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 Online Web Service (OWS)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 by encoding the encrypted data as images. Images have been selected as the target as more data can usually be stored in image posts than in text posts on OWSs. The filesystem, named The Fejk Filesystem (FFS), provides users with free, deniable, and cryptographic storage by exploiting the storage provided by online web services. The thesis analyzes and compares the performance of FFS against two other filesystems and a version of FFS that does not use an OWS. While FFS has performance limitations that make it non-viable as a general-purpose filesystem, such as a substitute to the local filesystem on a computer; however, FFS provides security benefits compared to other cloud-based filesystems specifically by providing end-to-end encryption, authenticated encryption, and plausible deniability of the data. Furthermore, being a cloud-based filesystem, FFS 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 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.

There exist multiple deniable filesystems that are designed to address this problem on digital storage 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 cryptographic and deniable 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 FFS that 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 explore how feasible it is 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 are 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 proposed filesystem is discussed and an analysis of the deniability of the developed filesystem is presented.

1.1 Problem

Current cryptographic filesystems are often based on local-disk solutions. Distributed filessystem services, such as Google Drive, might encrypt your data but it can be

considered unsafe storage as they can give out your data if as they do not always provide end-to-end encryption. 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 the social media service providers a potentially good host of the data for such a filesystem as they would not be able to access the unencrypted data nor have any idea how the posts are connected, and the usage of these services for data storage might even go unnoticed due to the existing heavy load of data from regular users of these services. This raises the following questions: Is it possible to use the storage offered by 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 concerning write and read speed, storage capacity, and reliability? Are there advantages to such a filesystem regarding security and deniability?

1.2 Purpose and motivation

The purpose of this research is to explore the possibility to create a cryptographically secure and deniable cloud-based filesystem that stores data on an Online Web Service (OWS) and to compare the performance, benefits, and disadvantages of such a filesystem to existing deniable filesystems and distributed filesystem services. A distributed filesystem service, such as Google Drive, provides data storage for users which can be free or 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. Because the service providers have the decryption keys, this opens up the possibility of hackers gaining access to the files without the user having any way to prevent this access.

The idea behind FFS 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. We also want adversaries to be unable to prove the amount of data and the existence of data, even when the images are posted publicly.

There are several deniable filesystems available but these lack certain aspects that FFS aims to solve. Some filesystems are based on the local disk of the device in use, such as the physical storage device on a computer or phone, or an external storage device connected to a computer or phone. While these filesystems have advantages compared to cloud-based solutions, such as low 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 lead to problems, such as it could 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 data is stored using standard steganographic methods in real images, meaning that there is a limit on how much data can be stored in a steganographic fashion. 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. FFS stores as much data as possible in the images, meaning that fewer images are needed to store a file bigger than 4 kB. This also means that the images produced by FFS does not look like a normal image, but instead have 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 an instance FFS if the filesystem is not mounted,
5. to analyze the write and read speed, storage capacity, and reliability of the filesystem and compare it to commercial cloud-based filesystems and local filesystems, and,
6. to analyze and discuss environmental and ethical aspects of the filesystem.

1.4 Research Methodology

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 FUSE 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 Limitations

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 FFS. The focus is to implement a subset of the POSIX standard functions, containing only crucial functions for a simple filesystem, specifically, the FUSE functions *open*, *read*, *write*, *mkdir*, *rmdir*, *readdir*, and *rename*. However, file

access control is not a necessity and will therefore not be implemented, thus functions such as *chown* and *chmod* are not going to be implemented. The reason is that the goal is to present and evaluate the *possibility* of creating a cryptographically secure and deniable filesystem with a storage medium based on online web services and thus FFS only aims to implement a minimal filesystem.

1.6 Structure of the thesis

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

2.1 Filesystems and data storage

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

2.1.1 Unix-like filesystems

A Unix-like 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 a mapping between filenames and inodes for the files and directories in the directory. The filesystem can with an inode find information about the file or directory using the inode table. Each inode can contain any metadata that might be relevant for the filesystem, 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 number of bytes. Listing 2.1 describes a simple implementation of an inode, an inode table, and directory entries.

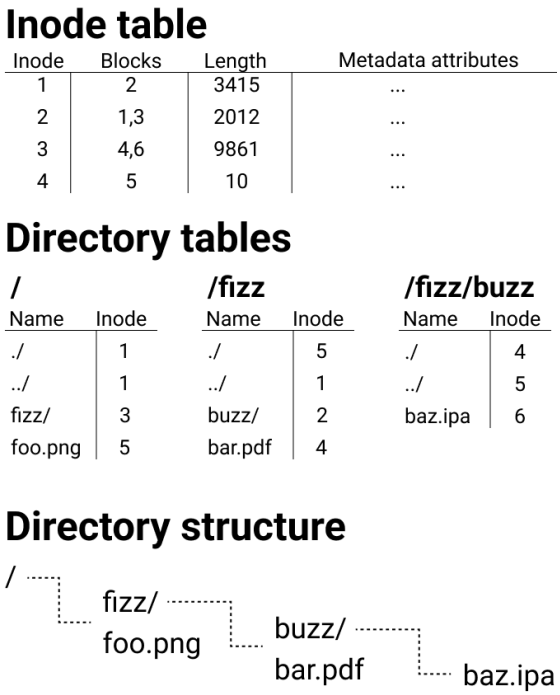

Figure 2.1: Basic structure of inode-based filesystem

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 Linux operating 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 ext3 only supports timestamps with a resolution of one second. Additionally, ext4 natively supports encryption at the directory level through the use of the fscrypt API [8].

The Apple Filesystem (APFS) is a modern filesystem that is used on iPhones and Macs 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 Solid-State drive (SSD) [10]. It also supports modern features that its predecessor Mac OS Extended (HFS+) does not support, such as Snapshots and Space Sharing. APFS natively supports encryption of the filesystem volume [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. Thus, even if the user would lose their computer or if it would malfunction, the data on the cloud-based filesystem can still be accessed which means that the data could still be recovered. These filesystems are often owned by companies, such as Google Drive and Apple's iCloud, as they are big companies

that can provide reliable storage. This also means that they have their own agenda and policies, and as they are hosting the data they have the possibility of accessing your data. The data can be encrypted on the filesystem, for instance by the service provider, 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 allows users to enable end-to-end encryption for some parts of the service, but not for the whole suite [16]. For instance, backup data and iCloud drive can be end-to-end encrypted while contacts and iCloud mail cannot.

2.1.3 Data storage and encoding

Different file types have different formats that describe of how they should be encoded and decoded, for instance, a JPEG and a PNG file can be used to display visually similar images 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).

A file as (string of bytes) can be encoded into text or as an image by representing the bytes as pixel color data and adding a suitable image header for a given image format. This image can in turn be posted on, for instance, social media. However, there is a possibility that the social media services compress the uploaded images 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 use methods such as error-correcting codes. The error-correcting codes would have to be stored in an ensured lossless format. For instance, text posts on the OWS can be used as long as the posts are not removed or the text is modified.

2.2 FUSE

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 using FUSE are generally slower than kernel-based filesystems, using FUSE simplifies the process of creating filesystems. macFUSE is a port of FUSE that operates on Apple's macOS operating system and it extends the FUSE API [19]. macFUSE provides an API for C and Objective C. During the research, experiments were not conducted to see if filesystems developed using macFUSE can be mounted on non-macOS operating systems. As macFUSE is an extension of the FUSE library, there is a possibility that it is easy to port the macFUSE filesystems to the normal FUSE 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 FUSE works. FUSE consists of a kernel space part and a userspace part that perform different tasks [20]. The kernel part of FUSE operates with the Virtual Filesystem (VFS) which is a layer in both the Linux kernel and the macOS kernel that exposes a filesystem interface for userspace applications [21, 22]. The VFS interface is independent of the underlying filesystem and is an abstraction of the underlying filesystem operations which can be used on any filesystem the VFS supports. The userspace part of FUSE communicates with the kernel space part through a block device. Operations on a mounted FUSE filesystem are sent to the VFS from the user application, which is then sent to the kernel part of FUSE. If needed, the operations are transmitted to the userspace part of FUSE where the operation is handled and a response is sent back to the VFS and the user application through the FUSE kernel module. However, some actions can be handled by the FUSE kernel module directly, such as if the file is cached in the kernel part of FUSE [20]. The response is then sent back to the user application from the kernel module through the VFS.

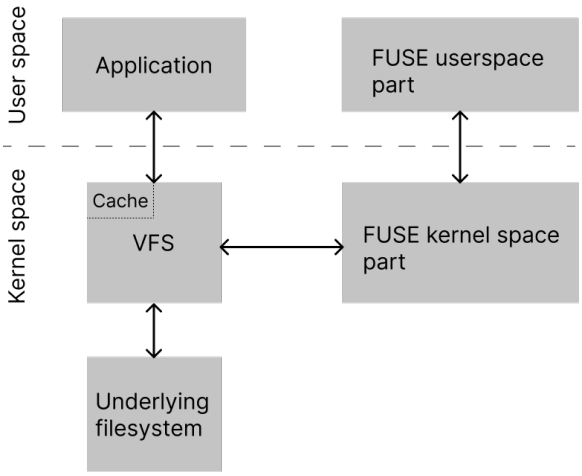

Figure 2.2: Simple visualization of how FUSE operations are executed

Figure 2.2 also visualizes the kernel cache, also known as the Unified Buffer Cache (UBC). The data stored in a macFUSE filesystem can be cached by the kernel to provide faster filesystem operations [20, 23, 24]. This can be disabled, for instance, when the filesystem can be modified independently of filesystem operations on the computer, such as for a distributed filesystem where changes can be made by another user at any time.

FUSE allows large files in the filesystems. The file operation offset, used to read or write at a byte location inside a file, is an unsigned 64 bit integer, meaning the filesystem operations can read and write to files as large as $1.8e^{19}$ bytes.

2.3 Online web services

This section presents two OWSs, Twitter and Flickr, where one can create free-tier accounts. On both of these OWSs, free-tier accounts can make numerous posts for free. The OWSs each provide a free-to-use API 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 uniqueIDassociated with it [25]. Text posts are limited to 280 characters while images can be up to 5 MB and videos up to 512 MB [26]. 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 API [27]. It is possible to create threads of Twitter posts where multiple tweets can be associated in chronological order.

Twitter's API defines technical limits of how many times certain actions can be executed by a user [28]. 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 [29]. 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. This corresponds to a maximum of 200 GB of storage if we are only storing images or 1 TB of storage if we store only videos. A mix of images and videos can also be uploaded, which would mean that the maximum storage capacity would be between 200 GB and 1 TB. 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 [30]. 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 [31]. 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 [32]. 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. Even if the original image cannot be downloaded by anonymous users when such restrictions are set by the original uploader, they can still download other versions of the image. The other versions of the image do not contain the same data as the original image but they look similarly, potentially enabling reverse engineer of the original pixel data. The original video can only be downloaded by the user [32]. Flickr does not state if it will always be possible to download the original versions of the uploaded image or video. Further, Flickr states that it retains the right to remove user content from the service at any time [33].

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

2.4 Cryptography

The Advanced Encryption Standard (AES) is a symmetric key encryption standard established by the The U.S. National Institute of Standards and Technology (NIST),

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

While AES 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]. Galois/Counter Mode (GCM) is a block cipher mode of operation which provides authenticated encryption [40]. GCM can be used together with AES to provide secure, authenticated encryption of data. To encrypt using GCM, the encryption function requires a key, a randomized Initialization Vector (IV), and the data to encrypt. The output is the encrypted cipher text and an authentication tag. The decryption function of GCM requires the same key and IV as was used as input in the encryption function, as well as the authentication tag and the cipher text received as output by the encrypting function. Further, both the encryption function and the decryption support Additional authentication data (ADD) to be provided. ADD is data that should be authenticated, but not encrypted. If ADD is provided to the encryption function, it must also be provided to the decryption function.

The key used when encrypting using AES can be derived from a password that the user provides. A Password-Based Key Derivation Function (PBKDF) is a function that can be used to derive a key to be used for, for instance, AES. The input to a PBKDF is a secret, such as a password [41]. An example of a PBKDF schema is the Hashed Message Authentication Code based Key Derivation Functions (HKDF) presented by Krawczyk [42, 43] which utilizes a hashing algorithm that provide a pseudo-random key. HKDF supports multiple hashing algorithms. The security of HKDF is partially dependent on the security of the hashing algorithm used. A well-defined suit of hashing algorithms is the The Secrure Hash Algorithms (SHA), which covers, among

other hash functions, SHA-256 [44]. SHA-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, HKDF uses a salt to improve the security of the provided secret. The salt is random data used to further diffuse the produced key, making two keys with the same secret but different salts, different [45]. The salt does not have to be secret and is sometimes stored with the produced cipher so that the decryption function can easily 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 text cannot be decrypted.

An alternative encryption solutions is Rivest-Shamir-Adleman (RSA). RSA 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 AES is a faster algorithm for encryption and decryption than RSA, while maintaining very good security.

2.5 Threats

To consider a filesystem secure it is important to imagine different potential adversaries who might attack the system. Considering that FFS has no real control of the data stored on the different services, all the data must be considered to be stored in an insecure system. Even if we could hide the posts made on 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 FFS, 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 FFS is not addressed in FFS but remains for future work.

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

The deniability of FFS is an important aspect of the filesystem. Potential threat adversaries are agents that the user is trying to hide the data from, such as governing states. For the system to be 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 FFS is unmounted there should be no trace of FFS 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 FFS has been mounted previously, but that they do not have access to the machine while FFS is mounted. It is assumed that the adversary might have snapshots of the user's computer before and after FFS was mounted, but that no snapshots were taken while FFS 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 FFS during the time inside the country. This requires FFS to not change any state of the operating system while it is mounted, such as information about when it was last mounted. Such state changes could be noticed using snapshots of the operating system state. During the development of FFS, no traces of previously mounted FFS instances have been found, although it has not been researched fully. Future work includes analyzing potential lingering modifications to the operating system state FFS and FUSE produce while mounted.

Chapter 3

RELATED WORK

The research area of creating filesystems to improve security, reliability, and deniability is not new and has been well worked on previously. This chapter presents previous work that is related to this thesis. This includes other filesystems that share similarities with the idea of FFS, for instance 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 FFS will not include malicious code in its images, this stegomalware problem has fostered the development of detection techniques of steganography in for instance social media, and it is well-researched.

