FFS: A CRYPTOGRAPHIC CLOUD-BASED DENIABLE 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 deniable 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 deniable filesystem through exploitation of online web services

Glenn Olsson

Many s today, such as Flickr and Twitter, provide users with the possibility to post images which are stored on the platform for free. This thesis explores the idea of creating a cryptographically secure filesystem which stores its data on an online web service using encoded and encrypted images. More data can usually be stored in image posts than in a text posts on s. The filesystem, named , provides users with free, deniable, and cryptographic storage by exploiting the storage provided by these online web services. The thesis compares the performance of against two other filesystems. It can be concluded that has limitations in mainly speed, making it unviable as a general-purpose usage, such as a substitute to the local filesystem on a computer. However, it provides security benefits compared to other cloud-based filesystems such as end-to-end encryption, authenticated encryption, and plausible deniability of the data. Furthermore, being a cloud-based filesystem, can be mounted on any computer with the same operating system, given the correct secrets.

ACKNOWLEDGMENTS

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

To the people who said it could not be done.

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 [1].

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 [2]. 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 to 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 exist 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 a 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 [3]. 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 on 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 [4].

This project created a cryptographic and deniable cloud-based filesystem called which takes advantage of free online web services, such as Twitter and Flickr, for the actual storage. The idea was to save the user's files by posting an encrypted version of the file as images and text posts on these web services. The intention was 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 is 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 is discussed and an analysis of the steganographic capability of the developed filesystem is presented.

1.1 Problem

Current cryptographic filesystems are mainly based on local-disk solutions, and while services such as 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 regard to write and read speed, storage capacity, and reliability? Are there advantages to such a filesystem in regard 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 s 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 [5]. 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 is to have a decentralized cloud-based filesystem where only the user has access to 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 ensure 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 unreadable without the decryption key. An interesting aspect of this is that online web services, such as social media, provide users with essentially an unbounded 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 stored in the open but only accessible by the user, and as can be unmounted to hide its existence, it is steganographic.

There are several steganographic filesystems available but these lack certain aspects that 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 [6] where data is stored in images posted on web services. The images are actual images representing something, meaning that there is 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. stores as much data as possible in the images, meaning that less images are needed to store a file bigger than 4 kB. It also means that the images produced by do not look like a normal image, but instead has seemingly randomly colored pixels. More examples of similar filesystems will be presented in Chapter 3.

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 the local disk,
3. for the system to be secure in the sense that even with access to the uploaded files and the software, the plain-text data is unreadable 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 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

A literature review was carried out to examine existing cryptographic, deniable, and cloud-based filesystems, as well as state of the art security standards. This created a basis for the technologies and security principles used in the produced filesystem, including as the filesystem library. Furthermore, experiments were carried out to gather quantitative data used to compare the performance of the produced filesystem against other relevant filesystems.

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 implemented in . The focus is to implement a subset of the POSIX standard functions, containing only crucial functions for a simple filesystem, specifically, the 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 will only aim to implement a minimal filesystem.

1.6 Structure of the thesis

Chapter 2 presents theoretical background information of filesystems and the basis of while Chapter 3 mentions and analyzes 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 . We first present the idea of inode-based filesystems and how data is stored in a filesystem. Following is the introduction of which will be used to implement . Later sections present background information about Twitter and the potential threat adversaries of .

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 .

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

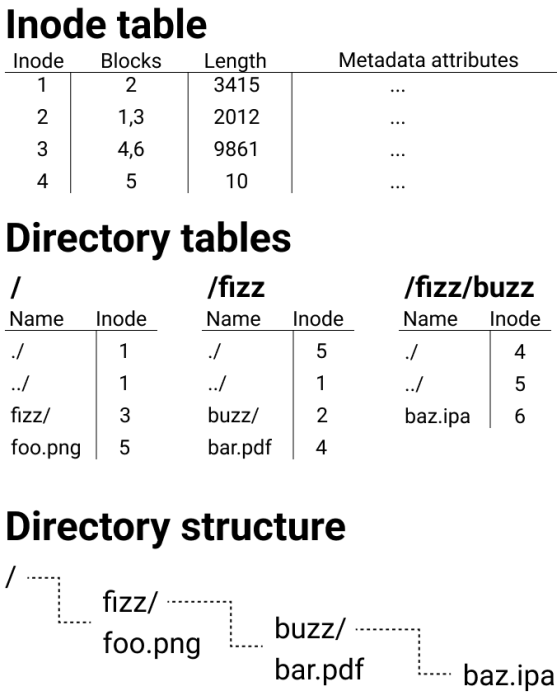

Figure 2.1: Basic structure of inode-based filesystem

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 an 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 [7]. 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 [8].

is a modern filesystem that is used on iPhones and Mac and can store files with a size up to 9 EB [9]. It supports timestamps in units of nanoseconds and is built to be used on [10]. It also supports modern features that its predecessor Mac OS Extended (HFS+) does not support, such as Snapshots and Space Sharing. natively supports encryption of the filesystem volume [11].

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 [12] 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 [13]. 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 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 [5]. While they mention in their Terms of Service that the user retains ownership of the data [14], 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 [15]. Controlling the encryption and decryption keys 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 suite [16]. For instance, backup data and iCloud drive are not end-to-end encrypted while the Keychain and Memoji data are.

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 [17], 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 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. 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 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 eighth bytes would represent the second pixel's 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 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 .

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. The error-correcting codes would have to be stored in a ensured lossless format, for instance, as a text post on an .

2.2 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 [18]. 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 are generally slower than kernel-based filesystems, using simplifies the process of creating filesystems. macFUSE is a port of that operates on Apple's macOS operating system and it extends the [19]. macFUSE provides an for C and Objective C. During research, it has not been found if filesystems developed using macFUSE can be mounted on non-macOS operating systems. While macFUSE is an extension of the library, it is possible that it is easy to port the macFUSE filesystems to the normal glsAPI and mount them on, for instance, a Linux operating system. However, information about this has not been found.

Figure 2.2 presents an overview how works. consists of a kernel space part and a userspace part that perform different tasks [20]. The kernel part of operates with the which is a layer in both the Linux kernel and the macOS kernel that exposes a filesystem interface for userspace applications [21, 22]. The interface is independent of the underlying filesystem and is an abstraction of the underlying filesystem operations which can be used on any filesystem the supports. The userspace part of communicates with the kernel space part through a block device. Operations on a mounted filesystem are sent to the from the user application, which is then sent to the kernel part of . If needed, the operations are transmitted to the userspace part of where the operation is handled and a response is sent back to the and the user application through the kernel module. However, some actions can be handled by the kernel module directly, such as if the file is cached in the kernel part of [20]. The response is then sent back to the user application from the kernel module through the .

Figure fig:fuse$_d escalsovisualizesthekernelcache.Thedatastoredinamac FUSE filesystemcanbecach$
$user support for is not covered by the scope of this thesis.$

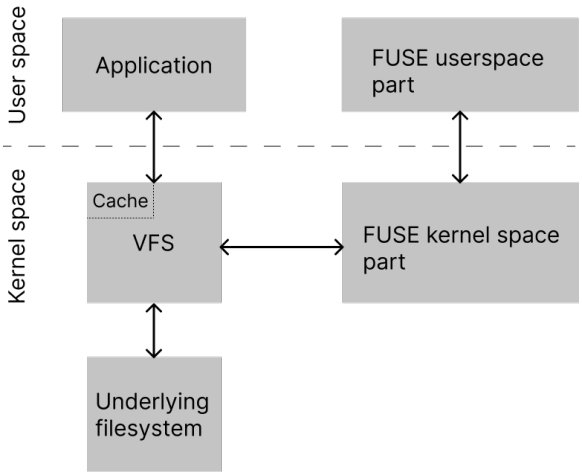

Figure 2.2: Simple visualization of how operations are executed

2.3 Online web services

This section presents two s, Twitter and Flickr, where one can create free-tier accounts. On both of these s, free-tier accounts can make numerous posts for free. The s each provide a free-to-use 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 [24]. Text posts are limited to 280 characters while images can be up to 5 MB and videos up to 512 MB [25]. A post with images can contain up to four 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 [26]. It is possible to create threads of Twitter posts where multiple tweets can be associated in chronological order.

Twitter's defines technical limits of how many times certain actions can be executed by a user [27]. 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 an 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 [28]. Size 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 a maximum of 1 GB. Further, free-tier accounts can only have a 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 [29]. Flickr Pro costs between EUR 7.49 to EUR 5.49 per month, depending on the subscription time the user signs up for. The description of a post has a limit of 65 535 characters according to Shhexy Corin [30]. This has been verified through testing. The title of a post has also been discovered through testing to have a limit of 255 characters.

The images and videos uploaded to Flickr are stored in their original form **without any compression** and can be downloaded by the user as the same file as was uploaded [31]. Flickr also stores other formats of the file, such as thumbnails. User accounts can restrict who, other than themselves, can download the original image. Restricting who can download the file helps ensure that no-one else can read the original file data, but also requires the user to authenticate with Flickr to download the image meaning it is not possible to anonymously download the image data. However, a restricted original image can still be downloaded by a knowledgeable person [32]. The original video can only be downloaded by the user [31]. Flickr does 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 [33].

The Flickr defines a query limit of 3 600 requests per hour, per application, across all calls [34]. However, according to Sam Judson in 2013, this is not a hard limit [35]. There is no official information from Flickr about what happens if you break the hourly request limit. The Flickr states that the is monitored on other factors as well [34]. If abuse is detected, Flickr reserves the right to revoke keys.

2.4 Cryptography

is an encryption standard established by the , more specificity specifying the Rijndael block cipher [36]. is a symmetrical cipher, meaning that the same key is used for encryption and decryption. is used to make the data confidential so that no one except the person with the key can access the unencrypted data. produces 128-bit encrypted cipher blocks and supports key sizes of 128 bits, 192 bits, or 256 bits. The security of has been heavily researched since its introduction in the early 2000s, and literature has found it is well resistant to quantum attacks as well [37].

While is a good standard for the confidentiality of the data, confidentiality is often not enough to secure the data [38]. The 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 [39]. is a block cipher mode of operation which provides authenticated encryption [40]. can be used together with to provide secure, authenticated encryption of data. To encrypt using , the encryption function requires a key, a randomized and the data to encrypt. The output is the encrypted cipher text and an authentication tag. The decryption function of requires the same key and 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 to be provided. is data that should be authenticated, but not encrypted. If is provided to the encryption function, it must also be provided to the decryption function.

The key used when encrypting using is often derived from a password that the user provides. s are functions that can be used to derive a key used for, for instance, .

The input to a is a secret, such as a password [41]. An example of a schema is the presented by Krawczyk [42][43] which utilizes a hashing algorithm that provide a pseudo-random key. supports multiple hashing algorithms. The security of is partially dependent on the security of the hashing algorithm used. A well-defined suit of hashing algorithms is the , which covers, among other hash functions, -256 [44]. -256 is a cryptographic hash function that outputs a 256-bit pseudo-random cipher from its input, which can, for instance, be a password. Further, 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 [45]. 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.

Alternative encryption solutions are, among others, and . is an asymmetrical cipher, meaning that it uses a public key and a private key for encryption and decryption. According to Mahajan and Sachdeva, asymmetric encryption techniques are more computationally intensive than symmetrical encryption techniques and are almost 1 000 times slower than symmetrical techniques [46]. Mahajan and Sachdeva found that is the fastest algorithm for encryption and decryption between , , and while maintaining very good security. This further makes a good choice as the cryptography technique for .

2.5 Threats

To consider a filesystem secure it is important to imagine different potential adversaries who might attack the system. Considering that 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 the online web service, 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. 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 [47]. Therefore, to achieve security the data stored must always be encrypted. We assume that an adversary has access to all knowledge about , including how the data is converted, encrypted, and posted. We also assume they know which websites and accounts could host data from the filesystem - but we assume they do **not** have the decryption key. However, even though the data is encrypted, other properties such as your IP address can be known which can expose the user's identity. The problem of these other sources of information external to is not addressed in but remains for future work.

Other than adversaries for , 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 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 completely deniable, an adversary should not be able to gain any information about the potential data in the system, this includes even the existence of data. When is unmounted there should be no trace of ever being present in the device. We will assume that an adversary is competent and can analyze the software and hardware completely. We assume that the adversary can gain access to the user's computer where has been mounted previously, but that they do not have access to the machine while is mounted. It is assumed that the adversary might have snapshots of the user's computer before and after was mounted, but that no snapshots were taken while was mounted. For instance, a country's border agents might take a snapshot of the computer's storage device every time the user passes through the border, but the user might mount during the time inside the country.

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 , for instance with the idea of unconventional storage media and 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 [48]. While 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 [49]. 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 [50]. 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.

Other examples of steganographic data storage on s include the paper presented by Ning et al. where the authors build a system for private communication on pub-

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

lic photo-sharing web services [51]. 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. Beato, De Cristofaro, and Rasmussen also explores the idea of undetectable communications over s in another paper [52]. 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 . 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 [53]. 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. 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 [54]. 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 will use 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 to that the user has access. For instance, Timothy Peters created DEFY, a deniable filesystem using a log-based structure in 2014 [55]. DEFY was built to be used exclusively on 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 [53, 56, 57, 1], 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, 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 [6]. The paper lacks implementation of the filesystem but they present an implementation plan which includes using . 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 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 [58].

In 2016, Szczypiorski introduced the idea of StegHash - a way to hide steganographic data on by connecting multimedia files, such as images and videos, with hashtags [59]. 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%. Bieniasz and Szczypiorski later presented SocialStegDisc which was a filesystem application of the idea presented with StegHash [60]. 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 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 [61], 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 2.1.3 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, Jones created GmailFS - a mountable filesystem that uses Google's Gmail to store the data [62, 63]. The filesystem was written in Python using 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 [64]. GMail Drive has been declared dead by its author since 2015.

is a filesystem that stores its data on Google Drive, built using [65, 66]. On the other hand, Google Drive provides a desktop application [67] that presents a mounted volume in the local filesystem, representing 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, interacts with the Google Drive REST rather than the mounted filesystem volume. One benefit of always synchronizing the data with Google Drive is that the duration of a filesystem operation can be measured easily. For instance, a write operation on a file in will not complete before the new file data has been completely stored on Google Drive. Therefore, the duration from the start of the filesystem operation until its end includes the time it takes to upload the file. On the other hand, the duration of a filesystem operation on the mountable volume provided by the Google Drive Desktop application does not always include the time it takes to upload the file, this can occur at a later time. One difference between and the idea of is that 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 . The data stored on is also stored as its original files in Google Drive, not as images as intends to store the data. The Google Drive filesystem architecture is utilized by , for instance by using its directory hierarchy structure. This allows to avoid creating its own inode table and directory structures, as Google Drive provides the functionality these structures similarly provide , through the Google Drive . The development of started in 2018 [66], and the repository in GitHub has around 2 300 starts as of writing.

Another Google Drive-based filesystem is google-drive-ocamlfuse [68], developed for Linux using . The project is well received online. The repository has around 6 700 stars on GitHub at the time of writing and there are multiple articles online about the project [69, 70, 71]. The filesystem is well developed and, as of writing, well maintained. The filesystem supports filesystem operations such as symbolic links, Unix ownership, and multiple account support. According to the author of , tends to be faster than google-drive-ocamlfuse for certain operations, including reading cached files [72, 73]. google-drive-ocamlfuse has no native support of macOS but is focused on Linux.

Zadok, Badulescu, and Shender created Cryptfs, a stackable Vnode filesystem that encrypted the underlying, potentially unencrypted, filesystem [74]. 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 [75]. This approach enables one to create not only an encrypted filesystem but also to provide redundancy by replicating data to different underlying filesystems. If these filesystems are independent, then this potentially increases availability and reliability. aims to achieve stackability through the use of .

3.4 Filesystem benchmarking

IOzone is a filesystem benchmarking tool that is used to measure performance and analyze a filesystem [76]. It is built for, among other platforms, Apple's macOS where will be built, run, and tested. However, 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 2011 Tarasov et al. presents a paper where they criticize several papers due to their lack of scientific and honest filesystem benchmarking [77]. The problem with benchmarking a filesystem is all the different components that are involved when interacting with a filesystem. For instance, they mention how benchmarking the 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.

Tarasov et al. 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 [77]. IOZone is listed as being compatible with multiple different benchmarking types and as it is simpler to use [78] and still maintained. Due to these factors, IOZone was chosen as the benchmarking tool for .

3.5 Summary

As presented, different filesystems provide different features and drawbacks. In Table 3.1 we display a summary of characteristics and features of some filesystems mentioned above and how compares. As can be seen, mainly lacks certain filesystem operations which are not the focus of as it is a proof of concept.

Table 3.1: Comparison between features present in related filesystems and . 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 chapter presents the methodology of implementing and the specifications of the development environment. We also present the benchmarking tools and methodology used to acquire the quantitative data for the evaluation of the filesystem.

4.1 Development environment specification

Development of was done on a 15 inch 2016 year model Macbook Pro laptop with a 2.6 GHz Quad-Core Intel Core i7 processor and 16 GB 2133 MHz LPDDR3 memory. The storage device of the computer was a 250 GB running an encrypted partition as the filesystem. Apple claims that the has a read speed of 3.1 GB/s and a write speed of 2.2 GB/s [79]. The computer used to develop FFS was running macOS.

Table 4.1 presents the version of the libraries, s and tools used by FFS. FFS was developed using C++ and compiled using Apple clang. uses the ImageMagick Magick++ library [80] for image processing. macFUSE [19] is used for to use the . cURLpp [81] is a cURL [82] C++ wrapper used by to make HTTP requests. libOauth [83] is used by to sign and encode HTTP requests according to the OAuth [81] standard. Flickcurl [84] is a C library used by to communicate with parts of the Flickr . Crypto++ [85] is a C++ library providing cryptographic schemes. uses Crypto++ to encrypt end decrypt the data stored in , and to derive the keys used in the encryption and decryption algorithm.

was developed for use 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 the development of the filesystem started. To avoid re-writing the source code to handle new designs, these versions remained the same throughout the development process.

Table 4.1: The versions of the libraries, s, and tools used by

Library, , or tool	Version
C++	20
Apple clang	13.0.0
Apple clang target	x86_64appledarwin21.4.0
ImageMagick Magick++	7.1.029
macFUSE	4.2.5
FUSE	26
cURLpp	0.8.1
libOauth	1.0.3
Flickcurl	1.26
Crypto++	8.6
macOS Monterey	12.5

4.2 FFS

The artifact that was developed as a result of this thesis is the (). It uses an to store the data but behaved as a mountable filesystem for the users. As mentioned in Section 1.5 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. Additionally, when comparing to distributed filesystems such as Google Drive, many of these other filesystems also do not support functionality such as links.

4.2.1 Design overview

uses images to store the data of files, directories, and the inode table of the filesystem. These images are uploaded to an , such as Flickr, as image posts. As mentioned in Section 2.3, there can be limitations of the size of these posts for certain s. To support file sizes bigger than these limitations, bigger files will be split into multiple posts, requiring to keep track of a list of posts. Figure 4.1 presents the basic outline of and an example content of the filesystem. is based on the idea of inode filesystems and uses an 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 will instead keep track of the id numbers of the posts on the 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.

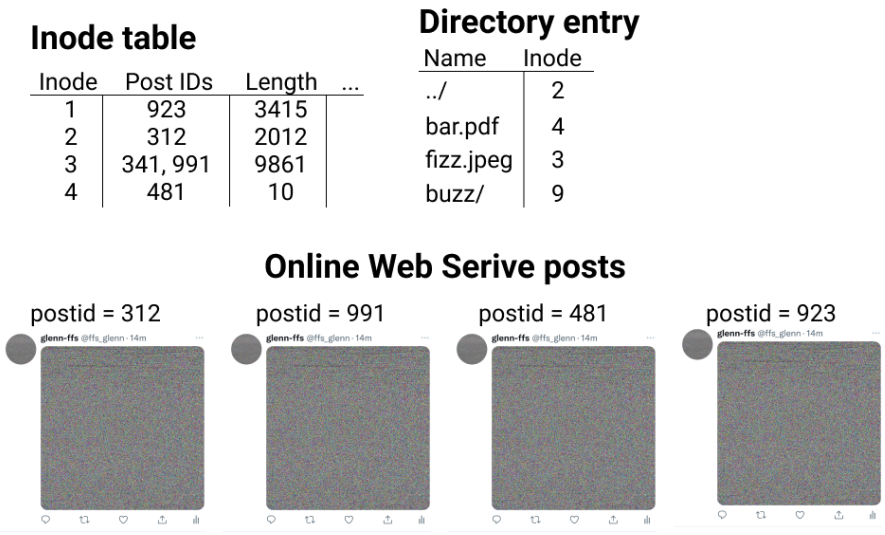

Figure 4.1: Basic structure of inode-based structure

The directories and inode table are represented as classes in C++. Appendix A visualizes the main attributes of the `Directory`, `InodeTable`, and `InodeEntry` classes. There can be multiple `Directory` and `InodeEntry` objects in the computers' memory and the filesystem, but there will only exist one relevant `InodeTable` instance. 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 file's `InodeEntry`. The `InodeTable` has always 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 always the most recent image saved on the , making it easy to find it on the .

To read the content of a known filename in a directory has three steps using these data structures:

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 file can be located.

The location of a file or directory is an ordered list of unique IDs of the image posts on the . The data received by downloading these images, decoding them (as described in Subsection 4.2.3), and concatenating them, can be read as a file or represented as a `Directory` object, depending on whether the `InodeEntry` is marked as a file or a directory.

As directories only know the filename's inode, the `Directory` object does not have to be updated (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 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 thus 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 A, the inode is represented as an 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 greater inode value than the ones currently in the inode table. This naïve approach to inode generation does not take into account that there might be an available inode less than the greatest inode in the inode table (for instance, due to the 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, would need to have saved more than four billion files before the inode value would overflow. This scenario is outside the scope of this proof-of-concept filesystem.

does not support all filesystem operations that are implementable through , instead, implements a subset of them as shown in Table 4.2. The implemented operations are the most essential operations required for a working filesystem [86]. Operations such as `chown` provide extended capabilities of the filesystem but these are not required for a proof-of-concept filesystem. The functionality of the filesystem operations implemented by and their implementation details are described in Subsection 4.2.5.

Table 4.2: Filesystem operations implementable trough the , and whether or not implements them

Filesystem operations implemented by FFS	**Filesystem operations *not* implemented by FFS**
open	readlink
opendir	symlink
release	link
releasedir	chmod
create	chown
mkdir	fsync
read	fsyncdir
readdir	lock
write	bmap
rename	setxattr
truncate	getxattr
ftruncate	listxatt
unlink	ioctl
rmdir	flush
getattr	poll
fgetattr	
statfs	
access	
utimens	

A file, a `Directory`, or the `Inode Table` has to be uploaded to the when it is modified to save its current information. As it takes time to make requests to the , is designed to make as few requests as possible while still saving the data required. Therefore, only the directory or file that is affected by a change is uploaded to the system, while those 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.

can be mounted to the local filesystem using , similar to how you can mount a network drive or a server. The mounted volume operates similarly to any other drive and can be accessed using, for instance, Mac's Finder or a shell terminal.

4.2.2 Cache

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

Cache Map a mapping between a post ID and its image data, and

Cache Queue a queue keeping track of the cached post IDs.

The cache stores a maximum of 20 image posts. The data stored in the cache is the encrypted image data. To avoid using 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 exceeds 20 elements, the last element 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 cache is limited to 20 entries, and by using the valuation method, the queue ensures that the oldest element in the cache is removed when the cache exceeds the limit. When a file or directory is removed from the filesystem, all its data is also removed from the cache, if it is stored there.

Before a post with a specified post ID is downloaded from the , the cache is checked to see if the cache is storing this post ID. If so, the stored image is returned. Otherwise, the process continues by downloading the image from the and then adding it to the cache. 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 a download of the data from the .

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 . They are stored as an instance of an `InodeTable` object and an instance of a `Directory` object.

separately also caches the inode of open files and the inode the open file's parent directory. The open file's data is also cached in memory if it has have been read or written to while it is open. A file is opened with the use of the `open` or `create` filesystem operation, as described further down in Section 4.2.5. When a file is opened it is associated with a file handle identifier which is used for subsequent filesystem operations to refer to the file rather than using the path to the file in the filesystem. When a user is reading or writing data to a file, multiple `read` or `write` file operations might be executed. For instance, when writing a 100 B file two `write` operations might be executed:

- One with `offset = 0` and with a buffer size of 50 B bytes, and
- One with `offset = 50` and with a buffer size of 50 B.

The amount of `read` or `write` operations required to read or write data depends on the amount of data to read or write, the buffer sizes used by the file operation which depends on the buffer sizes supported by the filesystem. macFUSE can be mounted with a maximum buffer size of 32 MB [87]. To save computation time by not having to download the file from the , or even decrypt the image data found in s regular cache, stores the file data separately in memory. When a file with a file handle is modified or read, checks if the file handle has any cached data associated with it, before it is checking the regular cache for the post ID. If there is data associated with the file handle, then this data is used for the file operation. If the file operation was a modifying operation, such as a `write` operation, the new data is associated with the file handle and stored in memory. When the file is closed with a subsequent `close` file operation, the modified data is encoded, encrypted and uploaded to the . There is no limit to how many files can be open in the filesystem at the same time, nor how much modified data can be associated with a file handle. Further, if a file is not closed, the associated data is not disassociated with the file handle and is kept in the memory until the filesystem is shut down.

Furthermore, an additional filesystem cache is provided by the operating system kernel [**MountOptionsOsxfuse**, 23]. can not control this cache other than disabling it when it is mounted. However, as it provides faster filesystem operations, it is beneficial for to use this. While this could be a problem if another user modifies the state of the filesystem on the while is mounted, multi-user support is not within the scope of this thesis. The size of the kernel cache has not been found during research.

4.2.3 Encoding and decoding objects

Entities that stores on the , and therefore also encodes and decodes, are: files, `Directory` objects, and the `Inode Table` object. All of these entities are stored on the using PNG images with 16-bit 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. The files saved to are read into memory in a binary format before being encoded into images. All the data encoded into images are encoded similarly, and a detailed description of the binary structures can be found in Appendix B.

The input to the image encoder is the binary data do encode as an image. A header (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 header and the input data are encrypted using authenticated encryption, utilizing and . The key used for the encryption is derived using the function utilizing the -256 hashing algorithm, along with a random 64 B salt vector, re-generated every time new data is 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 is a password provided by the user. also uses a random , re-generated every time new data is encrypted. The length of the is set to 12 bytes. The resulting data from the encryption is the salt, the , and 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 .

The dimensions of an image is based on the amount of bytes stored, as described in Section 2.1.3. The stored data is the CED, prepended with the Length of the (the) using four bytes. For an image of $X = ceil(\frac{4+LCED}{6})$ pixels, 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 as pixels in the image is, in order:

- four bytes representing the ,
- The data, and
- Filler bytes.

The content of the filler bytes are randomized.

The data consisting of the , CED, and filler bytes are encoded into for a PNG with 16 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 because most pixels are encrypted data and therefore the bytes representing this data are seemingly random.

To decode an image, the decoder first interprets the four first bytes as the . The salt and are retrieved from the as they are of known length. The decryption key is derived using the and salt and results in the same key as the encryption key because is a symmetric cipher algorithm. The remaining bytes of the ($-len(IV) - len(salt)$ bytes) are decrypted using the decryption key. The decrypted data consists of the header concatenated with the original stored data. The header is asserted to be in the correct format before the original binary data is returned from the decryption function. Figure 4.2 visualizes the encoder and decoder for all data saved in .

The encryption and decryption methods used are state-of-the-art solutions as defined and implemented by Crypto++ [85]. Crypto++ is a widely-used and well-maintained C++ library for cryptography, and as of writing has no reported CVE security vulnerabilities for the functionality used by [88].

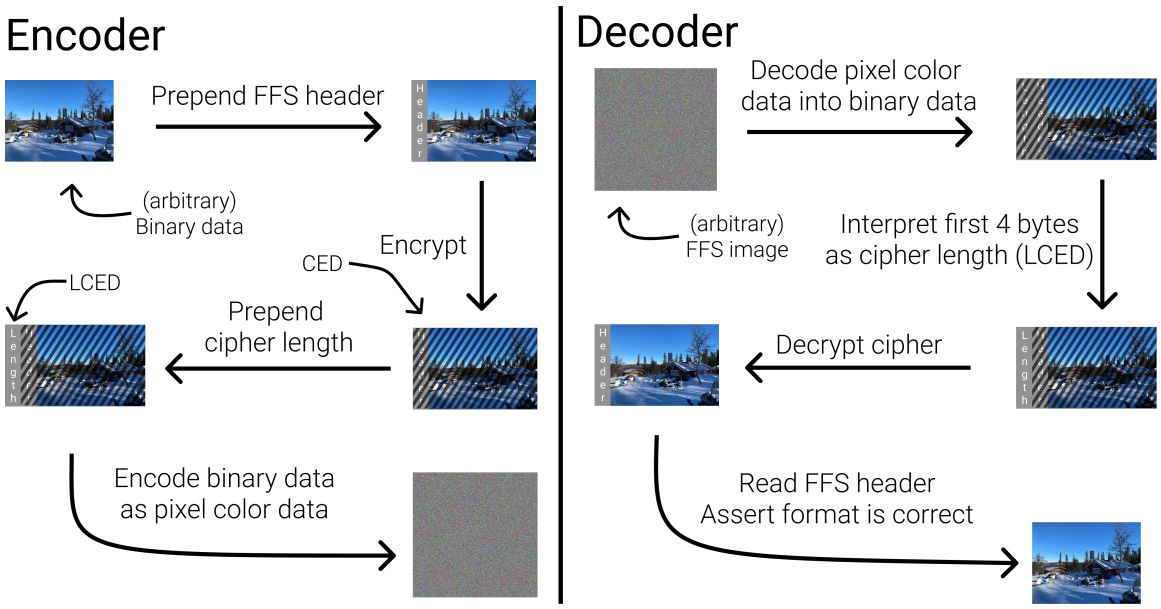

Figure 4.2: Simple visualization of the encoder and decoder of . The input of the encoder is the binary data to store in , eg. a file, and the output is the image to upload to the . The input to the decoder is an image, and the output is the binary data stored on , eg. a file

An image has an upper size limit, defined by the used. If the data to be stored in , such as a file, exceeds this limit, it is split into multiple data arrays of sizes less than this limit. Each data array is encrypted and encoded as images independently of each other, and will be encrypted using different salts and s. Only the inode table stores the different post IDs in the order they are encoded in. While files and directories stored in can be separated into multiple images, the inode table is limited to only one image for simplicity when interacting with the . This introduces a size limit of the inode table, limiting the filesystem. More details about the limits of are found in Subsection 4.2.6.

