FFS is an important aspect of the filesystem. Potential threat adversaries are agents that the user is trying to hide the data from, such as governing states. For the system to be deniable, an adversary should not be able to gain any information about anything about the potential data in the system, this includes even the existence of data. When the filesystem is unmounted there should be no trace of the filesystem ever being present in the device. We will assume that an adversary is competent and can analyze the software and hardware completely.

Chapter 3

RELATED WORK

The research area of creating filesystems to improve security, reliability, and deniability is not new and has been well worked on previously. This chapter presents previous work that is related to this thesis. This includes other filesystems that share similarities with the idea of FFS, for instance within the idea of unconventional storage media and the area of steganography.

3.1 Steganography and deniable filesystems

Steganography is the art of hiding information in plain sight and has been around for ages. Today, a major part of steganography is hiding malicious code in for instance images, called stegomalware or stegoware. Stegomalware is an increasing problem and in a sample set of examined real-life stegomalware, over 40% of the cases used images to store the malicious code [**stichtingcuingfoundationSIMARGLStegwarePrimer2020**]. While FFS will not include malicious code in its images, this stegomalware problem has fostered the development of detection techniques of steganography in for instance social media, and it is well researched.

Twitter has been exposed to allowing steganographic images that contain any type of file easily [**TwitterImagesCan**]. David Buchanan created a simple python script of only 100 lines of code that can encode zip-files, mp3-files, and any file imaginable in an image of the user's choosing [**buchananTweetablepolyglotpng2022**]. He presents multiple examples of this technique on his Twitter profile*. The fact that the images are available for the public's eye might be evidence that Twitter's steganography detection software is not perfect. However, it is also possible that Twitter has chosen to not remove these posts.

* `https://twitter.com/David3141593`

Other examples of steganographic data storage on Open Social Networks (OSNs) include the paper presented by **ningSecretMessageSharing2014** where the authors build a system for private communication on public photo-sharing web services [**ningSecretMessageSharing2014**]. Due to the web services processing of uploaded multimedia, they first researched how the integrity of steganographic data could be maintained after being uploaded to these services. Following this, they presented an approach that ensured the integrity of the hidden messages in the uploaded images, while also maintaining a low likelihood of discovery from the steganographic analysis. **beatoUndetectableCommunicationOnline2014** also explores the idea of undetectable communications over OSNs in another paper [**beatoUndetectableCommunicatio**
While implementation is not carried out, they present an idea where messages are encoded together with a cover object and a cryptographic key to produce a steganographic message which is then posted to the OSN. A web-based user interface client with a PHP server backend is presented as the method the users would use to create and share their secret messages.

A steganographic, or deniable, filesystem is a system that does not expose files stored on this system without credentials - neither how many files are stored, their sizes, their content, or even if there exist any files in the filesystem [**andersonSteganographicFileSystem199**
This is also known as a rubber hose filesystem because of the characteristic that the data only can be proven to exist with the correct encryption key which only is accessible if the person is tortured and beaten with a rubber hose because of its simplicity and immediacy compared to the complexity of breaking the key by computational techniques.

3.2 Cryptography

Some papers choose to invent their encryption methods rather than using established standards. **chumanEncryptionThenCompressionSystemsUsing2019** proposes a scrambling-based encryption scheme for images that splits the picture into multiple rectangular blocks that are randomly rotated and inverted, both horizontally and vertically, along with shuffling of the color components [**chumanEncryptionThenCompressionSyst**

This is used to demonstrate the security and integrity of images sent over insecure channels. The paper uses Twitter and Facebook to exhibit this. Despite its improvement and compatibility of a common image format, such as bitstream compliance, due to its well-proven security FFS will use AES as its encryption method.

3.3 Related filesystems

Multiple steganographic filesystems have been presented previously but many of these are focused on filesystems for physical storage disks that the user has access to. For instance, Timothy Peters created DEFY, a deniable filesystem using a log-based structure in 2014 [**petersDEFYDeniableFile2014**]. DEFY was built to be used exclusively on Solid State Drives (SSD) found in mobile devices to provide a steganographic filesystem that could be used on Android phones. Further examples of local disk-based filesystems can be found in [**andersonSteganographicFileSystem1998**, **mcdonaldStegFSSteganographicFile2000**, **domingo-ferrerSharedSteganographicFile200**
hanMultiuserSteganographicFile2010], among other papers. However, this paper aims to create a filesystem that is not based on a physical disk but rather a cloud-based steganographic filesystem that uses online web services as its storage medium.

In 2007, **baliga2007web** presented an idea of a covert filesystem that hides the file data in images and uploads them to web services, named CovertFS [**baliga2007web**]. The paper lacks implementation of the filesystem but they present an implementation plan which includes using FUSE. They limit the filesystem such that each image posted will only store a maximum of 4 kB of steganographic file data and the images posted on the web services will be actual images. This is different from the idea of FFS where the images will be purely the encrypted file data and will therefore not be an image that represents anything but will instead look like random color noise. An implementation of CovertFS has been attempted by **sosaSuperSecretFile2007** which also used Tor to further anonymize the users [**sosaSuperSecretFile2007**].

In **szczypiorskiStegHashNewMethod2016**, **szczypiorskiStegHashNewMethod2016** introduced the idea of StegHash - a way to hide steganographic data on Open Social

Networks () by connecting multimedia files, such as images and videos, with hashtags [**szczypiorskiStegHashNewMethod2016**]. Specifically, images were posted to Twitter and Instagram along with certain permutations of hashtags that pointed to other posts through the use of a custom-designed secret transition generator. StegHash managed to store short messages with 10 bytes of hidden data with a 100% success rate, while longer messages with up to 400 bytes of hidden data had a success rate of 80%. **bieniaszSocialStegDiscApplicationSteganography2017** later presented SocialStegDisc which was a filesystem application of the idea presented with StegHash [**bieniaszSocialStegDiscApplicationSteganography2017**]. Multiple posts could be required to store a single file and each post referenced the next post like a linked list, which means that you only need the root post to read all the data. This is unlike the idea of FFS where a table will be kept to keep track of which posts store a certain file, and in what order they should be concatenated, similar to the idea of an inode table. SocialStegDisc lacks actual implementation of the filesystem but similar to CovertFS presents the idea of a social media-based filesystem.

TweetFS is a filesystem created by Robert Winslow that stores the data on Twitter [**winslowTweetfsTweetfsMaster**], created in 2011. It was created as a proof of concept to show that it is possible to store file data on Twitter. The filesystem uses sequential text posts to store the data. The filesystem is not mounted to the operating system, instead, the user interacts with a Python script through the command line. This makes the filesystem less convenient from a user perspective, compared to a mounted filesystem where the files can be browsed using a user interface or command line. There are two commands available: `upload` and `download` which upload and download files or directories, respectively. Names and permissions of files and directories are maintained throughout the upload and download process. The tweets are not encrypted but are enciphered into English words which makes them look like nonsense paragraphs, similar to what we mentioned in Section **??** about how arbitrary data can be encoded as plain text. This makes the filesystem less secure than an encrypted version as it can be read by anyone with access to the decoder. However, it does introduce a steganographic element to the filesystem.

In 2006, **jonesGoogleHackUse2006** created GmailFS - a mountable filesystem that uses Google's Gmail to store the data [**jonesGoogleHackUse2006**, **jonesGmailFilesystemImple
The filesystem was written in Python using FUSE and was presented well before the introduction of Google Drive in 2012. It does not support encryption as the plain file data is stored in emails. Today, Gmail and Google Drive share their storage quota and GmailFS has since become redundant as Google Drive is an easier filesystem to use. GMail Drive is another example of a Gmail-based filesystem and it was influenced by GmailFS [**viksoeViksoeDkGMail2004**]. GMail Drive has been declared dead by its author since 2015.