Twitter has been exposed to allowing steganographic images that can easily contain any type of file [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[1]. The fact that the images are available publicly 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 Open Social Networks (OSNs) include the paper by Ning et al. where the authors build a system for private commu-

[1] `https://twitter.com/David3141593`

nication on public 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 explored the idea of undetectable communications over OSNs [52]. While they did not carry out an implementation, 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 OSNs. They presented a web-based user interface client with a PHP server backend as the method the users would use to create and share their secret messages.

A 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]. A rubber-hose filesystem is a filesystem where if an adversary is only given one key out of n keys in total, they cannot prove that more data exists and the filesystem is therefore deniable. This is known as a rubber-hose filesystem because of the idea behind rubber-hose cryptanalysis where an adversary beats the user with a rubber-hose to extract the encryption key. The adversary should have no way of knowing how many keys are used to encrypt the data. Steganographic methods can be used to hide the data for a deniable filesystem, and some deniable filesystems are also steganographic filesystems. However, deniability can be accomplished using other techniques than steganography. This thesis proposes a deniable filesystem that, while storing the data in images, is not steganographic as it does not hide the data in the images, but rather use the images as the storage medium.

3.2 Cryptography

Some work choose to invent their encryption methods rather than using established standards. Chuman, Sirichotedumrong, and Kiya proposed 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. Because of the well-proven security of AES, FFS will use AES as its encryption method.

3.3 Related filesystems

Many steganographic and deniable filesystems have been presented previously but many of these are focused on filesystems for physical storage disks to which the user has access. For instance, Peters et al. created DEFY, a deniable filesystem using a log-based structure in 2014 [55]. DEFY was built to be used exclusively on SSD found in mobile devices to provide a deniable filesystem that could be used on Android phones. Further examples of local disk-based filesystems can be found in [53, 56, 57, 1]. However, this thesis aims to create a filesystem that is not based on a physical disk but rather a cloud-based filesystem that uses images on 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 FUSE. They limit the filesystem such that each image posted will only store a maximum of 4 kB of steganographic file data and the images posted on the web services will be actual images. This is different from the idea of FFS where the images will be purely the encrypted file data and will therefore not be an image that represents anything but will instead look like random color noise. An implementation of CovertFS has been attempted by Sosa, Sutton, and Huang which also used Tor to further anonymize the users [58].

In 2016, Szczypiorski introduced the idea of StegHash - a way to hide steganographic data on OSNs 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 FFS where a table will be kept to keep track of which posts store a certain file, and in what order they should be concatenated, similar to the idea of an inode table. SocialStegDisc lacks actual implementation of the filesystem but similar to CovertFS presents the idea of a social media-based filesystem.

TweetFS is a filesystem created by Robert Winslow that stores the data on Twitter [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's 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. 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 FUSE and was presented well before the introduction of Google Drive in 2012. It does not support encryption as the plain file data is stored in emails. Today, Gmail and Google Drive share their storage quota and GmailFS has since become redundant as Google Drive is an easier filesystem to use. GMail Drive is another example of a Gmail-based

filesystem and it was influenced by GmailFS [64]. GMail Drive has been declared dead by its author since 2015.

Google Conduce Sistem de Fișiere (GCSF) is a filesystem that stores its data on Google Drive, built using FUSE [65, 66]. 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, GCSF interacts with the Google Drive REST API 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 GCSF 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. In contrast, 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, as this upload can occur at a later time. One difference between GCSF and the idea of FFS is that GCSF does not encrypt the data stored in the filesystem. While the data is, as mentioned previously, encrypted by Google Drive, the encryption keys are controlled by Google Drive, not the user of GCSF. The data stored on GCSF is also stored as its original files in Google Drive, not as images as FFS intends to store the data. The Google Drive filesystem architecture is utilized by GCSF, for instance by using its directory hierarchy structure. This allows GCSF to avoid creating its own inode table and directory structures, as Google Drive provides the functionality these structures similarly provide FFS, through the Google Drive API. The development of GCSF started in 2018 [66], and the repository in GitHub has around 2 300 starts as the time of writing this thesis.

Another Google Drive-based filesystem is google-drive-ocamlfuse [68], developed for Linux using FUSE. The project has been well received online. The repository has around 6 700 stars on GitHub at the time of writing this thesis and there are multiple articles online about the project [69, 70, 71]. The filesystem is well developed and

well maintained. The filesystem supports filesystem operations such as symbolic links, Unix ownership, and multiple account support. According to the author of GCSF, GCSF tends to be faster than google-drive-ocamlfuse for certain operations, including reading cached files [72, 73]. google-drive-ocamlfuse has no native support for 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. FFS aims to achieve stackability through the use of FUSE.

3.4 Filesystem benchmarking

IOzone is a filesystem benchmarking tool that is used to measure performance and analyze a filesystem [76]. It is built for, among other platforms, Apple's macOS where FFS 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. presented 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 In- and output (I/O) of the filesystem, such as bandwidth and latency, is different from benchmarking on-disk operations, such as the performance of file read and write operations. The benchmarking tools can for instance rarely affect or determine how the filesystem handles caching and pre-fetching. This means

that benchmarking the read and write performance of different filesystems can be misleading as they might handle this differently, meaning that the result could be different depending on for instance the distance between the files on the disk. Two files could be adjacent on the disk on one filesystem and therefore one could be pre-fetched into the cache when the other one is read. Considerations of such factors must be done 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 it is simpler to use [78] and still maintained. Due to these factors, IOZone was chosen as the benchmarking tool for FFS.

3.5 Summary

As presented in this chapter, different filesystems provide different features and drawbacks. Table 3.1 gives a summary of characteristics and features of some of the filesystems mentioned above and how they compare to FFS. As can be seen, FFS mainly lacks certain filesystem operations which are not the focus of FFS as FFS is a proof-of-concept deniable filesystem storing data on OWSs.

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

	ext4	**Google Drive**	**DEFY**	**TweetFS**	**FFS**
Mountable	X	X	X	-	X
Read/Write/Remove file	X	X	X	X	X
Read/Write/Remove directory	X	X	X	X	X
Hard links	X	-	X	-	-
Soft links	X	-	X	-	-
File and directory access control	X	X	-	X	-
Encrypted	X	X*	X	-	X
Deniable	-	-	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 FFS and the specifications of the development environment. We also present the benchmarking tools and methodology used to acquire quantitative data for the evaluation of the filesystem.

4.1 Development environment specification

Development of FFS 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 SSD running an encrypted APFS partition as the filesystem. Apple claims that the SSD 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, APIs, operating systems, and tools used by FFS. FFS was developed using C++ and compiled using Apple clang. FFS uses the ImageMagick Magick++ library [80] for image processing. macFUSE [19] is used by FFS to use the FUSE API. cURLpp [81] is a cURL [82] C++ wrapper used by FFS to make HTTP requests. libOauth [83] is used by FFS to sign and encode HTTP requests according to the OAuth [81] standard. Flickcurl [84] is a C library used by FFS to communicate with parts of the Flickr API. Crypto++ [85] is a C++ library providing cryptographic schemes. FFS uses Crypto++ to encrypt end decrypt the data stored in FFS, and to derive the keys used in the encryption and decryption algorithm.

FFS was developed for use on a single computer for simplicity, and the versions of the operating system, libraries, and tools were the most recent up-to-date versions when the development of the filesystem started (January 2022). To avoid re-writing the

source code to handle new API designs, these versions remained the same throughout the development process.

Table 4.1: The versions of the libraries, APIs, operating system, and tools used by FFS

Library, API, 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 API	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 degree project is the Fejk Filesystem (FFS). It uses an OWS to store the data and behaves 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 usable system. Additionally, when comparing FFS to distributed filesystems such as Google Drive, many of these other filesystems also do not support functionality such as links.

This section presents the implementation details of FFS. General overview of the filesystem and its internal structures are presented in Section 4.2.1. Following, the different caches of the filesystems are presented in Section 4.2.2. The encoding and encryption methods are presented in Section 4.2.3. Section 4.2.4 presents Flickr as the OWS used by FFS and why Twitter could not be used as the OWS. Following, Section 4.2.5 presents the implementation details of each FUSE filesystem operation implemented by FFS. Finally, Section 4.2.6 presents the limitations of the filesystem.

4.2.1 Design overview

FFS uses images to store the data of files, directories, and the inode table of the filesystem. These images are uploaded to an OWS, such as Flickr, as image posts. As mentioned in Section 2.3, there can be limitations on the size of these posts for certain OWSs. To support files larger than these limitations, these files will be split into multiple posts, requiring FFS to keep track of a list of posts. Figure 4.1 presents the basic outline of FFS and an example content of the filesystem. FFS 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 FFS instead keeps track of the IDs of the posts (i.e. post ID) on the OWS where the file or directory is located. The inode table entry for each file or directory will also contain metadata about the entry, such as its size and a boolean indicating if the entry is a directory or not.

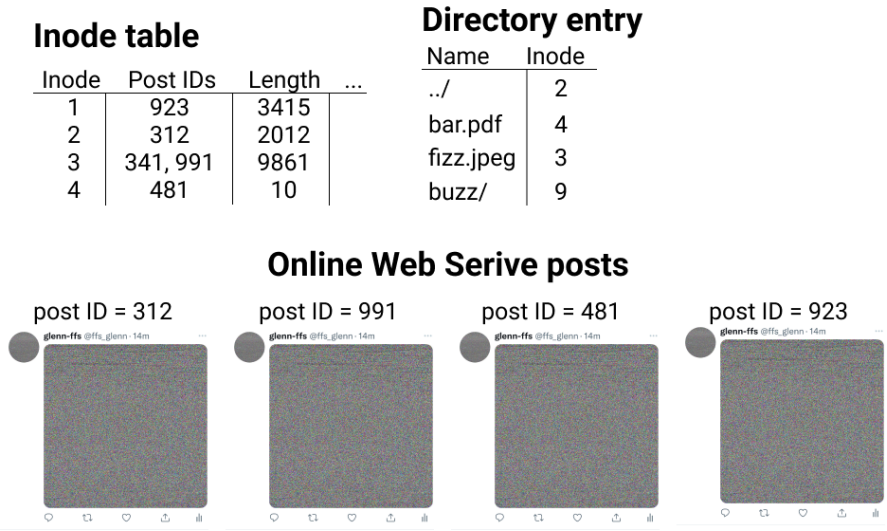

Figure 4.1: Basic structure of FFS inode-based structure

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 for 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 OWS, making it easy to find it on the OWS. The other images stored on the OWS can be images stored in FFS, or images posted by the user that does not contain FFS data as long as the most recent image saved on the OWS is the inode table. The inode table will keep track of which images are stored in the filesystem, and ignore all the other files.

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 OWS. The data received by downloading these images, decoding them (as described in Section 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 execution time as every request to the OWS takes time. However, if the filename is changed or the file or directory is moved to another location, the parent directory of the file or directory would have to be updated, 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. The inode is represented as an unsigned 32-bit integer (as shown in Appendix A). 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 those 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 unsigned integer, FFS would need to have saved more than four billion files and directories before the inode value would overflow. This scenario is outside the scope of this proof-of-concept filesystem.

FFS does not support all filesystem operations that are implementable through FUSE, instead, FFS implements the subset of them 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 FFS and their implementation details are described in Section 4.2.5.

Table 4.2: Filesystem operations implementable through the FUSE API, and whether or not FFS 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 OWS when it is modified to save its current information. As it takes time to make requests to the OWS, FFS is designed to make as few requests as possible while still saving the required data. Therefore, only the directory or file that is affected by a change is uploaded to the system, while those files and directories unaffected can remain the same. The inode table has to be updated with every change of a file or directory as the inode table contains the location of the file or directory.

FFS can be mounted to the local filesystem using FUSE, similar to mounting a network drive or a File Transfer Protocol (FTP) server. The mounted FFS 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

FFS implements a simple in-memory Least Recently Used (LRU) 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 FFS using too much memory, the cache is configured so that images greater than 5 MB are not cached. As a result the cache can store a maximum of $20 * 5\,\text{MB} = 100\,\text{MB}$ of encrypted image data. 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 First In First Out (FIFO) 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 OWS, the cache is checked to see if the cache is currently storing this post ID. If so, the stored image is returned. Otherwise, the process continues by downloading the image from the OWS 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 OWS.

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

FFS separately also caches the inode of open files and the inode of the open file's parent directory. The open file's data is also cached in memory if it has 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 in Section 4.2.5. When a file is opened it is associated with a file handle identifier that 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 number of `read` or `write` operations required to read or write data depends on the amount of data to read or write, and the buffer size 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 [23]. To save computation time by not having to download the file from the OWS, or even decrypt the image data found in FFS's regular cache, FFS stores the file data separately in memory. When a file with a file handle is modified or read, FFS checks if the file handle has any cached data associated with it, before 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 OWS. FFS does not 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. However, such limits can be imposed by the operating system. If a file

in FFS 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. Implications of this are discussed in Section 4.2.6.

Furthermore, an additional filesystem cache is provided by the operating system kernel [23, 24]. FFS can not control the kernel cache other than disabling the cache the filesystem is mounted. FFS does not disable the UBC when FFS is mounted but the UBC can be disabled by the user mounting FFS using FUSE command line arguments.

4.2.3 Encoding and decoding objects

Entities that FFS stores on the OWS, and therefore also encodes and decodes, are: files, `Directory` objects, and the `Inode Table` object. All of these entities are stored on the OWS using PNG images with 16-bit Red Green Blue (RGB) color depth, without an alpha channel. 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 FFS 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 to encode as an image. A header (FFS header) is prepended to the binary data, containing among other things, the size of the data and a timestamp of when the data was encoded. The FFS header and the input data are encrypted using authenticated encryption, utilizing GCM and AES. The key used for the encryption is derived using the HKDF function utilizing the SHA-256 hashing algorithm, along with a 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 HKDF is a password provided by the user. HKDF also uses a random IV, re-generated every time new data is encrypted. The length of the IV is set to 12 bytes. The resulting data from the encryption is the salt, the IV, 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 Complete Encrypted Data (CED).