4.2.4 Online web services

As is a proof-of-concept filesystem, it only uses one as its storage medium. However, for a production filesystem, multiple s would be beneficial. This would enable features such as redundancy by using replication over multiple s, for instance in case one would stop working.

The initial intention of was to use Twitter as the . Initial research for the thesis 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 were compressed or re-coded which led to data loss when downloading the image. As the decoder of images relies on a specific binary representation of the image, this meant that the images could not be decoded into the previously uploaded data. Twitter has previously publicly announced changes to the way they store images [89] and even suggested workarounds [90] for users who are concerned about the potential data loss. However, during research for the thesis, it was concluded that the workarounds mentioned in [90] no longer work on Twitter. For instance, some PNG images less than 900x900px that have been uploaded to Twitter, have not been able to be downloaded as the same image, which contradicts the workaround mentioned by the Twitter employee. Further changes may have been made to the data management of images on Twitter since the initial research for the thesis; however, an official announcement has not been found.

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 data that is encoded into an -encoded image can be uploaded, downloaded, and decoded into the same data 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 is. A free-tier Flickr account is therefore used for . However, as was noted in Section 2.3.2, only the user can download the original file - other users might get another file when downloading the image post.

Flickr provides an extensive free REST 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 for the calls that require authentication.

Flickr provides the ability to search for all the images posted by a user and to sort these results by the time of posting. In , every time an image is uploaded to Flickr, it is due to some modification in the filesystem, for instance, a write operation to a file or a creation of a new directory. For every modification in the filesystem, the inode table will have to be updated. Therefore, we can ensure that the inode table is always the most recently uploaded image to Flickr by configuring to upload all other images first, for instance, the newly written file. This provides with a simple way of querying the inode table from Flickr - by simply requesting the most recently uploaded image on the Flickr account.

While the Flickr is extensive in its functionality, only uses a few of the provided capabilities; specifically uses:

- Upload an image and return the post ID,
- Query the most recent image by a user, and return the URL and post ID of the original uploaded image,
- Get the URL to the original uploaded image given a post ID,
- Remove an image given a post ID, and,
- Get the image data of the image given its URL.

For instance, to download the original image given a post ID, two requests are required:

1. Get the URL to the original uploaded image given a post ID,
2. Get the image data of the image given its URL.

For benchmarking purposes, a fake variant of , , has also been developed. uses a , which stores the data on the local filesystem. The is used by just as Flickr is used by , by storing encoded images on it. By storing the images on the local filesystem, the filesystem operation's duration is shorter as the local filesystem operations are in general faster than the network requests. This allows us to analyze the theoretical performance limit of , and how it would perform if the used had very low latency and the network connection to the had very high bandwidth and low delay. By analyzing , we can also estimate how much of the filesystem operation time is affected by the time of the network requests. The time T of an filesystem operation can be modeled like:

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

where t_{ffs} is the time that takes, for example for a read operation on a file associated with a file handle;

- 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 for the same file size. However, computer memory cache misses/hits and process scheduling, among other factors, can fluctuate the value of t_{ffs}. In contrast, t_{ows} is the total time required to complete all requests to the for a filesystem operation. For instance, for a similar read operation as above this consists of:

- to download all the directories in the file path,
- query the Flickr for the URL pointing to the most recently uploaded image, and,
- to download the images representing the file to read.

Depending on the , the latency and bandwidth of the internet connection between the user's machine and the 's server can differ a lot. Duplicate requests to the same can also differ significantly due to, for instance, server load balancing and a difference in number of requests from other users at the time of the requests. Further, the request could be replaced by a fast cache hit in the cache. However, for a , t_{ows} can be replaced by t_{fows} which will have approximately consistent values for duplicate operations, because the local filesystem is not affected by the network connection or the current traffic by other users of the . The local filesystem requests by other applications on the machine can also be minimized by not using other applications on the machine while running the benchmarking tool to ensure filesystem requests by the 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, process scheduling, and cache hits/misses which can still affect the value of t_{fows}.

Due to limitations in the library `Flickcurl` used for uploading images to Flickr, the image to be uploaded to Flickr first has to be saved to the local filesystem. `Flickcurl` reads the image from the disk, before uploading it. Therefore, saves a temporary file on the local filesystem when data is uploaded to Flickr. This temporary file is stored

in the `/tmp` directory of the local filesystem and is removed by immediately after the file has been uploaded. However, it is not certain that the operating system removes or overwrites the file data on the storage device, and thus there are ways to recover the deleted data, by for instance adversaries [91, 92, 93]. Although, these methods require you to decrypt the volume, requiring the decryption password. Without this password, the data cannot be recovered. Even with the decryption password, it is not certain that the data is recoverable. If an adversary obtains proof that an image has been present in the `/tmp` directory, they could conclude that has been used to store data, reducing the deniability of the filesystem.

4.2.5 Implemented filesystem operations

This section gives a detailed description of all the operations implemented by , and how they are implemented by . Further explanations about the intended functionality of the operations can be found in Kuenning's report [86].

The path of a file is sometimes provided for the filesystem operation and traversed by to understand the requested location. An example path is `/foo/bar/buz.txt` or `/foo/bar/baz/`. A path is traversed with the following pseudo-code shown in Listing 4.1.

When traversing a path, has to fetch all parent directories in the hierarchy. The file or directory with the filename is not fetched while traversing the path, as it might not be necessary for the operation. All operations that rely on the path of a file or directory have to download all parent directories of the path. However, the directories in the path could be cached and therefore would not be required to be downloaded from the . Furthermore, the `open`, `opendir`, and `create` operations associate a file handle with a file or directory. This enables certain subsequent filesystem operations to use the file handle instead of traversing the string path. This saves time because the path traversing only occurs once for potentially multiple filesystem operations, and the result is saved in the filesystem state.

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 cache
        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 the cache
        curr_dir = cache.get_root_dir()