Google Conduce Sistem de Fișiere () is a filesystem that stores its data on Google Drive, built using FUSE [**puscassergiudanGCSFVIRTUALFILE2018**, **puscasHarababurelG**
Google Drive also provides a desktop application [**googleInstallSetGoogle**] which presents a mounted volume in the local filesystem, containing the user's Google Drive filesystem. The mountable volume provided by the desktop application does not always sync the stored data directly, but might instead store it locally until a later time. To enable direct synchronization of the data to Google Drive, GCSF interacts with the Google Drive REST API rather than the mounted filesystem volume. One difference between GCSF and the idea of FFS is that GCSF does not encrypt the data stored in the filesystem. While the data is, as mentioned previously, encrypted by Google Drive, the encryption keys are controlled by Google Drive, not the user of GCSF. The data stored on GCSF is also stored as its original files in Google Drive, not as images as FFS intends to store the data. The Google Drive filesystem architecture is utilized by GCSF, for instance by using its directory hierarchy structure. This allows GCSF to avoid creating its own inode table and directory structures, as Google Drive provides the functionality these structures similarly provide FFS, through the Google Drive API.

zadokCryptfsStackableVnode1998 created Cryptfs, a stackable Vnode filesystem that encrypted the underlying, potentially unencrypted, filesystem [**zadokCryptfsStackableVnode**
By making the filesystem stackable, any layer can be added on top of any other, and the abstraction occurs by each Vnode layer communicating with the one beneath. There is a potential to further stack additional layers by using tools such as

FiST [**FiSTStackableFile**]. This approach enables one to create not only an encrypted file system but also to provide redundancy by replicating data to different underlying filesystems. If these filesystems are independent, then this potentially increases availability and reliability. FFS aims to achieve stackability through the use of FUSE.

3.4 Filesystem benchmarking

IOzone is a filesystem benchmarking tool that is used to measure performance and analyze a filesystem [**IozoneFilesystemBenchmark**]. It is built for, among other platforms, Apple's macOS where the filesystem will be built, run, and tested. However, as mentioned previously, filesystem benchmarking is more complicated than one might imagine. Different filesystems might perform differently on small and big file sizes among other things, which means that we can never compare benchmarking outputs as just single numbers. We must instead compare different aspects of the filesystems. In **tarasovBenchmarkingFileSystem2011 tarasovBenchmarkingFileSystem2011** presents a paper where they criticize several papers due to their lack of scientific and honest filesystem benchmarking [**tarasovBenchmarkingFileSystem2011**]. The problem of benchmarking a filesystem is all the different components that are involved when interacting with a filesystem. For instance, they mention how benchmarking the in- and output (I/O) of the filesystem, such as bandwidth and latency, is different from benchmarking on-disk operations, such as the performance of file read and write operations. The benchmarking tools can for instance rarely affect or determine how the filesystem handles caching and pre-fetching. This means that benchmarking the read and write performance of different filesystems can be misleading as they might handle this differently, meaning that the result could be different depending on for instance the distance between the files on the disk. Two files could be adjacent on the disk on one filesystem and therefore one could be pre-fetched into the cache when the other one is read. Considerations about such factors must be present when analyzing the results of the benchmarking.

tarasovBenchmarkingFileSystem2011 also lists several different filesystem benchmarking tools available and used by the papers they reviewed, and how well the tools can analyze certain aspects of a filesystem [**tarasovBenchmarkingFileSystem2011**]. IOZone is listed as being compatible with multiple of the different benchmarking types and as it is simpler to use [**agarwalComparingIOBenchmarks2018**] and still maintained, this was chosen as the benchmarking tool for FFS.

3.5 Summary

As presented, different filesystems provide different features and drawbacks. In Table **??** we display a summary of characteristics and features of some filesystems mentioned above and how FFS compares. As can be seen, FFS mainly lacks certain filesystem operations which are not the focus of FFS as it is a proof of concept.

Table 3.1: Comparison between features present in related filesystems and FFS. X means that the feature is supported and - means that it is not supported

	ext4	**Google drive**	**DEFY**	**TweetFS**	**FFS**
Mountable	X	X	X	-	X
Read/Write/Remove file	X	X	X	X	X
Read/Write/Remove directory	X	X	X	X	X
Hard links	X	-	X	-	-
Soft links	X	-	X	-	-
File and directory access control	X	X	-	X	-
Encrypted	X	X*	X	-	X
Steganographic	-	-	X	X	X
Cloud-based	-	X	-	X	X

*As mentioned, the user has no control over this encryption

Chapter 4

METHOD

This section presents the methodology of implementing FFS and the specifications of the development environment. We also present how the quantitative data used for the evaluation is acquired. Also, the experiments on the filesystems are presented.

4.1 Development environment specification

Development of FFS is done on a 2016 year model Macbook Pro laptop with 2.6 GHz Quad-Core Intel Core i7 processor and 16 GB 2133 MHz LPDDR3 memory. The storage device of the computer is a 250 GB SSD, and the filesystem used is an encrypted APFS partition. The computer runs macOS Monterey 12.5

FFS is developed in C++20 and compiled using Apple clang version 13.0.0 using target x86_64appledarwin21.4.0. FFS uses the ImageMagick Magick++ library [**ImageMagick2022**] for image processing. Version 7.1.029 of Magick++ is used by FFS. macFUSE [**HomeMacFUSE**] version 4.2.5 is used for FFS to use the FUSE API. FUSE API version 26 is used. cURLpp [**barrette-lapierreCURLpp2022**] is a cURL [**CurlCurl2022**] wrapper used by FFS to make HTTP requests. Version 0.8.1 of cURLpp is used by FFS. liboauth [**Liboauth**] version 1.0.3 is used by FFS to sign and encode HTTP request according to the OAuth [**barrette-lapierreCURLpp2022**] standard. Flickcurl [**beckettFlickcurlLibraryFlickr**] version 1.26 is a C library used by FFS to communicate with parts of the Flickr API. Crypto++ [**CryptoLibraryFree**] is a C++ library providing cryptographic schemes. FFS uses Crypto++ to encrypt end decrypt the data stored in FFS, and to derive the keys used in the encryption and decryption algorithm. Crypto++ version 8.6 is used by FFS.

FFS is developed on a single computer for simplicity, and the version used for the operating system, libraries and tools were the most recent up-to-date versions when

development of the filesystem started. To avoid re-writing the source code, these versions will remain the same throughout the development process.

4.2 FFS

The artifact that was developed as a result of this thesis is the Fejk FileSystem (FFS). It uses an online web service () to store the data but behaved as a mountable filesystem for the users. The filesystem is a proof-of-concept and does not support all functionalities that other filesystems do, such as links or access permissions. The reasoning is that these behaviors are not required for a useable system, and when comparing the filesystem to distributed filesystems such as Google Drive, many of these other filesystems also often do not support functionality such as links.

4.2.1 Design overview

FFS uses images to store the data of files, directories and the inode table of the filesystem. These images will are uploaded to the OWS, such as Flickr, as image posts. As mentioned in Section **??**, there can be limitations to these posts for certain OWSs. To support file sizes bigger than these limitations, bigger files will be split into multiple posts, requiring FFS to keep track of a list of posts. Figure **??** presents the basic outline of FFS and a example content of the filesystem. FFS is based on the idea of inode filesystems and uses a inode table to store information about the files and directories in the filesystem. However, instead of an inode pointing to specific blocks in a disk, the inode table of FFS will instead keep track of the id numbers of the posts on the OWS where the file or directory is located. The inode table entry for each file or directory will also contain metadata about the entry, such as its size and a boolean indicating if the entry is a directory or not.