The dimensions of an FFS image is based on the number of bytes stored, as described in Section 2.1.3. The stored data is the CED, prepended with the Length of the CED (the Length of Complete Encrypted Data (LCED)) using four bytes. For an image of $X = \lceil \frac{4+LCED}{6} \rceil$[1] pixels, FFS will set the width w of the image as $w = \lceil \sqrt{X} \rceil$. Further, the height h of the image is set as $h = \lceil \frac{X}{w} \rceil$. This will require $(w * h) - X$ filler bytes and will create an image with a 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 LCED,
- the CED data, and
- filler bytes.

The content of the filler bytes are randomized.

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

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

[1] $\lceil x \rceil$ is the `ceil` function, rounding x up to the closest integer

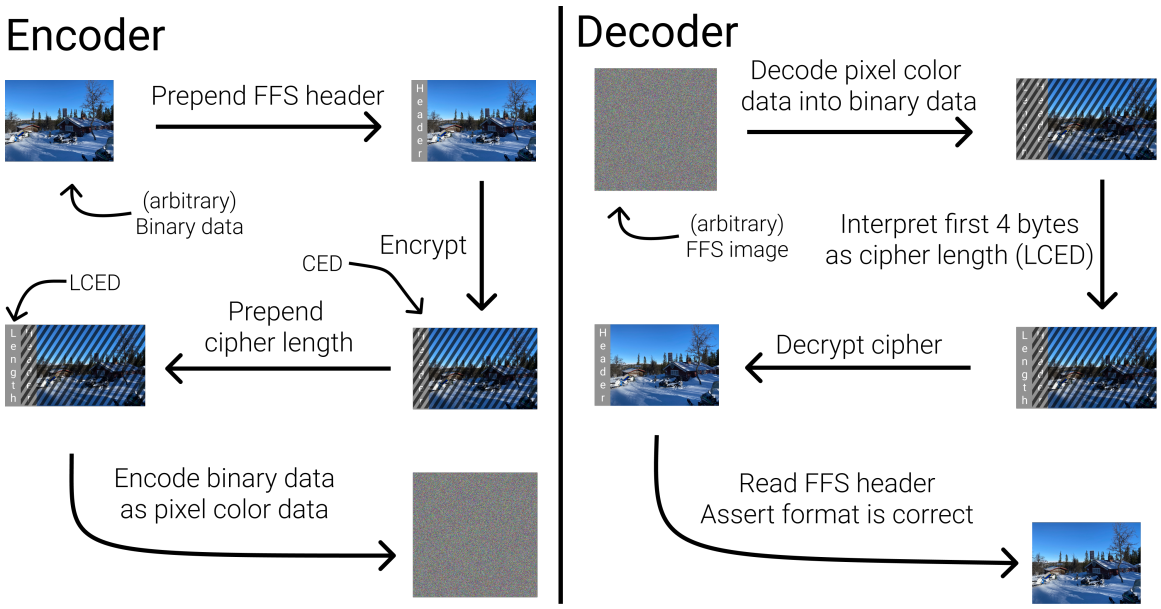

Figure 4.2: Simple visualization of the encoder and decoder of FFS. The input of the encoder is the binary data to store in FFS, eg. a file, and the output is the FFS image to upload to the OWS. The input to the decoder is an FFS image, and the output is the binary data stored on FFS, eg. a file

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 FFS [87].

An FFS image has an upper size limit, defined by the OWS used. If the data to be stored in FFS, 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 IVs. Only the inode table stores the different post IDs in the order they were encoded. While files and directories stored in FFS can be separated into multiple images, the inode table is limited to only one image for simplicity when interacting with the OWS. This introduces a size limit for the inode table, limiting the filesystem. More details about the limits of FFS are found in Section 4.2.6.

4.2.4 Online web services

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

The initial intention of FFS was to use Twitter as the OWS. 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 FFS images rely on a specific binary representation within 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 [88] and even suggested workarounds [89] for users who are concerned about the potential data loss. However, during research for the thesis, it was concluded that the workarounds mentioned in [89] no longer work on Twitter. For instance, some PNG images less than 900x900px that have been uploaded to Twitter, could not be downloaded as the same image, which contradicts the workaround mentioned in [89] 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 FFS-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 have not indicated that this would change. Therefore, Flickr can be used at this moment for FFS. A free-tier Flickr account is therefore used for FFS. Restrictions can be set on the Flickr account to enable or disable unauthenticated Flickr visitors to download the originally uploaded image. The advantage or enabling unauthenticated

Flickr visitors to download the images is that we can read the content of FFS even if we do not authenticate with Flickr, and the data can be shared with others using only the password used for encrypting the data in FFS. The disadvantage is that adversaries can also download these images and potentially gain information about the data stored in FFS even if they do not have the encryption password for FFS. However, even if restricting downloading of the original images, adversaries can still access the original images, for instance if Flickr provides them. Therefore, the Flickr account is set to allow unauthenticated visitors of Flickr to download the original versions of the images.

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

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 FFS, 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 FFS to upload all other images first, for instance, the newly written file. This provides FFS 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 API is extensive in its functionality, FFS only uses a few of the provided capabilities; specifically FFS 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 FFS, The Fejk Fejk Filesystem (FFFS), has also been developed. FFFS uses a Fake Online Web Service (FOWS), which stores the data on the local APFS filesystem. The FOWS is used by FFFS just as Flickr is used by FFS, 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 network requests. This allows us to analyze the theoretical performance limit of FFS, and how it would perform if the OWS used had very low latency and the network connection to the OWS had very high bandwidth and low delay. By analyzing FFFS, 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 FFS filesystem operation can be modeled like:

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

where t_{ffs} is the time that FFS 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, 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 OWS 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 API for the URL pointing to the most recently uploaded image, and,
- to download the images representing the file to read.

Depending on the OWS, the latency and bandwidth of the internet connection between the user's machine and the OWS's server can differ a lot. Duplicate requests to the same OWS can also differ significantly due to, for instance, server load balancing and a difference in the 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 FFS cache. However, for a FOWS, t_{ows} can be replaced by t_{fows} which will have approximately consistent values for duplicate operations, because the local filesystem is not affected by the network connection or the current traffic by other users of the OWS. 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 FOWS can be handled quickly by the operating system. However, t_{fows} is affected by, among other things, the underlying storage device of the local filesystem, process scheduling, and FFS 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, FFS 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 FFS 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 [90, 91, 92]. Although, these methods require you to decrypt the APFS 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 FFS image has been present in the `/tmp` directory, they could conclude that FFS 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 FUSE operations implemented by FFS, and how they are implemented by FFS. Further explanations about the intended functionality of the operations can be found in Kuenning's report [86].

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

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

All filesystem operations are synchronous unless specified. Further, FUSE is running in single-thread mode meaning that a filesystem operation call must complete before another can begin. This helps 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 OWS, only the parent directories are downloaded during the path traversing as explained above. An `open` call must, eventually, be followed by a `release` call. Although, multiple other operation calls can occur between these events.

4.2.5.2 create

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

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

4.2.5.4 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 OWS, only the parent directories are downloaded during the path traversing as explained above. An `opendir` call must, eventually, be followed by a `releasedir` call. Although, multiple other operation calls can occur between these events.

4.2.5.5 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 OWS, and the previous versions of the directory and inode table are deleted from the OWS. 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 OWS as it has no data associated with it. The parent directory is modified and updated in the OWS, and the old versions of the parent directory are deleted in the OWS. The parent directory entry in the inode table is modified with the new posts, and a new entry is created for the new directory. The inode table is updated in the OWS, and the old version of the table is removed from the OWS.

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

4.2.5.7 read

This operation reads s bytes into a data buffer a, starting at the provided offset o, from an open file with the provided file handle. If the file has not been read or written to since it was opened, the full file is downloaded and read into memory, even if just a small part of the file is requested. The file data is also cached and the cached data is associated with its file handle so that subsequential requests for the same file while it is open are faster. If the file has been read or written to since it was opened, the file data is accessed from the cache using the file handle.

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 buffer a, starting at the provided offset o, to the existing and open file with 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 OWS, this occurs instead in the subsequent `release` call. However, the data is associated with the file handle so that subsequent filesystem calls use this new file data.

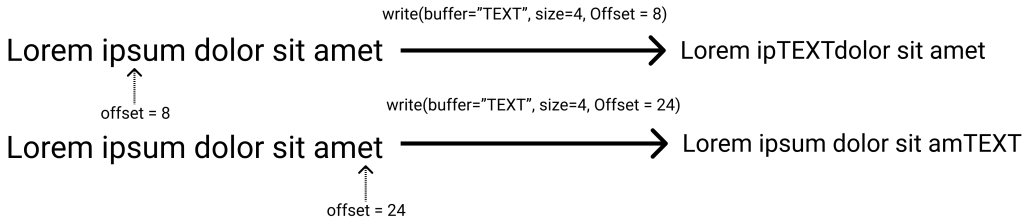

Figure 4.3: Visualization of how the write operation handles different offsets.

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

table is updated to the OWS and the old table is removed from the OWS. The old posts associated with the old parent directory and the new parent directory are removed from the OWS.

The new path could be in the same directory as the file or directory currently is in. This will not affect the process mentioned above; however, the path will only have to be traversed once, and the parent directory will only be removed and updated once. If the new path is the path of an existing file, the existing file is removed from the filesystem and the OWS.

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 OWS, and the old data is removed from the OWS. The inode table entry is updated with the new posts and uploaded to the OWS. The old inode table is removed from the OWS.

4.2.5.12 ftruncate

This operation is similar to `truncate`, but is called from a user context which means it has a file handle associated with it. The operation truncates or extends the file in the given file handle, to the provided integer s. The full current file is read into memory, either from the OWS 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 OWS, this occurs instead in the subsequent `release` call. However, the data is associated with the file handle so that subsequent filesystem calls use 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 OWS. The old parent directory data is removed on the OWS. The removed file's entry in the inode table is also removed, and the inode table updates the entry for the parent directory with its new posts. The inode table is then updated on the OWS and the old inode table is removed on the OWS. Finally, the data of the removed file is removed from the OWS. The last step is not necessary for a working filesystem; however, to save space on the OWS, this is done. If the OWS permits unlimited images and total storage, this step could be omitted to save time.

4.2.5.14 rmdir

Similar to `unlink`, this operation removes the directory at the given 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 OWS, and the inode table is updated with the new posts of the parent directory. Following this, the inode table is updated to the OWS. The old parent directory and the old inode table are removed from the OWS.

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

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, FFS 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 FFS. The file or directory pointed to by the path does not need to be downloaded, all the metadata that FFS stores is accessible through the inode entry in the inode table, 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 FFS. This includes, among other things, the maximum filename size and the filesystem ID. The operation has a short computation time as it does not have to download or upload any files. The only variable information is read from the inode table that is stored in memory and thus does not have to be downloaded from the OWS.

4.2.5.18 access

This operation, given a path, returns 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 OWS, and the old version is removed from the OWS.

4.2.6 FFS limitations

FFS has numerous limitations due to both implementation decisions and OWS limits. As Flickr allows a free-tier user account to store up to 1 000 images of up to 200 MB per image, this allows storage of up to 200 GB of images 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. Each image includes, among other things, a PNG header, other PNG attributes, and the CED which in total is of greater size than the unencrypted data. To ensure that the PCD along with the PNG header and other PNG attributes do not exceed the limit of 200 MB, FFS limits the PCD size to allow at least 10 MB for the PNG header and other PNG attributes, meaning that the PCD can be a maximum of 190 MB. The cryptographic variables

IV, salt, and the authentication tag are stored in the CED using 12, 16, and 64 bytes respectively, for a total of 92 bytes. The size limit means that these 92 bytes, along with the encrypted cipher text, cannot exceed 190 MB, meaning that the encrypted cipher text cannot exceed $190\,000\,000 - 92 = 189\,999\,908$ bytes. However, as AES 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 FFS allows is therefore $\lfloor \frac{189\,999\,906}{16} \rfloor * 16 = 189\,999\,904$[1] bytes. Due to plain text padding, the unencrypted plain text can be a maximum of one byte less than this value [93], 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 FFS 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 FFS 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 represent the root directory.

The inode table keeps the information about empty files and directories even though they store no data on the OWS. The inode of a file or directory is an unsigned 32-bit integer, meaning that the inode table could theoretically store 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 OWS, the inode table is limited to a maximum size of 189 MB. Further, the size of the inode table is 4 B plus the size of each entry, and one of these entries is the root directory. Even if a file is empty, it is still stored with its filename and inode in its parent directory. A non-empty file or directory in the inode

[1] $\lfloor x \rfloor$ is the `floor` function, rounding x down to the closest integer

table requires 37 B plus approximately (depending on the post ID length generated by the OWS) 12 more bytes per post ID representing the file or directory on the OWS. 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:

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

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 FFS could include multiple user accounts and multiple services. This could increase the limits on the filesystem at the cost of increasing the information stored in the inode table as it would need to keep information about the OWS of each post ID.

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 is 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 empty 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. Assuming each file has the same three character file extension (such as `pdf` or `png`), it requires four extra bytes per file name, including the `"."` separator. 5 108 106 directory entries of four-letter filenames, four-byte file extension, one NULL character, and four-byte inodes requires $5\,108\,106 * (4 + 4 + 4 + 1) = 66\,405\,378$ bytes ≈ 66.5 MB. Therefore, the directories do not need more than one image to describe all the possible files with four character filenames it can contain before FFS runs out of storage; however, if the filenames are longer than four characters, or if the file extensions are longer than three 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 is 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 $10 + 90 + 12 * 998 = 12\,076\,\text{B}$. As each image can use up to 189 MB, the maximum file size is $998 * 189 * 10^6 = 188.622\,\text{GB}$. $188.622\,\text{GB} + 12\,076\,\text{B} < 188.811\,\text{GB}$ meaning a file of 188.622 GB is allowed in the filesystem.

Limits to the file sizes also depend on the machine where FFS 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 FFS 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 of 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, if the computer has less physical memory available, more memory space can often be provided through swap on the hard disk. Apple ensures that the swapped data is securely encrypted on the hard disk [94]. However, using a swap space puts a constraint on the available storage of the hard disk used by the underlying filesystem as the available storage must be sufficient to store this data. Further, as FFS 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 FFS. If the local filesystem has no available storage, only a few filesystem operations can be performed on FFS 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.