        # While there are still directories to traverse,
        #  get the next directory in the list from the
        # 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)
```

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

All filesystem operations are synchronous unless specified. Further, is running in single-thread mode meaning that a filesystem operation call must complete before another can begin. This helps limit 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 file path multiple times. The file is not downloaded from the , 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 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 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 updated in the . The new posts representing the parent directory in the are associated with the inode entry of the parent directory in the inode table, and the old posts are deleted in the . A new inode entry is also created in the inode table, representing the new, empty, file. The inode table is updated in the , and the old inode table is removed.

4.2.5.3 release

Given a file handle, this operation closes the file in the filesystem, disassociating the file handle from the file. The current states of the file and the inode table are saved to the , and the previous versions of the file and inode table are deleted from the . 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.

4.2.5.4 opendir

Given a path to a directory, the directory is associated with a file handle. The file handle is used in subsequent operations to avoid traversing the file path multiple times. The directory is not downloaded from the , 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.

4.2.5.5 releasedir

Given a file handle, this operation closes the directory in the filesystem, disassociating the file handle from the directory. The current states of the directory and the inode table are saved to the , and the previous versions of the directory and inode table are deleted from the . Subsequent operations for the directory 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.

4.2.5.6 mkdir

This operation creates an empty directory in the filesystem given a path. The directory is not uploaded to the as it has no data associated with it. The parent directory is modified and updated in the , and the old versions of the parent directory are deleted in the . 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 , and the old version of the table is removed from the .

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 from a data array a, starting at the provided offset o, to the existing file at the provided file handle. All the data of the current file is read into memory. Starting from the offset, the new data from a 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 4.3 for a visualization of the result of a `write` operation given different offsets. The parent directory does not have to be modified.

The file and inode table are not updated on the , this occurs instead in the subsequent `release` call. However, the data is associated with the file handle so that subsequent filesystem calls uses this new file data.

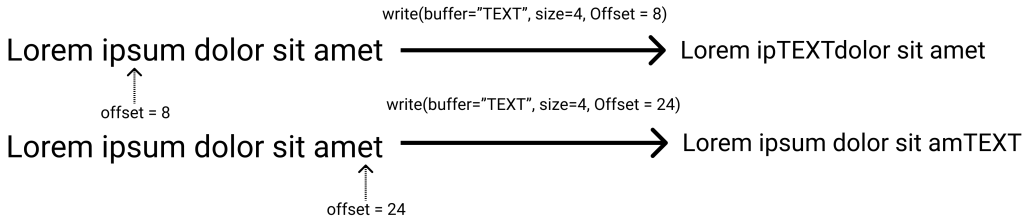

Figure 4.3: Visualization of how the write operation handles different offsets.

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 locate 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 . A new entry is created in the new parent directory, with the new filename. The new parent directory is updated to the . 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 , their inode entries need to be updated with the new posts. The inode table is updated to the and the old table is removed from the . The old posts associated with the old parent directory and the new parent directory are removed from the .

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, and the parent directory will only be removed and updated 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 written to a new buffer until either the file is fully written, or until s bytes have been written. If the current file's size is smaller than s, the remaining bytes are written as the NULL character. The new file data is uploaded to the , and the old data is removed from the . The inode table entry is updated with the new posts and uploaded to the . The old inode table is removed from the .

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 read into memory, either from the or from the cache. The data of the current file is written to a new buffer until either the file is fully written, or until s bytes have been written. If the current file's size is smaller than s, the remaining bytes are written as the NULL character.

The file and inode table are not updated to the , this occurs instead in the subsequent `release` call. However, the data is associated with the file handle so that subsequent filesystem calls uses this new file data.

4.2.5.13 unlink

This operation removes a file given the file path. The file is removed from the parent directory, and the parent directory is updated to the . The old parent directory data is removed on the . 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 and the old inode table is removed on the . Finally, the data of the removed file is removed from the . The last step is not necessary for

a working filesystem; however, to save space on the , this is done. If the permits unlimited images and sizes, this step could be omitted to 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 later. Following this, 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 , and the inode table is updated with the new posts of the parent directory. Following this, the inode table is updated to the . The old parent directory and the old inode table are removed from the .

The operation also starts a new thread, where all the posts of files and subdirectories inside the removed directory, are removed from the . They are removed to save space on the , and a separate thread is used to minimize the delay for subsequent file operations. There is no data race involved as the is thread-safe, and the posts are no longer associated with any data structures on the main thread and would not be accessed there. This also means that this thread can be run with a lower priority.

4.2.5.15 getattr

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

4.2.5.16 fgetattr

This operation is 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 . 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 .

4.2.5.18 access

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

4.2.5.19 utimens

This operation, provided new timestamps, 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 , and the old version is removed from the .

4.2.6 FFS limitations

has numerous limitations due to both implementation decisions and 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 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, which also limits the maximum storage of file- and directory images to 199.8 GB.

While Flicker supports each image to be up to 200 MB, it is not possible to use the full 200 MB to store the file or directory data. The image includes, among other things, a PNG header, other PNG attributes, and the which in total is of greater size than the unencrypted data. To ensure that the along with the PNG header and other PNG attributes does not exceed the limit of 200 MB, limits the size to allow at least 10 MB for the PNG header and other PNG attributes, meaning that the can be a maximum of 190 MB. The cryptographic variables , salt, and the authentication tag are stored in the 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 $190\,000\,000 - 92189\,999\,908$ bytes. However, as is a block cipher producing cipher blocks of 16 bytes, the resulting cipher text must be divisible by 16. The largest encrypted cipher text that 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 [94], 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. Therefore, 189 MB is set as the maximum amount of data will store per image. Data greater than 189 MB in size 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 image gives a maximum storage capacity of 188.811 GB using 999 files and directories on 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 entry of 188.811 GB stored in the filesystem, which would have to represent the root directory.

The inode table keeps the information about empty files and directories even though they store no data on the . The inode of a file or directory is an unsigned 32-bit integer, meaning that the inode table could theoretically store more than 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 number of images stored. An empty file requires 37 B in the inode table, consisting of the inode, length, and other variables that must exist for an inode entry. As the inode table is limited to one single image on the , 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 file or directory in the inode table requires 37 B plus approximately (depending on the post ID length generated by the) 12 more bytes per post ID representing the file or directory on the . Assuming only one non-empty directory which stores empty files and directories and only uses one post ID of 12 B, the maximum number of files and directories X that the inode table can store is:

$$X = floor(\frac{189\,000\,000 - 4 - (12 + 37)}{37}) + 1, X = 5\,108\,107$$

The additional directory is the root directory. Thus, the maximum number of files and directories that the inode table can store is more than five million; however, this requires all files and directories, except the root directory, to be empty. Otherwise the entries in the inode table will require more space due to the post IDs. These calculations are based on a single free-tier Flickr account. However, future work of could include multiple user accounts and multiple services. This could increase the limits on the filesystem.

A directory is encoded as four bytes to represent the number of entries in the directory, plus a directory entry for each file or directory in the directory. A directory entry consists of four bytes representing the inode, followed by the bytes representing the filename followed by the NULL character. The filename is limited to 128 characters, meaning that each directory entry can take up to $4 + 128 + 1 = 133$ bytes. While a directory is not limited to one image in the implementation, it can be interesting to calculate how many entries a directory can have using only one image. With 4 B for the number of entries and 133 B per entry, and a maximum size of 189 000 000 B, we get the equation:

$$4 + 133 * x < 189\,000\,000 \iff x < 1\,421\,052.60$$

Meaning that there can be a maximum of 1 421 052 entries using the full file-name limit using only one image, and that the directory can store at least that many files without exceeding the image size limit. With shorter filenames, more entries can be kept in the directory. The maximum number of files and directories one directory can store due to the inode table limitations mentioned above is 5 108 106, one less than the maximum number of files and directories the inode table can store. These files and directories must be empty as the inode table size limit would be exceeded otherwise. If the directory stores 5 108 106 empty files and directories, and the inode table uses one image, the directory can use up to 999 images on Flickr before the account exceeds the maximum number of images. However, these 5 108 106 files and directories can also fit in just one image representing the directory. This requires the filenames to be small enough that the directory does not exceed the maximum image limit of 189 MB. For instance, using filenames of four letters, and using both uppercase and lowercase letters, we can form $(26 * 2)^4 = 7\,311\,616$ unique filenames. 5 108 106 directory entries of four-letter filenames plus one NULL character and four byte inodes requires $5\,108\,106 * (4 + 4 + 1) = 45\,972\,954$ bytes ≈ 46 MB. Therefore, the directories does not need more than one image to describe all the possible files it can contain before runs out of storage; however, if the filenames are longer than four characters, the directory could exceed the maximum image limit of 189 MB before reaching 5 108 106 images, requiring multiple images.

The biggest possible file that can be stored in the filesystem can be 188.811 GB minus the size of a single-entry root directory and an inode table with two entries representing the root directory and the file. A single-entry directory requires four $4+4+1+x$ bytes for the number of entries, the inode of the file, the NULL character, and x as the number of bytes in the filename. The smallest filename size is one byte, meaning that the smallest possible single-entry directory requires 10 B. The inode table with two entries, assuming the post IDs are 12 B, requires:

- 4 bytes for the number of entries,
- $37 + 12$ bytes for the directory as it only requires one image, and
- $37 + 12 * y$, where y is the number of posts required to store the file.

This means that the inode table will require $90 + 12 * y$ bytes. The maximum value of y for a free-tier Flickr account is 998 as one image is required for the inode table and one for the root directory. The directory and inode table will therefore require $10+90+12*998 = 12\,076$ B. As each image can use up to 189 MB, the maximum file size is $998 * 189 * 10^6 = 188.622$ GB. $188.622\,\text{GB} + 12\,076\,\text{B} < 188.811\,\text{GB}$ meaning that file size is allowed in the filesystem.

Limits to the file sizes also depend on the machine where is mounted. When a file is read or written to, the complete file is read into memory. This requires the computer to provide at least as much memory as the size of the file. Further, the cache of can store up to 20 images with a size of 5 MB in memory, requiring up to 100 MB of memory. Furthermore, open files can be cached in-memory independent on their size, potentially occupying up to 188.622 GB of memory. The inode table and root directory are also cached in-memory so even more memory could be required. This is more memory than most computers are sold with today. However, even if the computer has less memory available, more memory can often be provided through swap on the hard disk. Apple ensures that the swapped data is securely encrypted on the hard disk [95]. However, using a swap puts a constraint on the available storage of the hard disk used by the underlying filesystem to be sufficient to store this data. Further, as temporarily saves the data in the local filesystem before it is uploaded to Flickr, the storage device must have sufficient storage available for this as well. A file larger than the available storage on the local filesystem cannot be saved to . If

the local filesystem has no available storage, only a few filesystem operations can be performed on as any operation that modifies the inode table requires the new inode table to be saved to the local filesystem before it is uploaded to Flickr.

stores data associated with file handles of modified and open files in memory. The amount of open files in is unbounded, and the amount of data can store per file handle is also unbounded. Further, the data associated with the file handle is not removed from memory until the file is closed. If many files are opened and modified without being closed, the memory of the computer could be filled fast. The file handle associated with an open file is the inode of the file, meaning that there is no risk for collision of file handles as the inodes are assured to be unique per file. One file can only be opened once before it is closed. `open` calls on an open file will be ignored.

Another limitation of is the rate limits presented by the Flickr . Flickr allows up to 3 600 requests per hour, after which the keys may be revoked by Flickr. 3 600 requests per hour equals 60 requests per minute, or 1 request per second. If the average request takes less than a second for constant, sequential Flickr calls, the keys could be revoked. Furthermore, some requests are sent concurrently to Flickr which means that could reach 3 600 calls faster. Reading a file, represented by one image which is not in the cache, stored in the root directory, requires two requests:

- Request the URL to the original image given a post ID,
- Downloading the image from Flickr.

With maximum one request per second, we can read 30 un-cached files per minute. If the file is in a directory which is in the root directory, and the directory is not in the cache, two similar requests are required to download the parent directory assuming that the directory is represented by one image. This would require four requests in total, limiting us to read 15 un-cached files per minute. For each level of depth in the directory, two more requests are required per directory not in the cache, assuming the directory is represented using one image.

If the file or directory is represented using more than one image, two more request are required per extra image. However, the bandwidth of the internet connection to the will affect the duration of the request significantly for big images. For instance,

downloading a 200 MB file with 100 Mbit/s download bandwidth to Flickr will take 16 s. To download a 5 MB (the cache file size limit) using the same connection would take 0.4 s. The response when requesting the URL to the original image is only a few kilobytes which would take a few millisecond depending on the latency. Constant read operations (including preceding open- and subsequent close operations) on a file just over 5 MB could therefore exceed the request limit with a 100 Mbit/s bandwidth.

A limitation of that is not possible to overcome 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 at a given moment and the geographic location of the user. For instance, given a 5 Mbit/s download bandwidth to Flickr, downloading a 5 MB would take 8 s, and a 200 MB file would take more than 5 min. The request limit is still a limitation when reading smaller files; for instance, any file of 625 kB or less takes 1 s or less to download using the same bandwidth.

4.3 Benchmarking

This section describes the methodology and execution of the different filesystem benchmarks. Two different filesystems that are relevant to : (1) and (2) , are compared with the result of two different instances of : (1) one instance that uses Flickr as its , and (2) one instance that uses a by storing the encoded images in the local filesystem on the test machine.

4.3.1 Filesystems

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

1. An encrypted partition on an SSD,
2. An instance of , and,
3. An instance of using an encrypted filesystem on an SSD as its .

The encrypted filesystem was used as a reference for a local filesystem without the required internet connection. It is the local filesystem of the development environment for . It was selected as it will give the analysis an example of a modern, well-used, and fast filesystem, and how the benchmark data of and other filesystems compare to this local filesystem.

was selected to compare against another network-based filesystem. While is not a steganographic filesystem, it is a filesystem that stores its data on an , namely Google Drive. The reason was used instead of, for instance, the official Google Drive mountable filesystem volume provided by the Google Drive Desktop application, is that provides instant upload of the files and directories to Google Drive using the Google Drive REST . The instant upload provided by enables us to easily measure the duration of a file operation. For instance, a write operation on a file in will not complete before the new file data has been completely stored on Google Drive. Another reason why was chosen was because it is a recent filesystem compared to other related filesystems. Some of the other filesystems discussed in Section 3.3 were developed many years before and thus no longer work as expected, for instance, due to changes in the , or because the manages the uploaded data differently than previously.

The instance of using a of an encrypted was chosen to be compared to so that the duration of the filesystem operations could be further analyzed. As the filesystem operations of are similar to the ones of , other than the network request being replaced by local filesystem operations, it is possible to analyze the effect of the latency, the internet connection bandwidth and latency, and the data processing speed has on the filesystem performance. Comparing the benchmark results of and allows us to analyze the overhead as is dependent on the performance of . Especially for file operations where must interact with the storage medium, for instance, write operations and read operations for files not in the cache, cannot outperform as it will

require the execution time of the file operation as well as the internal computation time. Both and are mounted with a maximum buffer size of 32 MB.

4.3.2 Tools

IOZone [76] is a filesystem benchmarking tool used to analyze the performance of filesystem file operations using different tests on a file [96]. Examples of tests that IOZone provides support for are: reading and writing, reading and writing randomly, and reading backward. Each test can be run with different file sizes and different buffer sizes for the read- or write operation. Normally, multiple buffer sizes are used for each test, for each file size tested. The buffer size starts at 4 kB and increases by a multiple of two up to a buffer size equal to the file size. Multiple file sizes are often used for benchmarking tests as well, which are also increased by a multiple of two. For instance, one could run the IOZone tests with file size 1024 kB and 2048 kB, which would utilize the following values of the file size and buffer size for each test specified:

1. File size = 1024 kB, buffer size = 4 kB,
2. File size = 1024 kB, buffer size = 8 kB,
3. File size = 1024 kB, buffer size = 16 kB,
4. File size = 1024 kB, buffer size = 32 kB,
5. File size = 1024 kB, buffer size = 64 kB,
6. File size = 1024 kB, buffer size = 128 kB,
7. File size = 1024 kB, buffer size = 256 kB,
8. File size = 1024 kB, buffer size = 512 kB,
9. File size = 1024 kB, buffer size = 1024 kB,
10. File size = 2048 kB, buffer size = 4 kB,
11. File size = 2048 kB, buffer size = 8 kB,
12. File size = 2048 kB, buffer size = 16 kB,
13. File size = 2048 kB, buffer size = 32 kB,
14. File size = 2048 kB, buffer size = 64 kB,
15. File size = 2048 kB, buffer size = 128 kB,
16. File size = 2048 kB, buffer size = 256 kB,
17. File size = 2048 kB, buffer size = 512 kB,
18. File size = 2048 kB, buffer size = 1024 kB, and
19. File size = 2048 kB, buffer size = 2048 kB.