The directories and inode table are represented as classes in C++. Appendix **??** visualizes the main attributes of the `Directory`, `InodeTable`, and `InodeEntry` classes. There can be multiple `Directory` and `InodeEntry` objects in the computers' memory and in the filesystem, but there will only exist one `InodeTable` instance which

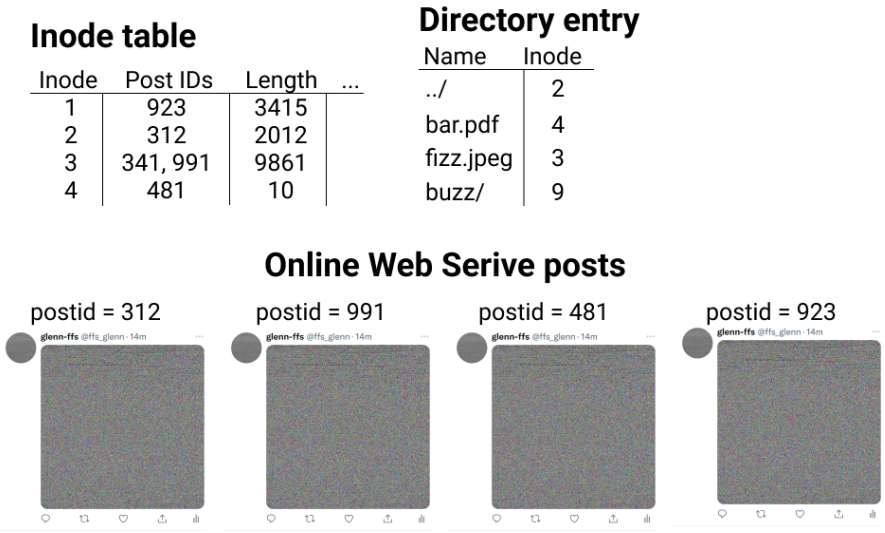

Figure 4.1: Basic structure of FFS inode-based structure

is relevant. The `Directory` class is a data structure that stores mappings between filenames and the files' and directories' inode for all files and directories stored in that directory. The `InodeEntry` is a data structure that keeps track of a file's or directory's information, such as where the data is stored and its metadata, such as size and creation timestamp. The `InodeTable` stores a mapping between an inode and the files' `InodeEntry`, and stores all the `InodeEntry` objects. The `InodeTable` always has at least one entry which is the root directory. This entry has a constant inode value of 0 for simplicity to look up the root directory. With the help of the root directory, all the files lower in the directory hierarchy can be found. The inode of all files and directories other than the root directory has a unique inode greater than 0. The `InodeTable` is saved on the OWS through such means that it can easily be found, for instance by tagging the image post it with a unique string so it can be found by a search.

To read the content of a known file in a directory has three steps:

1. The `Directory` object of the directory provides the inode of the given filename.
2. The inode is used to get the `InodeEntry` from the `InodeTable`.
3. Using the inode entry, the the file can be located.

The location of a file or directory is an ordered list of unique IDs of the image posts on the OWS. The data received by downloading these images, decoding them (as described in Subsection **??**), and concatenating them, can be read as a file or represented as a `Directory` object, depending on if the `InodeEntry` was marked as a file or a directory.

As directories only know the filenames inode, the `Directory` object does not have to be saved again (and thus uploaded) when a file or directory in it is edited, for instance adding data. Only the `InodeEntry`, and thus the `InodeTable`, needs to be updated with the new post IDs of the new file or directory. This saves computation time as every request to the OWS takes time. However, if the filename is edited or the file or directory is moved to another location, the parent directory of the file or directory would have to be edited, and such its corresponding `Directory` object has to be updated.

When a new file or directory is created, it is saved in its parent directory with its filename and an inode. The same inode is used in the inode table to keep track of the file's or directory's inode entry. As shown in Appendix **??**, the inode is represented as a unsigned 32-bit integer. The inode is calculated by adding one to the currently greatest inode. This means that new files and directories will always receive a higher greater inode than the ones currently in the inode table. This naïve approach to inode generation does not take in to account that there might be an available inode less than the greatest inode in the inode table (for instance, due to deletion of a previously created file). However, this inode generation approach is fast and will not be a problem until the integer overflows. As the inode is represented using a 32-bit integer, FFS would need to have saved over four billion files before the inode value would overflow. This scenario is not in the scope of this proof-of-concept filesystem.

FFS does not support all filesystem operations that are implementable through FUSE, instead FFS implements a subset of them. The implemented functions are shown in Table **??**. The implemented operations are the most vital operations required for a working filesystem [**kuenningCS135FUSEDocumentation2010**]. Operations such as `chown` provides extended capabilities of the filesystem but these are not re-

quired for a proof-of-concept filesystem. The functionality of the filesystem operations implemented by FFS and their implementation details are described in Subsection **??**.

Table 4.1: Filesystem operations implementable trough the FUSE API, and wether or not FFS implements them

Filesystem operation	Implemented by FFS
`open`	Yes
`opendir`	Yes
`release`	Yes
`releasedir`	Yes
`create`	Yes
`mkdir`	Yes
`read`	Yes
`readdir`	Yes
`write`	Yes
`rename`	Yes
`truncate`	Yes
`ftruncate`	Yes
`unlink`	Yes
`rmdir`	Yes
`getattr`	Yes
`fgetattr`	Yes
`statfs`	Yes
`access`	Yes
`utimens`	Yes
`readlink`	No
`symlink`	No
`link`	No
`chmod`	No
`chown`	No
`fsync`	No
`fsyncdir`	No
`lock`	No
`bmap`	No
`setxattr`	No
`getxattr`	No
`listxatt`	No
`ioctl`	No
`flush`	No
`poll`	No

A file, a directory, or the inode table has to be uploaded to the OWS when it is modified to save its current information. As it takes time to make requests to the OWS, FFS is created to make as few requests as possible while still saving the data required. Therefor, only the directory or file that is affected by a change is uploaded to the system, while the ones unaffected can remain the same. The inode table has to be updated with every change of a file or directory as it contains the location of the file or directory.

4.2.2 Cache

FFS implements a simple cache for the downloaded content. The cache consists of two data structures:

- a Cache Map - a mapping between a post ID and its image data, and
- a Cache Queue - a queue keeping track of the cached post IDs.

The cache stores a maximum of 20 image posts. To avoid FFS to use too much memory, the cache is configured so that images greater than 5 MB are not cached. Each time an image is uploaded or downloaded, it is added to the Cache Map with its post ID as the key. The post ID is also added to the beginning of the Cache Queue. If the Cache Queue exeeds 20 elements, the last elements of the queue is removed, and the corresponding entry in the Cache Map is erased, thus the entry is fully erased from the cache. The queue ensures that the chache is limited to 20 entries, and by using the first in first out valuation method, the queue also ensures that the oldest element in the cache is removed when the cache exeeds the limit. When a file or directory is removed from the filesystem, all its data is also removed from the cache, if it stored there.

Before a post with a specified post ID is downloaded from the OWS, the cache is checked to see if it is storing this post ID. If it is, the stored image is returned. Otherwise, the process continues by downloading the image from the OWS. When the thesis states that a file or directory is downloaded, it is implied that the cache is

also checked and the data is possibly returned by the cache instead of requiring to download the data from the OWS.

FFS separately caches both the root directory and the inode table. As both of these data structures are used in many of the filesystem operations, it is important that they can be accessed quickly and not be removed from the cache. Their cache entries are updated when the files are uploaded to the OWS.

4.2.3 Encoding and decoding objects

Objects that FFS stores, and therefore also encodes and decodes, are: files, directories, and the inode table. All of these objects are stored on the OWS using PNG images with 16 bit RGB color depth. The inode table and the directories are represented as C++ objects in memory during runtime, but are serialized into a binary representation before they are encoded into images. A detailed description of these binary formats is described in Appendix **??**. The files saved to FFS are also read in to memory in a binary format before being encoded and uploaded to the OWS.

The input to the image encoder is the binary data do encode as an image. A header (FFS header) is prepended to the binary data, containing among other things, the size of the data and a timestamp of when the data was encoded. The FFS header and the input data is encrypted using authenticated encryption, utilizing GCM and AES. The key used for the encryption is derived using the HKDF function utilizing the SHA-256 hashing algorithm, along with a 64 B salt vector, re-generated with random data every time new data is being encrypted. The salt is stored with the cipher to ensure that the decryption algorithm uses the same salt to derive the decryption key. The secret used in the HKDF is a password provided by the user. HKDF also uses a initialization vector, re-generated with random data every time new data is being encrypted. The length of the IV is set to 12 bytes. The resulting data from the encryption is the salt, the IV, the encrypted cipher (including the authentication tag). These three data points are concatenated into a string of bytes. This string of bytes is referred to as the Complete Encrypted Data ().