FFS stores data associated with file handles of modified and open files in memory. The number of open files in FFS is unbounded by FFS, but can be limited by the kernel of the operating system. The kernel of the operating system where FFS is run

for this degree project has a limit of 61 440 open files per process and this value can be configured [95]. The amount of data FFS can store per file handle is unbounded by FFS, and FUSE allows big file sizes as mentioned in Section 2.2. 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 quickly filled. The file handle associated with an open file is the inode of the file, meaning that there is no risk of 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; this, `open` calls on an open file will be ignored. If an edited file is not closed before FFS is shut down, the new file data is not updated on the OWS and the new file data is lost. When FFS is launched again, the data saved by the most recent close operation on the file will be used as this is the data saved on the OWS.

Another limitation of FFS is the rate limit imposed by the Flickr API. Flickr allows up to 3 600 API requests per hour, after which the API 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 API calls, the API keys could be revoked. Furthermore, some requests are sent concurrently to Flickr which means that FFS could reach 3 600 API calls faster. Reading a file, represented by one image that 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 a maximum of one request per second, we can read 30 un-cached files per minute. If the file is in a directory that 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 requests are required per extra image. Furthermore, the bandwidth of the internet connection to the OWS 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 milliseconds for the same bandwidth. Constant read operations (including preceding open- and subsequent close operations) on a file just over 5 MB could therefore exceed the API request limit with a 100 Mbit/s bandwidth.

A limitation of FFS that impossible to overcome is the bandwidth and latency of the network connection between the user and Flickr. The bandwidth and latency of this 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 image would take 8 s, and a 200 MB file would take more than 5 min. However, the API request limit would be a limitation when reading smaller files; for instance, any file of 625 kB or less takes 1 s or less to download given this same bandwidth.

4.3 Benchmarking

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

4.3.1 Filesystems

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

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

The encrypted APFS filesystem was used as a reference for a local filesystem without an internet connection. An encrypted APFS is the local filesystem of the development environment for FFS, it was selected as an example of a modern, well-used, and fast filesystem, and provide a baseline benchmark to which FFS and the other filesystems compared.

GCSF was selected to compare FFS against another network-based filesystem. While GCSF is not a deniable filesystem, it is a filesystem that stores its data on an OWS, namely Google Drive. The reason GCSF was used instead of, for instance, the official Google Drive mountable filesystem volume provided by the Google Drive Desktop application, is that GCSF provides instant upload of the files and directories to Google Drive using the Google Drive REST API. The instant upload provided by GCSF enables us to easily measure the duration of a file operation. For instance, a write operation on a file in GCSF will not complete before the new file data has been

completely stored on Google Drive. Another reason why GCSF was chosen is 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 FFS and thus no longer work as expected, for instance, due to changes in the API, or because the OWS manages the uploaded data differently than previously.

The instance of FFFS using a FOWS of an encrypted APFS was chosen to be compared to FFS so that the duration of the FUSE filesystem operations could be further analyzed. As the filesystem operations of FFFS are similar to the ones of FFS, other than the network request being replaced by local filesystem operations, it is possible to analyze the effect of the OWS latency, the OWS internet connection bandwidth and latency, and the OWS data processing speed has on the filesystem performance. Comparing the benchmark results of FFFS and APFS allows us to analyze the FFS overhead as FFFS is dependent on the performance of APFS. Especially for file operations where FFFS must interact with the storage medium, for instance, write operations and read operations for files not in the cache, FFFS cannot outperform APFS as FFFS requires the execution time of the underlying APFS file operation as well as the internal FFS computation time. Both FFFS and FFS were mounted with a maximum buffer size of 32 MB.

4.3.2 IOZone

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 and 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 checks 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 checks 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 of a filesystem. However, certain cases of the functionality of both FFS and GCSF were tested in order to see that they support directory hierarchies and multiple files as expected. Future work includes analyzing the functionality of FFS and GCSF. APFS is expected to have full functionality as it is a professionally developed and widely used filesystem.

While IOZone supports multiple different file operation tests, this thesis only used a subset of these for benchmarking. Among other reasons, certain tests failed when ran on GCSF. 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 FFS 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 the 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 any of the caches. 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 FFS has to download and read the entire file into memory before accessing a specific offset, FFS 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 FFS 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 GCSF[1], 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. The memory of the computer where the benchmarking is run is 16 GB which is a greater value than reasonable for testing FFS and GCSF due to the execution time of the tests. Each doubled file size takes exponentially more time as both the file size and buffer size are doubled for each test. Furthermore, it was found that GCSF will crash for large file sizes (as occurred for file sizes equal to or greater than 262 144 kB). The maximum file size for the benchmarking tests was set as 262 144 kB as this is bigger than the biggest image possible to store on Flickr, requiring FFS to split the

[1] Size limit of the cache entries of GCSF cannot be configured. However, the number of cache entries and the time-to-live per cache entry can be configured

image into more than one image on the OWS. This helped us test the functionality of FFS further. The file sizes used for the IOZone tests were 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 were:

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 actual maximum buffer size for each file size is limited to 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 does not make sense to run tests with a buffer size greater than 4096 kB for a file size of 4096 kB.

While GCSF does not define a maximum file size for its cache, one can configure the time-to-live of the cache entries, and the number of cache entries allowed in the cache [97]. The GCSF cache is stored in-memory so the cache will be cleared if the GCSF process is terminated. During benchmarking, The GCSF cache was configured to store up to 20 cache entries for up to 300 seconds per cache entry. The cache entry limit is the same limit as the cache entry limit of FFS. 300 seconds time-to-live limit was set for the entries is it was the default configuration for the time-to-live.

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. When conducting the benchmarking for this thesis, the file is stored in the root directory of FFS, FFFS, and GCSF. This means the path traversal is as short as possible for these filesystems. Furthermore, as FFS and FFFS constantly cache the root directory and both have 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 the file is small enough. The cache entry of the benchmarking file will not be removed unless the file is removed as the cache will not be filled by other entries. The file used for benchmarking APFS was in temporary directory in the root directory of the computer (`/tmp`). The reason is because the root directory of the APFS on the computer was a Read-only filesystem. Furthermore, FFFS saves its files in the same temporary directory, so the performance of saving the images for FFFS and the performance of the APFS write operation can be compared. Future work could compare benchmarking FFS, FFFS, GCSF, and APFS in different depths of directory hierarchies to analyze how the depth of a file impact 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 FFS, and potentially other filesystems, save the data to the storage medium only after the device is closed. In the case of FFS, 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 of each file operation with performance data for each file size and each buffer size used in the benchmark. The performance of the filesystem is measured in kilobytes per second. For this thesis, each filesystem is benchmarked 20 times with the kernel cache (UBC) enabled and 20 times with the kernel cache disabled. By comparing the benchmarks were the kernel cache is enabled and disabled, we expect to see better performance from the benchmarks were the filesystem has the kernel cache enabled. During testing, before running the experiments, it was found that one IOZone benchmark takes around three to five hours for FFS and GCSF. Therefore, a much large number of tests per filesystem would take very long time to complete. When benchmarking FFS, GCSF, and FFFS, the processes running the filesystems were terminated and the filesystems were unmounted and re-mounted between each iteration to clear their internal caches.

The benchmarking tests the filesystems were carried out in Amsterdam in The Netherlands using an ethernet connection to a fiber-connected router in a resedental home

with a bandwidth subscription of 100 Mbit download speed and 20 Mbit upload speed. The WiFi of the computer was disabled during benchmarking so all network traffic was sent through the ethernet interface the ethernet cable was connected to. The filesystem benchmarks were conducted during all hours of the day, during both week days and weekends. Other devices connected to the same network were not using a significant amount of internet bandwidth during this time to avoid affecting the available bandwidth for the computer running the tests. The computer on which the filesystem benchmarks were run was running as few processes as possible during the benchmarks to avoid other processes competing for resources of the computer, such as internet bandwidth and memory. For instance, the web browser was never running during any of the benchmarks. These processes were manually killed before the benchmarks started.

During the benchmarking of FFS and GCSF, the bandwidth used by the computer was measured using Wireshark [98]. Wireshark was run in the command line instead of in the graphical user interface to avoid Wireshark using more resources of the computer than necessary. The bandwidth analyzer was configured to capture all outgoing and incoming packets using TCP on port 80 or 443 (at either the source or the destination) over the ethernet interface the ethernet cable is connected to, during the benchmarking process. Port 80 and 443 are normally used for HTTP and HTTPS requests, respectively, which is how both FFS and GCSF communicate with their storage services. However, Wireshark captured all packets sent and received by the computer, not only the packets related to the filesystems. When researching tools that could measure the bandwidth over time, per application, no such tools were found that could be run on macOS. While the computer were running as few background processes as possible, some processes were found to sometimes start again after they have been killed by the user. This means that the result from Wireshark could include packets associated with other processes that are sending and receiving HTTP packets. In future work, it could be valuable to find a way to isolate the bandwidth measurement to only capture packets associated with the process. Wireshark reported the number of packets and bytes per ten-second interval to a file after the benchmarking was completed.

Wireshark consumes a lot of memory to capture all packets sent and received by the computer, even when the filtering is applied. Due to the long benchmarking execution time of FFS and GCSF, Wireshark was configured to save its measurements in temporary files during the benchmarking so the computer would not exceed its memory limit. Each temporary file was configured to be a maximum of 1 GB before a new temporary file was used. These files were summarized when the Wireshark measurement was ended. To avoid Wireshark competing for filesystems operations on APFS, these temporary files were saved on an external hard drive.

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 FFS, which uses Flickr as its OWS. The source code of FFS can be found on GitHub at github.com/GlennOlsson/FFS [99]. The source code consists of 4 184 lines of code excluding comments. The filesystem provides free cloud-based cryptographic and deniable storage as a mountable volume for a computer running macOS. The filesystem requires the user to provide their Flickr API keys and an encryption password. The API keys are used to authenticate with the Flickr API, and the password is used to derive the encryption and decryption keys. These values are passed to FFS as environment variables.

5.2 Benchmarking

This section presents the result from the IOZone benchmarking tests run on each filesystem. The benchmarking completed successfully for all file sizes for FFFS, and APFS. For the biggest file size, 262 144 kB, GCSF crashed multiple times, but it succeeded for the other file sizes. FFS crashed multiple times for the 262 144 kB file size when the UBC was disabled, but completed successfully for the file size when the UBC was enabled. It was found during debugging that FFS could not close a file of 200 MB without FFS crashing, seemingly due to a FUSE error. The reason behind these crashes was not found. Therefore, the results for the biggest file size was omitted from the tests of FFS with the UBC disabled, and for GCSF for both

states of the UBC. In total, each of the six filesystems were benchmarked 20 times with the UBC enabled and 20 times with the UBC disabled. Nine file sizes were used (eight file sizes for FFS with the UBC disabled and for GCSF) and up to 13 buffer sizes per file size. Each test per filesystem produced a table with nine rows (eight rows for FFS with the UBC disabled and GCSF) and 13 columns, where each cell is the average performance of the 20 tests with the specific file size and buffer size on the filesystem. The tables for FFFS and APFS with both states of the UBC and the table for FFS with the UBC enabled have 107 cells, while the table for FFS with the UBC disabled and the tables for GCSF with both states of the UBC have 94 cells. See Appendix C starting on page 120 for the tables.

The bandwidth used by the computer during the benchmarks was measured by Wireshark by capturing the number of bytes sent per ten second interval. Table 5.1 presents the bandwidth in kilobits per second recorded by Wireshark during the benchmarking of FFS and GCSF, with the UBC enabled and disabled. Each data point is the incoming and outgoing bandwidth per ten second window. For instance, it can be seen that the minimum bandwidth of GCSF is 0.00 kB/s, meaning that no data was sent or received over the internet on the network interface of the computer during at least one ten second window. It can be observed that the maximum bandwidth of either filesystem does not exceed 26 Mbit/s. 26 Mbit/s is higher than the upload bandwidth of the internet subscription (20 Mbit/s) but significantly lower than the download bandwidth of the subscription (100 Mbit/s).

Table 5.1: Network bandwidth during the benchmarks of the cloud-based filesystems

	Bandwidth (kbit/s)			
Filesystem	Minimum	Average	Median	Maximum
FFS, UBC Enabled	0.28	636.26	241.02	12214.97
FFS, UBC Disabled	0.11	310.61	185.28	17720.17
GCSF, UBC Enabled	0.00	590.87	471.35	15781.85
GCSF, UBC Disabled	0.00	547.30	477.68	25538.91

Combining all the data points from one benchmark test for a filesystem, over all 20 iterations, we get the overall performance of the test on the 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 FFS and GCSF with the UBC enabled are in general worse than the performance for FFFS and APFS with the UBC enabled. With the UBC disabled, the median performance of GCSF is higher than the median value of APFS, and the median performance of FFS and FFFS are significantly worse. APFS has a high spread of values when the UBC is disabled. All filesystems perform significantly better with the UBC enabled.

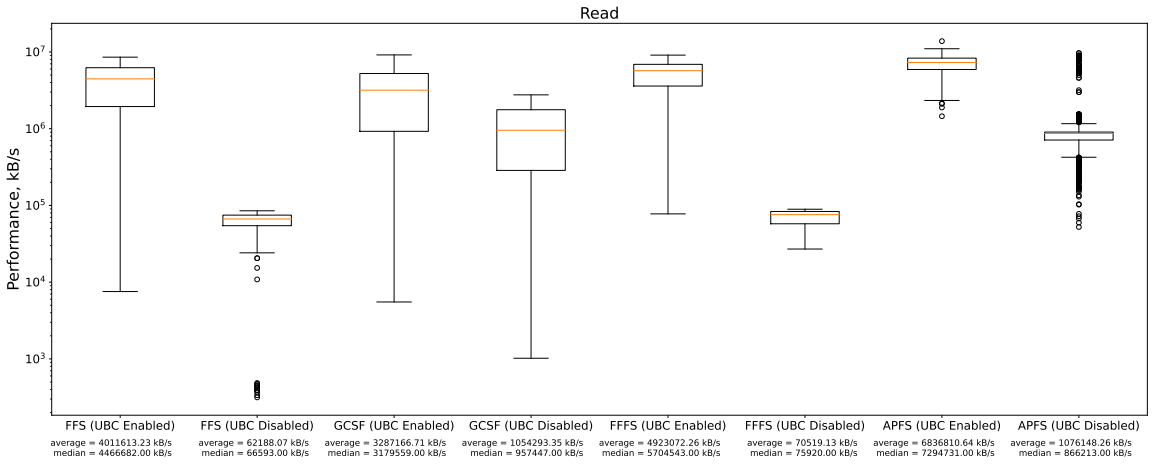

Figure 5.1: Box plot of the IOZone output for the Read test on the different filesystems

Figure 5.2 presents a box plot of the benchmarking results of the filesystems of the Write test. APFS has the best write performance of the four filesystems, both when the UBC is enabled and when it is disabled. FFFS performs better than the cloud-based filesystems for both states of the UBC. FFFS and GCSF have similar performance when the UBC is enabled and when it is disabled, while FFS and APFS perform significantly better with the UBC enabled.