Furthermore, each test is run for each file size-buffer size pair before the buffer size or file size is increased. This means that, for the example above using the tests: Read, Write, Re-Read, Re-Write, Random Read, and Random Write, IOZone will run the following in order:

1. Write test for: File size = 1024 kB, buffer size = 4 kB,
2. Re-Write test for: File size = 1024 kB, buffer size = 4 kB,
3. Read test for: File size = 1024 kB, buffer size = 4 kB,
4. Re-Read test for: File size = 1024 kB, buffer size = 4 kB,
5. Random Read test for: File size = 1024 kB, buffer size = 4 kB,
6. Random Write test for: File size = 1024 kB, buffer size = 4 kB,
7. Write test for: File size = 1024 kB, buffer size = 8 kB,
8. Re-Write test for: File size = 1024 kB, buffer size = 8 kB, et cetera.

When IOZone reads from a file it has written to, it asserts that the file content is what it wrote previously to verify that the filesystem stores the data properly. This is not documented in the IOZone documentation [96] but was discovered during testing. However, while it asserts that file operations function correctly, it does not verify all aspects of the filesystem functionality. Further, as IOZone does not state that the file operations are tested, it cannot be assumed that the file operations are correct. Additionally, IOZone does not test if directory hierarchies work as expected, nor if multiple files can be stored at the same time. IOZone is a benchmarking tool used for evaluating the performance of the file operations of a filesystem, not testing the functionality. However, certain cases of the functionality of both and was tested, and to support directory hierarchies and multiple files as expected. Future work could research the functionality of these filesystems utilizing online storage systems. is expected to have full functionality as it is a professionally developed and widely used filesystem.

While IOZone supports multiple different file operation tests, the thesis only uses a subset of these for benchmarking. Among other reasons, certain tests failed when ran on . Furthermore, tests such as backward reading lack relevance as it tests a rare case of filesystem operations. The documentation of IOZone [96] claims that the software MSC Nastran uses backward-read. The documentation also mentions that only a few operating systems provide enhancements for backward reading, although many operating systems provide enhancements for forward-reading. As is intended as a proof-of-concept filesystem and is not intended as a general-purpose filesystem, only relevant tests were chosen. The IOZone benchmarking tests used in the thesis are:

Forward- Read and Write, Forward- Re-Read and Re-Write, and Random- Read and Write. The *Forward* specifier will sometimes be omitted in the thesis when the tests are referenced. For instance, when mentioning the Read test, we refer to the Forward Read test. The Forward Write test is writing data to a new file, in order. The file is first created which can include overhead time. Forward Read tests reading the file in order. The Read test tests reading from same file in-order. The Re-Write test measures the performance of writing to an existing file in-order. In general, Re-Write is faster than Write as it does not have to create the file. Re-Read measures the performance of reading a recently read file in-order. In general, Re-Read is faster than Read as the file data can be kept in the filesystem cache. The Random Write and Random Read tests measure writing and reading data from random positions in a file. Depending on filesystem implementation, this can be influenced by many factors. For instance, as has to download and read the entire file into memory before accessing a specific offset, might provide lower performance for these tests than a filesystem that can choose to only read a part of a file at a specific location. The official descriptions of these tests can be found in the IOZone documentation [96].

The IOZone documentation [96] states that to get the most accurate performance results from the benchmarking, the maximum file size of the tests should be set to a value bigger than the filesystem cache. While the cache limit is known to be 5 MB, the cache size limit or the existence of such a limit for the other filesystems, such as , is unknown. The IOZone documentation states that when the cache is unknown, it should be set to be greater than the physical memory of the system. However, as the memory of the computer where the benchmarking is run is 16 GB, this is bigger than reasonable for testing and . Each doubled file size takes exponentially much more time as both the file size is doubled for each test and buffer size. Furthermore, it has been found that will crash for bigger file sizes. The maximum file size for the benchmarking tests is set as 262 144 kB as this is bigger than the biggest image possible to store on Flickr, requiring to split the image in to more than one image on the . This helps us test the functionality of FFS further. The file sizes used for the IOZone tests are therefore set as:

1. 1024 kB,
2. 2048 kB,
3. 4096 kB,
4. 8192 kB,
5. 16 384 kB,
6. 32 768 kB,
7. 65 536 kB,
8. 131 072 kB, and
9. 262 144 kB

The biggest buffer size IOZone uses is 16 384 kB, therefore the buffer sizes tested are:

1. 4 kB,
2. 8 kB,
3. 16 kB,
4. 32 kB,
5. 64 kB,
6. 128 kB,
7. 256 kB,
8. 512 kB,
9. 1024 kB,
10. 2048 kB,
11. 4096 kB,
12. 8192 kB, and
13. 16 384 kB

However, the maximum buffer size for each file size is the file size itself. For instance, for a file size of 4096 kB, IOZone will run the tests for buffer sizes up to, and including, 4096 kB. It can not run tests with a buffer size greater than 4096 kB.

IOZone reads and writes the same file. The file is created and removed during the tests, but there are never two files created by IOzone at the same time. The file is, when benchmarked for the thesis, stored in the root directory of , , and . This means the path traversal is as short as possible for these filesystems. Furthermore, as and constantly caches the root directory and has a cache limit of 20 entries, and as no other operations are performed on the filesystem at the same time as the benchmarking tests, the file can always be stored in the cache as long as it is small enough. The entry will not be removed unless the file is removed as the cache will not be filled by other entries. The file used for benchmarking is in a deep directory hierarchy. Future work could compare benchmarking in different depths of directory hierarchies to analyze if the depth has an impact on the performance of the filesystem.

When benchmarking the filesystems using IOZone, an argument is passed to include the time to close a file (using the `close` filesystem operation) in the total time of a test. This is important as , and potentially other filesystems, save the data to the storage medium only after the device is closed. In the case of , if the time of closing

the file was not included, the performance of the filesystem would appear to be higher than it is.

IOZone produces a log of the benchmarking results for the filesystem it benchmarked. This log contains a report of each test (file operation) with performance data for each file size, and for each buffer size for each file size benchmarked for the test. The performance of the filesystem is measured in kilobytes per second.

During the benchmarking of and , an automatic speed test was conducted every five minutes to survey the current internet connection. The speed test uses Bredbandskollen's command line interface tool [97] which measures the latency, upload-, and download speed of an internet connection to a measurement server in Sweden, Norway, or Denmark [98]. The benchmarking tests of and were carried out in Amsterdam in The Netherlands using an ethernet connection to a fiber-connected router in a corporate office after-hours when no other devices used the internet in the office. While the internet connection to the measurement server is not sure to be equal to the internet connection to the servers of Flickr or Google Drive, it is used as a reference point of the internet connection. Measuring the individual filesystem's internet connection's bandwidth would have been preferable. However, when researching tools that could measure the average bandwidth over time, per application, no tools were found that could be run on macOS. As there are multiple background processes running at all times, it is important that the internet connection can be filtered based on the application. If it could be ensured that no other processes used the internet at the same time, a tool such as Wireshark [99] could be used to measure the bandwidth. In future work the measurement of the individual applications bandwidth could be explored further which could provide further analysis of the comparisons between the filesystems.

Chapter 5

RESULTS

This section presents the results of the thesis. The resulting filesystem is presented in Section 5.1. The benchmarking data outputted from IOZone is presented in Section 5.2

5.1 FFS

The artifact developed as a result of the thesis is , which uses Flickr as its . The source code of can be found on GitHub at github.com/GlennOlsson/FFS [100]. The filesystem provides free cloud-based cryptographic and deniable storage as a mountable volume. The filesystem requires the user to provide their Flickr keys and an encryption password. The keys are used to authenticate with the Flickr , and the password is used to derive the encryption and decryption keys. These values are passed to as environment variables.

5.2 Benchmarking

This section presents the result from the IOZone benchmarking tests run on each filesystem. The output result is divided into a table for each test for each filesystem. Each table presents the benchmark performance of the test for each file size and each buffer size. The benchmarking completed successfully for all file sizes for , , and . For the biggest file size, 262 144 kB, crashed multiple times, but it succeeded for the other file sizes. Therefore, the biggest file size was omitted from the tests on . In total, six benchmarking tests were conducted on the filesystems, using nine file sizes (eight for) and up to 13 buffer sizes per file size. Each test per filesystem produces a table with nine rows (eight rows for) and 13 columns, where each cell is the performance

of the test with the specific file size and buffer size on the filesystem. Each table for , , and have 107 cells, and the table for has 94 cells.

The average latency of the internet speed when ran was 15.3 ms, the average download speed was 90.87 Mbit/s, and the average upload speed was 92.95 Mbit/s. The average latency of the internet speed when ran was 17.29 ms, the average download speed was 89.2 Mbit/s, and the average upload speed was 91.83 Mbit/s.

Combining all the data points of one table, we get the overall performance of a test on a filesystem. With this data we can use a box plot to present the values in the table, and combining the results from all filesystems for a specific test, we can compare their box plots in one figure. Figure 5.2 presents a box plot of the benchmarking results of the filesystems for the Read test. It can be observed that the read operation performance of and are in general worse than the performance for and . has by far the biggest spread of values, and has also a significant spread. The median performance of and are similar. and have less spread. The median performance of is significantly higher than the median performance of , , and .

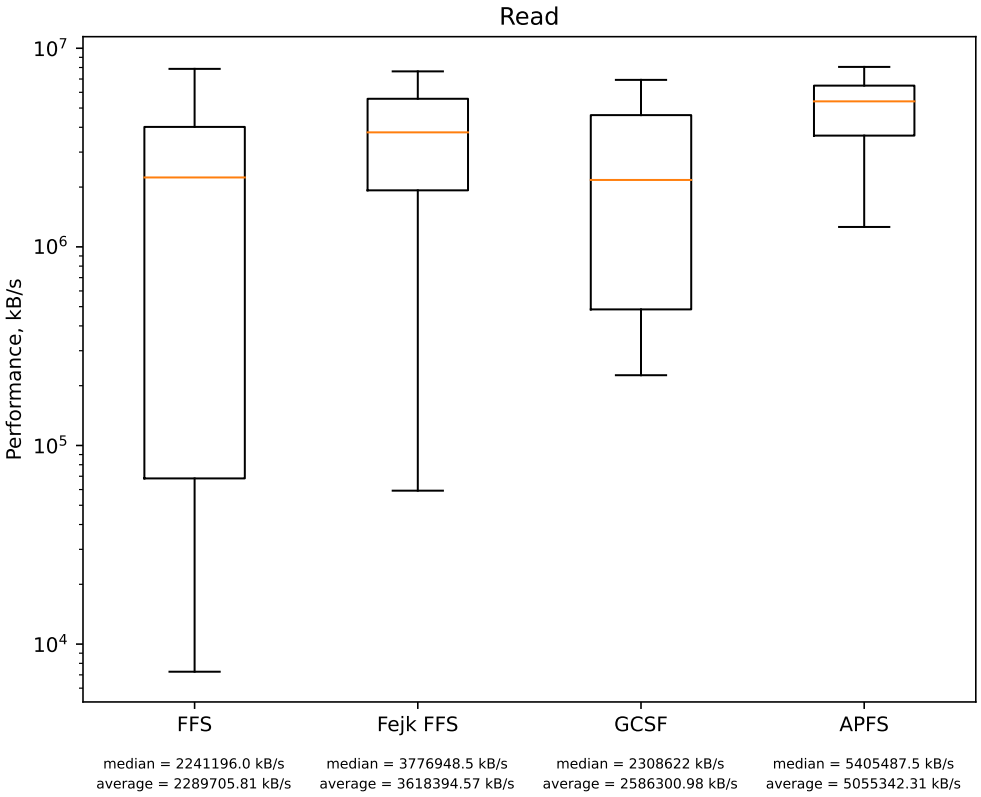

Figure 5.1: Box plot of the IOZone output for the Read test on the different filesystems

Figure 5.2 presents a box plot of the benchmarking results of the filesystems of the Write test. has the best write performance of the four filesystems, followed by . The cloud-based filesystems and have the worst median write performance, where has the worst performance of the filesystems. Comparing with the results of the Read test presented in Figure 5.2, it can also be noted that the write performance is significantly worse for the write operations than the read operations for all filesystems. The average performance of the Write test for is 12.9% of the Read performance. For , , and this percentage is 0.29%, 0.17%, and 0.06%.

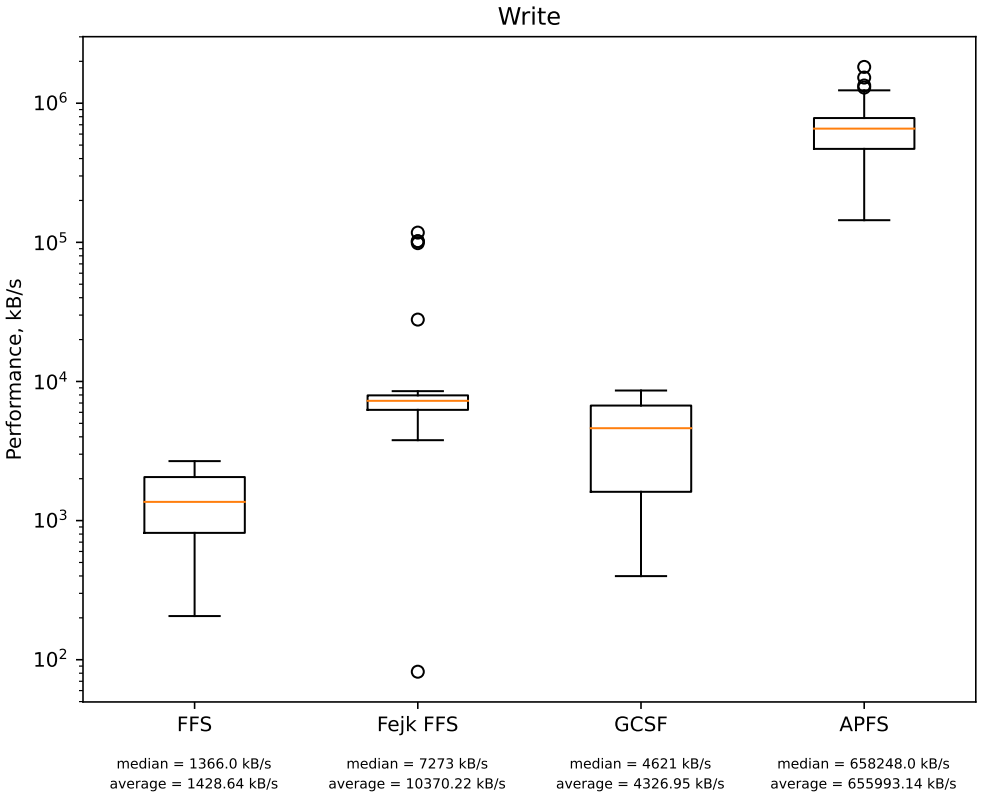

Figure 5.2: Box plot of the IOZone output for the Write test on the different filesystems

Figure 5.2 presents the result of the Re-Read test for the filesystems. The median performance of the filesystems are similar, with the performance of being lowest. The spread of the values for is greater than for the other filesystems but the filesystem still has the best median and average performance. While the average performance for the Re-Read test of is around 5 400 000 kB/s, the lowest performance was at 422 414 kB/s, namely for `file size = 32 768 kB, buffer size = 8 kB`. The spread of values is higher for than for and . has a higher average and median performance than .

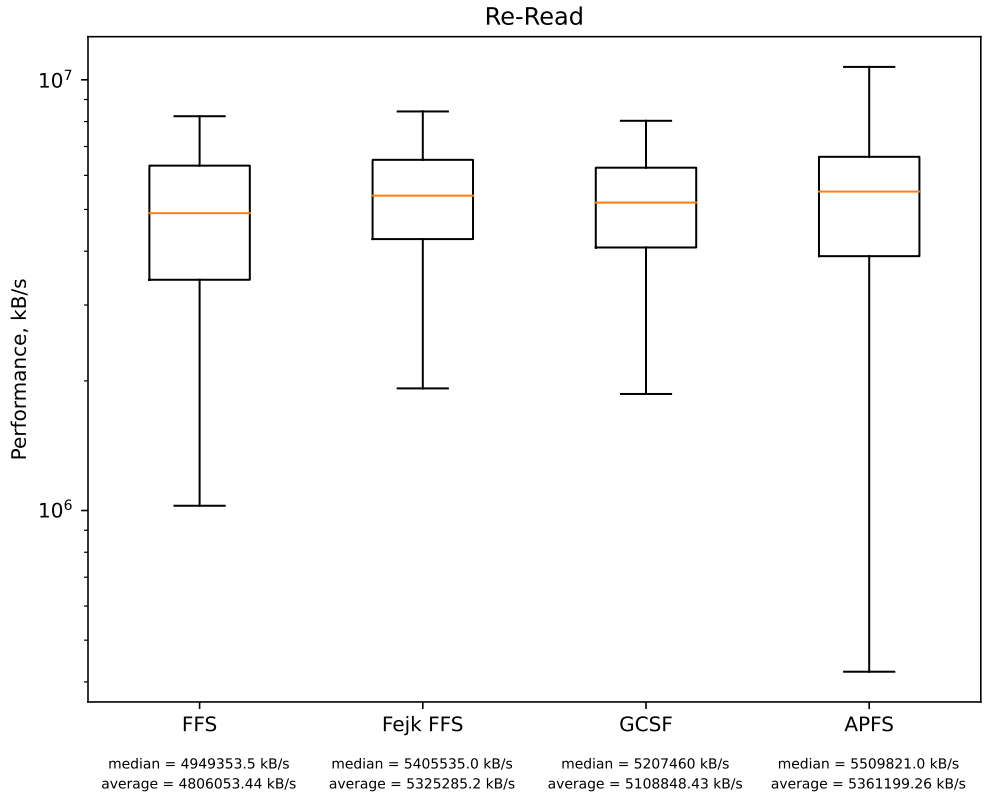

Figure 5.3: Box plot of the IOZone output for the Re-Read test on the different filesystems

Figure 5.2 presents a box plot for the Re-Write test for the filesystems. Similar to the Write test results presented in Figure 5.2, has the best performance of the filesystems, followed by and finally the two cloud-based filesystems. is the only filesystem that has higher average and median performance for the Re-Write test compared to its Write test, with 22% better median performance. The other filesystems has worse average and median performance for the Re-Write compared to their Write test. The median performance of the Re-Write test for , , and are 85%, 82%, and 64% of the median performance of their Write test.

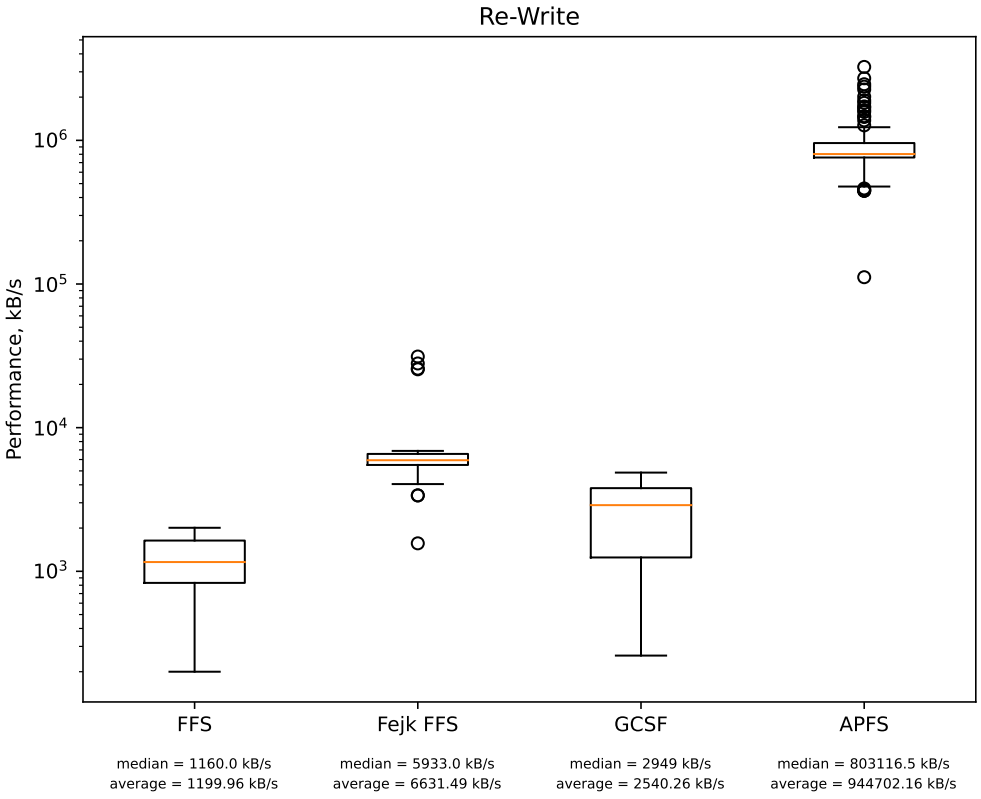

Figure 5.4: Box plot of the IOZone output for the Re-Write test on the different filesystems

Figure 5.2 presents a box plot for the Random read test for the different filesystems. The results are similar to the results for the Re-Read test presented in Figure 5.2, however, has bigger spread of values than has in the Re-Read test. has the best average performance of the filesystems, and has the best median performance. The median and average performance of is lower than the other filesystems.

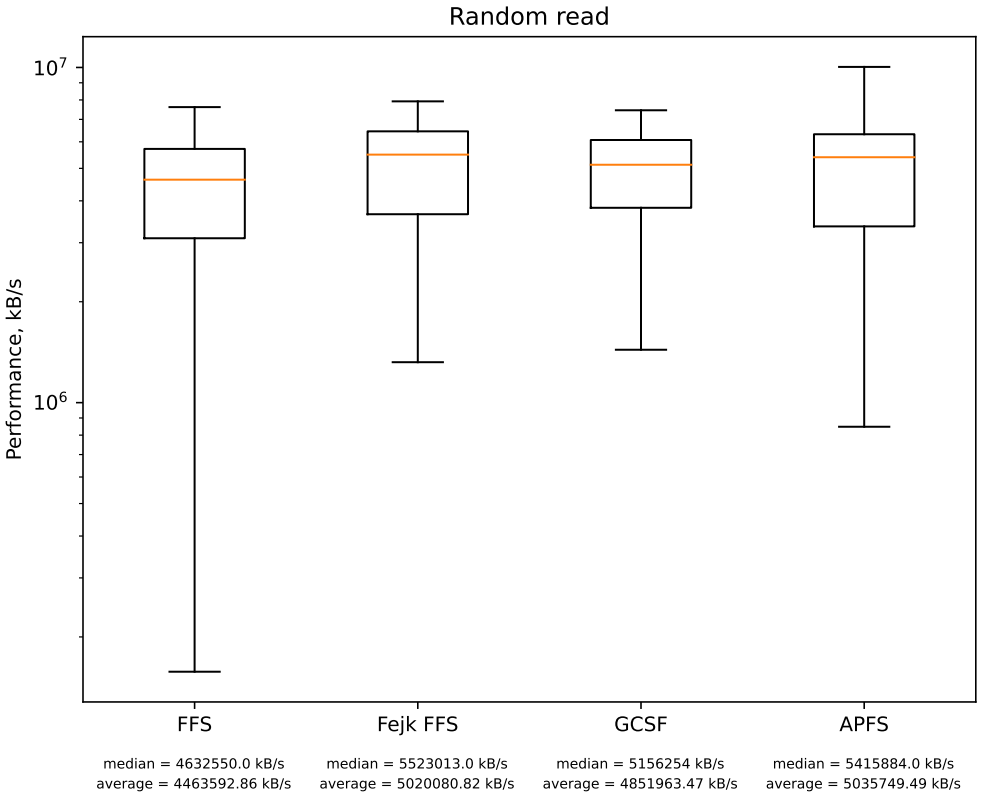

Figure 5.5: Box plot of the IOZone output for the Random read test on the different filesystems

Figure 5.2 presents a box plot for the Random read test for the different filesystems. has the highest performance, followed by and the cloud-based filesystems. has better performance than . The performances are similar to the Re-Write for all filesystems.

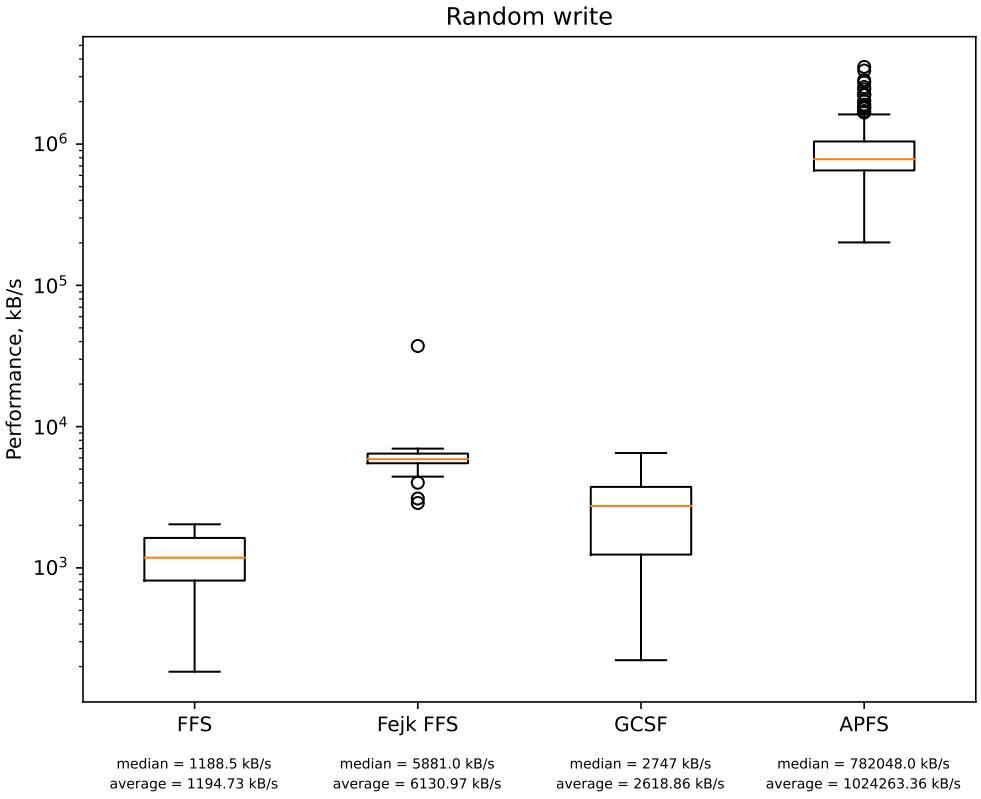

Figure 5.6: Box plot of the IOZone output for the Random write test on the different filesystems

does not have the best performance in any of the box plots presented. has better performance than for all tests run with the benchmarking. Following are diagrams presenting the performance of each benchmarking test per file size, for each filesystem. The raw data tables can be found in Appendix C. Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, and Figure 5.2 presents the performances of the Read, Write, Re-Read, Re-Write, Random Read, and Random Write tests for .

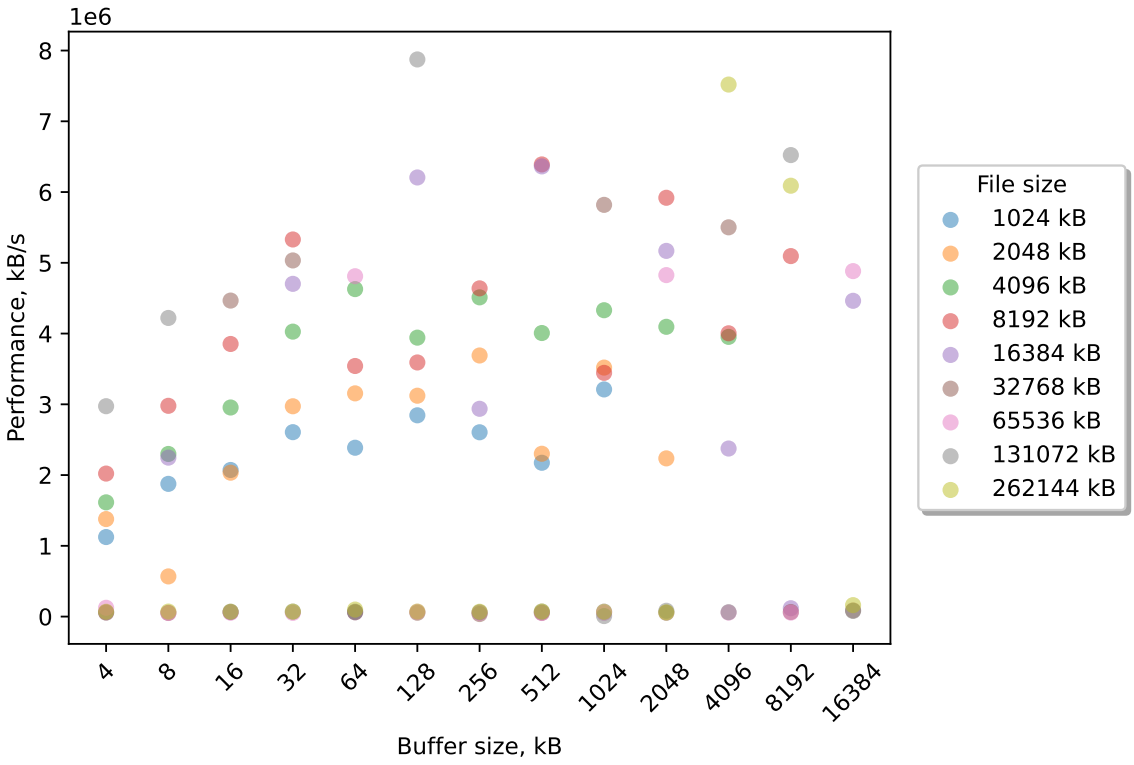

Figure 5.7: IOZone output for FFS Read

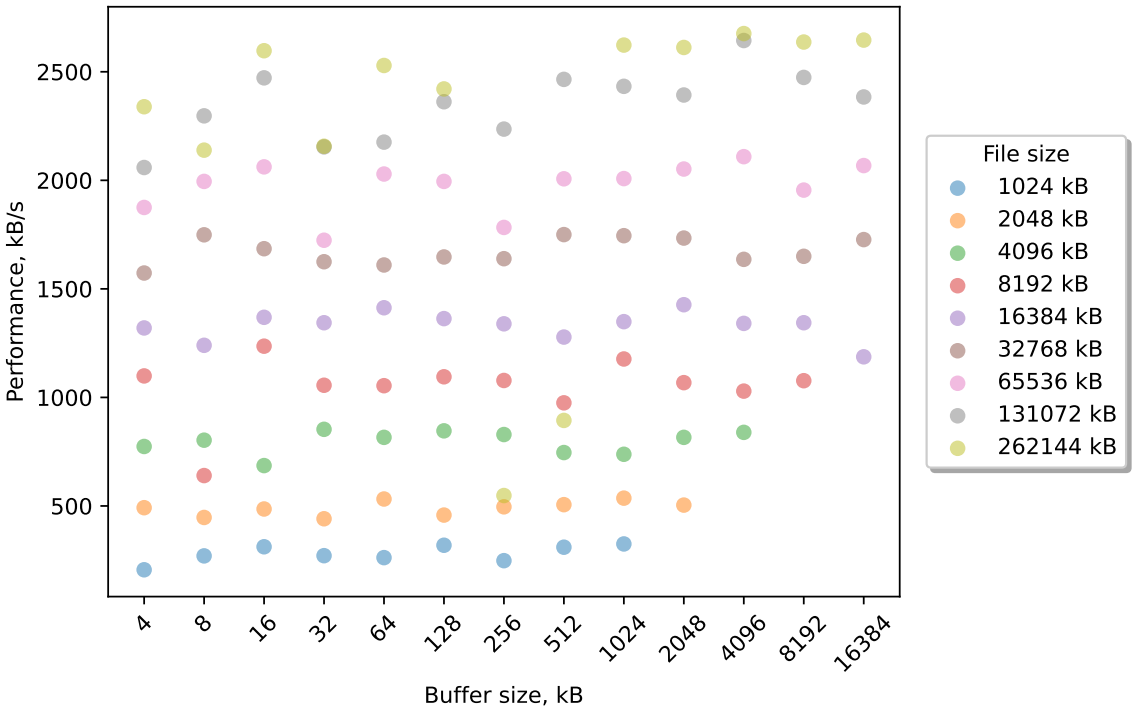

Figure 5.8: IOZone output for FFS Write

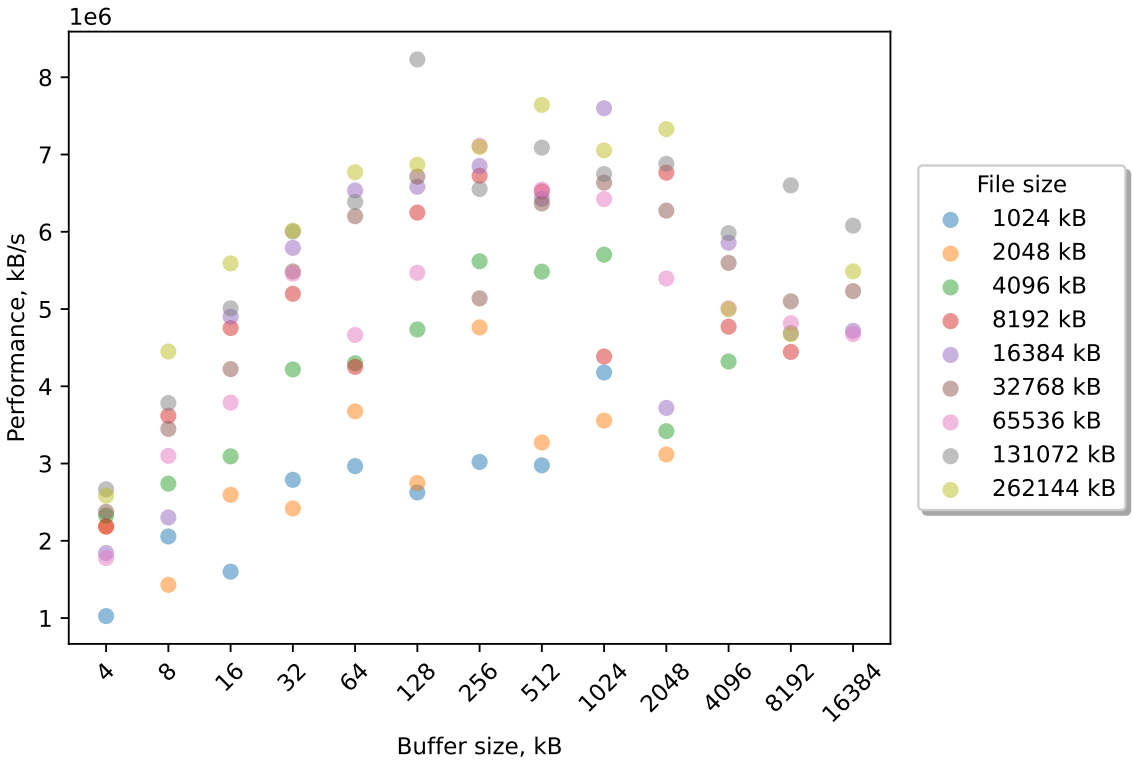

Figure 5.9: IOZone output for FFS Re-Read

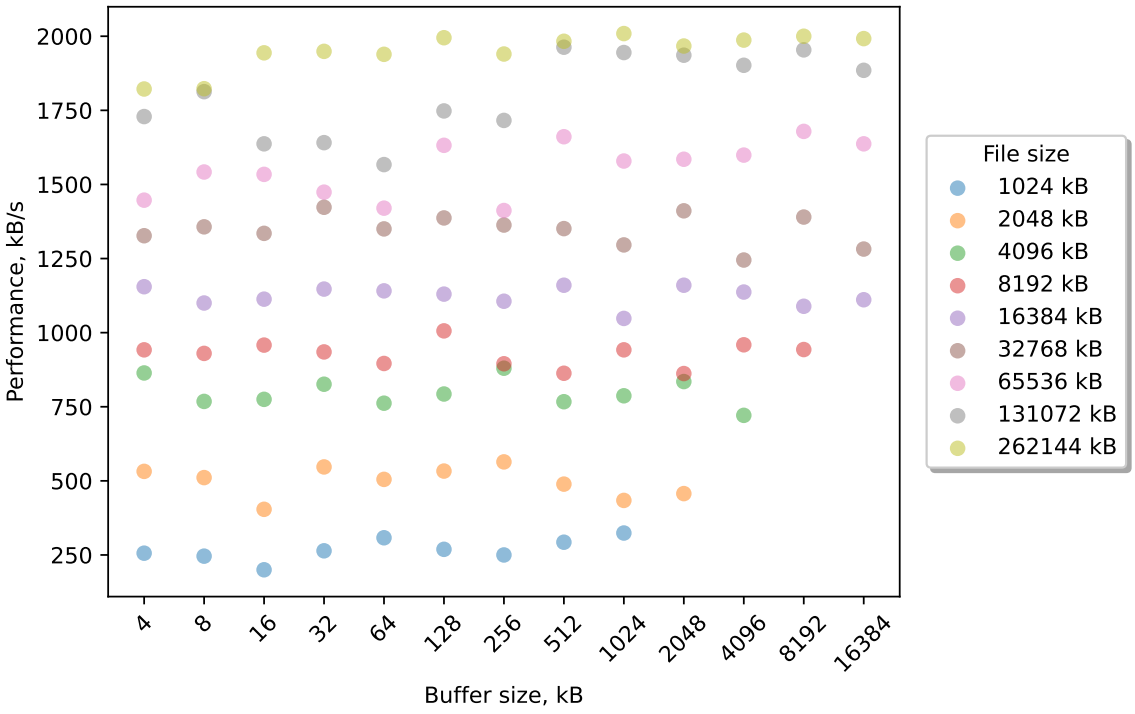

Figure 5.10: IOZone output for FFS Re-Write

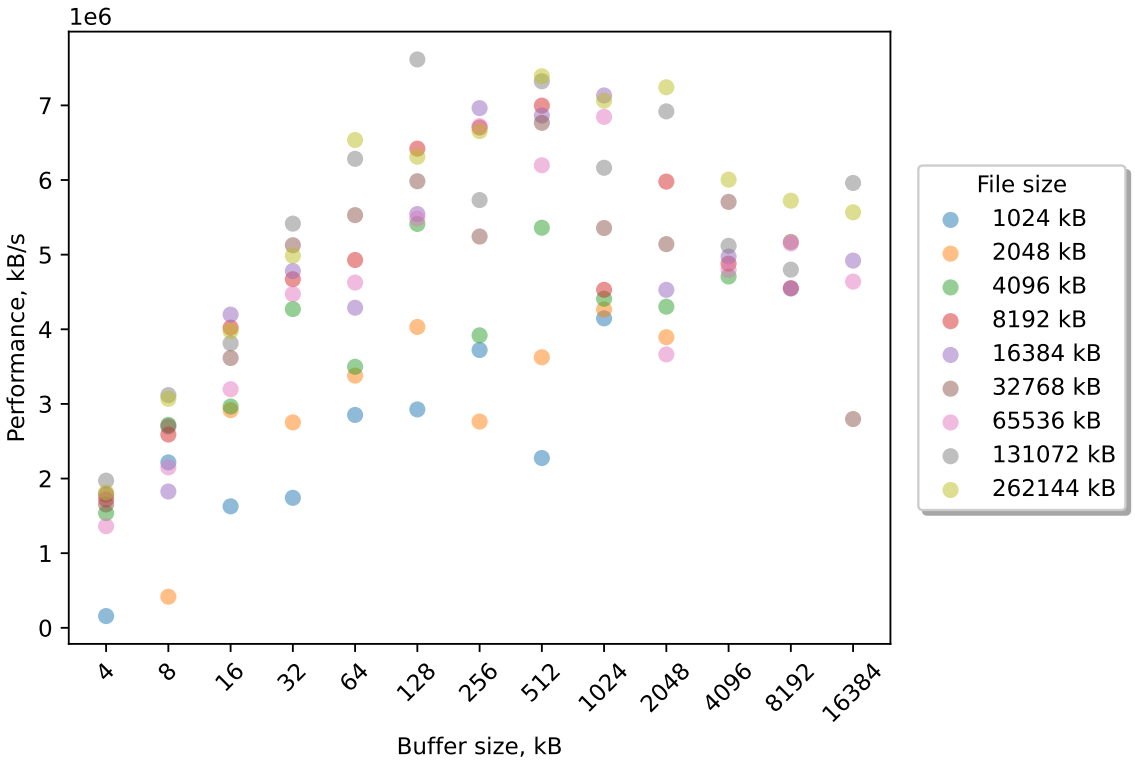

Figure 5.11: IOZone output for FFS Random read

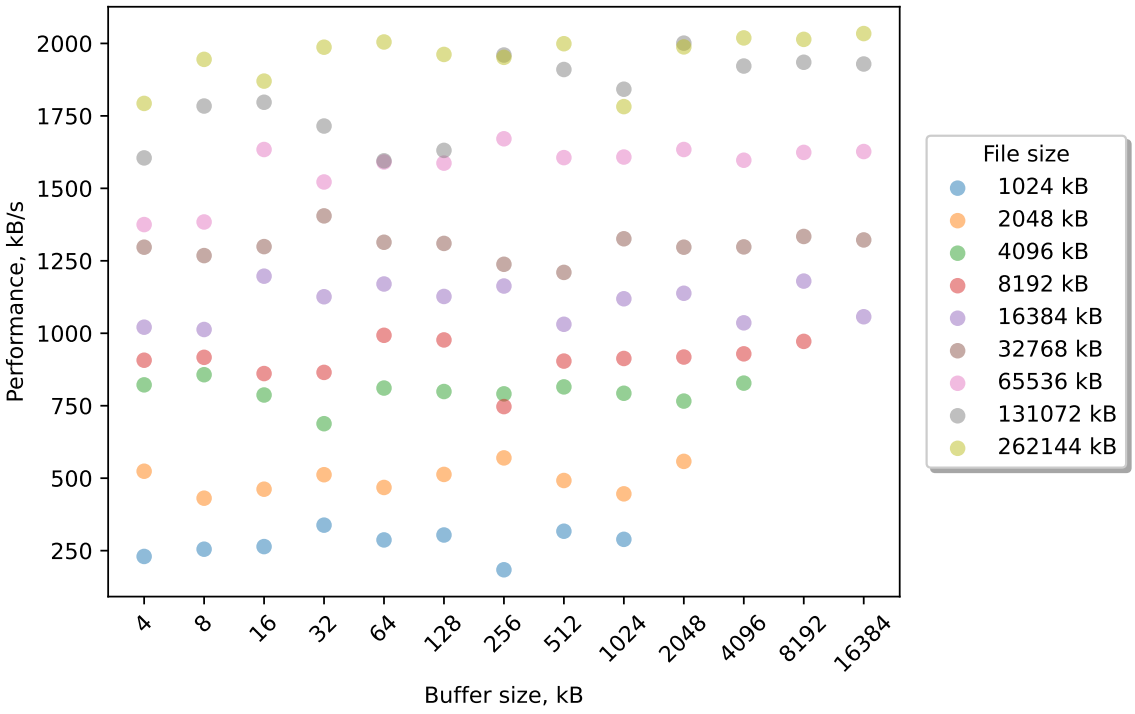

Figure 5.12: IOZone output for FFS Random write

Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, and Figure 5.2 presents the performances of the Read, Write, Re-Read, Re-Write, Random Read, and Random Write tests for .

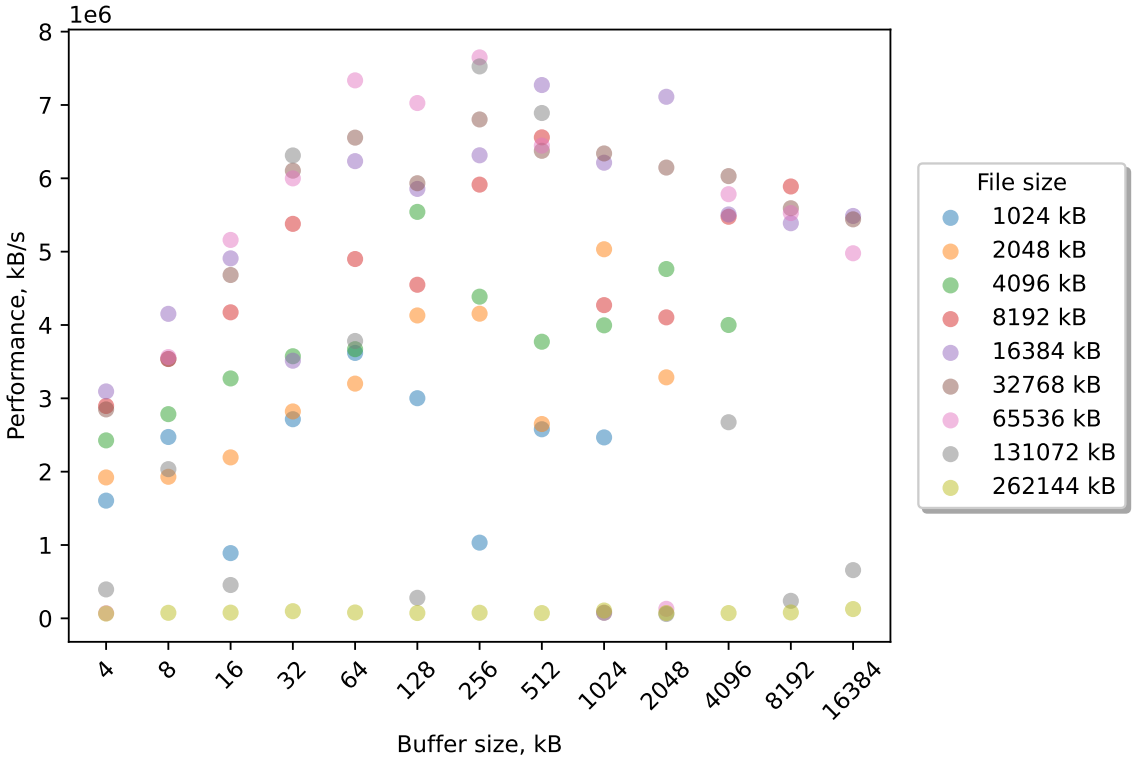

Figure 5.13: IOZone output for Fejk FFS Read

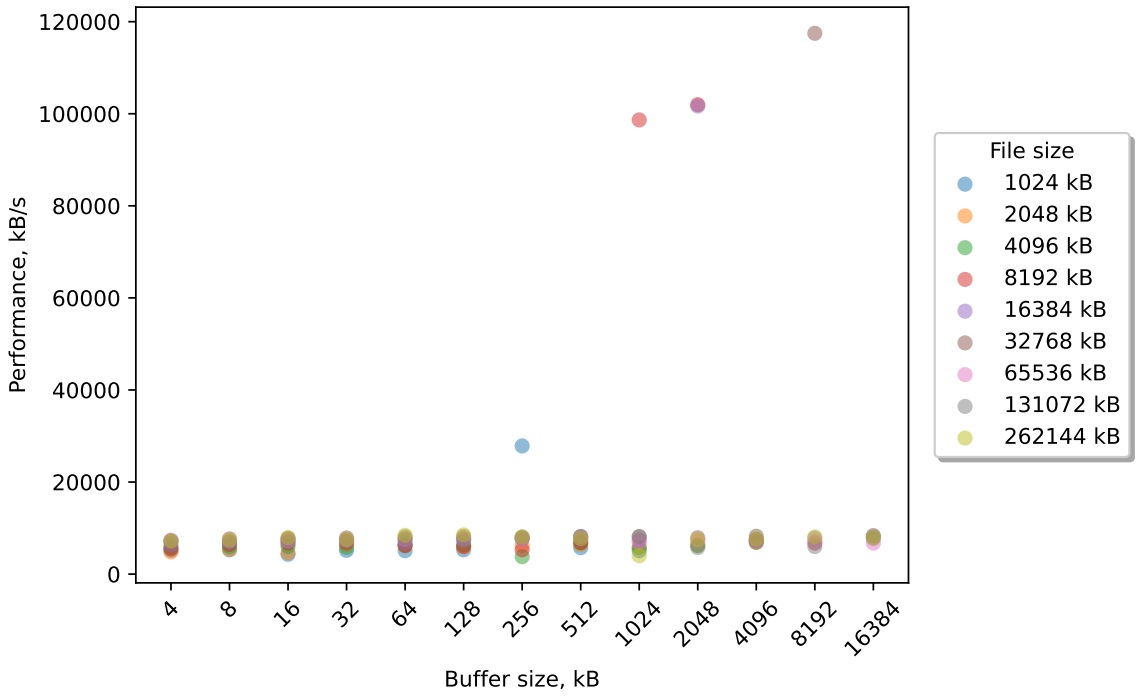

Figure 5.14: IOZone output for Fejk FFS Write

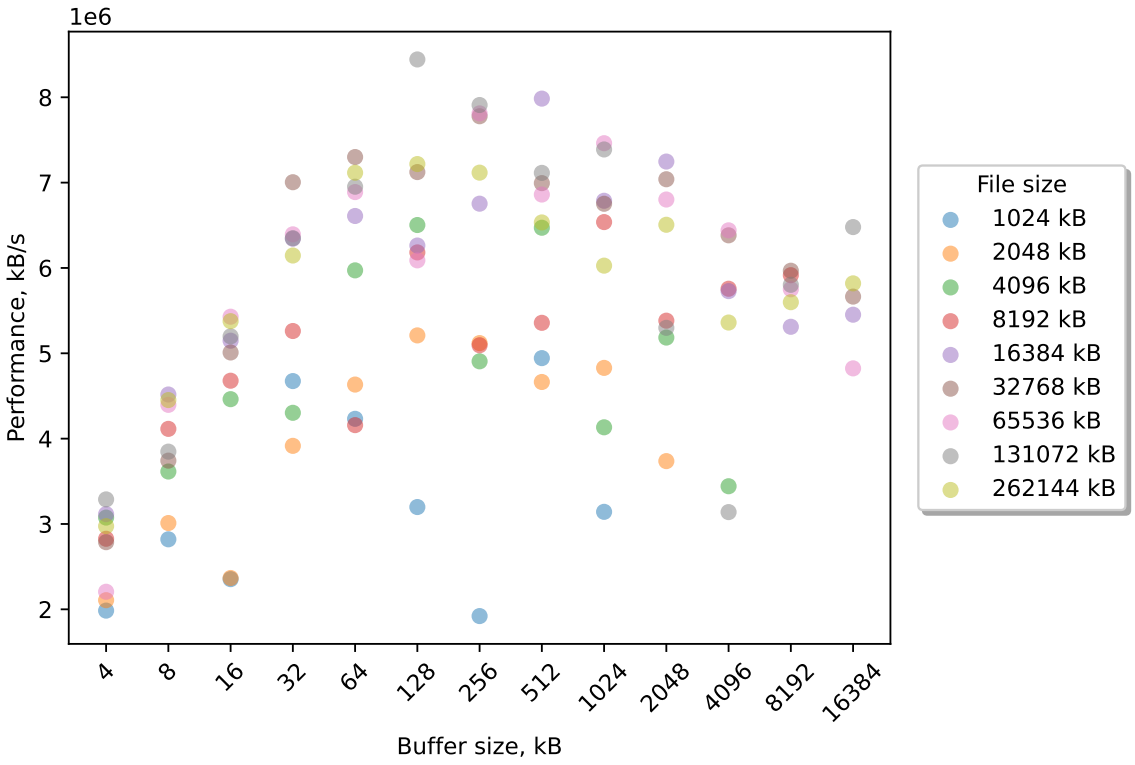

Figure 5.15: IOZone output for Fejk FFS Re-Read

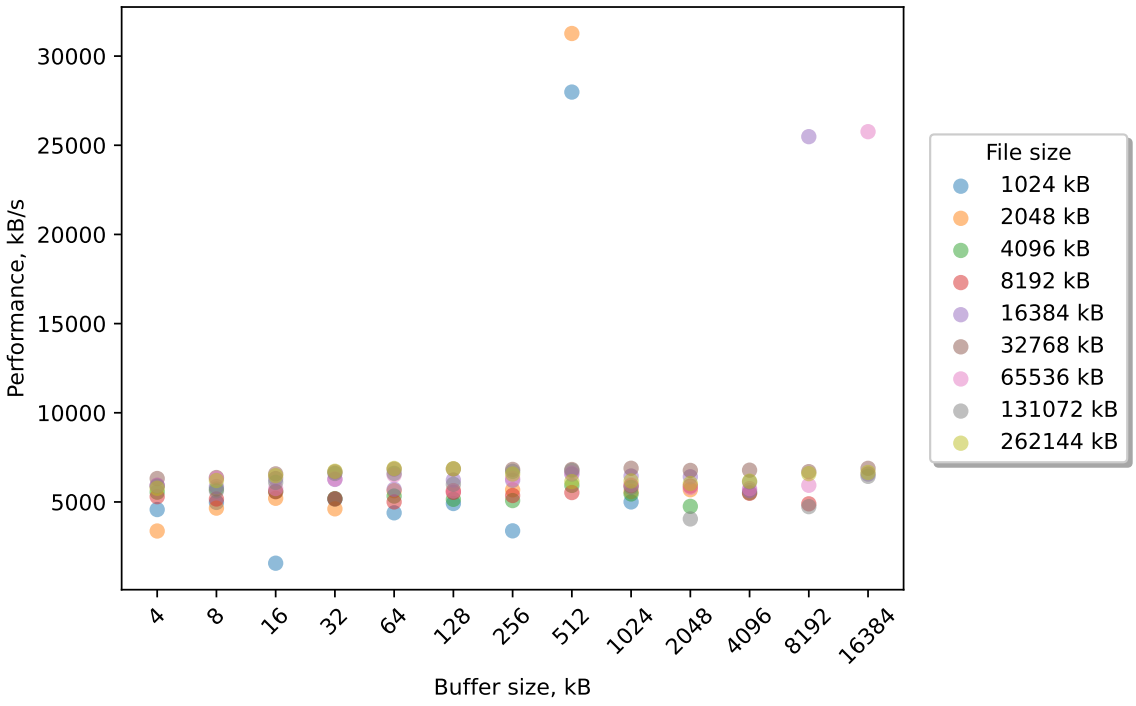

Figure 5.16: IOZone output for Fejk FFS Re-Write

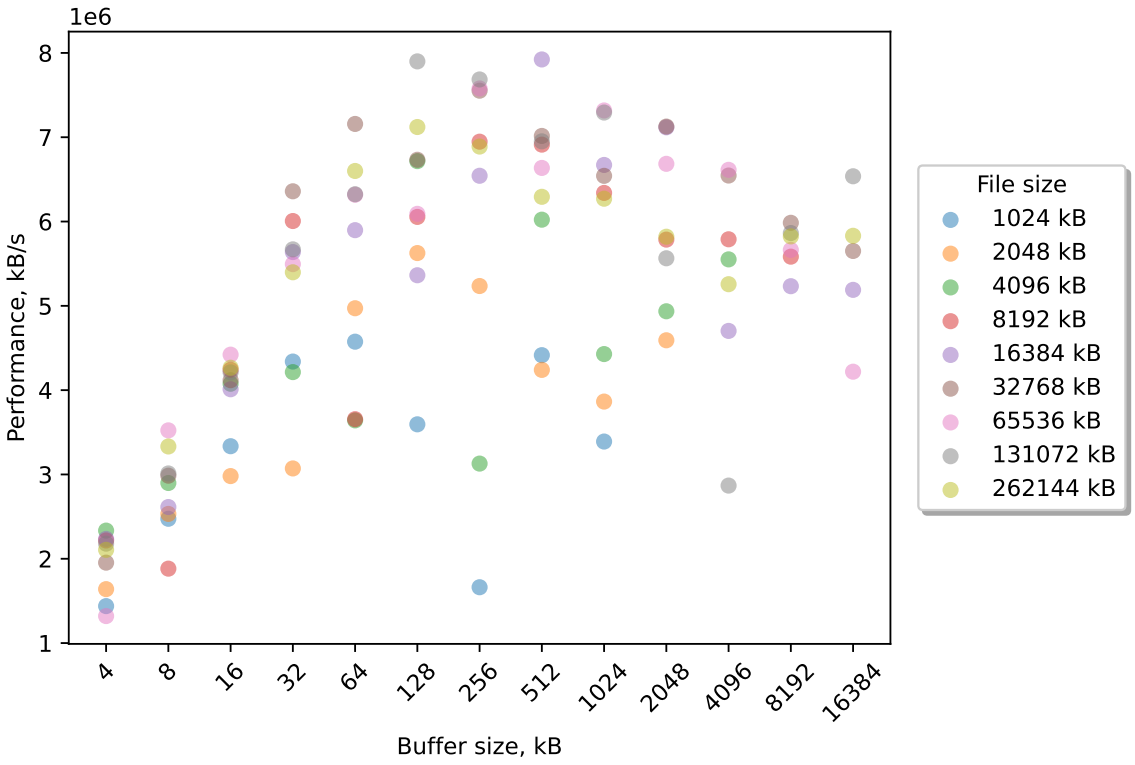

Figure 5.17: IOZone output for Fejk FFS Random read

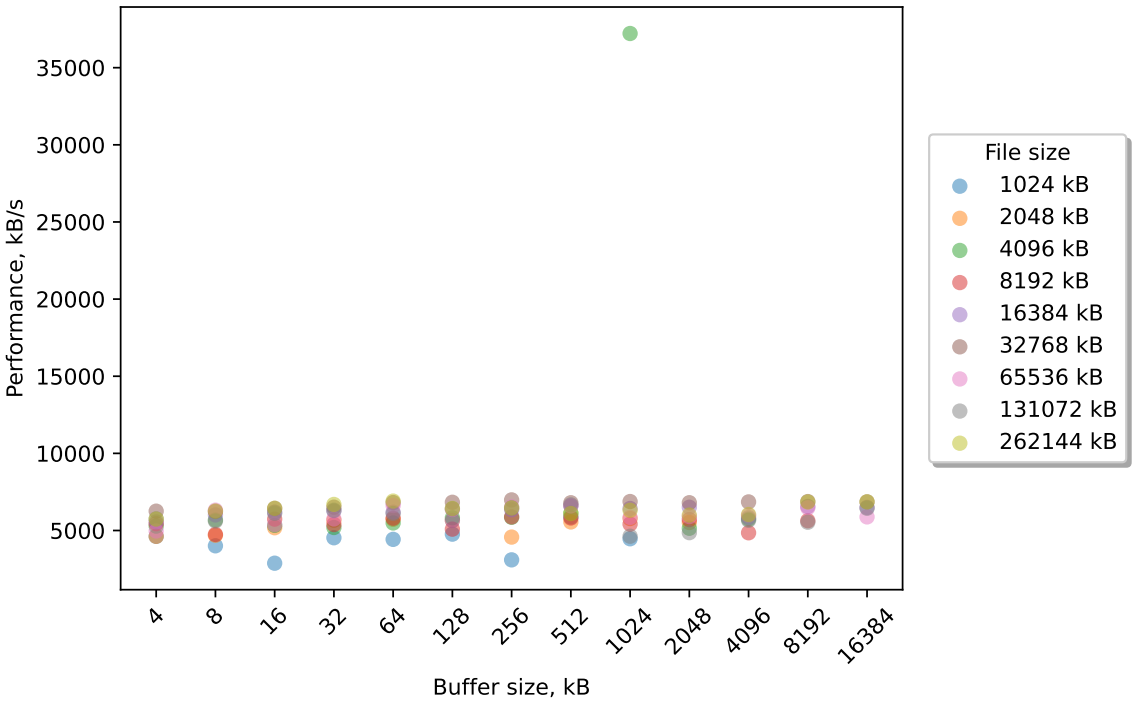

Figure 5.18: IOZone output for Fejk FFS Random write

Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, and Figure 5.2 presents the performances of the Read, Write, Re-Read, Re-Write, Random Read, and Random Write tests for .

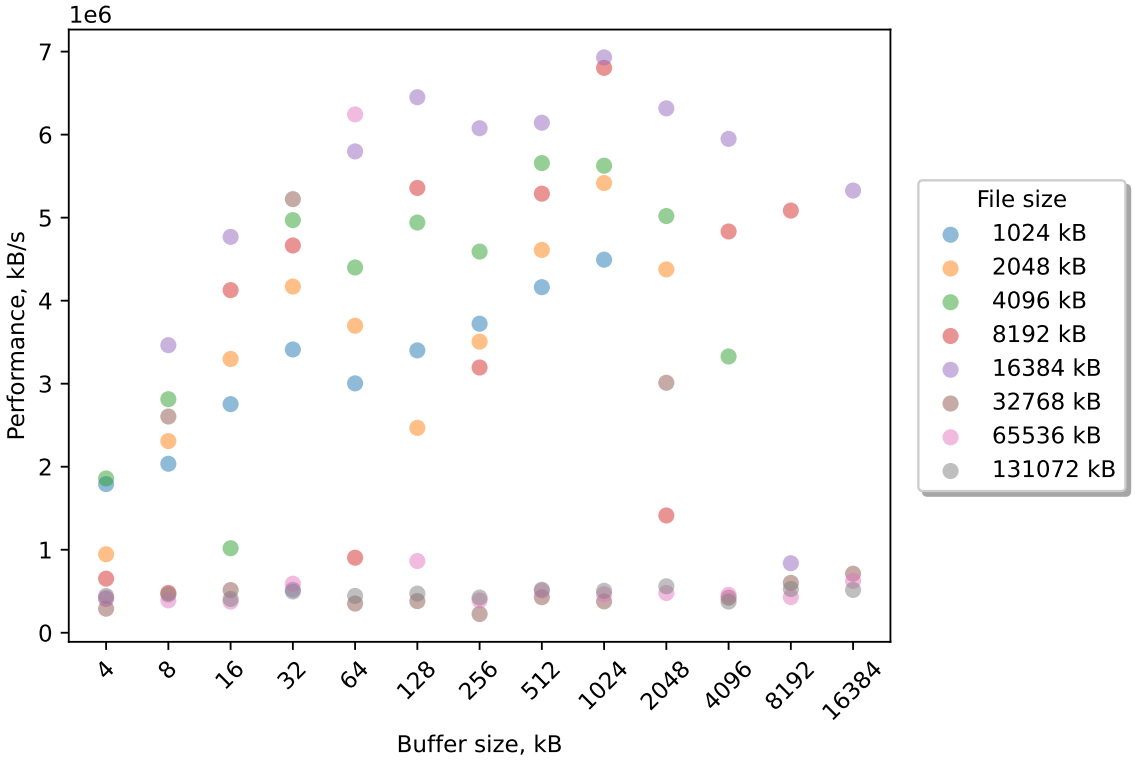

Figure 5.19: IOZone output for GCSF Read

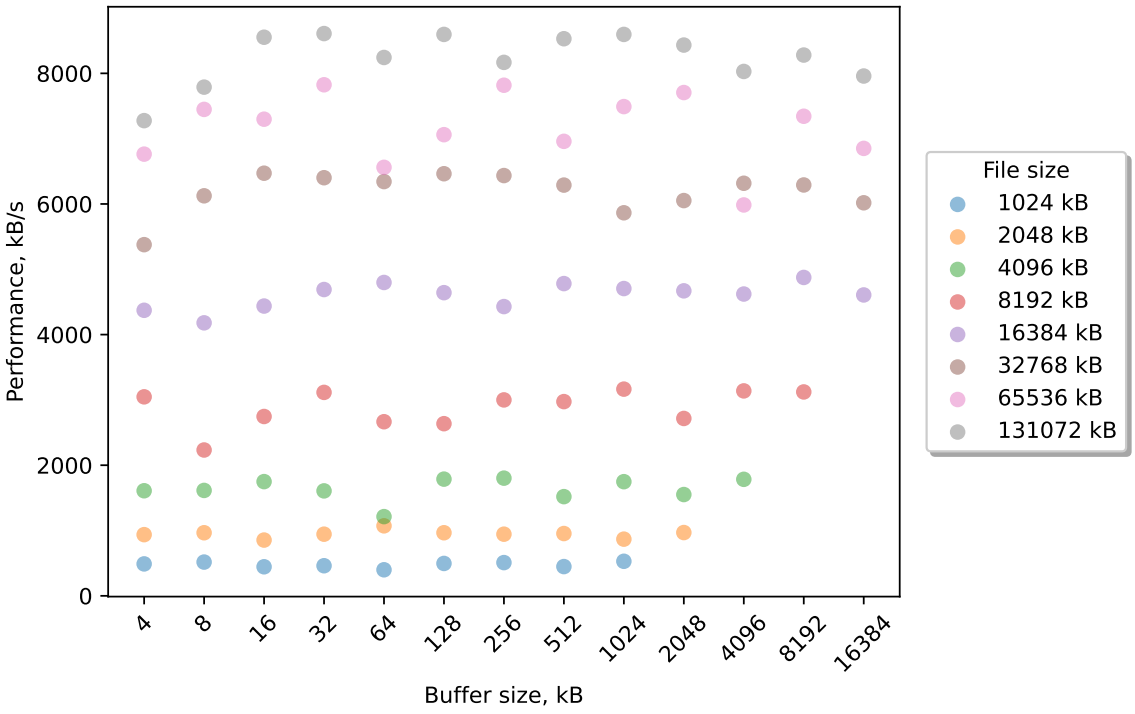

Figure 5.20: IOZone output for GCSF Write

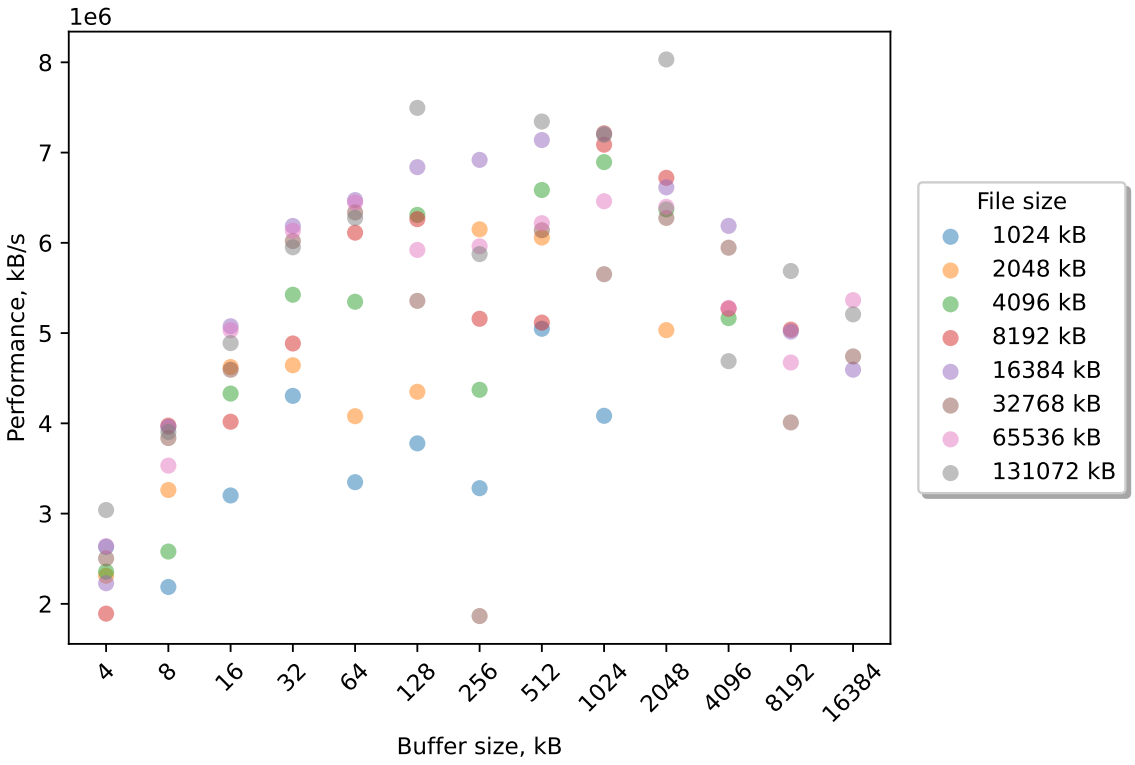

Figure 5.21: IOZone output for GCSF Re-Read

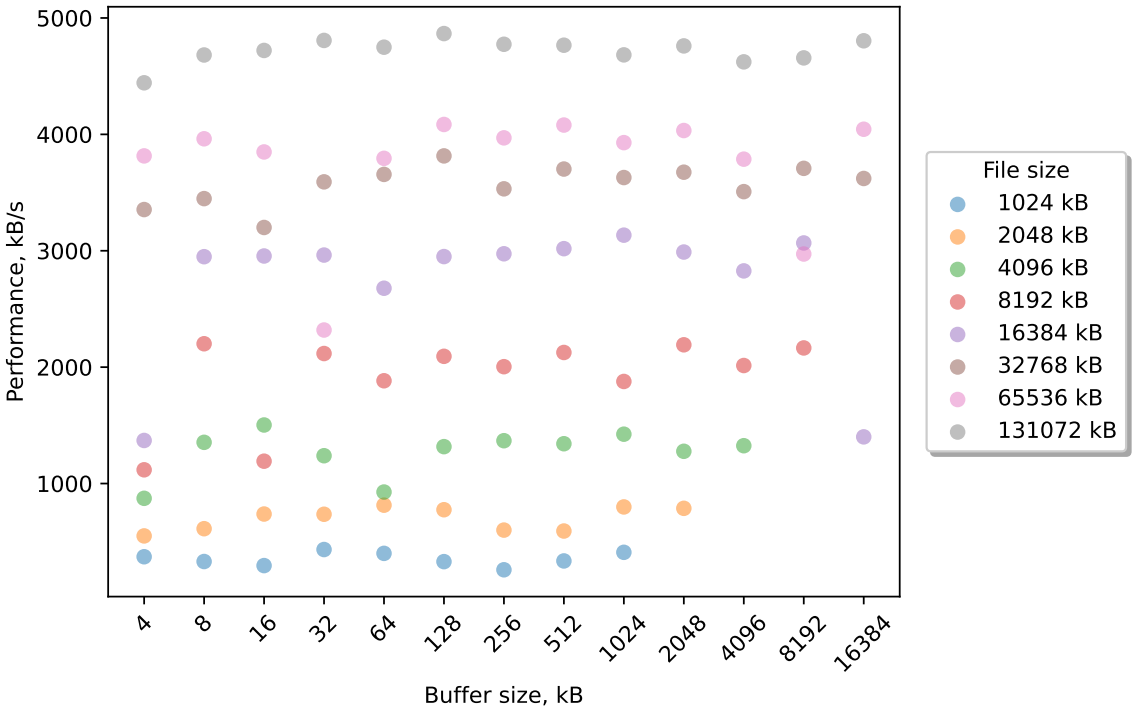

Figure 5.22: IOZone output for GCSF Re-Write

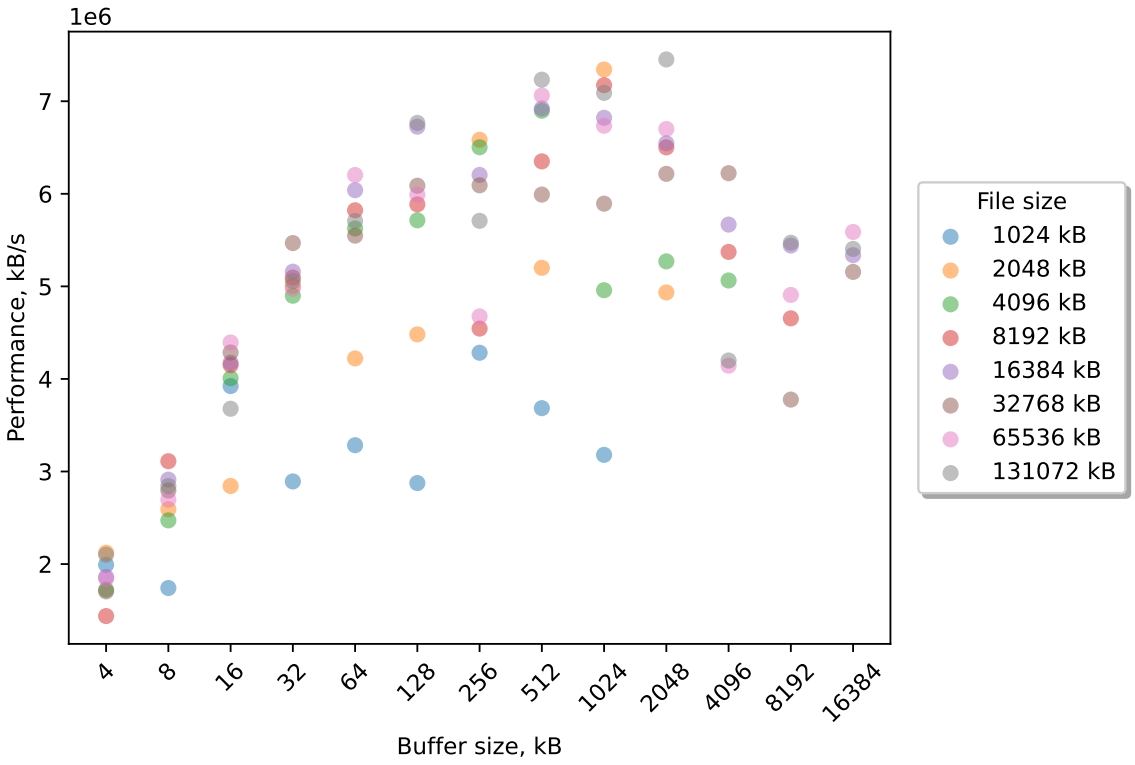

Figure 5.23: IOZone output for GCSF Random read

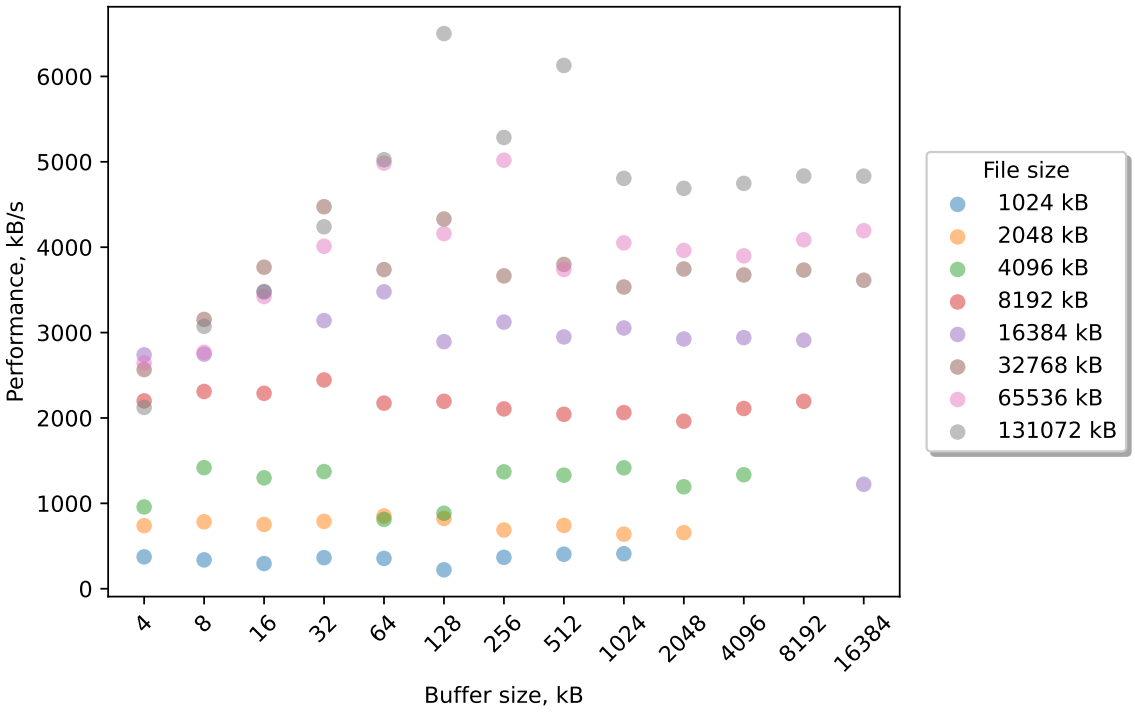

Figure 5.24: IOZone output for GCSF Random write

Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, Figure 5.2, and Figure 5.2 presents the performances of the Read, Write, Re-Read, Re-Write, Random Read, and Random Write tests for .

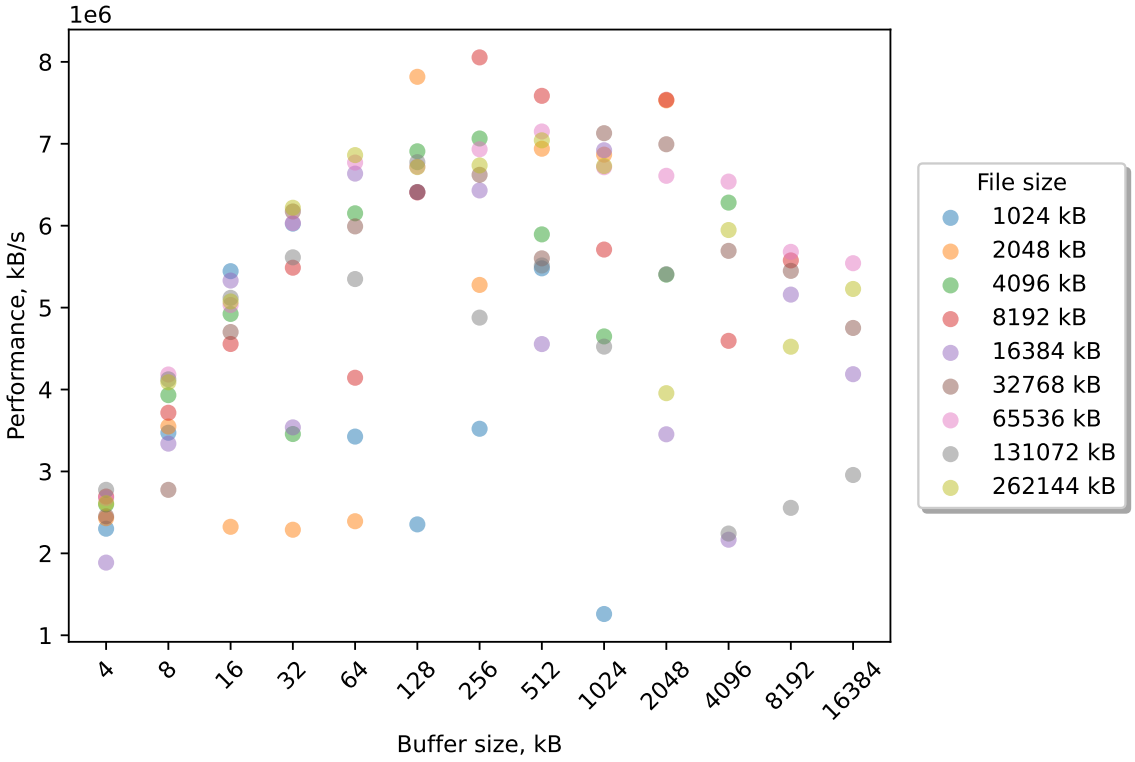

Figure 5.25: IOZone output for Read

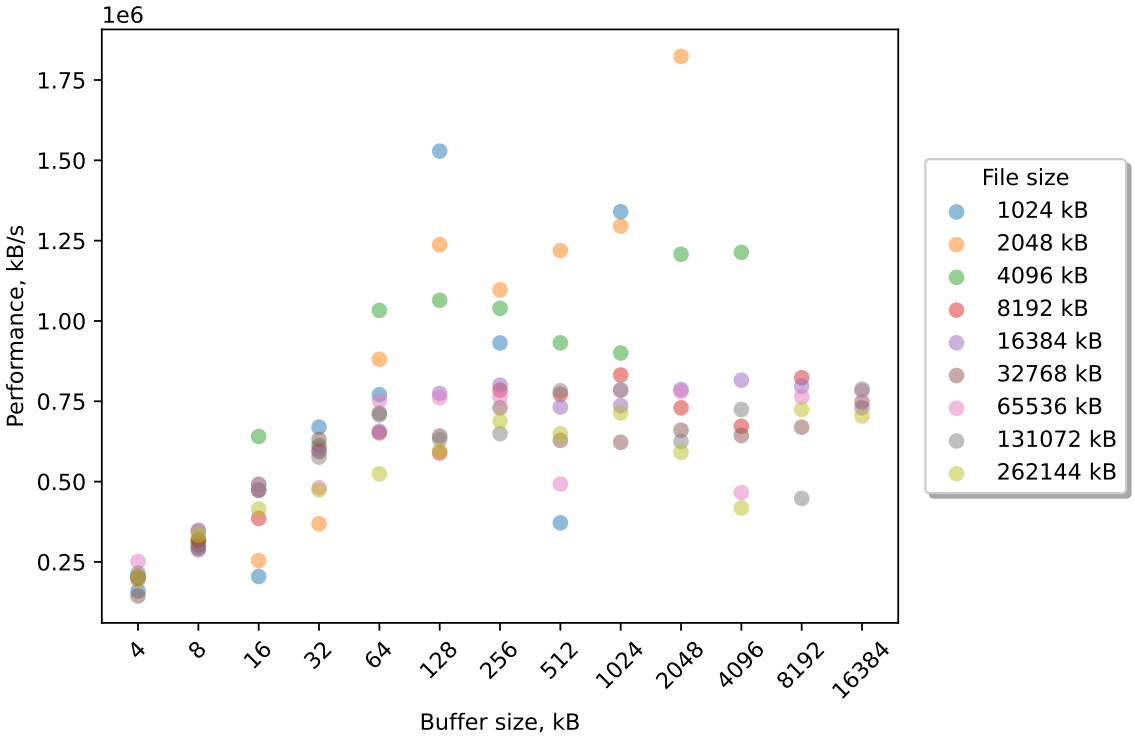

Figure 5.26: IOZone output for Write

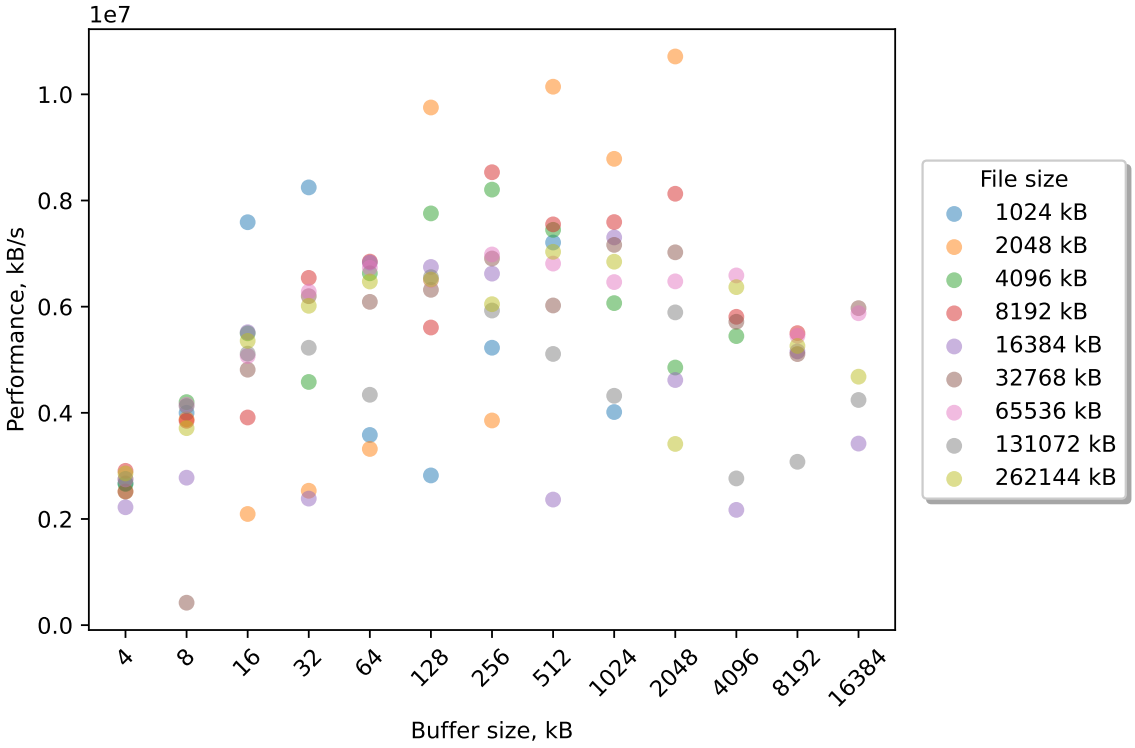

Figure 5.27: IOZone output for Re-Read

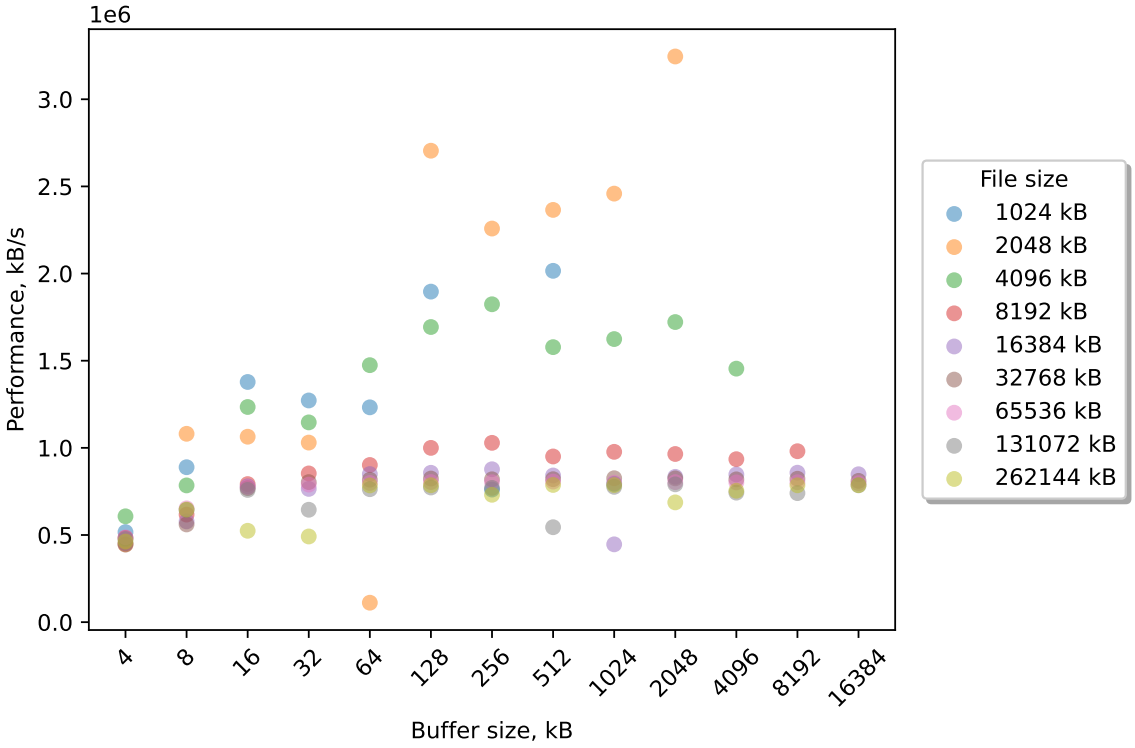

Figure 5.28: IOZone output for Re-Write

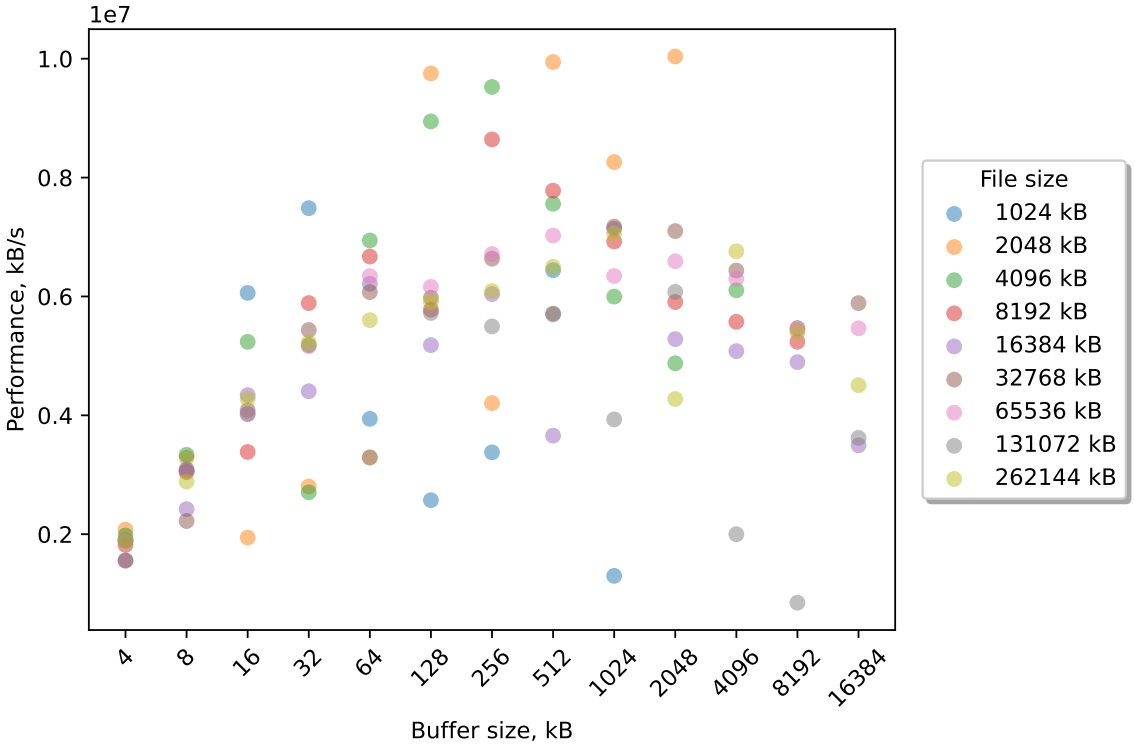

Figure 5.29: IOZone output for Random read

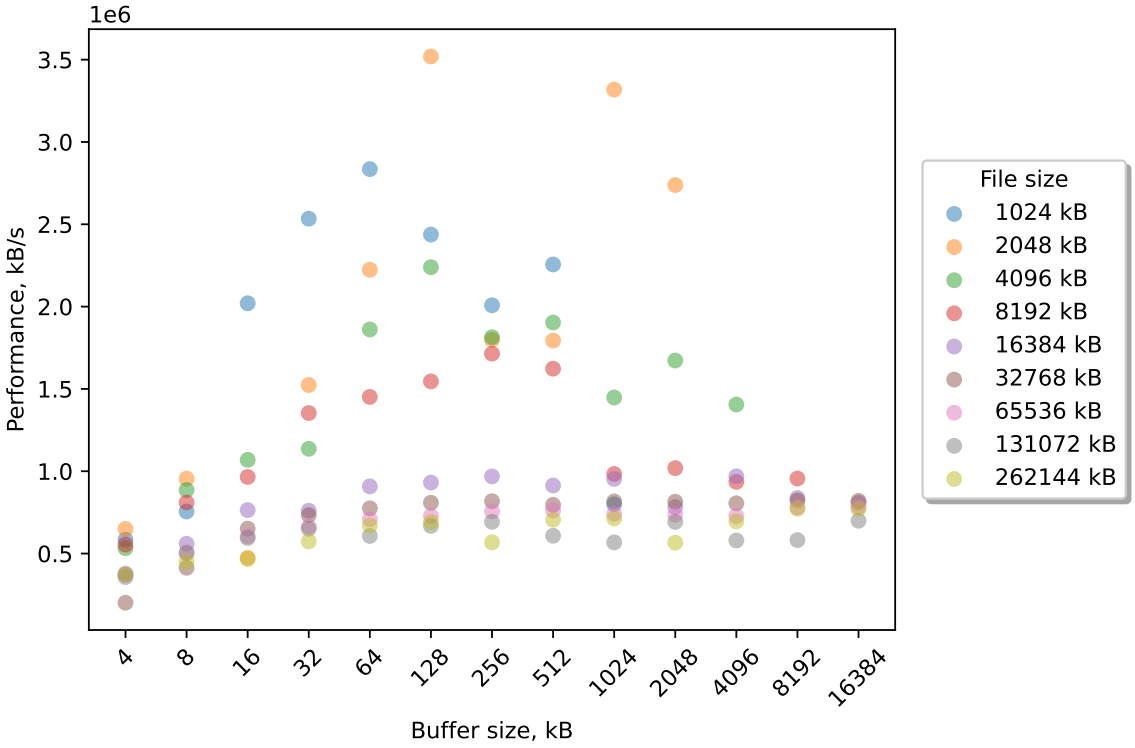

Figure 5.30: IOZone output for Random write

Chapter 6

DISCUSSION

The benchmarking results presented in the previous chapter are analyzed and discussed in Section 6.1. Furthermore, 's deniability and security are analyzed and discussed in Section 6.2. Potential societal and environmental impacts of are presented in Section 6.3.

6.1 Filesystems

Figure 5.2, Figure 5.2, and Figure 5.2 show that performs poorly for the Read tests with a small buffer size. Beginning at 4 kB buffer size, the performance in general increases with the first few buffer sizes. This could be influenced by the overhead of the read operation. Overhead of the read operation includes, among other things, the time to fetch the images from Flickr if they are not in the filesystem cache, and decrypting the images which is required even if the images are in the filesystem cache. Further, it is expected that the Re-Read tests performs better than the Read tests when the file size is small enough to fit in the filesystem cache. This is also a conclusion that can be drawn from the result. However, looking at Figure 5.2, and Figure 5.2, we can also see that the Re-Read tests overall performs better for file sizes bigger than the 5 MB cache file size limit as well. Especially for the file sizes 32 768 kB to 262 144 kB, the performance of the Read tests is in general very low, as opposed to the Re-Read tests. As Figure 5.2 and Figure 5.2 shows, the average performance of the Re-Read tests are more than double the average performance of the Read tests for . The reason why the Re-Read tests performs better than the Read test for files bigger than the filesystem cache size limit is most probably due to kernel caching of the files. The performance is higher than a usual internet connection bandwidth would be. An internet connection's bandwidth is often limited to a maximum of 1 Gbit/s = 125 000 kB/s by the ISP, depending on the subscription.

For the read operations of the cloud-based filesystems that exceed 125 000 kB/s, the data has most probably been provided my some kind of cache. Note that this limit is much higher than the measured reference bandwidth presented in Section 5.2, but we use this limit as a reference of a maximum bandwidth of the connection rather than as the actual bandwidth of the connection. All the data points for the Re-Read tests for and exceed 1 000 000 kB/s indicating that all these files were served by some cache. For , the files over 5 MB were most probably served from the kernel cache as they cannot fit in the filesystem cache. As the performances differ significantly between the filesystems, it is possible that not all data was provided by the kernel cache, but that some data was provided by internal filesystem caches. Otherwise, the data could be more similar, assuming the kernel cache has the same performance for all filesystems tested. Many data points of the Read tests for and also exceed 125 000 kB/s, indicating that some data returned when running this test were also served by some cache. This data could be saved in the cache after the Re-Write tests which precedes the Read tests. The data could also be cached after the first Read test of the file which could explain the increasing performance for the smaller buffer sizes in the beginning of the tests. The biggest file sizes have lower performance than the other file sizes for the Read tests of and , indicating that these file sizes might not have been stored in any cache.

Other than the data being provided by the kernel cache, it is also possible that IOZone does not close the file before it is read again, meaning that the file can be kept in memory for and as they cache open files that have been read. IOZone does not specify when the file is closed, however it could be assumed that the files are closed between tests. If the file would not be closed after a write test, the write tests of should have the same performance as the write tests of as neither of the two filesystems save the data in their storage medium until the file has been closed. As the write tests of and are not similar it is improbable that the file is not closed after the write tests. The Re-Read test is performed after the Read test, and as the Re-Read test is significantly faster than the Read test for both FFS and , even for files bigger than the cache limit of the filesystems, It is possible that the files are not closed between the two tests. However, if this was the case, the Re-Read tests would probably be more similar for the filesystems than the results show, as the data could be served just as fast by both

filesystems. The theory that the kernel cache is providing better performance of the Re-Read is more probable as the tests are less consistent between the two filesystems. If the kernel would not have the requested data in its cache, the filesystem would have to be called. This could be the explanation why the filesystem performances differ significantly between the two filesystems. Cache misses in the kernel cache requires and to get the data from their cache, if the files were small enough, or from their respective storage medium. This introduces performance differences between the filesystems. Cache misses in the filesystem cache is less probably as there are no more files being saved, meaning that the files will not be removed from the cache.

