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 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

Table Page

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 take even more space. With storage being cheap and easily usable, files do not needed 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 decendants 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 resonable cost. Along with increasing device storage cloud storage capacity 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 get 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[1]. A single user posting a few images per day does not significantly change the amount of data processed or saved at all for these tech giants. The difference between the photos posted on Twitter compared to the ones stored on cloud services such as iCloud is that the images on Twitter are stored for free for the user, for what seems to be indefinitely. While there is no obligation for these services to save it forever, and they do reserve the right to remove any content at any time, there is also no specified maximum lifespan of these posts. While iCloud and similar services often have a free-tier of storage, Twitter does not have a specified upper limit of how many images or tweets one can make, but such constraints can be inflicted on specific users whenever as according to their terms of service.

1.1 Project Overview

This project intends to create a filesystem called *Fejk FileSystem* (FFS) which takes advantage of 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 provides for free as a filesystem. The performance and limits of this filesystem will however be analyzed and compared to existing alternatives, such as Google Drive, to compare the benefits 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? What are the drawbacks of such a filesystem compared to commercial available solutions in regards to speed, throughput and reliability? Are there other advantages to such a filesystem than it providing free storage?

1.3 Purpose and motivation

The purpose of this paper 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 are reliant on other types of storage mediums, 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 which stores its data via online we services by taking advantage of the storage given to their 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, and,
3. to analyze the throughput, dependability and reliability of the filesystem and compare to commercial distributed filesystems.

In the following paragraph, I am trying to relay that the steganographic feature of the filesystem is not the goal, but rather a side effect of a publicly available data storage system. Is this a valid idea or should I rahter addess it some other way? Should I move it do be above the enumeration of subgoals? Or should it be a subgoal rather?

A side effect of such a filesystem which creates posts that are, while encrypted, publicly available is a steganographic filesystem in the sense that the data is hidden in plain sight. An imminent subgoal is therefore also to achieve and analyze the deniability of the system.

1.5 Research Methodology

The filesystem created through this thesis be written in C++11 and the FUSE MacOS library[2] which enables us to write a filesystem in user space rather than kernel space. The produced filesystem will be evaluated against other filesystems, both commercial distributed systems, such as Google drive, but also an APFS filesystem on a Macbook laptop. Quantitative data will be gathered from the different filesystems through the use of experiments with the filesystem benchmarking software Iozone[3]. We will look at attributes such as the difference in speed of read and write, as well as the speed of random read and random write.

Do I need to motivate the use of Iozone, as compared to Fio or FFSB? Should the attributes I will look at be motivated as well?

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. Focus will be to implement a subset of the POSIX standard functions, containing only crucial functions for a simple filesystem. This includes *open*, *read*, *write*, *mkdir*, *rmdir*, *readdir*, and *rename*. However, among other things, 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 minimalistic filesystem.

1.7 Structure of the thesis

Chapter 2 presents theoretical background information of filesystems and the basis of FFS while Chapter 3 mentions and analyses related work. Chapter 4 describes the

implementation and the design choices made for the system, along with the analysis methodology. Chapter 5 presents the results of the analysis and Chapter 6 discusses the findings and other aspects of the work. Lastly, Chapter 7 will finalize the conclusion of the thesis and discuss potential future work.

Chapter 2

BACKGROUND

2.1 Filesystems

2.1.1 Unix filesystems

A Unix filesystem uses a data structure called an *inode*. An inode keeps track of the metadata for the files in the filesystem, and a directory simply contains the file names and each file or directory's inode id. Using a lookup, the system can then learn about the file - for instance where it is located and how big it is as can be seen in Figure 2.1 Each inode entry can contain any metadata that might be relevant for the system, such as creation time and last update time.