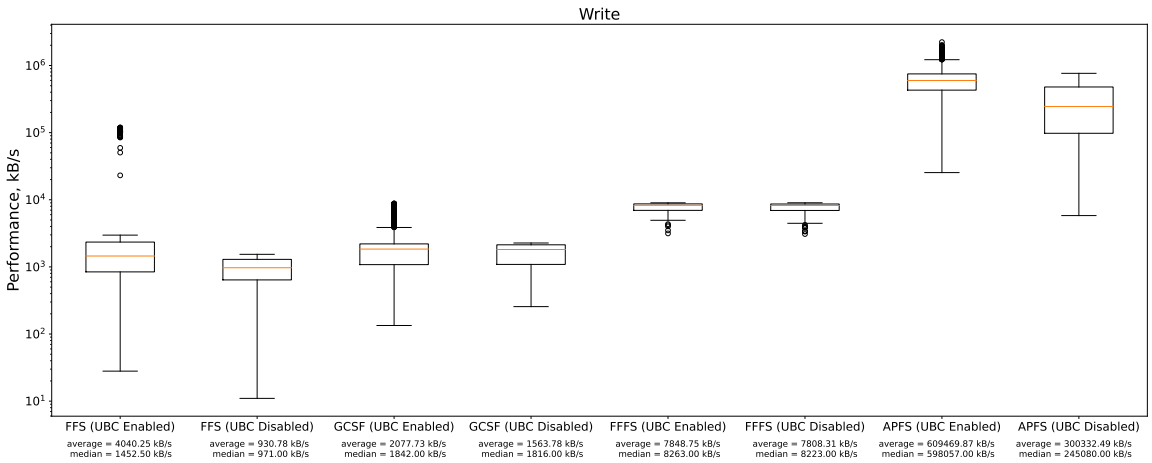

Figure 5.2: Box plot of the IOZone output for the Write test on the different filesystems

Figure 5.2 presents the result of the Re-Read test for the filesystems. FFS, GCSF, and FFFS perform significantly better with the UBC enabled compared to when it is disabled. APFS also performs better with the UBC enabled, but the performance difference compared to when the UBC is disabled is not as large as for the other filesystems. When the UBC is disabled, GCSF performs better than FFS and FFFS. Although, when the UBC is disabled, GCSF has a greater spread of values with its worst performance being lower than the worst performance of FFS and FFFS when the UBC is disabled.

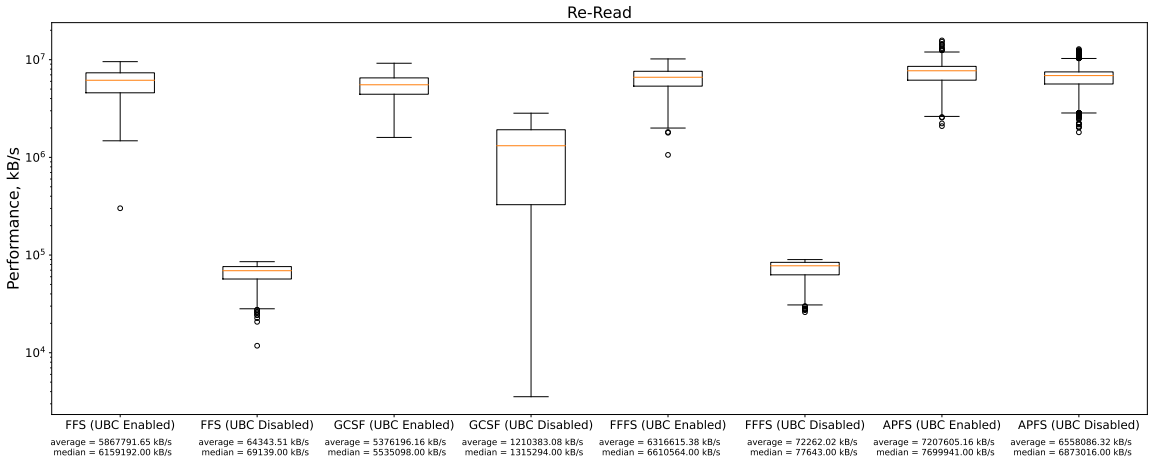

Figure 5.3: Box plot of the IOZone output for the Re-Read test on the different filesystems

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, APFS has the best performance of the filesystems, both when the UBC is enabled and when it is disabled. FFFS performs better than the cloud-based filesystems for both states of the UBC. FFS has slightly better average performance compared to GCSF when the UBC is enabled, but worse median performance. GCSF has better average and median performance than FFS when the UBC is disabled.

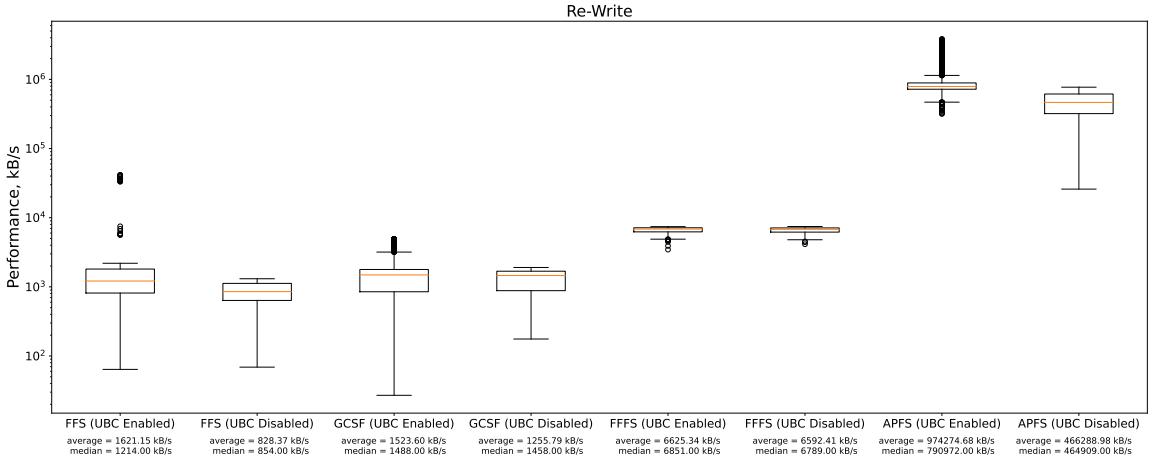

Figure 5.4: Box plot of the IOZone output for the Re-Write test on the different filesystems

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.

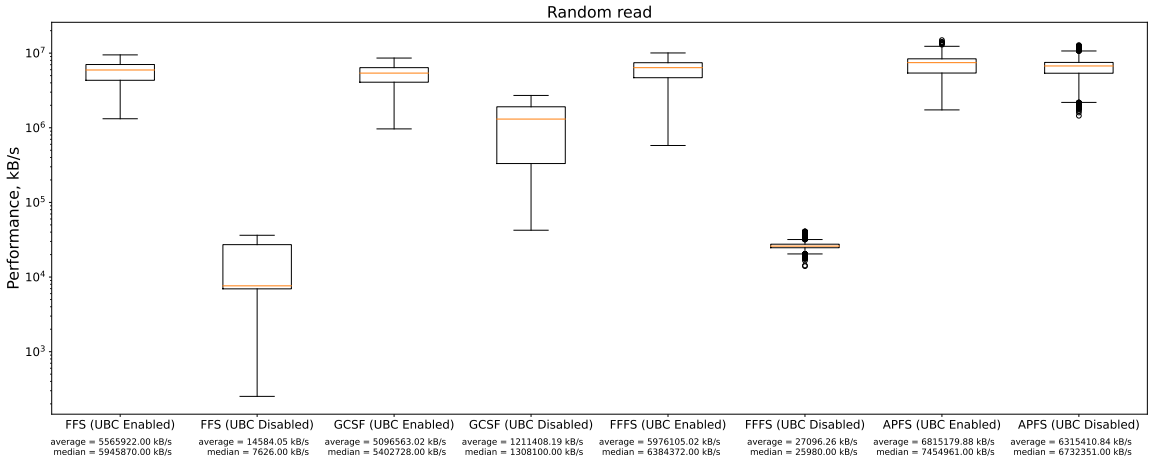

Figure 5.5: Box plot of the IOZone output for the Random read test on the different filesystems

Figure 5.2 presents a box plot for the Random read test for the different filesystems.The results are similar to the results for the Re-Write test presented in Figure 5.2.

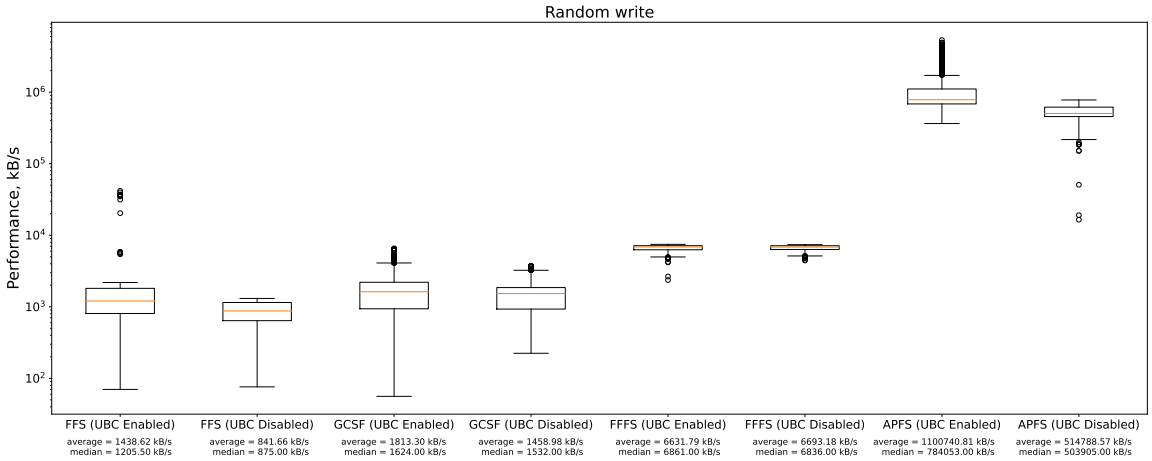

Figure 5.6: Box plot of the IOZone output for the Random write test on the different filesystems

Following are histograms for each filesystem and each state of the UBC. Each histogram presents the performance distribution for each file size in each file operation test. Figure 5.2 and Figure 5.2 presents the performance of FFS with the UBC enabled and disabled, respectively. Figure 5.2 and Figure 5.2 presents the performance of GCSF with the UBC enabled and disabled, respectively. Figure 5.2 and Figure 5.2

presents the performance of FFFS with the UBC enabled and disabled, respectively. Figure 5.2 and Figure 5.2 presents the performance of APFS with the UBC enabled and disabled, respectively.

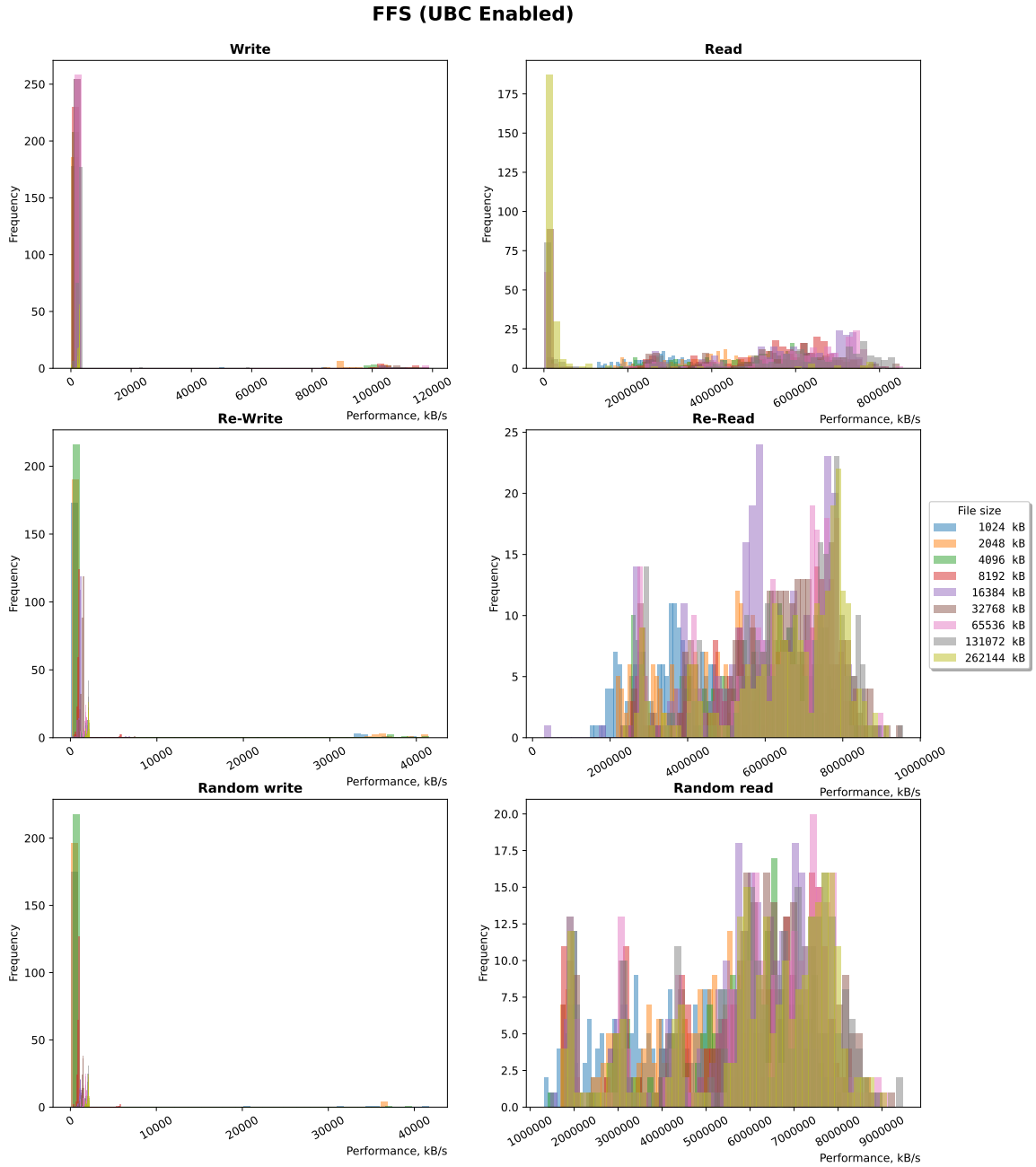

Figure 5.7: Performance comparison of different file sizes for FFS with the UBC enabled