While the Re-Read and Random Read tests increases in performance for the first buffer sizes for , the performance also decreases eventually. Looking at the data presented in Figure 5.2 and Figure 5.2, the buffer sizes 4096 kB to 16 384 kB have, in general, lower performance than the buffer sizes 256 kB to 2048 kB, for the same file size. This indicates that the optimal buffer size for read operations on previously read files is not the biggest possible buffer size, but rather around 512 kB, depending on the file size. The biggest file size has its best performance for a buffer size of 512 kB. Looking at the 131 072 kB file size, it peaks for both the Re-Read test and the Random read test at `buffer size = 128 kB`, and its performance for that buffer size for both tests are higher than any other performance of any file size or buffer size in both tests, for the filesystem. This is interesting because the 131 072 kB file size does not always outperform the other file sizes. Looking at the Re-Read test, the 262 144 kB has the best performance for seven of the 13 tests while the 131 072 kB file size has the best performance for five buffer sizes, namely the first one, the last three and for `buffer size = 128 kB`. The 16 384 kB file size has the best performance of the test once, for `buffer size = 1 024 kB`. Considering how fast the operations actually are, it is possible to understand why the values can fluctuate. For instance, the Re-Read test for using `file size = 16 384 kB, buffer size = 1 024 kB` has a performance of 7 599 353 kB/s. Transferring 16 284 kB at 7 599 353 kB/s takes 2.14 ms. If it would take 10% more time, the performance would instead be under 7 000 000 kB/s, meaning that the 262 144 kB file size would have better performance for the same buffer size. However, if it instead would take 10% less time to perform the `file size = 16 384 kB, buffer size = 1 024 kB` Re-Read test, it would reach a performance of

8 443 726 kB/s which would be the highest performance of the test on of all file sizes and buffer sizes. Small time differences can significantly affect the performance of the tests. The time of the filesystem operations can be fluctuated by process scheduling and the performance of the kernel cache, among other things.

The performance of the write operations of are highly influenced by the file size. As shown in Figure 5.2, Figure 5.2, and Figure 5.2, bigger file sizes implicates better performance for the write operations, generally. The best performing file size was most often the largest file size, 262 144 kB. Furthermore, the biggest file size of the tests perform better for the Write test than for the Re-Write and Random Write test. This is interesting as the Write test includes the overhead of creating the files before writing to them, which Re-Write and Random Write does not.

One interesting comparison is between the benchmark results of and . Both filesystems are cloud-based filesystems dependent on an internet connection to their respective storage servers. Looking at the box plots in Section 5.2, we can see that outperforms in both average performance and median performance for all tests. However, does not have any data for the biggest file size while has data for it. Looking at Figure 5.2 and Figure 5.2, we can see that performs better than for many of the bigger file sizes for the Re-Read test. It is possible that would perform better than for the 262 144 kB file size test if could run the test. However, even if that would be the case, it is also possible that would still perform better overall. One reason that generally outperforms could be because the cache stores the encrypted version of the image, meaning that before the data is read, the image must first be decrypted and decoded. As Google Drive provides the raw data of the file stored, can store the raw data in its cache meaning that the data in the read operation can be returned faster. If Google Drive caches the raw file data as well, it does not have to decrypt the data when serving it to . also outperforms in all the write tests. The reason could be that does not have to encrypt the data nor encode it as images. This could save significant computation time. The average (reference point) bandwidth measured when the two filesystem benchmarks were run are similar, indicating that there was not a big difference in the internet connection to the measurement servers. However, as this does not measure the actual internet connection to the , the

actual bandwidth of the filesystem could be different from this value. However, even assuming that the bandwidth of the internet connections of and are equal, can still benefit from fewer REST calls. As Google Drive is a filesystem, the inode table of the filesystem, or how ever the filesystem is organized, can be stored on the service without exposing it to the user. For instance, assuming that Google Drive uses an inode table like , the inode table would never have to be downloaded and uploaded by . By simply uploading a file and specifying its path and filename, the inode table does not have to be modified by the user but can be handled by Google Drive in the background, potentially after the request has completed requiring less time for the file upload request. Meanwhile, has to upload the inode table after every file modification. Additionally, the old version of the file and inode table must be removed. This requires to perform at least four requests for all modifications:

- Upload a new image with the new file content,
- Upload a new image with the new inode table content,
- Remove the old file, and,
- Remove the old inode table

Although, removing the old images is performed on another thread and does not affect the filesystem operation time. However, it can affect the congestion of the bandwidth to the if it is performed at the same time as another request, such as a file upload. Meanwhile, uploading a modified file to Google Drive requires one call using the file's ID [101] which will perform the same functionality as the four requests required for . The ID of the file could be stored locally in memory by to be able to serve file ID's quickly, but this data structure does not have to be uploaded to Google Drive. Furthermore, when downloading a file, parts of the file can be downloaded rather than the full file [102]. This can reduce the time as the full file does not have to be downloaded every time a file is read. Even if we could download parts of a file from Flickr, it would not make sense for as we need the full file content to decrypt it. Futhermore, with Google Drive's 800 million daily users [103] versus Flickr's 60 million monthly users [104], Google Drive is a much bigger service. This could mean that it has better infrastructure which can process uploaded data faster than Flickr can.

Certain data points in the graphs presented in Section 5.2 are outliers from groups of data points. For instance, looking at the Random Write test for in Figure 5.2 for `file size = 65 536 kB, buffer size = 16 384 kB`, the test data point has significantly lower performance than the the other data points for the other buffer sizes in the same test with the same file size. There are many possible reasons behind this drop in performance. One reason could be a slow internet connection to Google Drive in the point of time when the specific data point was benchmarked, for instance due to a higher load of other user requests to the service. Due to the being an external service that other users use at the same time, it is always possible that the of the cloud-based filesystems experiences a high user-demand at any time. Another reason of the data outlier can also be because the operating system scheduler scheduled the process unfavorable at that time. This is an especially possible reason to the

outlier data points for the non-cloud-based filesystems and as they do not rely on an internet connection. For instance, Figure 5.2 shows two outliers for for `file size` = 131 072 kB, namely `buffer size` = 4 096 kB and `buffer size` = 8 192 kB. They could also have lower performance due to disk scheduling and cache management. The files in the cache could be invalidated and removed if other processes are reading other files from the disk at the same time, resulting in a cache miss of the benchmark file.

Other outlier data points have much higher performance than the other data points in a test. For instance, looking at the Write test for in Figure 5.2, there are data points for `file size` = 8 192 kB, `file size` = 16 384 kB, and `file size` = 32 768 kB that have much higher performance than the other data points. While most data points are approximately between 6500 kB/s to 8300 kB/s, two of these file sizes have one outlier, and one has two outliers, of approximately 100 000 kB/s. Outliers can also be seen in Figure 5.2 and Figure 5.2 for the Re-Write and Random Write tests on . is not a cloud-based filesystem and is therefore not affected by a fluctuating bandwidth of an internet connection. Rather, this is possibly the result of favorable process scheduling and disk operation scheduling. As both file sizes exceed the cache limit of , the cache of the filesystem should not affect the value. However, it is a possibility that IOZone did not close the file or that the `close` operation was not performed correctly for the preceding test, which would keep the open file in memory regardless of size. This would result in faster read operations as the file would not have to be read from the disk. Furthermore, if the `close` operation was not called for a test, the filesystem would not write the data to the disk resulting in shorter execution time, leading to a higher performance for the write tests. Therefore, if there was a missed `close` operation, the following test and preceding test should both have higher performance than the other data points. However, looking at, for instance, the preceding Read test for `file size` = 32 768 kB, buffer size = 2 048 kB in Figure 5.2 and the subsequent Re-Read tests for `file size` = 32 768 kB, `buffer size` = 2 048 kB in Figure 5.2, those data points are not outliers which indicates that it the outlier in the Write test probably is not due to a missed `close` call. Rather, it is possible that the higher performing read tests outliers are due to the kernel cache providing the data. However, this is not a possible reason to the high-performing write tests as the cache cannot increase the performance of a write call.

Comparing benchmarking results against the benchmarking results, we can compare the theoretical best performance of against a general-purpose widely-used filesystem. Furthermore, we can compare against the underlying filesystem in which it is storing its data. In Figure 5.2 and Figure 5.2 we can see that the read operation perform similarly for and , where is in general faster than . However, for certain data points, such as `file size = 131 072 kB, buffer size = 256 kB`, has higher performance than with 7 525 973 kB/s for and 5 927 107 kB/s for . As the file size is bigger than 5 MB it was not stored in the cache of . The reason why outperformed at this data point is could be due to process scheduling or because the buffer size used by the test is less efficient for than the one called on by . The buffer size used by to read the file from is not certain to be the same as was called with. The buffer size used by on is set by the implementation of `basic_filebuf` [105]. It is also possible that the data was provided by the kernel cache to , which could be faster than reading from . Another possibility is that read the data from , but it was provided by a cache of rather than being read from the storage device, while the same test for read the data from the storage device.

The cache of a filesystems can generally influence the performance of the read operations. However, there is no significant drop in performance for when reading a file that fits in the cache, and one that does not. In fact, the performance of the Read test for `file size = 8 192` is in general better than for `file size = 4 096` and `file size = 2 048` on , even though the 8192 kB file cannot fit in the cache while the other file sizes can, as long as the encrypted data and the PNG attributes do not make the file bigger than 5 MB. The reason why these files can be provided fast is most probably due to the kernel caching of the files. However, the performance of the `file size = 262 144 kB` compared to many of the other file sizes is much lower, indicating that these files were not, at least entirely, cached by the kernel, possibly due to size constraint. As the 131 072 kB file size has similar performance to the 262 144 kB file size for some of the Read tests buffer sizes, but significantly higher performance for other buffer sizes, it is possible that the kernel cache size limit is around 131 072 kB. However, as it seems to fluctuate, the size limit might be dependent on factors such as the available memory of the system at the time. When the memory usage is high, for instance by other processes running at the same time, it is possible that the data

stored in the kernel cache for is removed, which decreases the performance for certain tests.

The only implementation difference between and is that stores the produced images on Flickr while stores the produced images on the local filesystem. Therefore, the time difference of an operation compared to an operation should only depend on the internet connection to Flickr and how fast Flickr can process the requests. For instance, looking at the Write tests for the two filesystems, outperforms significantly. Looking at one file size, for instance, `file size = 8 192` has an average performance of approximately 15 800 kB/s and has an average performance of approximately 1050 kB/s for the Write test. This means that the test took on average 518 ms for and 7802 ms for . The same test for the same file size for had an average performance of 612 000 kB/s, meaning it took on average 13 ms for to save the 8192 kB file. The time the test takes for is the same as the write operation overhead plus the time takes to save the file. Subtracting the write time from the test time of , we get the average overhead of the Write test for both and as they have the same overhead. The average overhead time of the Write test for and is therefore 505 ms, meaning that the requests and the request's overhead by took on average $7\,802 - 505 = 7297$ ms which is approximately 94% of the computation time for this test. Assuming the upload bandwidth to Flickr is the same as the measured reference bandwidth of 92.95 MB/s, uploading 8192 kB would take 705 ms. This means that the remaining 6592 ms were used for request overhead, such as preparing for the request, waiting for Flickr to process the data, and receiving the response from Flickr, including the post ID. Preparing the request includes first saving the image to the local filesystem, and then reading it when uploading it. Assuming the file was saved with the same performance as above, it would take 13 ms. The remaining 6579 ms are used for creating the request, receiving the request response, and for Flickr to process the data. The most significant time consumption of these three tasks is most probably the data processing of Flickr. This indicates that with faster data processing by Flickr, could potentially be faster. Furthermore, it indicates that the bandwidth of the internet connection to the is not the most important factor of the performance of . Even if uploading the file over the internet to the would be instant, it would reduce the file operation time by less than 10%. To improve the filesystem opera-

tion performance, using a which can process the data faster is of more importance. For instance, if the would process the data in the background and instantly return, the write operation performance could be reduced. This could mean, however, that the image is not seen on the instantly which could mean that it is not possible to download the file instantly after it has been uploaded. The calculations above assume that the bandwidth to Flickr was approximately the same as the bandwidth to the measurement servers. It is possible that the bandwidth to Flickr was much lower, which would mean that the bandwidth has more impact of the filesystem operation time. Future work includes using multiple s to compare their performances, as well as measuring the individual filesystem's bandwidth for more precise calculations.

While the values of the read operation for and are comparable to each other, this is not the case for all tests. For instance, the write operation of is much slower than the write operation of . The write operation performance average of is approximately 1.5% of the average performance of the write operation on . While it is understandable that the write performance is lower than the write performance as all data written to must also be written to , the difference in performance is significant. The reason for this could be the fact that has to encrypt the data stored, including creating all the cryptographic variables such as the salt and the . While is also an encrypted filesystem, it is possible that the filesystem prepares the cryptographic variables before they are needed. For instance, the next cryptographic key and its salt could be derived while the filesystem is idle as it does not depend on the content of the stored data. It is also possible that is using multiple threads to write the data which could speed up the operation. For instance, it is possible that requires multiple `write` calls to the underlying filesystem. If each `write` call to is computed on a different thread, the multiple `write` calls could possibly be completed faster rather than if they were completed sequentially.

and are comparable in some tests, which is interesting as is dependent on an internet connection and an while is not. The median performance of the Re-Read test on is slightly worse than the medium performance of the Re-Read test on . Meanwhile, the median Read performance of is significantly less than the median Read performance of . This indicates that a lot of the data served by and was served by the kernel

cache of the filesystems. Furthermore, it indicates that is faster than for files not in the kernel cache. However, as the Read tests performance is higher than a reasonable internet bandwidth, it is probable that the data is still served from an internal cache rather than the data being fetched from Google Drive. As the Read tests performances are higher for than for , it indicates that has a faster internal cache than , or that the kernel cache was used for the Read tests while it was not used for .

The Write, Re-Write, and Random Write tests on outperform the same tests on . This is reasonable as the data written to must be uploaded to Google Drive, while the data written to is stored on the local disk. Uploading 16 MB of data with the average (reference point) upload speed of 91.83 Mbit/s would take approximately 1.4 s. Meanwhile, we can see in Figure 5.2 that can write 16 MB of data as fast as 6 921 222 kB/s = 6921.2 MB/s, meaning it would take approximately 2 ms to write the data. Meanwhile, the maximum Write performance of is 101 677 kB/s, meaning that can write 16 MB in approximately 157 ms. With this data, we can see that can write the 16 MB of data approximately 7 800% slower than can, and that can write the 16 MB of data approximately 800% slower than or approximately 70 000% slower than .

It is easy to see, and it is not unexpected, that outperforms in performance. As a professional local filesystem, will always have better performance than FFS. Further, like , the performance of depends on the performance of as the file which is uploaded to Flickr first needs to be saved on disk. This dependency could be removed, for instance by providing the temporary file to the FlickCURL library via a filesystem. Further, the median performance of the Re-Read test on is approximately 72% of the performance of the Re-Read test on . This is probably high for many of the tests because of the kernel cache of , but as it differs significantly from , calls are also performed. With higher bandwidth and with another with faster data processing, it is possible that could increase its read performance. In contrast, the median performance of the Re-Read test on is approximately 76% of the median performance of the same test on .

Generally, the figures of the benchmark data presented in Section 5.2 have visibly similar patterns for the Write tests and similar patterns the Read tests, per filesystem. For instance, the patterns of the Read, Re-Read, and Random Read follow a similar pattern with a similar curve of the data points, while the Write, Re-Write, and Random Write follow another distinct pattern. The Read, Re-Read, and Random Read figures of follow another distinct pattern, as well as the three Write test data follow a fourth distinct pattern. The three write test patterns of are dissimilar to the patterns of the write tests even though both filesystems are implemented very similarly, other than the storage medium. However, some patters are similar even though they are from different filesystem, for instance, the Read tests of and . In Figure 5.2 and Figure 5.2, the Read test shows that certain file size data points are found on the lower spectrum of the plot, while the other file sizes follow a somewhat similar curve for both filesystems. However, the pattern of the Re-Read test data of and differ significantly. It might be possible to use these distinct patterns to create a fingerprint of a filesystem. This could be used, for instance, to identify a filesystem based on its performance when the filesystem is unknown. This could be useful when the filesystem is masked by an overlaying filesystem such as Cryptfs. However, by benchmarking the overlying filesystem, the pattern of the underlying filesystem might be lost. Analyzing if it is possible to identify filesystems based on benchmark fingerprints and identifying underlying filesystems using this technique is part of future work.

6.2 Security and Deniability

The data stored in images is encrypted with state-of-the-art encryption standards. Using -, not only provides confidentiality of the data, but it also provides the authenticity of the data. The cryptographic algorithms are implemented using good cryptographic standards, such as cryptographic secure number generators [106]. However, the security of is dependent on, among other things, the password the user chooses. A bad password, for instance, short or commonly used, is easily breakable for an adversary. An adversary who has access to an encrypted image could brute-force the bad password used to derive the encryption key much faster than they could

