DISTRIBUTED FILE SYSTEM BY EXPLOITING 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:	Distributed file system by exploiting 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

Distributed file system by exploiting 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 Twitter, 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

Year after year, people increase their total data storage used for obvious reasons. Cameras increase their resolution leading to images and videos taking even more space. With storage being cheap and easily usable, files do not need to be deleted thus the data accumulates. This means that users will require more and more storage throughout their lifetime, and even potentially beyond their lifetime if their descendants want to keep these files. System storage in our hardware devices often increases with new product cycles. Today you can keep hundreds of gigabytes in your pocket at a reasonable cost. Along with increasing device storage and cloud storage, the aggregate storage capacity available to users is increasing. For instance, Apple's iCloud service allows users to store up to 2 TB of data in the cloud for a few U.S. Dollars per month. Even though the cost per month is not a lot, after many months this cost accumulates and you as a user become more and more dependent on this storage, especially as you do not want to spend time looking through all your data and remove some files to save space. With increased pricing or increased space, the total cost will be even higher.

Social media platforms such as Twitter, Flickr, and Facebook 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 difference between the photos posted on Twitter compared to the ones stored on cloud services such as Google Drive is that the images on Twitter are stored for free for the user, for what seems to be an indefinite period. However, there is no obligation for these services to save it forever, they do reserve the right to remove any content at any time, as stated in their terms of service*. There is also no specified maximum lifespan of these posts. While Google Drive and similar services often have a free-tier of storage with a certain maximum storage limit, Twitter does not have a specified upper limit of how many images or

* `https://twitter.com/en/tos`

tweets one can make. However, such constraints can be imposed on specific users whenever Twitter wishes as is stated in their terms of service[†].

1.1 Project Overview

This project intends to create a filesystem called *Fejk FileSystem* (FFS) which takes advantage of free online web services, such as Twitter, for the actual storage. The idea is to save the user's files by posting or sending an encrypted version of the file as posts or private messages on 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 for free as a filesystem. Additionally, the performance and limits of this filesystem will be analyzed and compared to existing alternatives, such as Google Drive, to compare the advantages and disadvantages of this free storage compared to a professional system that might cost money. The security of the filesystem will also be discussed, as well as an analysis of the steganographic capability of the developed filesystem.

1.2 Problem

Is it possible to create a secure, distributed filesystem that takes advantage of online services to store the data through the use of free user accounts offered by various online web services? What are the drawbacks of such a filesystem compared to commercially available solutions with regards to write and read speed, storage capacity, and reliability? Are there other advantages to such a filesystem than simply providing free storage?

[†] `https://twitter.com/en/tos`

1.3 Purpose and motivation

The purpose of this research is to explore the possibility to create a filesystem that stores data on online services and to compare the performance of such a filesystem to an actual distributed filesystem service. The interesting aspect of this is that 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. The thesis explores the use of such a filesystem despite potentially being slower and less dependable than filesystems that utilize other types of storage media, such as filesystems that costs a few dollars per month. Further, is it ethically defendable to create and use such a system?

1.4 Goals

The project aims to create a secure, mountable filesystem that stores its data via 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 free mountable filesystem where files can be stored, read, and deleted,
2. 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,
3. to analyze the write and read speed, storage capacity, and reliability of the filesystem and compare it to commercial distributed filesystems, and,
4. to analyze and discuss environmental and ethical aspects of the filesystem.

A side effect of such a filesystem that creates posts that are, while encrypted, publicly available is a steganographic filesystem in the sense that the data is hidden in plain sight. Therefore, an additional subgoal is to achieve and analyze the deniability of

the filesystem. This could make the filesystem useful for persons who need to hide their data, such as spies, journalists, and political actors where freedom of speech is non-existent.

1.5 Research Methodology

The filesystem created through this thesis will be developed on a Macbook laptop running macOS Monterey, version 12.0.1. It will be written in C++11 and use the Filesystem in Userspace (FUSE) MacOS library [**HomeMacFUSE**] which enables 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 11 is well established and used. Further details about the development environment is 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) [**appleAppleFileSystem**] 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 of 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 drawing the conclusions.

1.6 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 and 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 filesystem with a variety of different storage subsystems and thus FFS will only aim to implement a minimal filesystem.

1.7 Structure of the thesis

Chapter **??** presents theoretical background information of filesystems and the basis of FFS while Chapter **??** mentions and analyses 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 (FUSE) 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 file names 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 how 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.