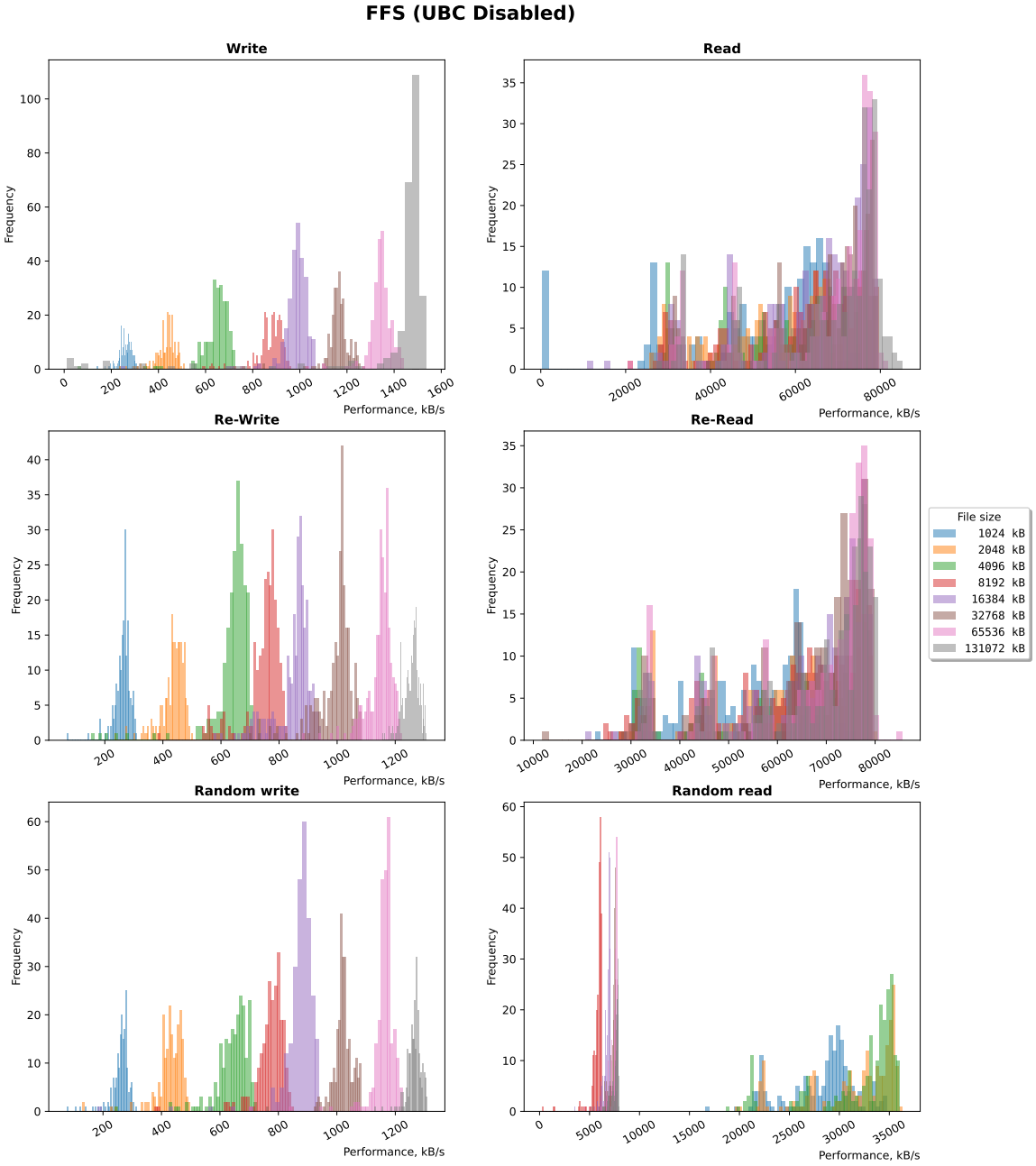

Figure 5.8: Performance comparison of different file sizes for FFS with the UBC disabled

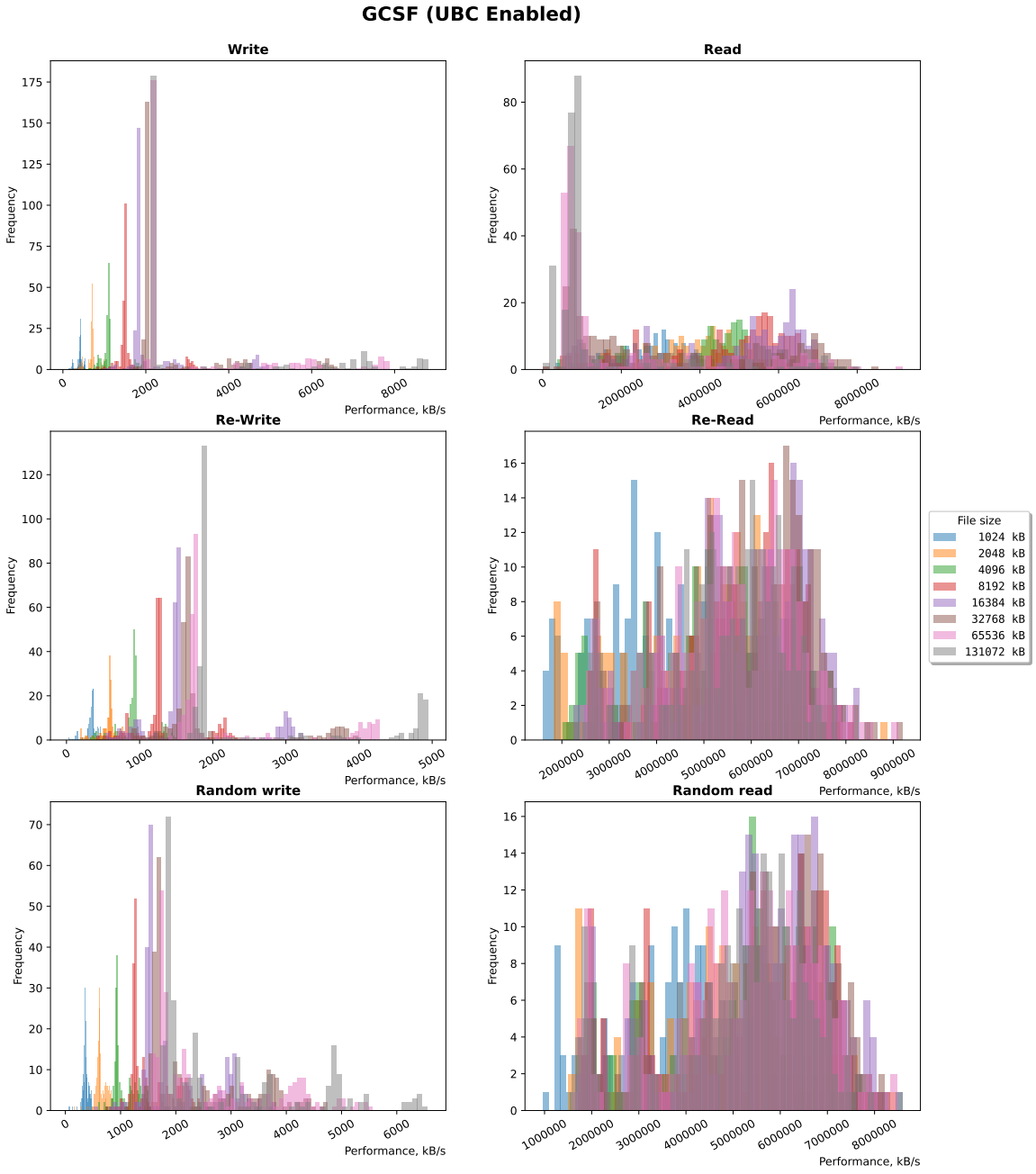

Figure 5.9: Performance comparison of different file sizes for GCSF with the UBC enabled

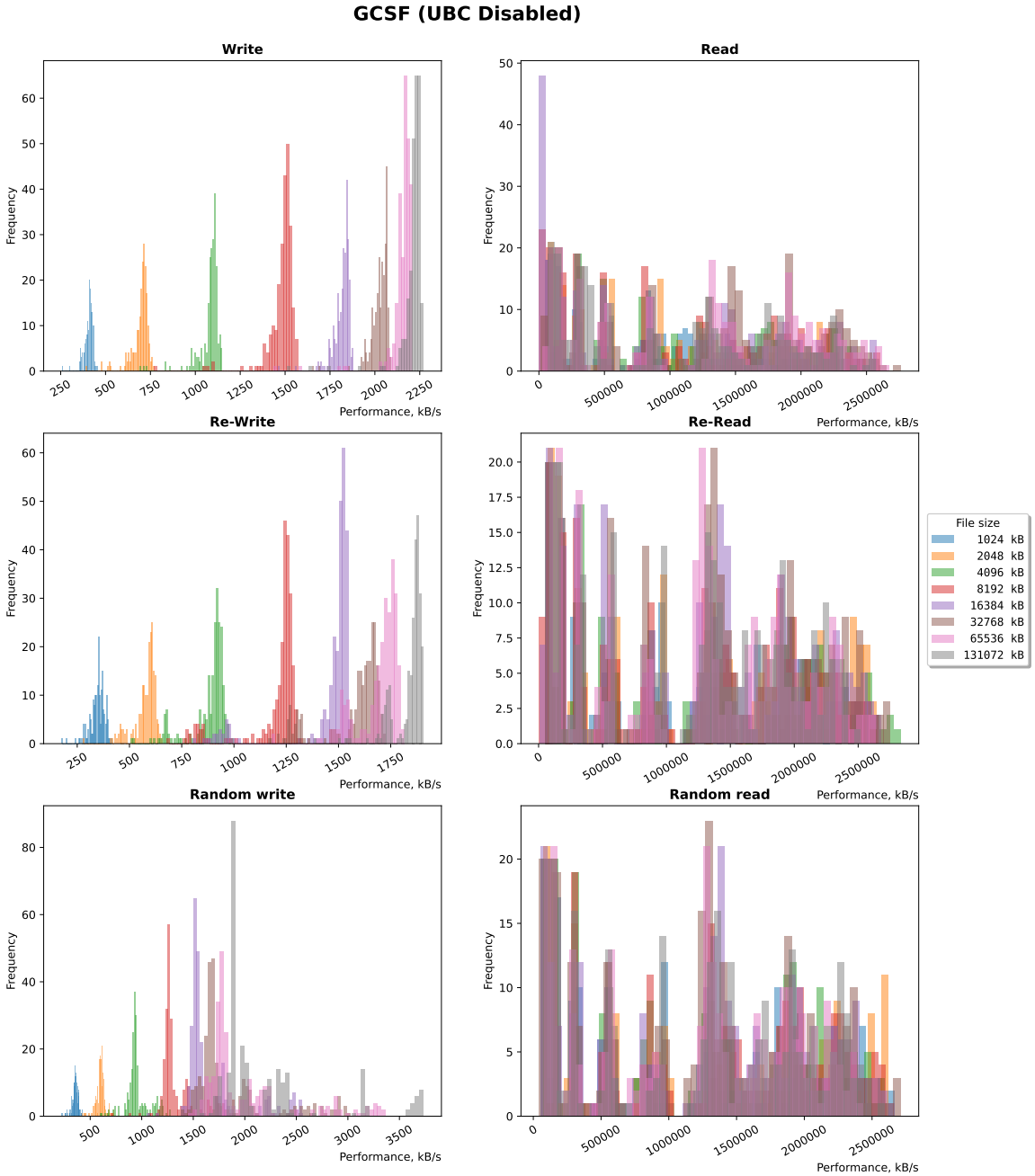

Figure 5.10: Performance comparison of different file sizes for GCSF with the UBC disabled