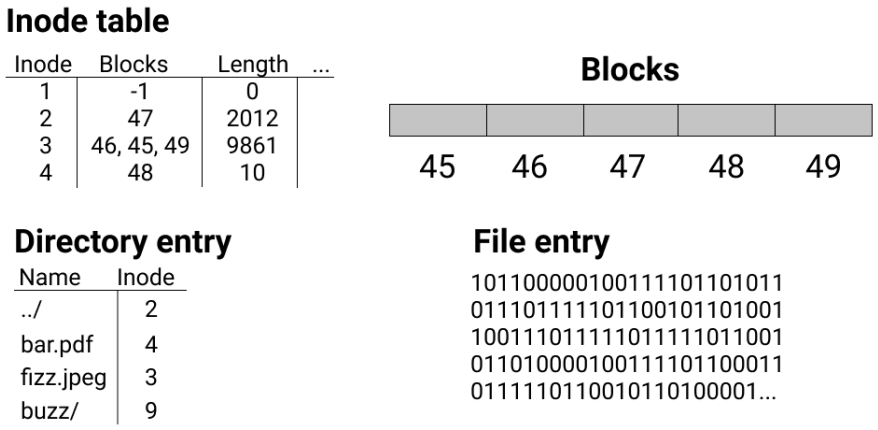

Figure 2.1: Basic structure of inode based filesystem

Looking at the four main file systems of Microsoft's Windows, there are functionalities present in some systems that are not found in others, for instance links and named streams. There is also other trade-offs such as a defined theoretical maximum file size[4] for the filesystems. This is set to 16 exbibytes for NTFS, exFAT, and UDF, while for FAT32 it is four gigabytes.

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[5] using their clusters of distributed storage devices, meaning that the data is saved on their servers which can be located wherever[6]. Paying customers can have a greater amount of storage using the service. Apple's iCloud and Microsoft's OneDrive are both other examples of distributed filesystems where users have the option of free-tier and paid-tier storage.

2.1.3 Image structures

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, files often consist of a string of binary digits no matter the file type. 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.

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 8-bit RGB image, each pixel consist 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. 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. 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 text or as an image, which can be posted on for instance social media.

2.2 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[7]. 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. It is also possible to create threads of Twitter posts where multiple tweets can be associated in a chronological order.

2.3 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 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[8]. Therefore, 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[9]. However, even though the data is encrypted, other properties such as your IP address can be compromised which can expose the user's identity. This problem is not addressed in FFS but is something for future work.

Other than adversaries for just FFS, we might also imagine that the underlying services might receive attacks that can potentially harm the security of the system or even have it go offline 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

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. Stegomalware is an increasing problem and in a sample set of images over 40% of real-life stegomalware was found in digital images[10]. 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 contains any type of file easily[11]. David Buchanan created a simple python script of only 100 rows of code that can encode zip-files, mp3-files, and really any file imaginable in an image of the user's choosing[12]. He presents multiple examples of this technique on his Twitter profile*. The fact that the images available for the public's eye is evidence that Twitter's steganography detection software might not be perfect. However, it is also possible that Twitter has chosen to not remove these posts.

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 on the filesystem[13]. This is also known as a rubberhose 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.

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

3.2 Cryptography

Some papers choose to invent their own encryption methods rather than using established standards. Chuman, Sirichotedumrong, and Kiya proposes a scrambling-based encryption scheme for images that split the picture into multiple rectangular blocks that are randomly rotated and inverted, both horizontally and vertically, along with shuffling of the color components[14]. 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 of the compatibly of the image format, such as bitsteam compliance, due to its well proven security FFS will use AES as its encryption method.

3.3 Related filesystems

Peters created a deniable filesystem using a log-based structure in 2014[13]. The filesystem of my project could be seen as a deniable system in the sense that the data is not stored on the device, and if the filesystem is not mounted it could be hard to prove that the user has access to the data, even if someone were for instance to find the Twitter account. The deniable system developed by Peters was also developed using FUSE[15] which we also will be using.

Badulescu, Shender, and Zadok created Cryptfs, a stackable Vnode filesystem that encrypted the underlying, potentially unencrypted, filesystem[16]. 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[17].

Chapter 4

METHOD

4.1 FFS

The artifact that will be developed as a result of this thesis is the Fejk FileSystem (FFS) which uses online services to store the data but behaves as a mountable filesystem for the users. The filesystem will however be very basic 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 4.1 describes the basic outline of FFS which is based on the idea of inode filesystems. 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 to online services where the file is located.