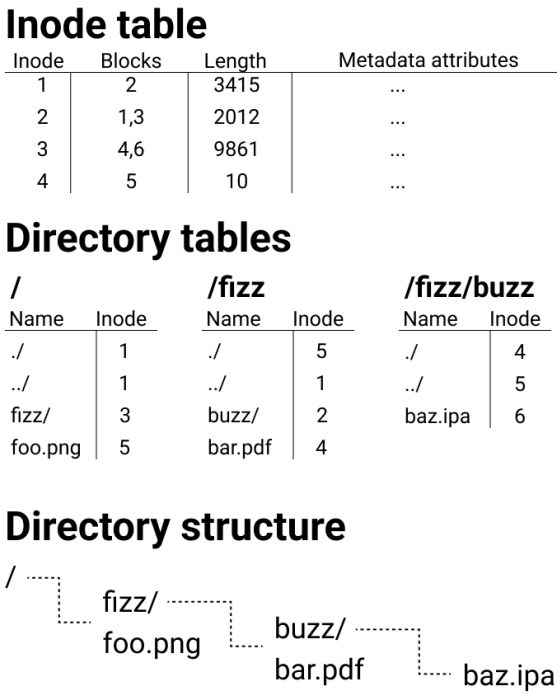

Figure 2.1: Basic structure of inode-based filesystem

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. This filesystem can theoretically store files up to 16 TiB while ext3 can store files up to 2 TiB [**salterUnderstandingLinuxFilesystems2018**]. Additionally, ext4 supports timestamps in units of nanoseconds while et3 only supports timestamp with resolution of one second.

The Apple Filesystem (APFS) is a modern filesystem which is used on iPhone and Mac and can store files with a size up to 9 EB [**APFSAppleFile2017**]. It supports timestamps in units of nanoseconds and is built to be used on solid state drives (SSD) [**nelsonWhatAPFSDoes**]. It also supports modern features which its predecessor Mac OS Extended (HFS+) does not support, such as Snapshots and Space Sharing. However, as opposed to ext it is often not used on Linux, while ext is the standard of many Linux distributions.

2.1.2 Distributed filesystems

Filesystems are used to store data on for instance a hard drive of a computer locally or in the cloud. For example, Google Drive is 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 hard drive, is 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 users location or device as long as a connection to the filesystem can be established. Thus, even if the user would loose

their computer or if it would malfunction, the data on the cloud-based filesystem can still be accessed.

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 could 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, with random words one after another leading 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 8-bit Red-Green-Blue (RGB) color depth, each pixel consists of three 8-bit values, i.e. three 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 three bytes as the first pixel's color, the next three bytes as the following pixel's color, and so forth.

This means that X bytes of data can be represented as

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

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 333 334 pixels in an image with 8-bit RGB color depth. The values of h and w are arbitrary but if we for instance want a square image we can set $h = w = 578$ which means that there will be 334 084 pixels in total, and the remaining 750 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 $w = 333\,334$ which would mean a very wide image but would not require filler pixels.

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.

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.

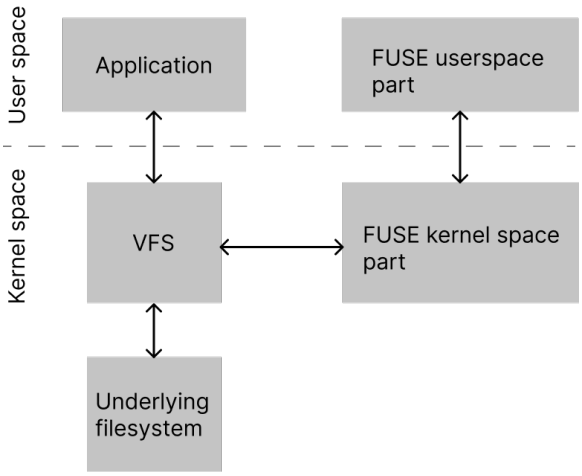

Figure 2.2: Simple visualization of how FUSE operations are transmitted

Figure **??** shows an overview how FUSE works. FUSE consists of a kernel space part and a user space part that perform different tasks [**vangoorFUSENotFUSE2017**]. The kernel part of FUSE operates with the Virtual Filesystem (VFS) 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 are then sent to the kernel part of FUSE. If needed, the operations is 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 handeld 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 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. Text posts are limited to 280 characters while images can be up to 5 MB and videos up to 512 MB [**MediaBestPractices**]. 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 on 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.