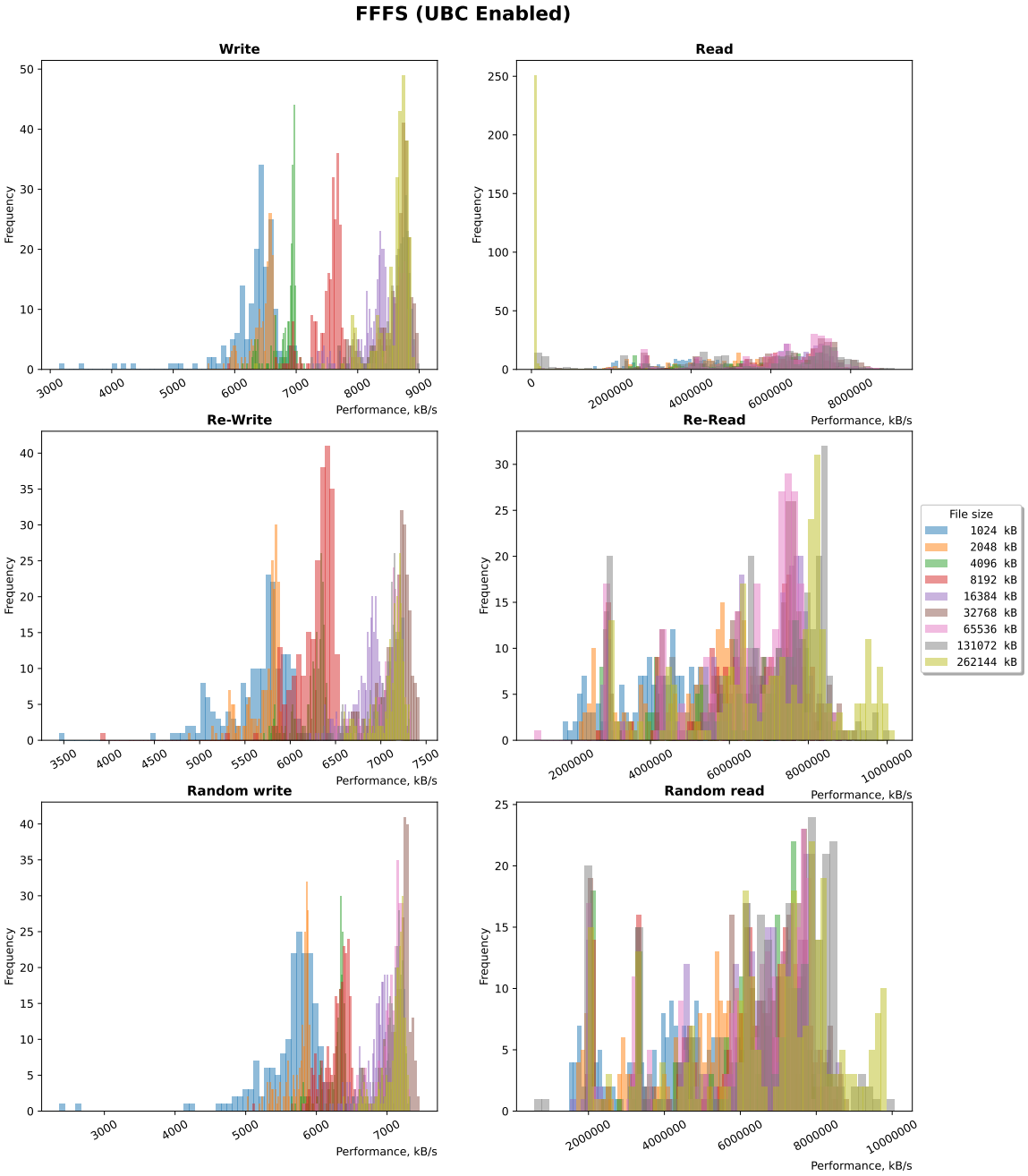

Figure 5.11: Performance comparison of different file sizes for FFFS with the UBC enabled

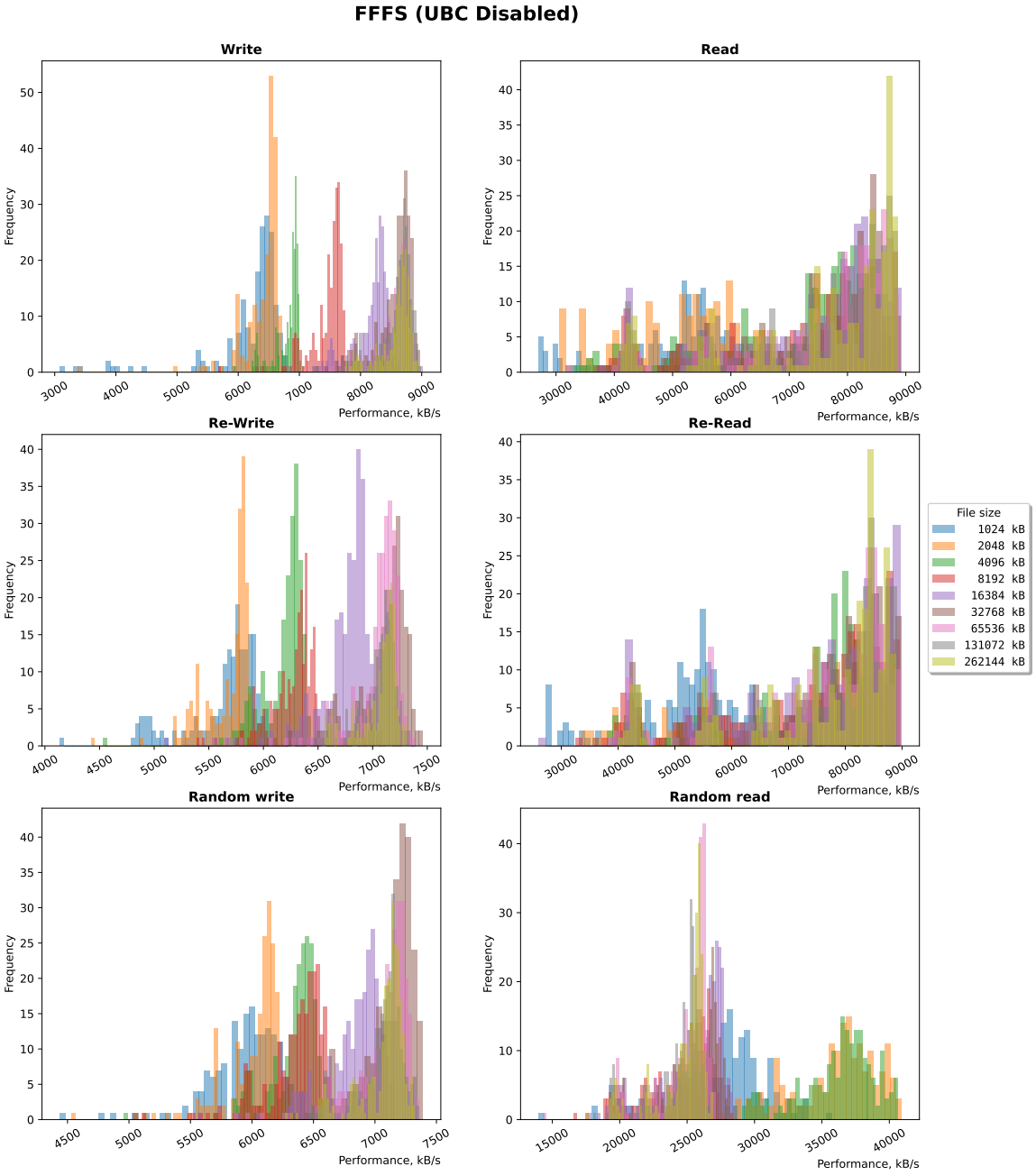

Figure 5.12: Performance comparison of different file sizes for FFFS with the UBC disabled

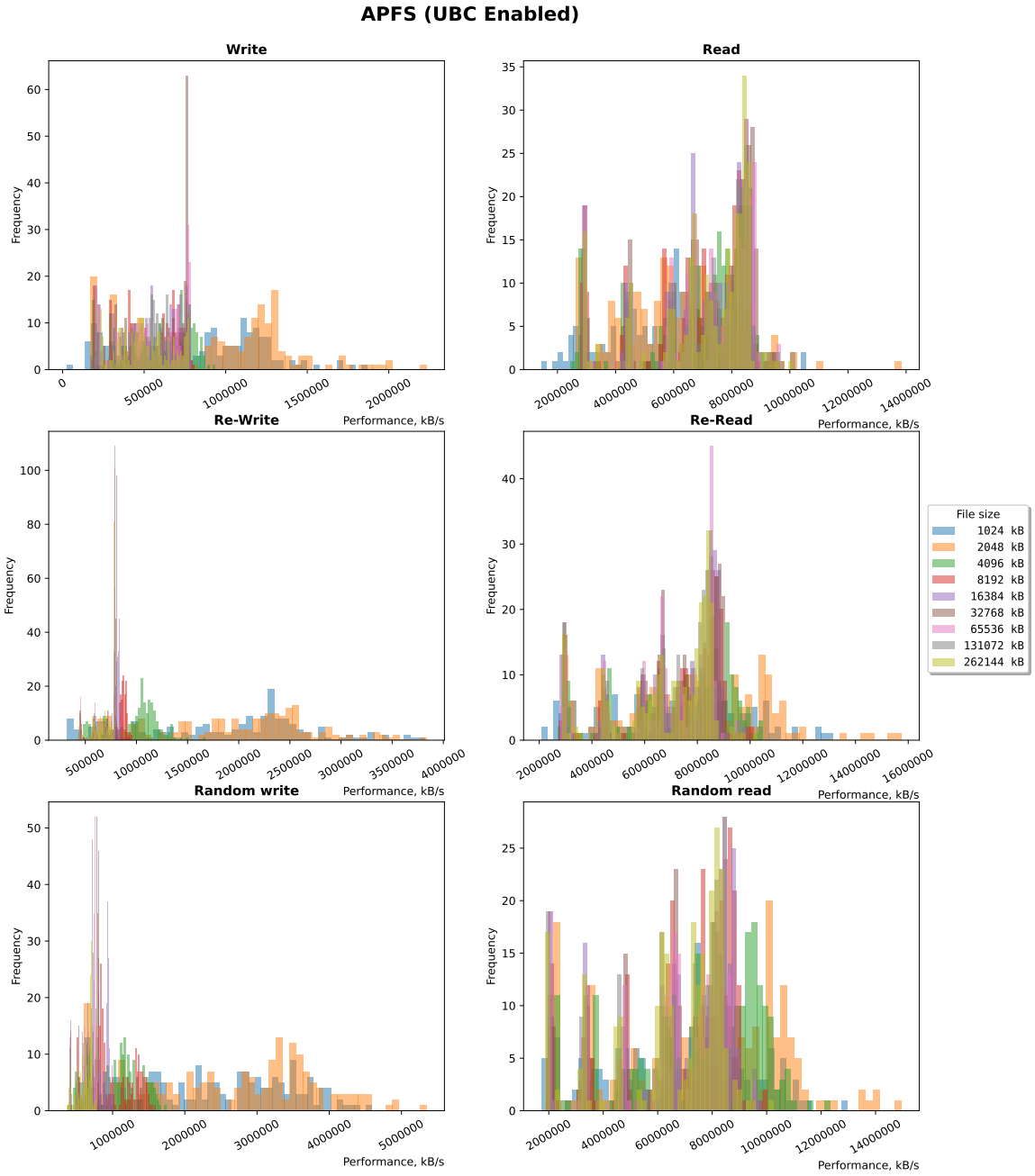

Figure 5.13: Performance comparison of different file sizes for APFS with the UBC enabled

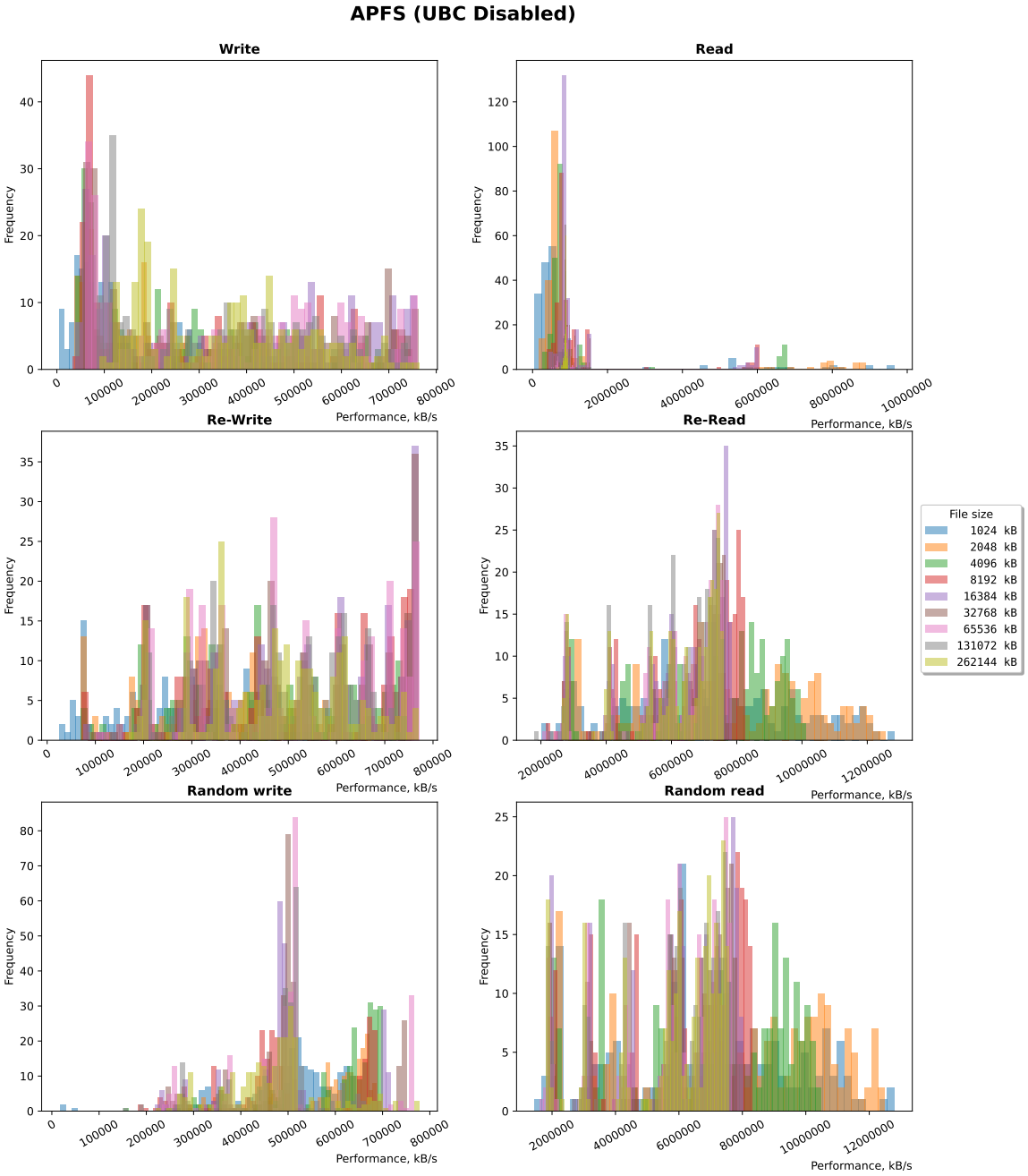

Figure 5.14: Performance comparison of different file sizes for APFS with the UBC disabled

Chapter 6

DISCUSSION

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

6.1 Filesystems

The read test of the filesystems have significantly lower performance when the UBC is disabled compared to when it is enabled. This is expected as disabling the cache provided by the operating system kernel requires the process running the filesystem to be called with the arguments. The data read during the read test is the same data written to the file during the write test, so the data can be cached and reported back without accessing the underlying storage device. The write test is not influenced as much by the state of the UBC for most of the filesystems. FFS, GCSF, and FFFS have similar performances for when the UBC is enabled and disabled for the write test. APFS has a slightly lower performance when the UBC is disabled compared to when it is enabled. It is understandable that the UBC does not influence the write operation performance as significantly as it does influence the read performance. The slight difference for APFS could be outliers due to, for instance, difference in memory usage and CPU usage by other processes during the benchmarks. This could be confirmed by running more benchmarks and increasing the confidence level of the data. There could also be benefits with an enabled UBC for the write operation that makes APFS perform better with the UBC enabled.