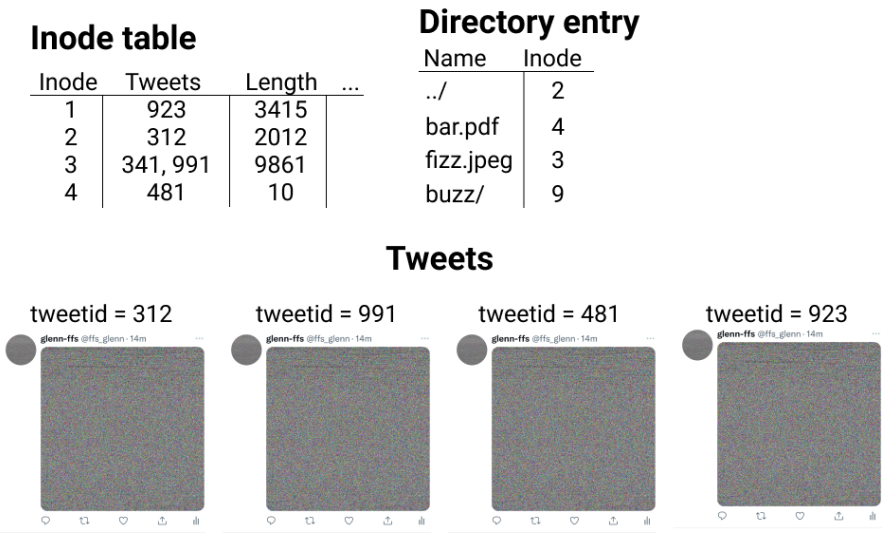

Figure 4.1: Basic structure of FFS inode-based structure

Chapter 5

RESULTS AND ANALYSIS

Chapter 6

DISCUSSION

Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Future work

Bibliography

[1] *Mobile @Scale London Recap - Engineering at Meta*. URL: https://engineering.fb.com/2016/03/29/android/mobile-scale-london-recap/.

[2] *Home - macFUSE*. URL: https://osxfuse.github.io/ (visited on 03/07/2022).

[3] *Iozone Filesystem Benchmark*. URL: https://www.iozone.org/ (visited on 03/07/2022).

[4] mikben. *File System Functionality Comparison - Win32 Apps*. URL: https://docs.microsoft.com/en-us/windows/win32/fileio/filesystem-functionality-comparison (visited on 02/07/2022).

[5] *Cloud Storage for Work and Home – Google Drive*. URL: https://www.google.com/intl/sv/drive/ (visited on 10/26/2021).

[6] *Distributed Storage: What's Inside Amazon S3?* Cloudian. URL: https://cloudian.com/guides/data-backup/distributed-storage/ (visited on 10/26/2021).

[7] *Media Best Practices - Twitter*. URL: https://developer.twitter.com/en/docs/twitter-api/v1/media/upload-media/uploading-media/media-best-practices (visited on 10/26/2021).

[8] *Privacy Policy*. URL: https://twitter.com/en/privacy (visited on 02/15/2022).

[9] Dr Prerna Mahajan and Abhishek Sachdeva. "A Study of Encryption Algorithms AES, DES and RSA for Security". In: *Global Journal of Computer Science and Technology* (Dec. 7, 2013). ISSN: 0975-4172. URL: https://computerresearch.org/index.php/computer/article/view/272 (visited on 02/07/2022).

[10] *SIMARGL: Stegware Primer, Part 1*. URL: https://cuing.eu/blog/technical/simargl-stegware-primer-part-1 (visited on 02/09/2022).

[11] *Twitter Images Can Be Abused to Hide ZIP, MP3 Files — Here's How*. URL: https://www.bleepingcomputer.com/news/security/twitter-images-can-be-abused-to-hide-zip-mp3-files-heres-how/ (visited on 02/09/2022).

[12] David Buchanan. *Tweetable-Polyglot-Png*. Feb. 9, 2022. URL: https://github.com/DavidBuchanan314/tweetable-polyglot-png (visited on 02/09/2022).

[13] Timothy M Peters. "DEFY: A Deniable File System for Flash Memory". San Luis Obispo, California: California Polytechnic State University, June 1, 2014. DOI: 10.15368/theses.2014.76. URL: http://digitalcommons.calpoly.edu/theses/1230 (visited on 10/19/2021).

[14] Tatsuya Chuman, Warit Sirichotedumrong, and Hitoshi Kiya. "Encryption-Then-Compression Systems Using Grayscale-Based Image Encryption for JPEG Images". In: *IEEE Transactions on Information Forensics and Security* 14.6 (June 2019), pp. 1515–1525. ISSN: 1556-6021. DOI: 10.1109/TIFS.2018.2881677.

[15] *Libfuse*. libfuse, Oct. 26, 2021. URL: https://github.com/libfuse/libfuse (visited on 10/26/2021).

[16] Ion Badulescu, Alex Shender, and Erez Zadok. "Cryptfs: A Stackable Vnode Level Encryption File System". In: (1998). DOI: 10.7916/D82N5935. URL: https://doi.org/10.7916/D82N5935 (visited on 03/04/2022).

[17] *FiST: Stackable File System Language and Templates*. URL: https://www.filesystems.org/ (visited on 02/02/2022).