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

APPENDICES

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 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 [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 an indefinite period. While there is no obligation for these services to save it forever, they do reserve the right to remove any content at any time*. There is also no specified maximum lifespan of these posts. While Google Drive and similar services often have a free-tier of storage, Twitter does not

---

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

have a specified upper limit of how many images or tweets one can make, but 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 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. However, the performance and limits of this filesystem will 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 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 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 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, and,

3. to analyze the write and read speed, storage capacity, and reliability of the filesystem and compare it to commercial distributed filesystems.

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 written in C++11 and use the Filesystem in Userspace (FUSE) MacOS library [2] which enables writing of a filesystem in userspace rather than in kernel space. C++ is chosen because the FUSE API is available in C, and version 11 is well established and used. 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)[3] on a Macbook laptop, as the development of the filesystem will be conducted on a Macbook laptop. Quantitative data will be gathered from the different filesystems through the use of experiments with the filesystem benchmarking software IOzone [4]. IOzone was chosen because it is, compared to tools such as Fio and Bonnie++, simpler to use while still powerful [5]. 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, benchmarking filesystems using benchmarking tools is difficult to perform in a standardized way according to Tarasov et al.[6] which will be taken into consideration during the evaluation and conclusion.

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 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

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

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, and 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 2.1 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 2.1 describes a simple implementation of an inode, an inode table, and directory entries.

Inode table

Inode	Blocks	Length	Metadata attributes
1	2	3415	...
2	1,3	2012	...
3	4,6	9861	...
4	5	10	...

Directory tables

/			**/fizz**			**/fizz/buzz**	
Name	Inode		Name	Inode		Name	Inode
./	1		./	5		./	4
../	1		../	1		../	5
fizz/	3		buzz/	2		baz.ipa	6
foo.png	5		bar.pdf	4			

Directory structure

```
/ ┄┄┄┐
     └┄ fizz/ ┄┄┄┄┄┄┄┐
        foo.png     └┄ buzz/ ┄┄┄┄┄┄┄┐
                       bar.pdf      └┄ baz.ipa
```
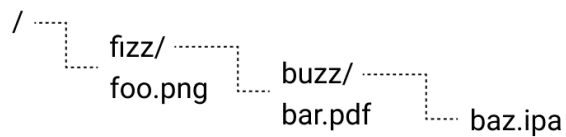
Figure 2.1: Basic structure of inode-based filesystem

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. Extended Filesystem (ext) exist in four different versions; ext, ext2, ext3, and ext4, and 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[7]. ext4 also supports timestamps at nanoseconds while et3 only supports timestamp down to the second. The Zettabyte filesystem (ZFS) introduces features that no version of ext supports, such as block-level cryptographic checksumming.

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 [8] using their clusters of distributed storage devices, meaning that the data is saved on their servers which can be located wherever they have data centers [9]. 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.

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, storage devices often represent files as a string of binary digits no matter the file type (however there are non-binary storage devices too [10], but this will not be addressed). 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 an actual text document, however with random words after another with a high probability of creating 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 $1\,\text{MB}$, 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 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.

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 [11]. The reason 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, it simplifies the process of creating filesystems. macFUSE is a port of FUSE that operates on Apple's macOS operating system that extends the FUSE API [2]. macFUSE provides an API for C and Objective C.

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[12]. 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 [13]. 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 [14]. 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 until the time 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 [15]. 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 [16]. 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, then 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 [17]. 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 [18]. David Buchanan created a simple python script of only 100 lines of code that can encode zip-files, mp3-files, and really any file imaginable in an image of the user's choosing [19]. 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.

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

---

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

content, or even if there exist any files in the filesystem [20]. 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. Chuman, Sirichotedumrong, and Kiya 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 [21]. 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, Baliga, Kilian, and Iftode presented an idea of a covert filesystem that hides the file data in images and uploads them to web services, named CovertFS [22]. 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 Sosa, Sutton, and Huang which also used Tor to further anonymize the users [23].

In 2006, Jones creates GmailFS - a mountable filesystem that uses Google's Gmail to store the data [24, 25]. 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 its launch become redundant. GMail Drive is another example of a Gmail-based filesystem and it is influenced by GmailFS [26]. GMail Drive has been declared dead by its author since 2015.

Timothy Peters created a deniable filesystem using a log-based structure in 2014[20]. 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 to find the web service account. The deniable system developed by Peters was developed using FUSE [11] which we also will be using.

Badulescu, Shender, and Zadok created Cryptfs, a stackable Vnode filesystem that encrypted the underlying, potentially unencrypted, filesystem [27]. 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 [28]. This approach enables one to create not only an encrypted file system but also to provide redundancy by replicating data to different underlying file systems. If these file systems are independent, then potentially this increases availability and reliability.

Chapter 4

METHOD

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

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 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.