brute-force the encryption key. does not put any constraints on the password used - as long as it is at least one byte it is acceptable for . This puts the responsibility on the user for the choice of password.

puts a lot of trust in the open-source library Crypto++ [85]. Crypto++ provides cryptographic functions that uses for, among other things, deriving the encryption key, encrypting the data, and verifying the authentication tag. While there are no reported CVE security vulnerabilities as of the time of this writing [88], there may be vulnerabilities that have not yet been discovered or that have been found but not published in the CVE database. There is also a possibility that has vulnerabilities, such as side channels, which could be exploited. was developed by a single author without a review from anyone.

Anyone with access to Flickr.com can view and download the original images stored by , both registered users on Flickr, and anonymous visitors, even if the user has restricted access to their originally uploaded images [32]. An example of how the profile might look is shown in Figure 6.1. The images found on the account present little information about the filesystem. For users unaware of who view the Flickr profile, they see different sizes of images with seemingly randomly generated pixel colors. However, for adversaries who know about the details of , more information can be retrieved. For instance, they could assume that the most recently uploaded image to Flickr is the inode table. However, as we assume the adversary does not have access to the decryption key, they cannot decode the plain-text data of the image and thus cannot verify that this is indeed the inode table. The exact number of files and directories in cannot be known precisely without access to the content of the inode table. Even if the Flickr account has, for instance, 15 images stored, and we know that one represents the inode table and one represents the root directory, it is not possible to conclude if other images stores file data or directory data. The remaining 13 images in the example could represent:

- one big single file split over 13 images, or
- one big single directory split over 13 images, or
- 13 different files, or
- 13 different directories, or
- one directory and 12 different files, or
- 13 copies of the same file, et cetera.

It is also not possible to know if an image stored on Flickr has been uploaded by or by the user manually to further inflate the amount of data stored on the service. For instance, by encrypting random data using 's encoder and uploading the images to Flickr, but without saving the posts in the inode table or in a directory of , the images will look indistinguishable from the other images on Flickr. Only with access to the decrypted inode table can one know if the image is relevant to or not. However, it is possible that Flickr could have logs about the uploaded images, and be able to distinguish images uploaded from the from the user interface. Future research could extend to post encrypted random data at random time intervals to automatically diffuse the knowledge about the images on the . This would mean that even Flickr would be unable to distinguish the uploaded data as it would all be uploaded from the same service. would be required to upload the diffusing data using the same pattern as is used when a file or directory is updated or created so that the upload pattern for the diffusing data is undistinguishable from the regular data for users without the decryption keys. For instance, a write operation on a file will remove the old file and the old inode table, and upload one new image as the new file data and one new image as the new inode table. Similarly, for a new file created in the filesystem, the new file content is uploaded as an image, one image is uploaded as the parent directory's new data, one image is uploaded as the new inode table, and the old directory image and inode table image are removed from Flickr. To simulate file write and file create filesystem operations, two or three images must be uploaded and two images must be deleted. One problem with this could be that the last image uploaded should always be the inode table. While the data of the inode table can easily change by changing encryption key, the size of the inode table could remain the same which could be suspicious for adversaries with access to snapshots of the Flickr account's images. This could be solved by always adding a random amount of

filler bytes when encoding the image. The filler bytes are not read by the decoder anyway. However, adding filler bytes can increase the resulting file size of the images, decreasing the overall storage capacity of . Furthermore, storing diffusing images on Flickr that are not stored in the filesystem decreases the storage capacity of .

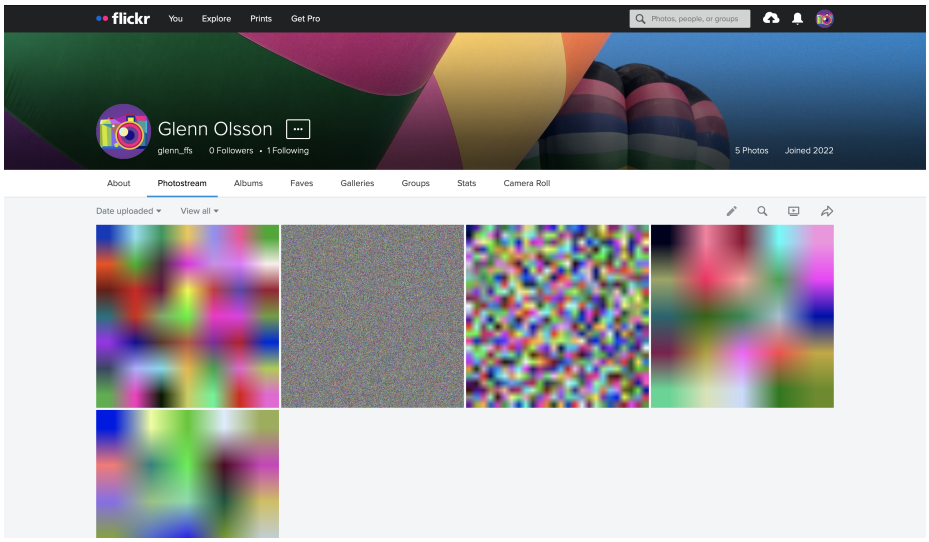

Figure 6.1: Screenshot of the Flickr profile used for . At the moment of the screenshot, the filesystem is storing a previous version of this thesis in a directory inside the root directory. The images seen are the inode table, the thesis data, the root directory data, the subdirectory (containing the thesis) data, and a temporary file containing extra attributes of the thesis document created by macOS while was mounted (this file is sometimes referred to as a *turd* [107]).

The size of data stored in an image is not completely hidden. While the exact number of bytes of unencrypted data that the image stored is not possible to know without the decryption key, it is possible to get an estimate. If you know the binary structure of the image (as presented in Appendix B), you can find out how many bytes the encrypted cipher is, the value of the data, the value of the salt used for the encryption key derivation, and the value of the authentication tag. By knowing the length of the cipher, the length of the unencrypted data can be placed in a range. The length of the cipher L_c in bytes is divisible by 16 (as is a 16-byte block cipher), and the length of the plain text must be less than L_c due to the requirement of at least one bit of padding [94]. The smallest possible size for the length of the plain text is $L_c - 16$.

Therefore, the length of the plain text L_p is:

$$L_c - 16 \leq L_p < L_c$$

By examining all the images stored on Flickr and the possible values of L_p, it is possible to know the upper limit and lower limit of all the data stored on Flickr at a certain time, assuming that all images on Flickr are stored in . However, it is **not** possible to know if all this data is stored on through entries in the inode table. It is also **not** possible to know if the plain text represents a file or directory without the decrypted data of the inode table. One image can be assumed to be the inode table of which a size-esitamte is also possible.

If a user supplies a different password when mounting than used previously, the images stored on Flickr cannot be decrypted. When tries to read the image it believes represents the inode table (the most recently uploaded image) and it fails, it will simply create a new inode table representing an empty filesystem, and upload the image representing this inode table, essentially replacing the potentially previous inode table (if it existed). As it is not possible to know if the images already uploaded to Flickr represent an inode table without the correct decryption key, it is impossible to determine if the image that could represent the inode table is indeed an inode table encrypted with another password, or if it is some arbitrary data. In a potential rubber-hose situation*, the user of the filesystem could easily claim that they uploaded images with arbitrary data, using randomly generated keys that they do not remember and that the filesystem is empty. There is no way to prove the existence of any meaningful data on Flickr without the decryption key. As the encoder also uses random salting for the encryption key, it is not even possible to prove that the images are encrypted with the same password as the encryption keys will differ for all images, even when the same password is used.

However, as mentioned earlier, we assume that an adversary has access to the structure of images as well. To counter this, the user who wants to hide their data could, after creating a filesystem containing meaningful information, mount again

* When an adversary might torture the user, with for instance a rubber hose. See Section 3.1

with another password. would then create a new inode table and upload this table, creating a dummy . In a rubber-hose situation, the user could give up the password to the dummy instance, which is empty. The adversary can verify that this password indeed decrypts the most recently uploaded image and that the unencrypted image data represents an empty inode table. If the user proceeds to claim that they do not know the passwords of the other images, the adversary cannot prove that they contain meaningful data nor that they have been uploaded by the user. These images could, for instance, have been uploaded by another user of . Further, with no password constraints by , a user could also create a dummy with a password that is easily breakable, to make the adversary believe they found the correct password if they perform a brute-force attack. As long as the user remembers which post represents the inode table, the images uploaded after this inode table could simply be removed from Flickr before mounting with the correct password when the user wants to access their actual instance. Alternatively, the user could save the image representing the inode table in another storage medium and upload it again when they want to access their actual instance.

One aspect where is better than is its security against the potential adversary of the store owners. stores the data in its original format on Google Drive, essentially providing an overlay filesystem for Google Drive. While this can be desired in certain situations, such as using on one machine and the Google Drive website on another, it gives Google Drive access to your data. As mentioned, Google Drive encrypts your data from outside agents, but as they control the encryption and decryption keys themselves, the data stored can be accessed by the company. The data could also be given to authorities who are requesting it with a subpoena. However, to use Google Drive while maintaining your own encryption methods, an overlay filesystem could be mounted on top of using a stackable filesystem, such as Cryptfs [74]. Future work could include researching the performance of such a filesystem and compare it to .

 on the other hand gives the user control of all its data. While Flickr can give out the images uploaded by , this data can be accessed by anyone with access to Flickr.com anyway. The only way to access unencrypted data is by using the password that the user controls. This provides with one aspect of better security than , but this

might also be a factor why is slower than . By requiring the data to be encrypted when it is written and uploaded, and decrypted when it is downloaded and read, will need to compute new cryptographic variables every time a file or directory is written which requires a lot of computations. Further, every time an image is read it must be decrypted, even if it is in the cache of . Decrypting an image requires a lot of computations as well as, other than decrypting the data, the decryption key must first be derived from the password. Meanwhile, it is possible that Google Drive is caching the unencrypted files, or performing the cryptographic computations on high-performance computers requiring less computation time. So while gaining a security aspect of the filesystem, sacrifices the performance of the filesystem operations.

Every in Sweden are required to keep data generated or managed during internet access for ten months [108]. has declared that retained data should only be accessed when fighting serious crimes or if national security is at stake [109]. In the United States, there are no data retention laws requiring s to store identifiable information, however, no major American has enacted zero data retention policies [110]. Furthermore, American s can be forced to give out what information they have about a customer by US law, including the websites the customer have been visiting [111]. Using or , your can identify your internet traffic and know that you are accessing Flickr or Google Drive. Furthermore, by combining the data with the IP logs of Flickr or Google Drive, it could also be possible to identify you as the uploader of data to or . However, using technologies such as Tor or a could hide your identity from your . According to Swedish law, providers are not required to retain logs of its users [112]. However, they are not prohibited from storing such data. Furthermore, the of the provider must still log the network traffic originating from the service. Even though they cannot identify specific users without logs from the provider, they can determine the what websites are accessed from services. If the provides logs about you to adversaries, you could still be identified. Tor provides the user with multiple routing layers to hide the origin of the internet traffic [113]. The internet connection is encrypted and the source IP address is changed for each hop, and the final node has no information about the original source address of the request. Furthermore, the nodes used by the client are changed periodically. However, Tor is slow due to the layered connections and would therefore decrease the already low performance of further.

writes temporary files to the local filesystem when uploading images to Flickr due to limitations in the Flickr library used. Even though the images are stored for a short time in the filesystem and remove shortly after being created, there are possibilities to recover removed data from [91, 92, 93]. Furthermore, if would crash for some reason while uploading a file, the temporary file could remain readable in the filesystem. This could be solved by storing the temporary file in a temporary filesystem. For instance, a secure in-memory filesystem could be mounted on the computer and be used as the temporary file storage. This is part of future work for .

Flickr states in their community guidelines [114] that they do not allow non-photographic elements in their service. uploads images with seemingly random noise, not photographs. It is therefore probable that Flickr would remove the images uploaded by if they were discovered or reported by the community. Another adversary of is thus detection from Flickr. During testing of the filesystem, while big files have been uploaded, there have not been more than 50 images stored on Flickr by at a time which is far from their 1 000 image limit. No images have been removed from Flicker yet and it is possible that the Flickr account has not raised suspicion from Flickr due to the low number of images. If images were removed from Flickr, it could have very negative effects on the usability of the filesystem. For instance, if the image representing the inode table was removed, the post id of each filesystem would be lost unless was running at the same time as the inode table is cached in memory. While the images could still be decrypted by , the filename and path of each file and directory would be lost. If an image representing a directory was removed (but the inode table was not removed), the files and directories in the directory would not have a filename associated with them. They would also not be findable in the filesystem as does not support links, so there is only one filepath per file or directory in the filesystem. If an image representing a file was removed, the file data would be lost. If an image representing one part of a file was removed, only the data stored in that image would be lost. The remaining images could still be decoded and decrypted on their own, but the lost data cannot be recovered. It is important that the images are not removed from the for the data stored in to be safe. However, this guarantee is hard to achieve as the Flickr terms of service states that Flickr retains the right to remove user content from the service at any time [33]. If Flickr would have knowledge

about it is possible that they would implement detection techniques of the images, and remove them. Combating detection by the is part of future work.

6.3 Impact

This section presents the impact could have. Section 6.3.1 presents societal impacts that could introduce. Section 6.3.2 presents the environmental impacts of .

6.3.1 Societal impacts

Secure and hidden data is not only for the better good. As the data stored on cannot be decrypted by bad guys or good guys, illegal data could be stored on the system without anyone knowing about it. It is known that end-to-end encryption does not have only a positive impact on society, for instance, terrorist organizations are known to be using end-to-end encryption to spread their messages across the internet [115]. could potentially provide secure storage for illegal groups such as terrorist organizations and child pornography rings. It is not possible to limit who uses , by other means than not publishing the source code of the filesystem. However, this does not prevent criminal organizations using other end-to-end encrypted filesystems or develop their own. Some terrorist organizations consist of well-educated engineers who could develop similar technologies for their organization [116].

An ethical consideration of is that it is breaking the terms of services of Flickr by exploiting its free storage. Flickr and many other s provide users with a lot of free storage. By exploiting this storage, the costs for the s could increase requiring them to charge users for the storage or decrease the free storage quota. This would hurt honest users of the service who are following the guidelines. Content creators, such as photographers on Flickr, could have to pay money to use the previously free service which could deter the usage of the platform. If the would implement detection techniques to combat the images, these could falsely mark legit images for removal which again could affect honest users of the service. Less users of the could eventually require the company to reduce their staff due to loss of revenue.

provides free storage for all users. This benefits people who might not have money to spend on commercial cloud-based storage or physical hard disks. However, as requires the user to run macOS which natively only runs on Apple's computers which are often considered expensive, this might not benefit the poor. Furthermore, the cost of a hard disk or commercial cloud-based storage is often not expensive.

6.3.2 Environmental impact

uses Flickr's data centers to store its data. Globally, data centers have a large environmental impact. It has been estimated that they use over 2% of the world's electricity [117] and emit roughly the same amount of carbon dioxide as the global airline industry emits burning aircraft fuel [118]. However, the emissions of the data centers depend on their location. For instance, some data centers in Sweden are powered with 100% carbon-free energy [119, 120]. What data centers Flickr are using and where they are located has not been found.

As mentioned previously, encrypting and encoding the stored data as images requires more storage than the actual data stored. This means that more storage is required to store all the data in , as opposed to storing the same data in a local filesystem. It also means that the network request will carry more data than necessary, requiring more energy. This fact is also true when comparing to a cloud-based filesystem, such as Google Drive. While both Google Drive and store their data in data centers, the data that Google Drive stores can be less than the same file stored in due to only storing the encrypted file data rather than an encoded image. While the extra PNG data is expected to be less than 10% of the total image size, big files will require a lot of storage for just the PNG attributes. Furthermore, as Google Drive is a filesystem, operations and data structures could be optimized for a filesystem while is a layered filesystem on top of Flickr. As mentioned in Section 6.1, Google Drive could potentially implement a more efficient filesystem with its REST than Flickr does with its . is also developed by one developer during a limited time while Google Drive is maintained by multiple teams and was released over ten years ago. Google Drive also has requirements of efficient power- and storage usage as it is a massive

company where small improvements can save a lot of money. on the other hand has no such requirements and is developed with useability as its main focus.

Other than storing data in the cloud, data is often stored on physical devices such as memory sticks or hard disks. provides on-demand storage when the user needs it, and can be forgotten when not in use. It does not require the user to purchase hardware whenever they need more storage, possibly just temporarily. Such hardware could produce litter if the user disposes of it after use. While the storage devices can be cheap, it can promote single-use of the devices which in turn could increase the littering. However, with the low-performance of compared to a local filesystem, can often not be used as a substitute of a physical storage device. While a portable storage device was not included in the analysis of the thesis, will probably perform worse than most portable storage devices. Many modern storage devices are based solid-state disks which in general are very fast. Even hard-drive based storage devices are probably faster than a cloud-based filesystem in many cases.

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 Conclusions

is a cryptographic and deniable cloud-based filesystem with free storage through exploiting online web services. Compared to other filesystems, is slow and is not suitable as a multi-purpose filesystem, for instance as a hard drive for a computer. It performed poorly even compared to another cloud-based filesystem, . However, one key difference between these two filesystems is that manages the cryptography of the filesystem, while delegates this task to Google Drive. This provides security benefits for , but might also contribute to the slower computation time. The results also show that even when removing the dependency of an internet connection is performing poorly, especially for the read operations compared to and . The write operations of perform better than and . The read operations of and are more similar than the write operations, however, outperforms for every read operation test leading to the conclusion that the internet connection and the influence the file operations significantly. With better read performance than write performance, is best suited as a many-read-few-write filesystem.