The dimensions of an FFS image is based on the amount of bytes stored, as described in Section **??**. The stored data is the CED, prepended with the length of the CED () using 4 bytes. For an image of $X = ceil(\frac{4+LCED}{6})$ pixels, FFS will set the width w of the image as $w = ceil(\sqrt{X})$. Further, the height h of the image is set as $h = ceil(\frac{X}{w})$. This will require $(w * h) - X$ filler bytes, and will create an image with similar height and width. For certain values of X, h will be equal to w. For other values of X, $h = w - 1$. The resulting data encoded in the image is, in order:

- 4 bytes representing the LCED,
- The CED data, and
- Filler bytes

The filler bytes are randomized bytes.

The data consisting of the LCED, CED and filler bytes is encoded in to pixel color data for a PNG with 16 RGB bit color depth using the Magick++ library. The result is an image, with a high probability, of what looks like randomized colors for each pixel. This is due to the fact that most pixels are encrypted and therefore the bytes representing this data is seemingly random.

To decode an FFS image, the decoder first interprets the 4 first bytes as the LCED. The salt and IV are retrieved from the CED as they are of known length. The decryption key is derived using the IV and salt, and results in the same key as used in the encryption step because AES is a symmetric cipher algorithm. The remaining bytes of the CED are decrypted using the decryption key. The unencrypted data consists of the FFS header concatenated with the original stored data. The FFS header is asserted to be in the correct format, before the stored binary data is returned from the decryption function. Figure **??** visualizes the encoder and decoder for all data saved in FFS.

The encryption and decryption methods used are state-of-the-art solutions as defined and implemented by Crypto++ [**CryptoLibraryFree**]. Crypto++ is a well-used and well maintained C++ library for cryptography, and as of writing has no reported CVE security vulnerabilities for the functionality used by FFS [**CryptoppSecurityVulnerabilities**].

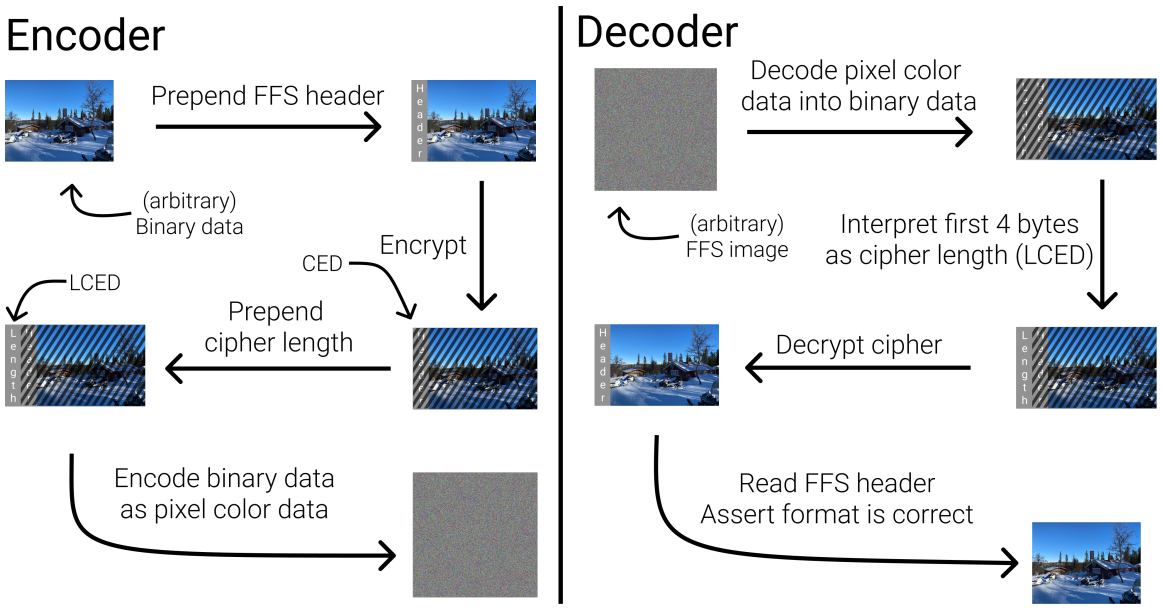

Figure 4.2: Simple visualization of the encoder and decoder of FFS. The input of the encoder is the binary data to store in FFS, eg. a file, and the output is the FFS image to upload to the OWS. The input to the decoder is an FFS image, and the output is the binary data stored on FFS, eg. a file

An FFS image has an upper size limit, defined by the OWS used. This limit is defined further down in this section. If the data to be stored in FFS, such as a file, exceeds this limit, it is split into multiple encoded images. These images will have no association with each other and will be encrypted using different salts and IVs. Only the inode table stores the different post IDs in the order they are encoded in. Files and directories stored in FFS can be separated in to multiple images, however the inode table is limited to only one image for simplicity when interacting with the OWS. This introduces a size limit of the inode table, limiting the filesystem further. More details about the limits are found in Subsection **??**.

4.2.4 Online web services

As FFS is a proof-of-concept filesystem, it only uses one OWS as its storage medium. However, for a production filesystem, multiple OWSs would be beneficial. This would

enable features such as redundancy by using replication over multiple OWSs, for instance in case one OWS would stop working.

The initial intention of FFS was to use Twitter as the OWS. Initial research found that it was possible to upload a file and download the same file without any data loss. However, it was later found that this was not a reliable conclusion. Some images uploaded to Twitter were converted to another image format when they were stored by Twitter, which meant that the decoder could not decode the data as it expected another image format. Other images where compressed or recoded which led to data loss when downloading the image. As the decoder of FFS images relies on a specific binary representation of the image, this meant that the images could not be decoded into the previously uploaded data.

Flickr saves the original version of the uploaded image and thus it can be used to download the same image as was uploaded. This also means that a file that is encoded into an FFS-encoded image can be uploaded, downloaded, and decoded into the same file as before. While they do not assure that they will always support original images, they also do not indicate that this would change. Therefore, Flickr can be used at this moment for the proof-of-concept filesystem that FFS is. A free-tier Flickr account is therefore used for FFS.

Flickr provides an extensive free REST API for non-commercial use. A user can create applications and generate access tokens for the application. These application tokens are later used to request tokens from users who authenticate using Flickr's web interface, and allow the application to do requests for the user. The application will then receive access tokens for the user, which are used to authenticate with the API for the API calls that require authentication.

As mentioned previously, Flickr provides the ability to tag an image with a number of custom tags. Every tag is a string that the user defines. By tagging the image containing the data of the inode table with a pre-defined string we can identify the image in the OWS easily. The Flickr API supports searching for images with a specific tag, uploaded by a specific user. This allows FFS to easily find the images tagged with the pre-defined inode table tag on Flickr, uploaded by FFS. Further, ensuring

that only the inode table is tagged with this tag, and by deleting the old version of the inode table on Flickr when a new version is uploaded to Flickr, we can ensure a singelton pattern of the inode table in the OWS.

While the Flickr API is extensive in its functionality, FFS only uses a few of the provided capabilities. The Flickr API capabilites that FFS utilizes are:

- Upload an image, optionally with a tag, and return a post ID,
- Search for a tag of an image, uploaded by a specific user, and return the URL to the original uploaded image,
- Get the URL to the original uploaded image given a post ID, and
- Remove an image given a post ID.

To download the original image given a post ID or a tag, two requests are required:

- Getting the URL to the original image using the post ID or tag,
- Downloading the image from the URL received from the previous request.

The second request does not require authentication as the previous request returns a public URL to the image.