Inode table

Inode	Tweets	Length	...
1	923	3415	
2	312	2012	
3	341, 991	9861	
4	481	10	

Directory entry

Name	Inode
../	2
bar.pdf	4
fizz.jpeg	3
buzz/	9

Tweets

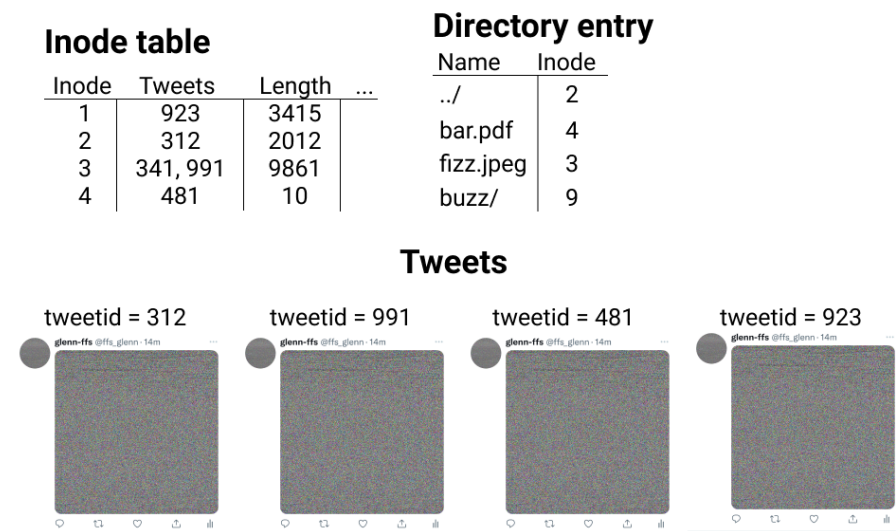tweetid = 312   tweetid = 991   tweetid = 481   tweetid = 923

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 6. 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.

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] Apple. *About Apple File System — Apple Developer Documentation*. URL: `https://developer.apple.com/documentation/foundation/file_system/about_apple_file_system` (visited on 03/13/2022).

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

[5] Udit Kumar Agarwal. *Comparing IO Benchmarks: FIO, IOZONE and BONNIE++*. FuzzyWare. May 19, 2018. URL: `https://uditagarwal.in/comparing-io-benchmarks-fio-iozone-and-bonnie/` (visited on 03/13/2022).

[6] Vasily Tarasov et al. "Benchmarking File System Benchmarking: It *IS* Rocket Science". In: *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*. Napa, CA: USENIX Association, May 2011. URL: `https://www.usenix.org/conference/hotosxiii/benchmarking-file-system-benchmarking-it-rocket-science`.

[7] Jim Salter. *Understanding Linux Filesystems: Ext4 and Beyond*. Opensource.com. Apr. 2, 2018. URL: `https://opensource.com/article/18/4/ext4-filesystem` (visited on 03/09/2022).

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

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

[10]   *Multi-State Data Storage Leaving Binary behind: Stepping 'beyond Binary' to Store Data in More than Just 0s and 1s.* ScienceDaily. Oct. 12, 2020. URL: `https://www.sciencedaily.com/releases/2020/10/201012115937.htm` (visited on 03/10/2022).

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

[12]   *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).

[13]   *Retrieving Older than 30 Days Direct Messages (Direct_messages/Events/List) - Twitter API / Standard APIs v1.1.* Twitter Developers. Apr. 27, 2018. URL: `https://twittercommunity.com/t/retrieving-older-than-30-days-direct-messages-direct-messages-events-list/104901` (visited on 03/11/2022).

[14]   *Understanding Twitter Limits — Twitter Help.* URL: `https://help.twitter.com/en/rules-and-policies/twitter-limits` (visited on 03/11/2022).

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

[16]   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).

[17] Stichting CUING Foundation. *SIMARGL: Stegware Primer, Part 1*. Feb. 14, 2020. URL: `https://cuing.eu/blog/technical/simargl-stegware-primer-part-1` (visited on 02/09/2022).

[18] *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).

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

[20] 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).

[21] 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`.

[22] Arati Baliga, Joe Kilian, and Liviu Iftode. "A Web Based Covert File System". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. HOTOS'07. USA: USENIX Association, May 7, 2007.

[23] Chris Sosa, Blake Sutton, and Howie Huang. "The Super Secret File System". 2007. URL: `https://www.cs.virginia.edu/~evans/wass/projects/ssfs.pdf` (visited on 03/09/2022).

[24] Richard Jones. *Google Hack: Use Gmail as a Linux Filesystem*. Computerworld. Sept. 15, 2006. URL: `https://www.computerworld.com/article/`

2547891/google-hack--use-gmail-as-a-linux-filesystem.html (visited on 03/09/2022).

[25] Richard Jones. *Gmail Filesystem Implementation Overview*. Apr. 11, 2006. URL: https://web.archive.org/web/20060411085901/http://richard.jones. name/google-hacks/gmail-filesystem/gmail-filesystem-implementation. html (visited on 03/09/2022).

[26] Bjarke Viksoe. *Viksoe.Dk - GMail Drive Shell Extension*. Apr. 10, 2004. URL: http://www.viksoe.dk/code/gmail.htm (visited on 03/09/2022).

[27] 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).

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