While the filesystem is slow, it provides security aspects such as end-to-end encryption and deniability. As long as the filesystem is not mounted to the computer, it is not possible to prove how much data is stored on , or even prove that data is stored on . However, it is possible to get a upper-limit amount of data stored. End-to-end cryptography provides the user with confidential data. Further, by using authenticated encryption, provides the user with proof of the authenticity of the data it stores.

7.2 Future work

As mentioned previously, does not implement all features that the POSIX standard defines. Future development for could be to implement more of these functions, such as links and file permissions. This could make resemble a regular filesystem further. Another improvement could be to move from userspace using , 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 anonymity of the user is not considered but for 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 and increase the storage capacity of , 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, could achieve even more storage. Future work includes comparing such a filesystem with the current state of .

To increase the storage capacity further, could take advantage of storing videos on the as well. Flickr allows videos up to 1 GB on its system. Future work could include researching how much steganographic data can be stored in videos and how efficient a filesystem using encoded videos could be.

If the would pursue identifying images stored on their service to remove them, it would be a problem for . Even removing a single image could remove the full functionality of the filesystem. Future work includes finding evasion techniques to hide the hidden data even further. For instance, hiding less data in the images is a possibility which could mean that the images could look like actual images, such as photographs. This is similar to the idea of CovertFS [6] where a maximum of 4 kB per

image would be used. However, this would significantly decrease the storage capacity of . Part of future work is to explore if more data could be stored in the images, or if multiple s could be used to overcome the decreased storage capacity.

The benchmark tests showed visible patterns, seemingly distinct between filesystems. Future work includes analyzing these patterns to create fingerprints which could be used to identify filesystems based on their benchmark data.

Bibliography

[1] Jin Han et al. "A Multi-User Steganographic File System on Untrusted Shared Storage". In: *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC '10*. The 26th Annual Computer Security Applications Conference. Austin, Texas: ACM Press, Dec. 6, 2010, p. 317. ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920309. URL: http://portal.acm.org/citation.cfm?doid=1920261.1920309 (visited on 01/27/2022).

[2] Rick Westhead. "How a Syrian Refugee Risked His Life to Bear Witness to Atrocities". In: *The Toronto Star. World* (Mar. 14, 2012). ISSN: 0319-0781. URL: https://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_witness_to_atrocities.html (visited on 04/13/2022).

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

[4] Twitter. *Twitter Terms of Service*. Aug. 19, 2021. URL: https://twitter.com/en/tos (visited on 05/09/2022).

[5] Dave Johnson. *Is Google Drive Secure? How Google Uses Encryption to Protect Your Files and Documents, and the Risks That Remain*. Business Insider. Feb. 25, 2021. URL: https://www.businessinsider.com/is-google-drive-secure (visited on 04/13/2022).

[6] 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.

[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] *Fscrypt - ArchWiki*. URL: https://wiki.archlinux.org/title/Fscrypt (visited on 04/25/2022).

[9] iGotOffer. *APFS (Apple File System) Key Features | iGotOffer*. About Apple | iGotOffer. July 16, 2017. URL: https://igotoffer.com/apple/apfs-apple-file-system-key-features (visited on 04/11/2022).

[10] Tom Nelson. *What Is APFS and Does My Mac Support the New File System?* Lifewire. URL: https://www.lifewire.com/apple-apfs-file-system-4117093 (visited on 04/11/2022).

[11] Apple Inc. *File System Formats Available in Disk Utility on Mac*. Apple Support. URL: https://support.apple.com/guide/disk-utility/file-system-formats-dsku19ed921c/mac (visited on 04/25/2022).

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

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

[14] Google. *Google Drive Terms of Service - Google Drive Help*. URL: https://support.google.com/drive/answer/2450387?hl=en (visited on 04/25/2022).

[15] Google. *Google Terms of Service – Privacy & Terms – Google*. URL: https://policies.google.com/terms?hl=en#toc-content (visited on 04/25/2022).

[16] Apple Inc. *iCloud Security Overview*. Apple Support. URL: https://support.apple.com/en-us/HT202303 (visited on 04/25/2022).

[17] *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).

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

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

[20] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To {FUSE} or Not to {FUSE}: Performance of {User-Space} File Systems". In: 15th USENIX Conference on File and Storage Technologies (FAST 17). Feb. 27–Mar. 2, 2017, pp. 59–72. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor (visited on 04/06/2022).

[21] Richard Gooch. *Overview of the Linux Virtual File System — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/filesystems/vfs.html (visited on 04/12/2022).

[22] Amit Singh. *Mac OS X Internals: A Systems Approach*. Pearson, 2006. ISBN: 0-321-27854-2. URL: https://flylib.com/books/en/3.126.1.136/1/ (visited on 04/11/2022).

[23] Raghavendra Gowdappa, Csaba Henk, and Manoj Pillai. "Experiences with FUSE in the Real World". Feb. 2019.

[24] Twitter. *Twitter IDs*. URL: https://developer.twitter.com/en/docs/twitter-ids (visited on 07/15/2022).

[25] *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).

[26] *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).

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

[28] *Flickr Upload Requirements*. Flickr. July 26, 2022. URL: https://www.flickrhelp.com/hc/en-us/articles/4404079649300-Flickr-upload-requirements (visited on 07/31/2022).

[29] Flickr, Inc. *Upgrade Everything You Do with Flickr*. Flickr. URL: https://www.flickr.com/account/upgrade/pro (visited on 07/31/2022).

[30] *Flickr: The Help Forum: Captions/Text In Flickr*. Flickr Help Forum. Jan. 2, 2009. URL: https://www.flickr.com/help/forum/en-us/88316/ (visited on 07/31/2022).

[31] Flickr, Inc. *Download Permissions*. Flickr. URL: https://www.flickrhelp.com/hc/en-us/articles/4404079715220-Download-permissions (visited on 07/31/2022).

[32] *Flickr: The Help Forum: Download Blocking?* Mar. 25, 2020. URL: https://www.flickr.com/help/forum/en-us/72157713620256173/ (visited on 11/20/2022).

[33] Flickr, Inc. *Flickr Terms & Conditions of Use*. Flickr. Apr. 30, 2020. URL: https://www.flickr.com/help/terms (visited on 07/31/2022).

[34] Flickr, Inc. *Flickr: The Flickr Developer Guide - API*. URL: https://www.flickr.com/services/developer/api/ (visited on 07/31/2022).

[35] *What Are the API Limits, Actually? | Flickr API | Flickr*. Flickr Help Forum. Oct. 2, 2013. URL: https://www.flickr.com/groups/51035612836@N01/discuss/72157636113830065/72157636114473386 (visited on 07/31/2022).

[36] Harsh Kumar Verma and Ravindra Singh. "Performance Analysis of RC6, Twofish and Rijndael Block Cipher Algorithms". In: *International Journal of Computer Applications* 42 (Mar. 1, 2012), pp. 1–7. DOI: 10.5120/5773-6002.

[37] Xavier Bonnetain, María Naya-Plasencia, and André Schrottenloher. "Quantum Security Analysis of AES". In: *IACR Transactions on Symmetric Cryptology* 2019.2 (Dec. 6, 2019), p. 55. DOI: 10.13154/tosc.v2019.i2.55-93. URL: https://hal.inria.fr/hal-02397049 (visited on 08/24/2022).

[38] John Ross Wallrabenstein. *When It Comes to Data Integrity, Can We Just Encrypt the Data? - EngineerZone Spotlight - EZ Blogs - EngineerZone*. ADI EngineerZone. Nov. 17, 2021. URL: https://ez.analog.com/ez-blogs/b/engineerzone-spotlight/posts/data-integrity-encrypt-data (visited on 08/24/2022).

[39] Dmitry Khovratovich. *Answer to "Why Should I Use Authenticated Encryption Instead of Just Encryption?"* Cryptography Stack Exchange. Dec. 7, 2013. URL: https://crypto.stackexchange.com/a/12192 (visited on 08/24/2022).

[40] David McGrew and John Viega. "The Galois/Counter Mode of Operation (GCM)". In: *submission to NIST Modes of Operation Process* 20 (2004), pp. 0278–0070.

[41] Gaurav Kodwani, Shashank Arora, and Pradeep K. Atrey. "On Security of Key Derivation Functions in Password-based Cryptography". In: *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2021 IEEE

International Conference on Cyber Security and Resilience (CSR). July 2021, pp. 109–114. DOI: 10.1109/CSR51186.2021.9527961.

[42] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. 264. 2010. URL: https://eprint.iacr.org/2010/264 (visited on 08/24/2022).

[43] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. Request for Comments RFC 5869. Internet Engineering Task Force, May 2010. 14 pp. DOI: 10.17487/RFC5869. URL: https://datatracker.ietf.org/doc/rfc5869 (visited on 08/24/2022).

[44] Tony Hansen and Donald E. Eastlake 3rd. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. Request for Comments RFC 6234. Internet Engineering Task Force, May 2011. 127 pp. DOI: 10.17487/RFC6234. URL: https://datatracker.ietf.org/doc/rfc6234 (visited on 08/24/2022).

[45] Dan Arias. *Adding Salt to Hashing: A Better Way to Store Passwords*. Auth0 - Blog. Feb. 25, 2021. URL: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/ (visited on 08/27/2022).

[46] 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).

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

[48] 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).

[49] *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).

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

[51] Jianxia Ning et al. "Secret Message Sharing Using Online Social Media". In: *2014 IEEE Conference on Communications and Network Security*. 2014 IEEE Conference on Communications and Network Security. Oct. 2014, pp. 319–327. DOI: 10.1109/CNS.2014.6997500.

[52] Filipe Beato, Emiliano De Cristofaro, and Kasper B. Rasmussen. "Undetectable Communication: The Online Social Networks Case". In: *2014 Twelfth Annual International Conference on Privacy, Security and Trust*. 2014 Twelfth Annual International Conference on Privacy, Security and Trust. July 2014, pp. 19–26. DOI: 10.1109/PST.2014.6890919.

[53] Ross Anderson, Roger Needham, and Adi Shamir. "The Steganographic File System". In: *Information Hiding*. Ed. by David Aucsmith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 73–82. ISBN: 978-3-540-49380-8. DOI: 10.1007/3-540-49380-8_6.

[54] 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.

[55] 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).

[56] Andrew D. McDonald and Markus G. Kuhn. "StegFS: A Steganographic File System for Linux". In: *Information Hiding*. Ed. by Andreas Pfitzmann. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 463–477. ISBN: 978-3-540-46514-0. DOI: 10.1007/10719724_32.

[57] Josep Domingo-Ferrer and Maria Bras-Amorós. "A Shared Steganographic File System with Error Correction". In: *Modeling Decisions for Artificial Intelligence*. Ed. by Vicenç Torra and Yasuo Narukawa. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 227–238. ISBN: 978-3-540-88269-5. DOI: 10.1007/978-3-540-88269-5_21.

[58] 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).

[59] Krzysztof Szczypiorski. "StegHash: New Method for Information Hiding in Open Social Networks". In: *International Journal of Electronics and Telecommunications; 2016; vol. 62; No 4* (2016). ISSN: 2300-1933. URL: https://journals.pan.pl/dlibra/publication/116930/edition/101655 (visited on 04/13/2022).

[60] Jędrzej Bieniasz and Krzysztof Szczypiorski. "SocialStegDisc: Application of Steganography in Social Networks to Create a File System". In: *2017 3rd International Conference on Frontiers of Signal Processing (ICFSP)*. 2017 3rd International Conference on Frontiers of Signal Processing (ICFSP). Sept. 2017, pp. 76–80. DOI: 10.1109/ICFSP.2017.8097145.

[61] Robert Winslow. *Tweetfs/Tweetfs at Master · Rw/Tweetfs*. GitHub. URL: https://github.com/rw/tweetfs (visited on 04/06/2022).

[62] 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).

[63] 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).

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

[65] Puşcaş, Sergiu Dan. "GCSF – A VIRTUAL FILE SYSTEM BASED ON GOOGLE DRIVE". BABES, -BOLYAI UNIVERSITY CLUJ-NAPOCA, 2018. URL: https://harababurel.com/thesis.pdf (visited on 08/27/2022).

[66] Sergiu Puşcaș. *Harababurel/Gcsf*. Aug. 24, 2022. URL: https://github.com/harababurel/gcsf (visited on 08/27/2022).

[67] Google. *Install and Set up Google Drive for Desktop - Google Workspace Learning Center*. URL: https://support.google.com/a/users/answer/9965580?hl=en (visited on 08/27/2022).

[68] Alessandro Strada. *Google-Drive-Ocamlfuse*. June 17, 2022. URL: https://github.com/astrada/google-drive-ocamlfuse (visited on 09/08/2022).

[69] Xiao Guoan. *Install Google Drive Ocamlfuse on Ubuntu 16.04, Linux Mint 18*. LinuxBabe. May 21, 2021. URL: https://www.linuxbabe.com/cloud-storage/install-google-drive-ocamlfuse-ubuntu-linux-mint (visited on 09/08/2022).

[70] Joey Sneddon. *Mount Your Google Drive on Linux with Google-Drive-Ocamlfuse*. OMG! Ubuntu! May 10, 2017. URL: http://www.omgubuntu.co.uk/2017/04/mount-google-drive-ocamlfuse-linux (visited on 09/08/2022).

[71] Yawar Amin. *Use Google Drive as a Local Directory on Linux*. DEV Community. Feb. 18, 2021. URL: https://dev.to/yawaramin/use-google-drive-as-a-local-directory-on-linux-1b9 (visited on 09/08/2022).

[72] shubhamharnal. *In Short, GCSF Tends...* r/DataHoarder. July 2, 2018. URL: www.reddit.com/r/DataHoarder/comments/8vlb2v/google_drive_as_a_file_system/e1oh9q9/ (visited on 09/08/2022).

[73] harababurel. *Show HN: Google Drive as a File System | Hacker News*. Hacker News. July 1, 2018. URL: https://news.ycombinator.com/item?id=17430397 (visited on 09/08/2022).

[74] Erez Zadok, Ion Badulescu, and Alex Shender. "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).

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

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

[77] 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.

[78] 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).

[79] Ben Lovejoy. *New 15-Inch MacBook Pro Appears to Offer Fastest SSD Performance on the Market*. 9to5Mac. Nov. 1, 2016. URL: https://9to5mac.com/2016/11/01/2016-macbook-pro-ssd/ (visited on 10/16/2022).

[80] *ImageMagick*. ImageMagick Studio LLC, Aug. 26, 2022. URL: https://github.com/ImageMagick/ImageMagick (visited on 08/27/2022).

[81] Jean-Philippe Barrette-LaPierre. *cURLpp*. Aug. 23, 2022. URL: https://github.com/jpbarrette/curlpp (visited on 08/27/2022).

[82] *Curl/Curl*. curl, Aug. 27, 2022. URL: https://github.com/curl/curl (visited on 08/27/2022).

[83] *Liboauth*. URL: https://sourceforge.net/projects/liboauth/ (visited on 08/27/2022).

[84] Dave Beckett. *Flickcurl: C Library for the Flickr API*. URL: https://librdf.org/flickcurl/ (visited on 08/27/2022).

[85] *Crypto++ Library 8.7 | Free C++ Class Library of Cryptographic Schemes*. URL: https://www.cryptopp.com/ (visited on 08/27/2022).

[86] Geoff Kuenning. *CS135 FUSE Documentation*. 2010. URL: https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html (visited on 07/30/2022).

[87] Benjamin Fleischer. *Mount Options · Osxfuse/Osxfuse Wiki*. GitHub. Nov. 30, 2020. URL: https://github.com/osxfuse/osxfuse (visited on 11/07/2022).

[88] *Cryptopp : Security Vulnerabilities*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-15519/Cryptopp.html (visited on 08/27/2022).

[89] NolanOBrien. *Upcoming Changes to PNG Image Support*. Twitter Developers. Dec. 20, 2018. URL: https://twittercommunity.com/t/upcoming-changes-to-png-image-support/118695 (visited on 09/06/2022).

[90] NolanOBrien. *Feedback for "Upcoming Changes to PNG Image Support"*. Twitter Developers. Jan. 2019. URL: https://twittercommunity.com/t/feedback-for-upcoming-changes-to-png-image-support/118901/84 (visited on 09/06/2022).

[91] LLC SysDev Laboratories. *How to Recover Data from Encrypted Apple APFS*. UFS Explorer. Aug. 19, 2022. URL: https://www.ufsexplorer.com/articles/how-to/recover-data-apfs-encryption/ (visited on 09/11/2022).

[92] Cedric and Gemma. *APFS Data Recovery: How to Recover APFS Files on Mac/Windows*. EaseUS. May 16, 2022. URL: https://www.easeus.com/mac-file-recovery/recover-files-from-apfs-drive.html (visited on 09/11/2022).

[93] Alejandro Santos. *How to Recover Data from an APFS Hard Drive on Mac [a Full Guide]*. Macgasm. Dec. 10, 2021. URL: https://www.macgasm.net/data-recovery/apfs-data-recovery/ (visited on 09/11/2022).

[94] Z. Z. Coder. *Answer to "Size of Data after AES/CBC and AES/ECB Encryption"*. Stack Overflow. July 19, 2010. URL: https://stackoverflow.com/a/3284136/8138631 (visited on 09/11/2022).

[95] Apple Inc. *What Is Secure Virtual Memory on Mac?* Apple Support. URL: https://support.apple.com/en-gb/guide/mac-help/mh11852/mac (visited on 09/11/2022).

[96] IOZone. *Iozone Filesystem Benchmark Documentation*. URL: https://www.iozone.org/docs/IOzone_msword_98.pdf (visited on 09/08/2022).

[97] Internetstiftelsen. *Bredbandskollen CLI | Bredbandskollen*. URL: http://www.bredbandskollen.se/om/mer-om-bbk/bredbandskollen-cli/ (visited on 10/11/2022).

[98] Internetstiftelsen. *Mer om Bredbandskollen | Bredbandskollen*. URL: http://www.bredbandskollen.se/om/mer-om-bbk/ (visited on 10/11/2022).

[99] *Wireshark · Go Deep.* URL: https://www.wireshark.org/ (visited on 11/20/2022).

[100] Glenn Olsson. *Fejk File System*. July 28, 2022. URL: https://github.com/GlennOlsson/FFS (visited on 10/09/2022).

[101] *Files: Update | Drive API | Google Developers*. Oct. 5, 2022. URL: https://developers.google.com/drive/api/v3/reference/files/update (visited on 11/20/2022).

[102] Google. *Download Files | Drive API | Google Developers*. Sept. 27, 2022. URL: https://developers.google.com/drive/api/guides/manage-downloads (visited on 11/20/2022).

[103] Frederic Lardinois. *Google Updates Drive with a Focus on Its Business Users*. TechCrunch. Mar. 9, 2017. URL: https://techcrunch.com/2017/03/09/google-drive-now-has-800m-users-and-gets-a-big-update-for-the-enterprise/ (visited on 11/20/2022).

[104] Stefan Campbell. *Flickr Statistics 2022: How Many People Use Flickr? - The Small Business Blog*. Nov. 16, 2022. URL: https://thesmallbusinessblog.net/flickr-statistics/ (visited on 11/20/2022).

[105] cppreference.com. *Std::Basic_filebuf - Cppreference.Com*. July 19, 2020. URL: https://en.cppreference.com/w/cpp/io/basic_filebuf (visited on 11/20/2022).

[106] *RandomNumberGenerator - Crypto++ Wiki*. Apr. 13, 2021. URL: https://cryptopp.com/wiki/RandomNumberGenerator (visited on 09/11/2022).

[107] geekosaur. *Answer to "Why Are Dot Underscore ._ Files Created, and How Can I Avoid Them?"* Ask Different. May 29, 2011. URL: https://apple.stackexchange.com/a/14981 (visited on 09/11/2022).

[108] Post- och telestyrelsen. *Frågor och svar om datalagring | PTS*. pts.se. Sept. 27, 2019. URL: https://www.pts.se/sv/bransch/internet/sakerhet-och-skydd-av-uppgifter/Brottsbekampning/fragor-och-svar/Pts.se (visited on 11/20/2022).

[109] *European Court of Justice/Sweden: Invalidation of Data Retention Obligations*. Library of Congress, Washington, D.C. 20540 USA. Jan. 19, 2017. URL: https://www.loc.gov/item/global-legal-monitor/2017-01-19/european-court-of-justicesweden-invalidation-of-data-retention-obligations/ (visited on 11/20/2022).

[110] Law Offices of Salar Atrizadeh. *United States Data Retention Laws*. Internet Lawyer Blog. Dec. 13, 2021. URL: https://www.internetlawyer-blog.com/united-states-data-retention-laws/ (visited on 11/20/2022).

[111] McAfee Institute. *Data Retention Laws in the United States*. McAfee Institute. URL: https://www.mcafeeinstitute.com/blogs/articles/data-retention-laws-in-the-united-states (visited on 11/20/2022).

[112] Ray Walsh. *Internet Censorship in Sweden | Staying Secure Online*. ProPrivacy.com. Nov. 29, 2020. URL: https://proprivacy.com/guides/swedish-privacy (visited on 11/21/2022).

[113] E. Ramadhani. "Anonymity Communication VPN and Tor: A Comparative Study". In: *Journal of Physics: Conference Series* 983.1 (Mar. 2018), p. 012060.

ISSN: 1742-6596. DOI: 10.1088/1742-6596/983/1/012060. URL: https://dx.doi.org/10.1088/1742-6596/983/1/012060 (visited on 11/21/2022).

[114] Flickr, Inc. *Flickr Community Guidelines*. Flickr. June 27, 2022. URL: https://www.flickr.com/help/guidelines (visited on 11/21/2022).

[115] Amber Rudd. *Encryption and Counter-Terrorism: Getting the Balance Right*. GOV.UK. July 31, 2017. URL: https://www.gov.uk/government/speeches/encryption-and-counter-terrorism-getting-the-balance-right (visited on 09/11/2022).

[116] David Berreby. "Engineering Terror". In: *The New York Times. Magazine* (Sept. 10, 2010). ISSN: 0362-4331. URL: https://www.nytimes.com/2010/09/12/magazine/12FOB-IdeaLab-t.html (visited on 09/11/2022).

[117] Jessica McLean. *Data Centers Generate the Same Amount of Carbon Emissions as Global Airlines*. TNW | Syndication. Feb. 15, 2020. URL: https://thenextweb.com/news/data-centers-generate-the-same-amount-of-carbon-emissions-as-global-airlines (visited on 10/09/2022).

[118] Fred Pearce. *Energy Hogs: Can World's Huge Data Centers Be Made More Efficient?* Yale E360. URL: https://e360.yale.edu/features/energy-hogs-can-huge-data-centers-be-made-more-efficient (visited on 10/09/2022).

[119] Nicole Cappella. *Sweden and the Sustainable Data Centre*. Techerati. Jan. 7, 2022. URL: https://www.techerati.com/features-hub/opinions/sweden-and-the-sustainable-data-centre/ (visited on 11/21/2022).

[120] UNFCCC. *EcoDataCenter - Sweden | UNFCCC*. URL: https://unfccc.int/climate-action/momentum-for-change/activity-database/ecodatacenter (visited on 11/21/2022).

APPENDICES

Appendix A

DIRECTORY, INODETABLE, AND INODEENTRY CLASS AND ATTRIBUTES REPRESENTATION

This chapter present pseudo-code of the different data structures used by . Listing A.1 presents the attributes each C++ class stores in-memory, which is also encoded into the binary representation of the object when it is serialized before uploaded to the .

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