For benchmarking purposes, FFS also provides the possibility to compile the filesystem to use a Fake OWS (), which stores the data on the local filesystem. The FOWS is used by FFS similar to how Flickr is used, by storing encoded images in it. By storing the images on the local filesystem, the filesystem operations require much shorter execution time as the local filesystem operations are in general faster than the network requests. This makes it easier to conclude how much of the filesystem operation time is affected by the time of the network requests. The time T of an FFS filesystem operation can be modeled like:

$$T = t_{ffs} + t_{ows}$$

where t_{ffs} is the time that FFS takes to, for example for a file read operation;

- to find the file in the inode table,

- decode and decrypt the image data,
- read the specified amount of data, and,
- to output the data

This time will be approximately consistent for the same request. However, cache misses/hits in the filesystem and process scheduling can fluctuate the value of t_{ffs}. t_{ows} is the total time required to complete all requests to the OWS for a filesystem operation. For instance, for a similar read operation as above;

- to download all the directories in the file path,
- find the inode table image given the tag,
- download the inode table, and,
- to download the images representing the file to read

Depending on the OWS, the latency and bandwidth of the internet connection between the user's machine and the OWS's server can differ a lot. Duplicate requests to the same OWS can also differ significantly due to, for instance, server load balancing and a difference in request quantity from other users at the time of the requests. However, for a FOWS, t_{ows} can be replaced by t_{fows} which will have approximately consistent values for duplicate operations, because the local filesystem is not affected as much by load balancing. The local filesystem requests by other applications on the machine can also be influenced and minimized by not using other applications on the machine while running the benchmarking tool to ensure filesystem requests by the FOWS can be handled quickly by the operating system. However, t_{fows} is affected by, among other things, the underlying storage device of the local filesystem and process scheduling which can still fluctuate the value of t_{fows}.

4.2.5 Implemented filesystem operations

Following is a detailed description of all the FUSE operations implemented by FFS, and how they are implemented by FFS. Further explanations about the intended functionality of the operations can be found in [**kuenningCS135FUSEDocumentation2010**].

The path of a file is sometimes provided for the filesystem operation and traversed by FFS to understand the requested location. An example path is `/foo/bar/buz.txt` or `/foo/bar/baz/`. A path is traversed like the following pseudo code:

Listing 4.1: Pseudocode of traversing a given path, returning the `Directory` and the filename

```
# Traverse a given path and return the parent directory
    object
#  and filename of the path
traverse_path(path) -> (Directory, string):
        # Fetches inode table from the OWS
        inode_table := get_inode_table()

        split_path := path.split("/")
        # The filename could be either the name of a file
        #  or the name of a directory
        filename := split_path.last
        dirs := split_path.remove_last()

        # Get the root dir from cache
        curr_dir = cache.get_root_dir()

        # While there are still directories to traverse,
        #  get the next directory in the list from current
        #  directory
        while(!dirs.empty())
                dir_name := dirs.pop_first()
                inode := curr_dir.inode_of(filename=dir_name)
                inode_entry = inode_table.entry_of(inode=
                    inode)
                # Download the image posts defined by the
                #  post IDs in the inode entry
                curr_dir = download_as_dir(inode_entry)
```

```
return (curr_dir, filename)
```

By traversing a path, FFS has to fetch all parent directories in the hierarchy. The file or directory with the filename is not fetched during while traversing the path, as it might not be necessary for the operation. This implies that all operations that relies on the path of the file or directory has to download all parent directories of the path. However, the directories in the path could be cached and therefore not require a download from the OWS. Further, `open`, `opendir`, and `create` can associate a file handle with a file or directory, so that certain other operations can use the file handle instead of traversing the string path. This saves time because the path traversing result is saved in the filesystem state.

After every operation that modifies the inode table, the inode table is uploaded to the OWS and cached. Therefore, it is assumed that the inode table is always up to date in memory and on the OWS. This will be true as long as there are not multiple FFS instances working with the same OWS account at the same time. This scenario has undefined behavior as there is no locking implemented for FFS.

All filesystem operations are synchronous unless specified. Further, FUSE is running in single-thread mode meaning that a filesystem operation call must complete before another can begin. This helps limiting the risk of data races as two processes cannot call different operations that, for instance, modify the inode table at the same time.

4.2.5.1 open

Given a path to a file, the file is associated with a file handle. The file handle is used in subsequent operations to avoid traversing the filepath. The file is not downloaded from the OWS, only the parent directories are downloaded during the path traversing as explained above. An `open` call must, eventually, be followed by a `release` call. Although, multiple other operation calls can occur between these events.

4.2.5.2 release

Given a file handle, this operation closes the file in the filesystem, disassociating the file handle with the file. The current states of the file and the inode table are saved to the OWS, and the previous versions of the file and inode table are deleted from the OWS. Subsequent operations for the file will require path traversing as the file handle can no longer be used.

The file must have a file handle associated with it before `release` is called. This requires a preceding `open` or `create` call for the file.

4.2.5.3 opendir

Given a path to a directory, the directory and associated with a file handle. The file handle is used in subsequent operations to avoid traversing the filepath. The directory is not downloaded from the OWS, only the parent directories are downloaded during the path traversing as explained above. An `opendir` call must, eventually, be followed by a `releasedir` call. Although, multiple other operation calls can occur between these events.

4.2.5.4 releasedir

Given a file handle, this operation closes the directory in the filesystem, disassociating the file handle with the directory. The current states of the directory and the inode table are saved to the OWS, and the previous versions of the directory and inode table are deleted from the OWS. Subsequent operations for the file will require path traversing as the file handle can no longer be used.

The directory must have a file handle associated with it before `releasedir` is called. This requires a preceding `opendir` call.

4.2.5.5 create

This operation creates an empty file in the filesystem given a path, and associates a file handle with the file, similar to `open`. The empty file will not be uploaded to the OWS as it has no data associated with it. A new entry is added to the parent directory with the filename and a generated inode, and the parent directory is uploaded to the OWS. The new posts representing the parent directory in the OWS is associated with the inode entry of the parent directory in the inode table, and the old posts are deleted in the OWS. An new inode entry is also created in the inode table, representing the new, empty, file.

4.2.5.6 mkdir

This operation creates an empty directory in the filesystem given a path. The directory is not uploaded to the OWS as it has no data associated with it. The parent directory is modified so it is uploaded to the OWS, and the old versions of the parent directory is deleted on the OWS. The parent directory entry in the inode table is modified with the new posts, and a new entry is created for the new directory. The inode table is updated in the OWS.

As opposed to `create` for files, this operation does not associate a file handle with the directory.

4.2.5.7 read

This operation reads a number of bytes, starting from a set offset, from the file specified by the file handle. The data is read into a provided buffer. The full file is downloaded and read into memory, even if just a small part of the file is requested. The file is also cached so that subsequential requests for the same file are faster.

4.2.5.8 readdir

This operation reads the filenames inside the directory specified by a file handle. The result includes all filenames in the directory, and the special `"."` and `".."` directories.

4.2.5.9 write

This operation writes s bytes, starting at the provided offset o, to the existing file at the provided file handle. All the data of the current file is read in to memory. Starting from the offset, the new data overwrites the current data of the file, until s bytes have been written. If $o + s$ is greater than the file's size, the file size is set to $o + s$. If $o + s$ is less than the file's size, the data from position $o + s$ and forward remains the same, and the file size is not modified. See Figure **??** for a visualization of the result of a `write` operation given different offsets. The parent directory does not have to modified.

The file and inode table are not updated to the OWS, this occurs instead in the subsequent `release` call.

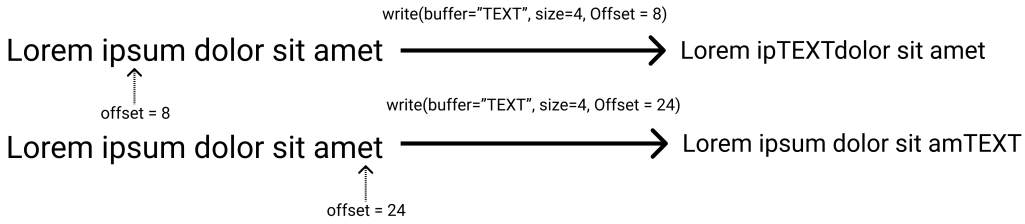

Figure 4.3: Visualization of how the write operation handles different offsets.