2.4 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 instance 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. In fact, 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 - 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.

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 within 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 does 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`

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 [**petersDEFYDeniableFile2014**]. This is also known as a rubber hose filesystem because of the characteristic that the data really 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 own 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 [**chumanEncryptionThenCompressio**
This is used to demonstrate the security and integrity of images sent over unsecure channels. In fact, the paper uses Twitter and Facebook to exhibit this. Despite its improvement and compatibly 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

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**].

TweetFS is a filesystem created by Robert Winslow which 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 filedata on Twitter. The filesystem uses sequential text posts to store the data. The filesystem is not mounted to the operating system but 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 uploads and downloads files or directories, respectively. Names and permissions of files and directories are maintained throughout the upload and download process.

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. Today, Gmail and Google Drive share their storage quota and GmailFS has since become redundant. 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.

Timothy **petersDEFYDeniableFile2014** created DEFY, a deniable filesystem using a log-based structure in **petersDEFYDeniableFile2014** [**petersDEFYDeniableFile2014**]. DEFY was built to be used exclusively on Solid State Drives (SSD) found in mobile devices to provide a steganographic filesystem which could be used on Android phones.

badulescuCryptfsStackableVnode1998 created Cryptfs, a stackable Vnode filesystem that encrypted the underlying, potentially unencrypted, filesystem [**badulescuCryptfsStackab**
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 potentially this increases availability and reliability.

3.4 Filesystem benchmarking

IOzone is a filesystem benchmarking tool which is used to measures 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.

3.5 Summary

As presented, different filesystems provide different features and drawbacks. Here we display a summary of characteristics and features of some filesystems mentioned above and how FFS compares.

	ext4	DEFY	FFS
Storage device	HDD or SSD	Built for SSD	Online Web Services height

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

4.2 FFS

The artifact that will be developed as a result of this thesis is the Fejk FileSystem (FFS). It uses online services to store the data but behaves as a mountable filesystem for the users. The filesystem will be minimal and not support all functionalities that other filesystems do, such as links. The reasoning is that these behaviors are not required for a useable system, and when comparing the system to distributed filesystems such as Google Drive, many of these other filesystems also often do not support links.

Figure **??** presents the basic outline of FFS and a example content of the filesystem. FFS is based on the idea of inode filesystems but instead of an inode pointing to specific blocks in a disk, the inodes of FFS will instead keep track of the id numbers of the posts on the online services where the file is located.

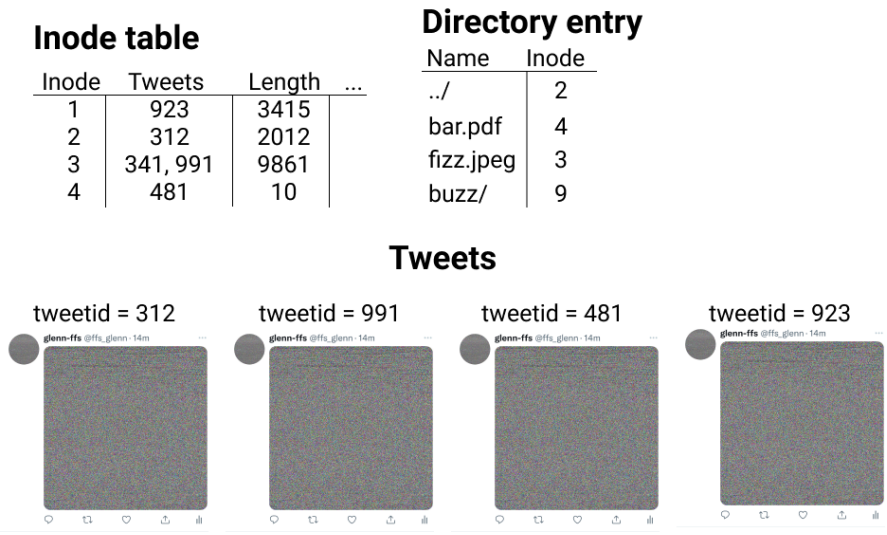

Figure 4.1: Basic structure of FFS inode-based structure

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 plausibly deniable, this should be addressed as the user could otherwise be identified by its IP address and other online fingerprints.

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.