```
# typedef inode_id = uint32_t

# Represents a directory in \gls{FFS}. Keeps track of the
    filename and inode of each file
class Directory
        # Map of (filename, inode id) representing the
            content of the directory
        map<string, inode_id> entries


# Represents an entry in the inode table, representing a file
    or directory
class InodeEntry
        # The size of the file (not used for directories)
        uint32_t length

        # True if the entry describes a directory, false if
            it describes a file
```

```
        uint8_t is_dir

        # When the file first was created
        uint64_t time_created
        # When the file was last accessed
        uint64_t time_accessed
        # When the file was last modified
        uint64_t time_modified

        # A list representing the posts of the file or
            directory.
        string[] post_ids

# Represents the inode table of the filesystem. The table
    consists of multiple inode entries
class InodeTable
        # Map of (inode id, inode entry) for each file and
            directory in the filesystem
        map<inode_t, InodeEntry> entries
```

Appendix B

BINARY REPRESENTATION OF 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 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 object's 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 visualize the serialized output of the different classes. Figure B.1 visualizes the serialized format of the `InodeTable`. Figure B.1 visualizes the serialized format of the `InodeEntry`. Figure B.1 visualizes the serialized format of the `Directory`.

B.2 FFS Images

An image consists of multiple binary structures, including the header and the encrypted data. This section visualizes these binary structures. Figure B.2 visualizes binary format of the header. Figure B.2 visualizes the of images stored on the .

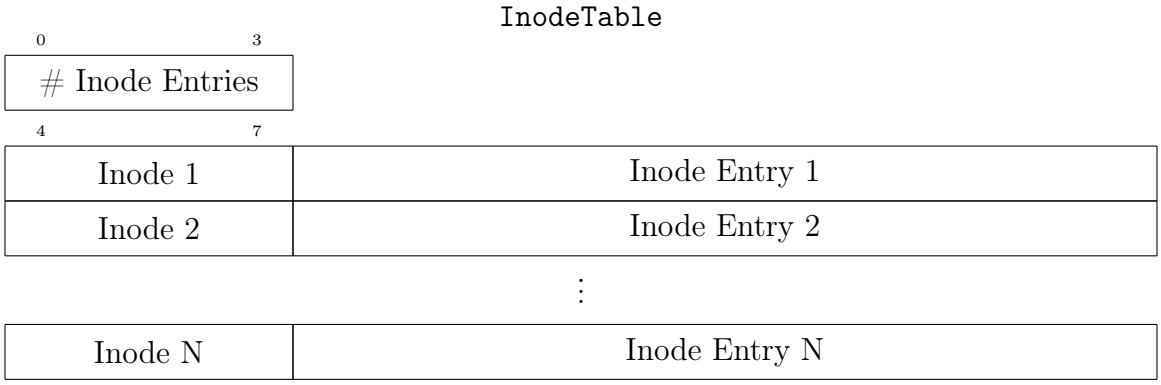

Figure B.1: `# Inode Entries` **is an unsigned integer representing the amount of inode entries the inode table contains. Following are** `# Inode Entries` **entries of an unsigned integer representing the inode of the inode entry, and the serialization of the corresponding** `InodeEntry` **object**

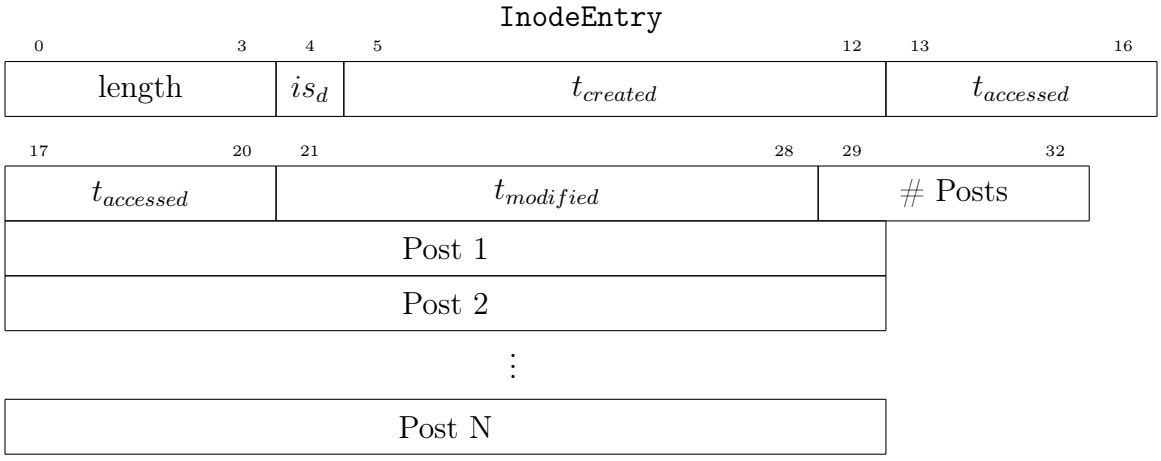

Figure B.2: Byte representation of a serialized `InodeEntry`**, representing a file or directory stored in . `length` is an unsigned integer representing the amount of data stored on by the file or directory, for instance the size of the file. is_d is a boolean with the value true ($\neq 0$) if the inode 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 . Following are** `# Posts` **NULL-terminated strings representing each post ID in the . The size of this field depends on the 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**

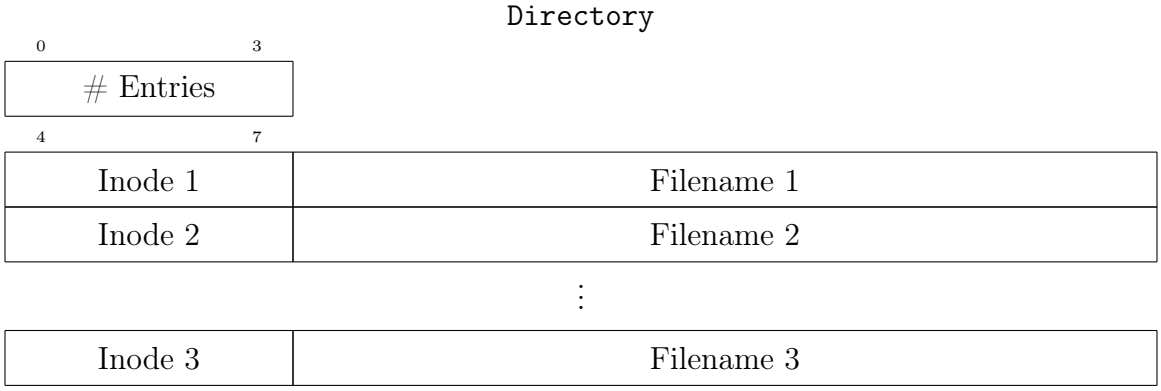

Figure B.3: Byte representation of a serialized `Directory`. `# Entries` is an unsigned integer representing the amount of entries in the directory. Following are `# Entries` inode-filename pairs. The `Inode` is an integer representing the inode of the file or directory, corresponding to the file's or directory's entry in the inode table. The filename is a NULL-terminated strings representing the filename of the file or directory in . The size of this field can vary from filename to filename, but the maximum size of the field is 129 B (128 characters + NULL character). As the strings are NULL-terminated, the deserializer can read the bytes until the NULL-character is found

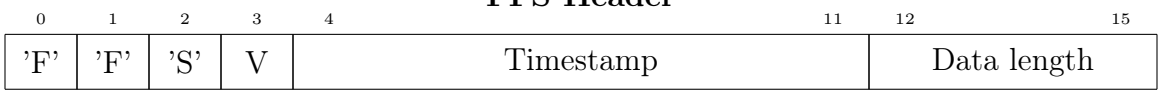

Figure B.4: `'F'` and `'S'` are the literal letters F and S in ASCII code. `V` is an integer representing the version of the image produced. `Timestamp` is an unsigned integer representing the number of milliseconds since Unix epoch when the image was encoded. `Data length` is an unsigned integer representing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

in images

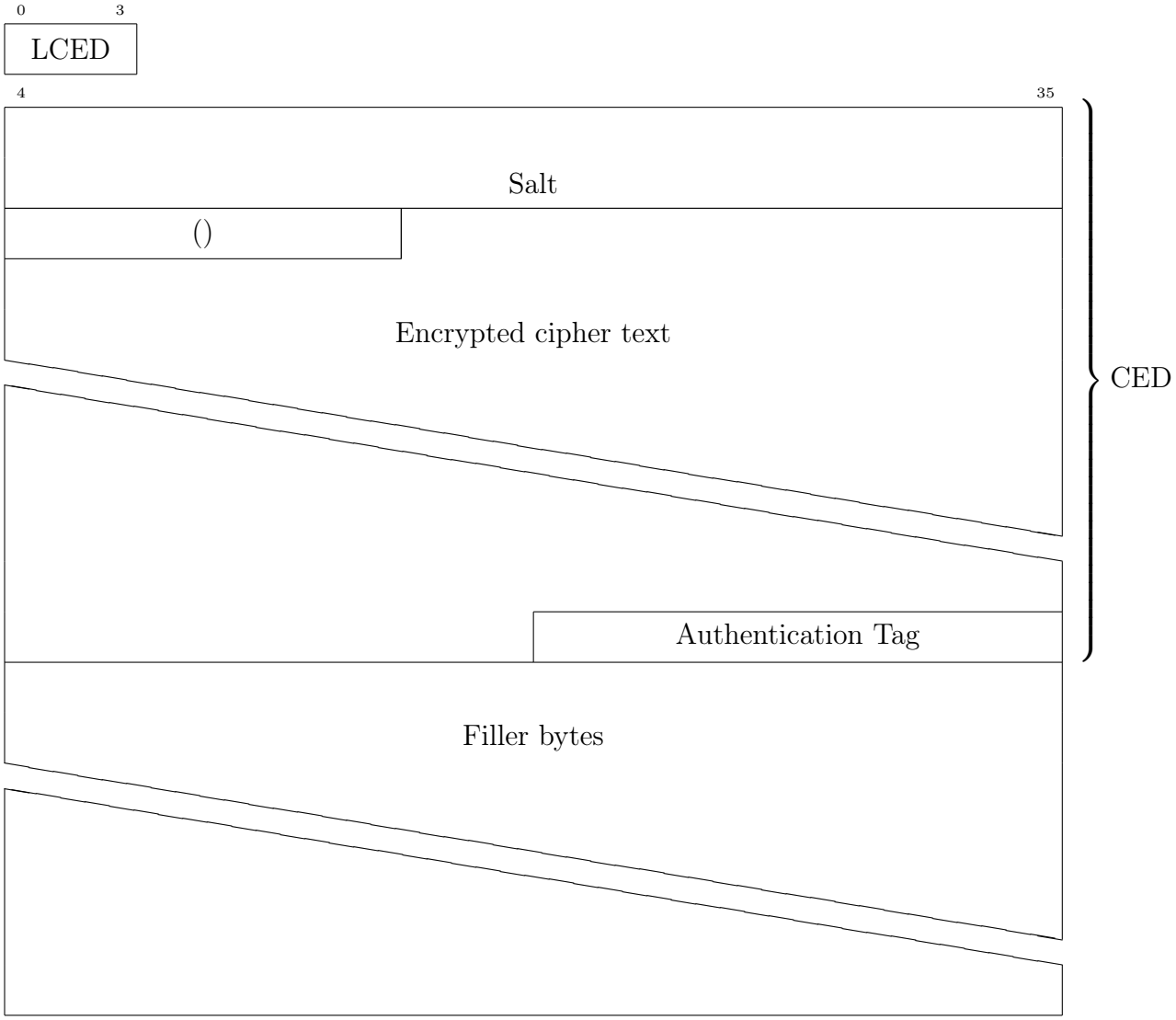

Figure B.5: Byte representation of the data stored as in images. `LCED` is an unsigned integer representing the Length of the . The `Salt` is a 64-byte randomized vector used to derive the encryption and decryption key. The `IV` is a 12-byte randomized vector used as the initial state of the encryption and decryption methods. Following is the `Encrypted cipher text` of variable size, depending on the size of the unencrypted data. The 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.

Appendix C

IOZONE BENCHMARKING DATA

This appendix contains tables of the IOZone benchmarking outputs produced. Each section contains the raw table output for the benchmarking results of the specific filesystem. Each table represents one IOZone test for a filesystem. Each table presents the throughput in kilobytes per second for a test, for the different file sizes and buffer sizes, both presented in kB.

C.1 FFS

Table C.1: IOZone result for the Read test on FFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1124099	1875690	2072054	2606476	2386350	2844715	2604895	2173780	3210456				
2048	1378181	567636	2033695	2972495	3154809	3121562	3690142	2301200	3519302	2235910			
4096	1614527	2298370	2955012	4027344	4628439	3942321	4511750	4007615	4329816	4095510	3953206		
8192	2021184	2978787	3852936	5329730	3541581	3591558	4638941	6390362	3444652	5919188	4003788	5095007	
16384	60625	2246482	63191	4702688	61471	6206148	2936754	6362439	69217	5168507	2375140	117941	4462939
32768	55818	52177	4466247	5033507	60178	54036	35729	54242	5819047	53512	5501562	64800	86733
65536	124344	50556	54524	52190	4810367	67341	53582	50273	52437	4826246	56654	56309	4882655
131072	2973359	4220618	68079	70359	69429	7874149	60407	70434	7260	81991	62072	6522596	79478
262144	68103	70685	68449	69849	99154	70227	68031	71294	68417	59723	7519272	6089594	161814

Table C.2: IOZone result for the Write test on FFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	206	270	312	271	262	319	248	310	325				
2048	492	447	486	441	532	458	496	506	536	504			
4096	774	803	686	853	816	846	829	746	738	816	839		
8192	1099	640	1236	1056	1054	1095	1078	975	1177	1068	1029	1077	
16384	1320	1240	1369	1344	1413	1363	1339	1278	1349	1427	1341	1344	1187
32768	1573	1749	1685	1625	1610	1647	1639	1750	1745	1734	1636	1650	1727
65536	1875	1995	2062	1724	2029	1995	1783	2007	2008	2052	2109	1955	2068
131072	2059	2297	2472	2154	2176	2362	2236	2465	2433	2393	2644	2474	2384
262144	2339	2139	2597	2157	2529	2421	548	894	2623	2612	2676	2637	2646

Table C.3: IOZone result for the Re-Read test on FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1025833	2056183	1600327	2789291	2966535	2625597	3020783	2976816	4178773				
2048	2183619	1430271	2595298	2418469	3675930	2748949	4762117	3272598	3555722	3117032			
4096	2326065	2740095	3093350	4218190	4298399	4735608	5618365	5483860	5704175	3419011	4321103		
8192	2185761	3618412	4754484	5197511	4250966	6248582	6725591	6522590	4385525	6765318	4771651	4445097	
16384	1841910	2301789	4900900	5791387	6534875	6582447	6852209	6427904	7599353	3720236	5858035	4687931	4717539
32768	2375173	3448145	4222679	5487065	6201613	6714556	5137750	6366213	6636098	6273801	5597460	5099433	5232021
65536	1777628	3099807	3788868	5457737	4664497	5469466	7114221	6547080	6423160	5395284	5009273	4816014	4676161
131072	2666778	3783414	5009036	6003942	6385321	8230640	6553543	7089250	6747555	6880134	5982251	6600438	6078894
262144	2583743	4449684	5590363	6013557	6771275	6868693	7097066	7641769	7053902	7329012	4997807	4674122	5487959

Table C.4: IOZone result for the Re-Write test on FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	256	246	200	264	308	269	250	293	324				
2048	532	511	404	547	505	533	564	489	434	457			
4096	864	768	775	826	762	793	880	767	787	835	721		
8192	942	930	958	935	896	1006	895	863	942	862	959	943	
16384	1155	1100	1113	1147	1141	1130	1106	1160	1048	1160	1137	1089	1111
32768	1327	1357	1335	1423	1350	1387	1363	1351	1296	1411	1245	1390	1282
65536	1447	1542	1534	1474	1420	1632	1412	1661	1579	1585	1599	1679	1637
131072	1729	1813	1637	1641	1567	1748	1716	1963	1945	1936	1902	1954	1885
262144	1822	1823	1944	1949	1939	1995	1940	1983	2009	1967	1987	2000	1992

Table C.5: IOZone result for the Random read test on FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	157493	2216407	1627617	1741810	2852271	2926114	3722435	2275110	4146499				
2048	1782585	416667	2916978	2752473	3379461	4031308	2763987	3624742	4265523	3894239			
4096	1536962	2719708	2965725	4270618	3497668	5411313	3918939	5360658	4409837	4302706	4707064		
8192	1719637	2589902	4021596	4670469	4928384	6420214	6703285	6996761	4528880	5978928	4879393	4551075	
16384	1790371	1826537	4196659	4780882	4288854	5544663	6963302	6863844	7133159	4528524	4973975	4546200	4920199
32768	1652689	2699602	3615596	5126252	5529450	5983722	5242798	6766123	5356822	5140824	5706695	5170803	2796839
65536	1361645	2149636	3197042	4471609	4626031	5484198	6721584	6197810	6846648	3663045	4794509	5149377	4639069
131072	1971894	3118619	3812353	5415790	6283727	7615527	5731410	7322871	6164024	6919971	5118969	4799231	5960782
262144	1813113	3068092	3975558	4985728	6536121	6311675	6655244	7390988	7063782	7242548	6004755	5722694	5567942

Table C.6: IOZone result for the Random write test on FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	230	255	264	338	287	304	184	317	289				
2048	524	431	462	512	468	513	570	492	446	558			
4096	822	857	787	688	811	799	791	815	793	766	828		
8192	907	917	861	865	993	977	747	904	913	918	929	972	
16384	1021	1013	1197	1126	1170	1127	1163	1031	1119	1138	1036	1180	1057
32768	1297	1268	1299	1405	1314	1310	1238	1210	1326	1297	1298	1334	1322
65536	1375	1384	1634	1522	1591	1587	1671	1606	1608	1634	1597	1624	1627
131072	1605	1784	1797	1715	1595	1631	1960	1910	1842	2001	1922	1935	1929
262144	1793	1945	1870	1987	2005	1962	1952	1999	1782	1988	2019	2014	2034

C.2 GCSF

Table C.7: IOZone result for the Read test on GCSF

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1790460	2035716	2753527	3411938	3003881	3401130	3722435	4162573	4493556				
2048	945007	2308622	3297725	4170264	3698085	2467800	3506373	4611288	5416763	4376355			
4096	1858527	2813233	1017897	4969868	4399673	4941279	4591331	5657217	5625724	5019235	3327622		
8192	651866	480214	4126896	4664763	903994	5357988	3195123	5288712	6804170	1412655	4833405	5085204	
16384	409261	3463891	4768276	518677	5797739	6450228	6077171	6143453	6930296	6316241	5948808	837411	5326345
32768	289995	2604351	514007	5222875	351689	380969	225834	428160	377558	3011251	422812	601160	710955
65536	420968	390311	374191	590919	6243421	863782	388813	506734	463735	478777	456922	429282	624033
131072	443068	462364	407350	496836	444001	471616	424145	518826	505112	559713	375452	526586	515828

Table C.8: IOZone result for the Write test on GCSF

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	489	517	445	461	399	496	509	448	529				
2048	937	966	854	944	1072	966	944	954	868	969			
4096	1609	1614	1749	1607	1213	1787	1802	1519	1748	1551	1784		
8192	3047	2233	2746	3115	2666	2636	3000	2974	3165	2716	3138	3122	
16384	4372	4180	4437	4690	4798	4642	4429	4781	4705	4670	4621	4875	4607
32768	5377	6125	6471	6402	6343	6463	6435	6289	5865	6052	6317	6291	6018
65536	6763	7448	7298	7825	6561	7061	7818	6959	7491	7706	5985	7343	6851
131072	7276	7789	8553	8609	8243	8596	8168	8530	8596	8433	8029	8280	7959

Table C.9: IOZone result for the Re-Read test on GCSF

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2632033	2187063	3200886	4304412	3348104	3778101	3281592	5048117	4083422				
2048	2311728	3261415	4623699	4643695	4079167	4349761	6149699	6058611	7213548	5032754			
4096	2356697	2579635	4329816	5424983	5347310	6309619	4371684	6585338	6894546	6370451	5165626		
8192	1891555	3974610	4017834	4884942	6112956	6262248	5158495	5116249	7087688	6720329	5270864	5038228	
16384	2228199	3960364	5075741	6187150	6473317	6838571	6919132	7139088	7211005	6614761	6187707	5016457	4594528
32768	2503678	3836954	4593731	6020418	6336861	5357031	1864671	6139832	5652480	6276093	5945927	4010262	4741431
65536	2639381	3531600	5031555	6133503	6445904	5921182	5961505	6217859	6461207	6398788	5278722	4674094	5364747
131072	3039494	3904216	4889073	5950781	6275836	7495395	5875292	7344197	7196622	8031869	4689156	5687415	5207460

Table C.10: IOZone result for the Re-Write test on GCSF

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	371	330	295	433	400	329	259	335	409				
2048	550	612	738	736	814	775	600	592	799	787			
4096	873	1354	1503	1239	927	1317	1368	1342	1424	1277	1325		
8192	1118	2201	1192	2117	1883	2093	2004	2126	1877	2192	2014	2165	
16384	1370	2949	2955	2963	2678	2950	2974	3018	3134	2989	2827	3067	1401
32768	3354	3448	3200	3592	3656	3815	3532	3702	3629	3675	3508	3708	3621
65536	3815	3962	3849	2319	3794	4085	3970	4080	3929	4033	3787	2972	4044
131072	4443	4682	4721	4807	4749	4866	4774	4766	4683	4760	4623	4657	4804

Table C.11: IOZone result for the Random read test on GCSF

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1992279	1741104	3923040	2892612	3284101	2875184	4282950	3684119	3179559				
2048	2122645	2592166	2843590	5006356	4221501	4481379	6583305	5200329	7343043	4934459			
4096	1723189	2471992	4007615	4899008	5625724	5713661	6503078	6897314	4958393	5270212	5063617		
8192	1438199	3111222	4149824	5095007	5821902	5885728	4543853	6350207	7173514	6501608	5372229	4654652	
16384	1859705	2912112	4175240	5158807	6039252	6725463	6203907	6921919	6820923	6546080	5667676	5441055	5336686
32768	1705159	2794224	4285618	5467855	5548200	6087347	6091664	5991547	5893405	6215636	6222672	3776853	5156254
65536	1843064	2695626	4393982	4975003	6203125	5990477	4676082	7062855	6735419	6700285	4143421	4907587	5587993
131072	2100647	2842719	3678063	5056629	5706720	6765741	5708261	7233171	7090347	7451506	4198987	5471360	5404768

Table C.12: IOZone result for the Random write test on GCSF

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	373	338	295	364	355	222	368	404	410				
2048	739	784	753	789	852	823	688	741	639	657			
4096	958	1418	1299	1372	812	884	1369	1330	1416	1194	1335		
8192	2200	2310	2289	2445	2174	2195	2106	2043	2063	1962	2111	2195	
16384	2739	2747	3476	3141	3478	2894	3123	2949	3054	2925	2941	2911	1223
32768	2567	3154	3766	4474	3738	4330	3664	3798	3534	3745	3675	3732	3613
65536	2646	2769	3422	4011	4985	4159	5019	3739	4051	3962	3899	4087	4193
131072	2124	3073	3481	4239	5025	6502	5285	6129	4806	4689	4747	4834	4832

C.3 Fejk FFS

Table C.13: IOZone result for the Read test on Fejk FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1605111	2472911	889631	2715230	3618930	3001782	1032244	2578312	2467229				
2048	1921339	1930406	2194777	2821176	3200654	4130162	4154130	2649737	5032754	3286370			
4096	2426601	2784507	3271230	3574071	3670280	5542240	4386193	3771811	3995498	4763181	4000150		
8192	2895693	3535750	4173008	5378957	4899570	4548665	5914094	6558696	4271046	4104220	5475825	5888754	
16384	3094244	4152032	4908602	3512941	6234299	5855539	6313919	7272051	6212881	7111751	5507336	5387732	5486670
32768	2848900	3537882	4681672	6104381	6554760	5933093	6802626	6375072	6339199	6146697	6030191	5591994	5440583
65536	70116	3562495	5159913	5999237	7335592	7026566	7648924	6445904	77043	129764	5783761	5526649	4978066
131072	394463	2034461	454646	6311863	3782086	280028	7525973	6890568	75477	59220	2674224	238745	657556
262144	65486	75529	77854	97658	79916	72736	76276	71780	106689	72269	72176	79924	126614

Table C.14: IOZone result for the Write test on Fejk FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5193	5287	4303	5160	5076	5282	27847	5736	5086				
2048	4807	5335	4665	6305	6258	5920	6228	6453	5811	6301			
4096	5767	5943	5979	5807	6265	6320	3784	6770	5759	6188	6844		
8192	5504	6464	6702	6783	6175	6015	5288	6874	98659	101980	7043	6738	
16384	6430	6955	7463	7442	6794	7637	7667	7628	7337	101677	7553	7824	7735
32768	7232	7656	7808	7859	8075	8142	8075	8062	8161	7896	8229	117474	8377
65536	7370	7246	7273	7350	7530	7664	7658	8170	6959	7196	7192	7019	6735
131072	7160	6974	6961	7229	7593	7217	7901	8196	8116	5763	7398	5990	8025
262144	7167	7419	7937	7545	8435	8520	8030	7742	3996	7468	7601	8066	7891
524288	7578	7967	8161	82	8189	8494	8454	8327	8466	8501	8370	8308	5935

Table C.15: IOZone result for the Re-Read test on Fejk FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1984913	2820430	2353657	4674510	4232305	3198502	1920993	4943530	3142339				
2048	2107025	3011047	2367151	3915540	4633675	5209791	5119743	4663865	4829046	3736694			
4096	3075079	3614679	4462528	4302706	5971855	6503078	4906002	6471234	4132949	5184332	3442303		
8192	2826843	4114541	4678737	5261179	4158362	6182248	5094251	5357152	6537482	5382328	5756551	5915112	
16384	3115426	4517212	5147601	6343059	6609036	6262707	6753222	7984318	6787239	7246745	5728625	5310703	5452279
32768	2786858	3742299	5009656	7004755	7299786	7123475	7777741	6994062	6752162	7039557	6382473	5967613	5664360
65536	2205856	4396090	5428742	6393876	6889031	6089074	7811071	6860318	7461842	6801920	6440769	5750246	4823451
131072	3287977	3847707	5198793	6348527	6951999	8443258	7909267	7114111	7388515	5298401	3139222	5802490	6479087
262144	2974653	4452026	5376537	6145825	7115944	7217446	7116911	6533364	6027138	6506108	5360469	5598732	5819467

Table C.16: IOZone result for the Re-Write test on Fejk FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	4574	4975	1566	5157	4391	4913	3380	27982	5001				
2048	3371	4654	5206	4614	5631	5635	5680	31266	5577	5685			
4096	5548	5728	5588	5188	5330	5148	5077	5928	5449	4753	5515		
8192	5296	5167	5574	5177	5002	5540	5362	5527	5913	5893	5485	4895	
16384	5933	5866	6309	6263	5714	6230	6215	6563	6454	6412	5640	25482	6565
32768	6314	6369	6575	6687	6841	6852	6824	6819	6889	6768	6782	6699	6885
65536	5933	6327	5753	6290	6514	5615	6329	6632	5962	5815	5749	5935	25761
131072	5885	5619	6099	6598	6601	5990	6717	6763	5816	4048	6124	4737	6436
262144	5769	6203	6474	6722	6879	6865	6565	6140	6177	6033	6164	6586	6653

Table C.17: IOZone result for the Random read test on Fejk FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1438461	2474336	3335105	4339202	4574926	3594699	1662264	4415030	3390391				
2048	1639674	2531808	2980747	3071337	4971585	5626082	5235193	4240255	3864455	4591569			
4096	2333965	2898672	4076076	4214051	3641495	6716643	3128843	6022095	4428023	4935601	5551194		
8192	2217071	1882745	4242044	6006101	3655755	6054786	6947247	6912307	6339662	5785631	5789530	5583494	
16384	2235520	2613523	4010753	5637917	5897753	5362924	6542963	7922644	6671272	7117644	4702688	5233056	5189585
32768	1953956	2985674	4118168	6358850	7157605	6732647	7552086	7015482	6541968	7126430	6545707	5985025	5650853
65536	1320279	3522910	4420978	5494832	6317898	6091233	7578280	6635907	7317626	6684480	6613236	5663404	4218907
131072	2179297	3011967	4211016	5669935	6324062	7899720	7685258	6951208	7292757	5564284	2867974	5863448	6536945
262144	2105727	3331986	4265958	5398262	6599796	7121152	6889437	6293359	6270783	5819868	5256751	5823875	5831010

Table C.18: IOZone result for the Random write test on Fejk FFS

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	4619	4008	2880	4530	4425	4764	3097	5879	4465				
2048	4620	4691	5179	5665	5728	5660	4573	5554	5775	5519			
4096	5284	5681	5333	5194	5488	5822	5854	6127	37216	5145	5738		
8192	5442	4747	5737	5386	5800	5088	5881	5811	5414	5708	4852	5635	
16384	5553	6011	6191	6267	6188	5654	6348	6585	6428	6509	5831	6558	6496
32768	6253	6238	6434	6511	6810	6829	6984	6800	6877	6801	6853	6875	6857
65536	4967	6318	5379	5753	6616	6361	6528	6611	5854	5900	5980	6472	5881
131072	5754	5611	6114	6317	6102	6407	6469	6662	4629	4861	5661	5536	6443
262144	5741	6259	6439	6689	6907	6427	6463	6100	6309	5998	6035	6849	6857

C.4 APFS

Table C.19: IOZone result for the Read test on APFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2300703	3472628	5444898	6025439	3425544	2353657	3521025	5479632	1259592				
2048	2429413	3548378	2324238	2288326	2392202	7817519	5277002	6939647	6867521	7529708			
4096	2597576	3931494	4922872	3456849	6151473	6908408	7064655	5894001	4648477	5402804	6281934		
8192	2693866	3716653	4555903	5487193	4143818	6405850	8054689	7585256	5708730	7537004	4594281	5576245	
16384	1887130	3339494	5330063	3537899	6635840	6409917	6430310	4554938	6921222	3453620	2166589	5158807	4187197
32768	2452479	2775321	4702013	6173478	5991547	6411354	6621073	5601338	7129387	6994417	5692749	5448563	4751758
65536	2673966	4183591	5032015	6032415	6770924	6720270	6932816	7149936	6714032	6609102	6538515	5680492	5542138
131072	2775537	4124081	5119016	5613831	5348765	6775497	4877838	5514443	4523778	5408171	2241853	2555460	2956665
262144	2612033	4088592	5074689	6218999	6861877	6715279	6735808	7042156	6728553	3955521	5947137	4522621	5227609

Table C.20: IOZone result for the Write test on APFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	159475	302147	204797	669331	770562	1528563	931702	371703	1340164				
2048	194475	310725	254254	368875	880855	1237420	1097081	1219155	1295263	1823453			
4096	199931	318406	640709	611714	1033076	1064440	1039326	931976	900421	1207570	1213627		
8192	204824	316817	385558	593660	651767	589312	784669	772167	831858	729741	672246	823307	
16384	204440	287428	473154	599882	656623	774480	800275	732212	736592	786598	815402	797276	729763
32768	144265	292913	473815	630616	713183	641806	729863	628379	622493	659873	643456	669350	747666
65536	251368	348690	492085	481153	751525	761160	762827	492373	784232	780599	465990	763618	781920
131072	214970	346696	490691	576345	707747	631216	649070	782725	785093	624839	724503	447659	787787
262144	204778	333219	414944	473988	524245	596270	688020	648681	713499	591206	417813	724537	704260

Table C.21: IOZone result for the Re-Read test on APFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2672984	3999762	7590887	8246774	3582705	2820430	5226256	7208671	4014717				
2048	2519185	3878414	2094183	2531808	3319389	9752360	3855782	10143926	8784909	10713237			
4096	2659914	4201683	5497899	4581535	6628532	7756829	8205092	7447471	6066754	4853336	5445618		
8192	2906963	3849914	3912159	6542461	6848927	5607184	8532752	7550253	7593638	8127085	5809106	5501249	
16384	2220998	2777913	5518393	2383791	6829736	6747917	6622411	2365574	7304517	4617684	2171450	5157259	3420444
32768	2515593	422414	4812321	6197698	6090584	6317346	6907241	6022528	7163948	7024087	5715476	5106444	5972021
65536	2746120	4150992	5069321	6282807	6731955	6501553	6983011	6811697	6463182	6476583	6588509	5450919	5876120
131072	2755173	4126310	5114587	5225974	4340138	6555887	5927107	5109264	4320967	5892231	2764011	3077523	4241949
262144	2857991	3711246	5355664	6016321	6473815	6511426	6046594	7033237	6847177	3413775	6368117	5259819	4680728

Table C.22: IOZone result for the Re-Write test on APFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	516668	888894	1378002	1271902	1232484	1896395	760735	2015654	793775				
2048	446281	1080657	1063926	1030239	111521	2704804	2258247	2365196	2458618	3245396			
4096	607259	784547	1234469	1146103	1474438	1693297	1823802	1577900	1624143	1721808	1454465		
8192	483301	617095	792195	853238	901716	999881	1028771	950560	977217	964787	934938	980703	
16384	451959	578063	779372	764570	849712	856906	877175	841410	446416	833672	847375	857398	848516
32768	445138	560634	769079	803450	817277	823935	819651	820120	825722	824999	819279	822821	810668
65536	486312	652780	774380	789258	798799	803976	802861	806467	800008	803372	801959	802725	797411
131072	476958	643974	758567	644890	762831	772610	770272	544649	776289	790748	743047	740269	785215
262144	462449	646059	524252	491535	784510	785636	730755	787167	786522	687147	754028	785252	785848

Table C.23: IOZone result for the Random read test on APFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1899750	3074849	6059442	7485055	3941039	2572135	3377062	6441107	1299219				
2048	2076967	3287628	1943069	2801852	3287628	9752360	4204968	9944290	8261095	10037248			
4096	1978835	3335375	5236478	2703870	6941906	8944095	9524234	7558881	5996870	4875373	6103394		
8192	1819630	3044239	3383593	5888754	6670750	5776876	8642207	7779318	6924845	5905961	5572627	5237920	
16384	1554007	2421507	4341144	4405431	6212881	5181367	6039252	3657858	7139088	5282129	5080244	4895314	3495607
32768	1558975	2220991	4084269	5435204	6073896	5979556	6635778	5711676	7174793	7100291	6436274	5468725	5888102
65536	1901416	3064454	4018870	5162336	6343120	6161136	6712557	7024950	6344145	6591827	6297922	5434646	5464464
131072	1887775	3091541	4023453	5183353	3293670	5722700	5496414	5696785	3931719	6076610	1999695	847392	3620272
262144	1883367	2885714	4278225	5223883	5600786	5905358	6090977	6498034	7061150	4274183	6760118	5397122	4505903

Table C.24: IOZone result for the Random write test on APFS

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	584523	755251	2020395	2534194	2835325	2437821	2008114	2255990	799241				
2048	649335	954776	474508	1523901	2223755	3519302	1798259	1793378	3318107	2738433			
4096	532497	886025	1069144	1136474	1860943	2239351	1813981	1903416	1447846	1672526	1405213		
8192	554530	810279	965247	1353555	1451443	1545660	1714146	1622824	983538	1018766	935600	955796	
16384	357261	559387	764001	760172	907993	931337	968783	913327	953102	781980	969179	837370	808209
32768	201371	413784	651295	733376	774802	808646	817647	795687	816427	815081	805050	822446	820184
65536	378858	501350	604459	663018	708627	728283	756312	761905	740855	735434	727814	779968	782116
131072	373346	506122	594740	650852	607028	667936	692831	608203	567929	692191	579639	581513	699404
262144	370122	450730	466516	573833	670580	691430	567649	705739	713318	566151	695148	772297	770628