4.2.5.10 rename

This operation renames a file or directory to a new path. Both the old path and the new path have to be traversed to located the parent directories and the file or directory to rename. The file or directory entry in the old parent directory is removed, and the old parent directory is updated to the OWS. A new entry is created in the

new parent directory, with the new filename. The new parent directory is updated to the OWS. The inode entry of the renamed file or directory does not have to be modified. However, as both the old parent directories and the new parent directory are updated in the OWS, their inode entries need to be updated with the new posts. The inode table is updated to the OWS and the old table is removed from the OWS. The old posts associated with the old parent directory and the new parent directory are removed from the OWS.

The new path could be in the same directory as the file or directory currently is in. This will not affect the process mentioned above, however the path will only have to be traversed once.

4.2.5.11 truncate

This operation truncates or extends the file in the given path, to the provided size s . The full current file is downloaded into memory. The data of the current file is read into a new buffer until either the file is fully read, or until s bytes have been read. If the current file's size is smaller than s, the remaining amount of bytes are added as the NULL character. The new file data is uploaded to the OWS, and the old data is removed from the OWS. The inode table entry is updated with the new posts and uploaded to the OWS. The old inode table is removed from the OWS.

4.2.5.12 ftruncate

This operation is similar to `truncate`, but is called from a user context which means it has a file handle associated with it. The operation truncates or extends the file in the given file handle, to the provided integer s. The full current file is downloaded into memory. The data of the current file is read into a new buffer until either the file is fully read, or until s bytes have been read. If the current file's size is smaller than s, the remaining amount of bytes are added as the NULL character.

The file and inode table are not updated to the OWS, this occurs instead in the subsequent `release` call.

4.2.5.13 unlink

This operation removes a file given the filepath. The file is removed from the parent directory, and the parent directory is updated to the OWS. The old parent directory data is removed on the OWS. The removed file's entry in the inode table is also removed, and the inode table updates the entry for the parent directory with its new posts. The inode table is then updated on the OWS and the old inode table is removed on the OWS. Finally, the data of the removed file is removed from the OWS. The last step is not necessary for a working filesystem; however, to save space on the OWS, this is done. If the OWS permits unlimited images and sizes, this step could be omitted to execution save time.

4.2.5.14 rmdir

Similar to `unlink`, this operation removes the directory at the path. The directory and all its subdirectories are traversed, and the post IDs of these files and directories are recorded for deletion in the OWS later. Following, the entry of the removed directory is removed from the parent directory. The inode entry for the removed directory is removed. The parent directory is updated to the OWS, and the inode table is updated with the new posts of the parent directory. Following, the inode table is updated to the OWS. The old parent directory and the old inode table are removed from the OWS.

The operation also starts a new thread, where all the posts of files and subdirectories inside the removed directory, are removed from the OWS. This occurs to save space on the OWS, and a separate thread is used to save computation time for subsequent file operations. There is no data race involved as the API is thread safe, and the posts are no longer associated with any data structures on the main thread.

4.2.5.15 getattr

This operations returns attributes about a file or directory given a path. This includes permissions, number of entries (if the provided path points to a directory), and timestamps of creation, last access and last modification. However, as mentioned previously, FFS does not implement all features, such as permissions. Instead of keeping track of a file's or directory's permissions, all calls to valid path will return full read, write, and execute permissions for everyone. However, the timestamps are stored in the inode table of FFS. The file or directory pointed to by the path does not need to be downloaded, all the metadata that FFS stores is accessible through the inode entry in the inode table.

4.2.5.16 fgetattr

This operation is the similar to `getattr`, but is called from a user program context meaning that the file has a file handle associated with it. Other than skipping the path traverse step, this operation returns the equivalent information as `getattr`.

4.2.5.17 statfs

This operation returns metadata information about FFS. This includes, among other things, the maximum filename size and the filesystem ID. The operation has a short computation time as it does not have to download or upload any files. The only variable information is read from the inode table which is stored in memory and thus does not have to be downloaded from the OWS.

4.2.5.18 access

This operation, given a path, returns wether or not the path can be accessed. As long as the path is valid, this always returns that it can be accessed.

4.2.5.19 utimens

This operation updates the last access timestamp, the last modified timestamp, or both, of the file or directory at the given path. The file or directory does not have to be downloaded. However, the inode entry for the file's or directory's inode is updated with the new timestamps if they are newer than the previous timestamps but not greater than the current time since epoch. The new state of the inode table is updated to the OWS, and the old version is removed from the OWS.

4.2.6 FFS limitations

FFS has numerous of limitations due to both implementation decisions and OWS limits. As Flickr allows a free-tier user account to store up to 1 000 images of up to 200 MB per image, this allows storage of up to 200 GB of images on per account on Flickr. However, as the inode table is required to be stored on the filesystem, a maximum of 999 images can be used to save file and directory data. This limits the filesystem to a maximum of 999 files and directories when utilizing one free-tier account on Flickr.

While Flicker supports each image to be up to 200 MB, it is not possible to use the full 200 MB as the file data to store. The image includes, among other things, a PNG header, other PNG attributes, and the CED which in total is of greater size than the unencrypted data. To ensure that the pixel color data along with the PNG header and other PNG attributes does not exceed the limit of 200 MB, FFS limits the pixel color data size to allow at least 10 MB for the PNG header and other PNG attributes, meaning that the pixel color data can be a maximum of 190 MB. The cryptographic variables IV, salt, and the authentication tag are stored in the CED using 12, 16, and 64 bytes respectively, for a total of 92 bytes. The size limit means that these 92 bytes, along with the encrypted cipher text, cannot exceed 190 MB, meaning that the encrypted cipher text cannot exceed $190\,000\,000 - 92 = 189\,999\,908$ B. However, as AES is a block cipher producing cipher blocks of 16 bytes, the resulting cipher text must be a divisible of 16. The largest encrypted cipher text that FFS allows is

therefore $floor(\frac{189\,999\,906}{16}) * 16 = 189\,999\,904$ bytes. Due to plain text padding, the unencrypted plain text can be a maximum of one byte less than this value, meaning that the plain text can be a maximum of 189 999 903 B. For simplicity, this is rounded down to 189 MB, leaving almost 11 MB in total for the PNG header and other PNG attributes. 189 MB is set as the maximum amount of datan FFS will store per image. Data greater than 189 MB is split into multiple encoded images. For instance, a file of 200 MB will be stored as 189 MB in one image, and 11 MB in another.

189 MB of usable data per images gives FFS a maximum storage capacity of 188.811 GB using one free-tier account on Flickr. Each file with data requires at least one image, thus there can be a maximum of 998 non-empty files and directories in the filesystem, excluding the root directory. However, there could also be just one single file of 188.811 GB stored in the filesystem.

The inode table also keeps information about empty files and directories even though they store no data on the OWS. The inode of a file or directory is an unsigned 32-bit integer, meaning that the inode table could theoretically store up to over four billion files and directories. However, due to the constraints mentioned above, most of these files and directories would have to be empty as Flickr limits the amount of images stored. An empty file requires 37 B in the inode table. As the inode table is limited to one single image on the OWS, the inode table is limited to a maximum size of 189 MB. Further, the size of the inode table is 4 B plus the size of each entry, and one of these entries is the root directory. Even if a file is empty, it is still stored with its filename and inode in its parent directory. A non-empty directory in the inode table requires approximately (depending on the post ID length generated by the OWS) 12 B per file or directories it contains. The maximum number of empty files and directories X that the inode table can store is therefore, approximately:

$$X = ceil(\frac{189\,000\,000 - 4 - (12 * X)}{37}) + 1, X = 3\,857\,143$$

The additional directory is the root directory. Thus, the maximum number of files and directories that the inode table can store is close to four million, however this requires all files and directories, except the root directory, to be empty.

All these calculations are based on a single free-tier Flickr account. However, a future expansion of FFS could include multiple user accounts, and multiple services. This could increase the limits on the filesystem.

A limitation of FFS that is not possible to quantify is the bandwidth and latency of the network connection from the user to Flickr. The connection can vary significantly depending on for instance the network load in a given moment and the geographic location of the user. A slow network connection is not something FFS can solve, but is left as an exercise for the reader.

4.3 Benchmarking

This sections describes the methodology and execution of the different filesystem benchmarks. Two different filesystems that are relevant to FFS are compared with the result of two different instances of FFS; one instance that uses Flickr as its OWS, and one instance that uses a FOWS by storing the encoded images in the local filesystem on the test machine.

4.3.1 Filesystems

To analyze the performance of FFS, filesystem benchmarking tools are used to compare FFS against other filesystems that are relevant to FFS. FFS is compared to:

- An encrypted APFS partition,
- An instance of GCSF, and,
- An instance of FFS using the local APFS filesystem as its FOWS.

4.3.2 Tools

Chapter 5

RESULTS AND ANALYSIS

Chapter 6

DISCUSSION

Chapter 7

CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions from the thesis from what has been discussed under Chapter **??**. Finally, future work on the topic is discussed.

7.1 Future work

As mentioned previously, FFS does not implement all features that the POSIX standard defines. Future development for FFS could be to implement more of these functions, such as links and file permissions. This could make the filesystem resemble a regular filesystem further. Another improvement could be to move from userspace using FUSE, to kernel space. This could speed up filesystem operations. Another feature that could be interesting to evaluate is the possibility to share files with other users, similar to Google Drive.

Even though the files are encrypted so that the data is confidential, further research could include hiding the user's online activity through the use of for instance Tor. Currently, the integrity of the user is not considered but for the filesystem to be further plausibly deniable, this should be addressed as the user could otherwise be identified by its IP address and other online fingerprints that could be provided by the online web services.

To improve the dependability of the filesystem, support for more online web services could be implemented. For instance, Github provides free user accounts with many gigabytes of data. Even free-tier distributed filesystems, such as Google Drive, could be utilized. If multiple user accounts are used in coordination over multiple services, the filesystem could achieve even more storage.

APPENDICES

Appendix A

DIRECTORY, INODETABLE AND INODEENTRY CLASS AND ATTRIBUTES REPRESENTATION

Listing A.1: The attributes classes representing directories and the inode table in FFS

```
// inode_id is an unsigned 32-bit integer
#typedef inode_id uint32_t

/**
* @brief Describes a directory in FFS. Keeps track of the
   filename and inode of each file
*/
class Directory {
public:
        /**
        * @brief Map of (filename, inode id) describing the
           content of the directory
        */
        std::map<std::string, inode_id> entries;

        /**
        * @brief Returns the size of the directory object in
           terms of bytes
        *
        * @return uint32_t the amount of bytes required by
           object
```

```
        */
        uint32_t size();
};

/**
* @brief Describes and entry in the inode table, representing
    a file or directory
*/
class InodeEntry {
public:
        /**
        * @brief The size of the file (not used for
           directories)
        */
        uint32_t length;

        /**
        * @brief True if the entry describes a directory,
           false if it describes a file
        */
        uint8_t is_dir;

        /**
        * @brief A list representing the posts of the file or
            directory.
        */
        std::vector<post_id> post_blocks;

        /**
         * @brief Returns the size of the object in terms of
            bytes
         *
```

```
         * @return uint32_t the amount of bytes occupied by
            object
         */
        uint32_t size();
};

/**
* @brief Describes the inode table of the filesystem. The
   table consists of multiple inode entries
*/
class InodeTable {
public:
        /**
        * @brief Map of (inode id, Inode entry) describing
           the content of the inode table
        */
        std::map<inode_id, InodeEntry> entries;

        /**
         * @brief Returns the size of the object in terms of
            bytes
         *
         * @return uint32_t the amount of bytes occupied by
            object
         */
        uint32_t size();
};
```

Appendix B

BINARY REPRESENTATION OF FFS IMAGES AND CLASSES

This appendix visualizes the binary structures produced when serializing the `InodeTable`, the `InodeEntry`, and the `Directory` objects, and the binary structure of the encoded FFS images. The models are in terms of bytes, index 0 indicating the first byte, index 1 indicating the second byte, etc.

B.1 Serialized C++ objects

The `InodeTable`, `InodeEntry`, and the `Directory` class all have one `serialize` and one `deserialize` method each. The `serialize` method converts the objects data into binary form, and the `deserialize` method converts the serialized data into an object. The deserializer expects the same format of its input data as the serializer produces. The figures in this section visualizes the serialized output of the different classes. Figure **??** visualizes the serialized format of the `InodeTable`. Figure **??** visualizes the serialized format of the `InodeEntry`. Figure **??** visualizes the serialized format of the `Directory`.

B.2 FFS Images

An FFS images consists of multiple binary structures, including the FFS header and the encrypted data. This section visualizes these binary structures. Figure **??** visualizes binary format of the FFS header. Figure **??** visualizes the pixel color data of FFS images stored on the OWS.

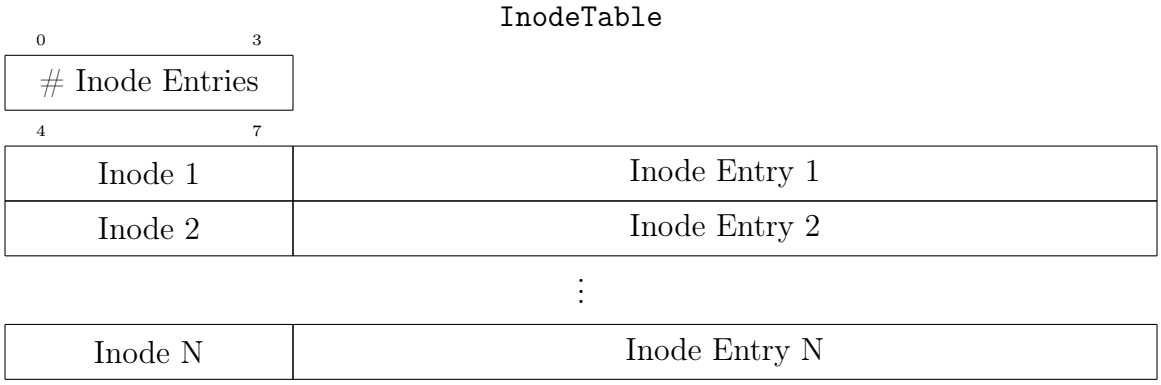

Figure B.1: `# Inode Entries` **is an unsigned integer representing the amount of inode entries the inode table contains. Following are** `# Inode Entries` **entries of an unsigned integer representing the inode of the inode entry, and the serialization of the corresponding** `InodeEntry` **object**

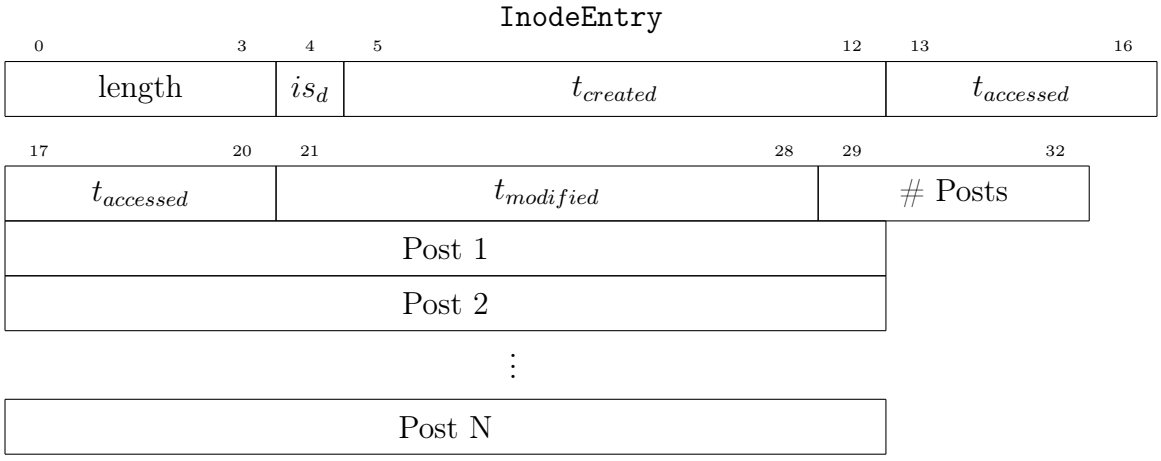

Figure B.2: Byte representation of a serialized `InodeEntry`**, representing a file or directory stored in FFS.** `length` **is an unsigned integer representing the amount of data stored on FFS by the file or directory, for instance the size of the file.** is_d **is a boolean with the value true** ($\neq 0$) **if the inode entry represents a directory, and false** ($= 0$) **if the inode entry represents a file.** $t_{created}$**,** $t_{accessed}$**, and** $t_{modified}$ **are unsigned integers represents timestamps of when the file or directory was created, last accessed and last modified, respectively.** `# Posts` **is an unsigned integer representing the amount of posts the file or directory is stored in on the OWS. Following are** `# Posts` **null-terminated strings representing each post ID in the OWS. The size of this field depends on the OWS used, for instance does Flickr often generate 11-byte post IDs. However, as the strings are null-terminated, the deserializer can read the bytes until the null-character is found**

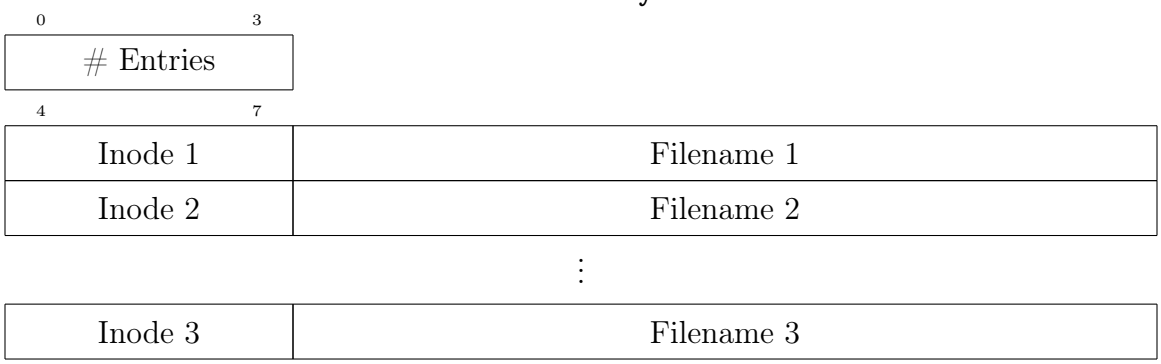

Figure B.3: Byte representation of a serialized `Directory`. `# Entries` is an unsigned integer representing the amount of entries in the directory. Following are `# Entries` inode-filename pairs. The `Inode` is an integer representing the inode of the file or directory, corresponding to the file's or directory's entry in the inode table. The filename is a null-terminated strings representing the filename of the file or directory in FFS. The size of this field can vary from filename to filename. However, as the strings are null-terminated, the deserializer can read the bytes until the null-character is found

Figure B.4: `'F'` and `'S'` are the literal letters F and S in ASCII code. `V` is an integer representing the version of the FFS image produced. `Timestamp` is an unsigned integer representing the number of milliseconds since Unix epoch when the image was encoded. `Data length` is an unsigned integer representing the number of bytes stored after the header. Following the heder is `Data length` bytes, containing the actual data stored in the image.

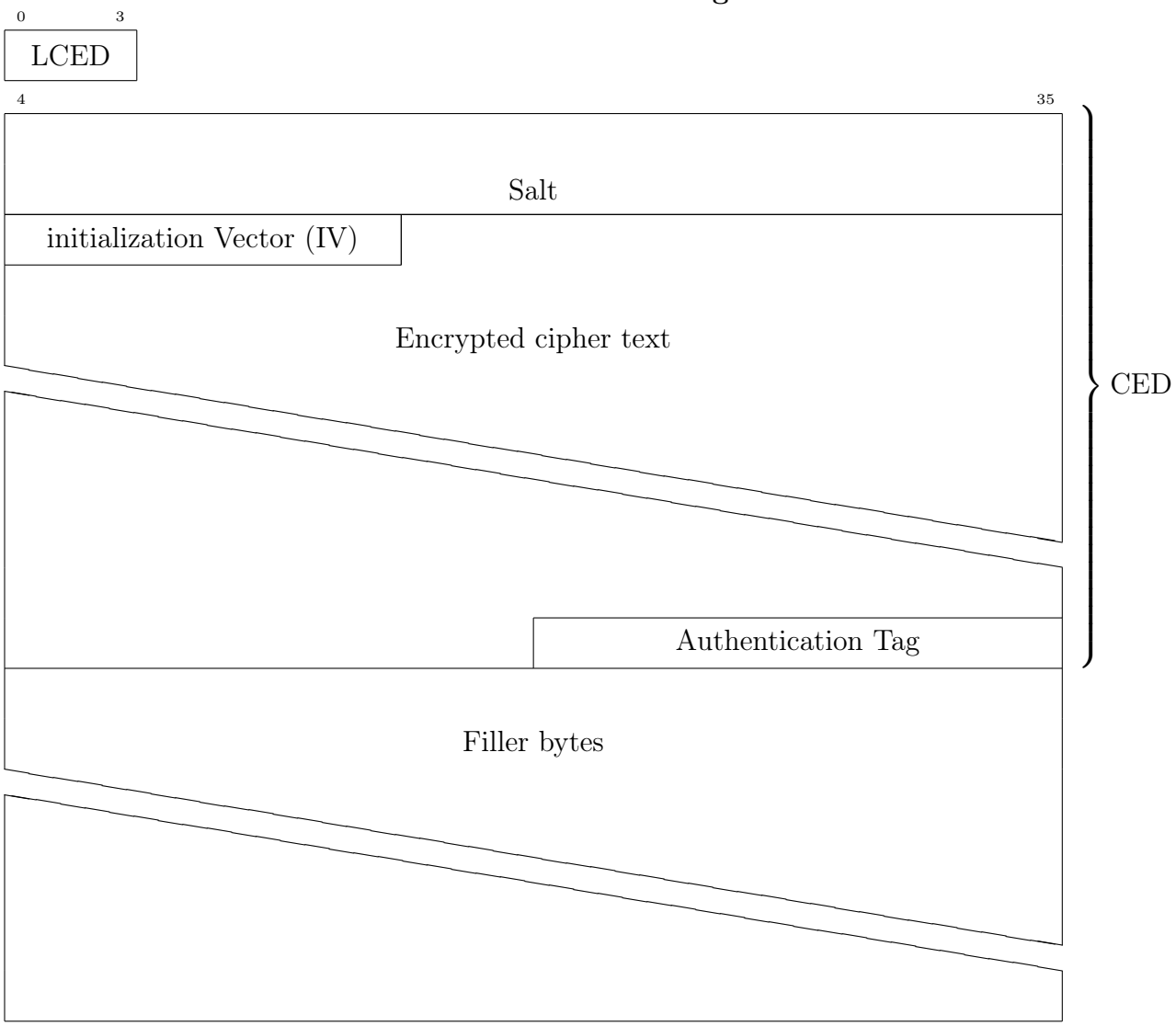

Figure B.5: Byte representation of the data stored as pixel color data in FFS images. `LCED` an unsigned integer representing the Length of the Complete Encrypted Data (). The `Salt` is a 64-byte randomized vector used to derive the encryption and decryption key. The `IV` is a 12-byte randomized vector used as the initial state of the encryption and decryption methods. Following is the `Encrypted cipher text` of variable size, depending on the size of the unencrypted data. The FFS header and the data to be stored, for instance the data of a file, is what is encrypted to become the `Encrypted cipher text`. The `Authentication Tag` is a 16-byte vector produced by the authenticated encryption method, and verified by the decryption method, to ensure data integrity has been upheld. Following is a number of filler bytes, depending on the size of the preceding data, to ensure the image has enough number of pixels for its calculated dimensions.