FFS and FFFS perform similarly for read test when the UBC is disabled, with FFFS performing slightly better. The spread of values by FFS is greater than the spread of FFFS, and the lowest 31 performance data points of FFS are worse than the lowest

performance data point of FFFS. The data of FFS with the UBC disabled does not include benchmarking of the biggest file size. For both FFS and FFFS with the UBC enabled, the 262 144 kB file size had significantly worse performance in the read test compared to the other file sizes of the same test for the filesystems. For FFFS with the UBC disabled, the read test for the same file size is similar to the performance of the preceding file sizes, but the performance is still significantly worse than for FFFS with the UBC enabled. It is therefore probable that FFS with the UBC disabled would perform similarly for the 262 144 kB file size as it preforms for the preceding file sizes but still preform significantly worse than for the same file size for FFS with the UBC enabled.

For the re-read and random read tests, APFS with the UBC disabled performs similar to APFS with the UBC enabled, while APFS with the UBC is significantly worse than APFS with the UBC enabled for the read test. This indicates that the cache of APFS is fast for previously read files. This is expected for a widely-used commercial filesystem.

Looking at Figure 5.2, Figure 5.2, and Figure 5.2, it is observable that all three write tests of FFS with the UBC enabled have high-performing outlier data points. This can also be seen in Figure 5.2 where the low-performance values have high bars and the different bars per file size are overlapping, but a few, shorter bars have much higher performance. These outliers are even outperforming the best performance data points of both GCSF and FFFS with both states of the UBC. The reason behind these outliers is unclear. As FFS with the UBC disabled does not have similar outliers, and as these extreme values are not coming from the 262 144 kB file size which the benchmarks of FFS with the UBC disabled does not include, it is possible the extreme values are due to the UBC. Furthermore, GCSF and APFS with the UBC enabled also have some outlier data points in the three figures similar to the style of the outliers of FFS with the UBC enabled. This further indicates that the outliers are due to the UBC. Although, FFFS does not have similar performance outliers for either state of the UBC. This could indicate that the performance of the FFFS writes are as high as possible and is not affected by potential performance increases by the UBC.

FFS with the UBC enabled performed better than GCSF with the UBC enabled for all the read tests. However, FFS with the UBC disabled performs significantly worse than GCSF with the UBC disabled. This indicates that the UBC is critical in the performance of FFS while the performance of GCSF depends less on the UBC. This could be due to a faster cache implemented by GCSF compared to the cache of FFS. It could also mean that Google Drive is a more time-efficient storage service than Flickr as both filesystems might have to communicate more with the OWSs when the UBC is disabled. As Google Drive provides the raw data of the file stored, GCSF 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 GCSF. Looking at the median performance of FFS and GCSF, GCSF with both states of the UBC outperforms FFS with both states of the UBC, in all the write tests. The reason could be that GCSF does not have to encrypt the data nor encode it as images. This could save significant computation time. Furthermore, it is possible that GCSF is faster because it requires fewer REST API 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 FFS, the inode table would never have to be downloaded and uploaded by GCSF. 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, FFS 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 FFS 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 OWS 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 API call using the file's ID [100] which will perform the same functionality as the four requests required for FFS. The ID of the file could be stored locally in memory by GCSF 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 [101]. 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 FFS as we need the full file content to decode the image pixel data as the encrypted cipher text. Even if we could download certain blocks of the cipher text, we need the whole cipher text to verify the authenticity of the AES-GCM cipher. We could also not change the content of this block and encrypt it again without reusing the cryptographic parameters, thus, for write operations we need the full file. Furthermore, by downloading the full image when it is read, subsequent reads of FFS are faster which is beneficial for the benchmarking tests as it often needs to read the file multiple times at different offsets. Google Drive has 800 million daily users [102] versus Flickr's 60 million monthly users [103], which means Google Drive is a much bigger service. This could mean that Google Drive has better infrastructure and that they can process the uploaded data faster and serve the data for download faster than Flickr can.

For the write tests on FFS and GCSF with the UBC disabled, the file size is significant for the performance of the filesystems, as can be seen in Figure 5.2 and Figure 5.2. In general, a larger file sizes has better performance than smaller file sizes for these filesystems. The histograms in Figure 5.2 and Figure 5.2 show almost isolated peaks for each file size, while the histograms for the other tests of the filesystems show more

overlap of the performances of the different file sizes. The distribution of these file sizes looks to be normally distributed. For FFFS with the UBC disabled, the bars in the histogram overlap more than the bars in FFS and GCSF, but the peaks of the different file sizes are still distinguishable (as shown in Figure 5.2). The distribution of these values also seem to be normally distributed. For APFS with the UBC disabled, the histogram does not show the same pattern as seen with FFS, GCSF, and FFFS. The values of APFS with the UBC disabled does not seem to be normally distributed.

The only implementation difference between FFS and FFFS is that FFS stores the produced images on Flickr while FFFS stores the produced images on the local filesystem. Therefore, the time difference of an FFS operation compared to an FFFS operation should only depend on the internet connection to Flickr and how fast Flickr can process the requests, if we assume the UBC is not enabled. For instance, looking at the write tests when the UBC is disabled for the two filesystems, FFFS outperforms FFS significantly. Looking at one file size, for instance, `file size = 8 192` FFFS has an average performance of approximately 7500 kB/s and FFS has an average performance of approximately 870 kB/s for the write test. This means that the test took on average 1092 ms for FFFS and 9416 ms for FFS. The same test for the same file size for APFS with the UBC enabled[1] had an average performance of around 534 000 kB/s, meaning it took on average 15 ms for APFS to save the 8192 kB file. The time the test takes for FFFS is the same as the write operation overhead plus the time APFS takes to save the file. Subtracting the APFS write time from the test time of FFFS, we get the average overhead of the Write test for both FFS and FFFS as they have the same overhead. This overhead includes encrypting the data and encoding the result as an image. The average overhead time of the Write test for FFS and FFFS is therefore 1077 ms, meaning that the request and the request's overhead by FFS took on average $9\,416 - 1\,077 = 8339$ ms which is approximately 88% of the computation time for this test. Assuming the upload bandwidth to Flickr while FFS with the UBC disabled was being benchmarked is the same as the measured maximum bandwidth of the computer, i.e. 17 720 kbit/s, uploading 8192 kB = 16 536 kB would take 933 ms. This means that the remaining 7406 ms were used for

[1] Even though we are looking at FFS and FFFS with the UBC disabled, filesystem calls to APFS will not be affected, thus the default state of the UBC will be used when FFS and FFFS interacts with APFS

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 APFS performance as above, it would take 27 ms to save the file. Reading the file with the average read test performance of APFS with the UBC enabled for a 8192 kB file of 6 906 000 kB/s would take 1 ms. The remaining 7378 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, FFS could potentially be faster. Furthermore, it indicates that the bandwidth of the internet connection to the OWS is not the most important factor of the performance of FFS. Even if uploading the file over the internet to the OWS would be instant, it would only reduce the file operation time by around 10%. To improve the filesystem operation performance, using a OWS that can process the data faster is of more importance. For instance, if the OWS 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 OWS 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 OWSs to compare their performances, as well as measuring the individual filesystem's bandwidth for more precise calculations.

6.2 Security and Deniability

The data stored in FFS images is encrypted with state-of-the-art encryption standards. Using AES-GCM, FFS 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 [104]. However, the security of FFS 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 FFS encrypted image could brute-force the bad password used to derive the encryption key much faster than they could brute-force the decryption key. FFS does not put any constraints on the password used - as long as it is at least one byte it is acceptable for FFS. This puts the responsibility on the user for a secure password.

FFS puts a lot of trust in the open-source library Crypto++ [85]. Crypto++ provides cryptographic functions that FFS 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 [87], 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 FFS has vulnerabilities, such as side channels, which could be exploited. FFS 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 FFS, both registered users on Flickr, and anonymous visitors, as the account is set to allow this. 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 FFS 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 FFS, 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 FFS 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 possible to see the size of an image, and a big image can suggest that a file or directory consists of multiple images on Flickr.

It is also not possible to know if an image stored on Flickr has been uploaded by FFS or by the user manually to further inflate and diffuse the amount of data stored on the service. For instance, by encrypting random data using FFS's encoder and uploading the images to Flickr, but without saving the posts in the inode table or in a directory of FFS, 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 FFS or not. However, it is possible that Flickr could have logs about the uploaded images, and can be able to distinguish images uploaded from the API from the user interface. Future research could extend FFS to post encrypted random data at random time intervals to automatically diffuse the knowledge about the images on the OWS. This would mean that even Flickr would be unable to distinguish the uploaded data as it would all be uploaded from the same service. FFS 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 the 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 FFS. Furthermore, storing diffusing images on Flickr that are not stored in the FFS filesystem decreases the storage capacity of FFS.

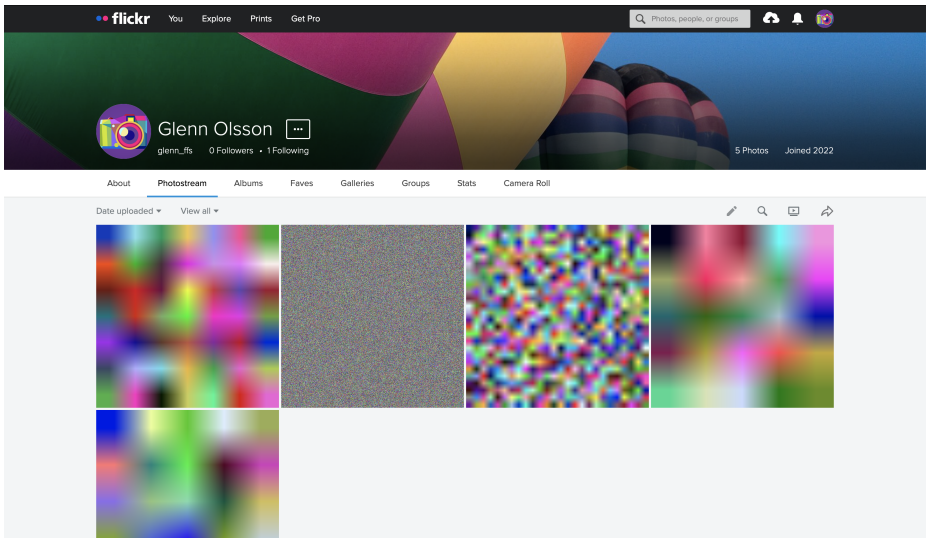

Figure 6.1: Screenshot of the Flickr profile used for FFS. At the moment of the screenshot, the filesystem is storing a previous version of this thesis in a directory inside the root directory. The images seen are the inode table, the thesis data, the root directory data, the subdirectory (containing the thesis) data, and a temporary file containing extra attributes of the thesis document created by macOS while FFS was mounted (this file is sometimes referred to as a *turd* [105]).

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 IV 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 AES 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 [93]. 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 FFS data stored on Flickr at a certain time, assuming that all images on Flickr are stored in FFS. However, it is **not** possible to know if all this data is stored on FFS 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-estimate is also possible.

If a user supplies a different password when mounting FFS than used previously, the images stored on Flickr cannot be decrypted. When FFS 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[2], the user of the filesystem could easily claim that they uploaded FFS 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 FFS 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.

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

However, as mentioned earlier, we assume that an adversary has access to the structure of FFS images as well. To counter this, the user who want to hide their data could, after creating a filesystem containing meaningful information, mount FFS again with another password. FFS would then create a new inode table and upload this table, creating a dummy FFS. In a rubber-hose situation, the user could give up the password to the dummy FFS 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 or that they have been uploaded by the user. These images could, for instance, have been uploaded by another user of FFS. Further, with no password constraints by FFS, a user could also create a dummy FFS 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 FFS with the correct password when the user want to access their actual FFS 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 FFS instance.

One aspect where FFS is better than GCSF is its security against the potential adversary of the storage owners. GCSF 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 GCSF 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 GCSF using a stackable filesystem, such as Cryptfs [74]. Future work could include researching the performance of such an overlay filesystem on GCSF and compare it to FFS.

FFS on the other hand gives the user control of all its data. While Flickr can give out the images uploaded by FFS, 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 FFS with one aspect of better security than GCSF, but this might also be a factor why FFS is slower than GCSF. By requiring the data to be encrypted when it is written and uploaded, and decrypted when it is downloaded and read, FFS 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 FFS. 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, FFS sacrifices the performance of the filesystem operations.

Every Internet Service Provider (ISP) in Sweden is required to keep data generated or managed during internet access for ten months [106]. European Court of Justice (ECJ) has declared that retained data should only be accessed when fighting serious crimes or if national security is at stake [107]. In the United States, there are no data retention laws requiring ISPs to store identifiable information, however, no major American ISP has enacted zero data retention policies [108]. Furthermore, American ISPs can be forced to give out what information they have about a customer by US law, including the websites the customer has been visiting [109]. Using FFS or GCSF, your ISP 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 FFS or GCSF. However, technologies such as Tor or a Virtual Private Network (VPN) could hide your identity from your ISP. According to Swedish law, VPN providers are not required to retain logs of its users [110]. However, they are not prohibited from storing such data. Furthermore, the ISP of the VPN provider must still log the network traffic originating from the VPN service. Even though they cannot identify specific users without logs from the VPN provider, they can determine what websites

are accessed from the VPN service, even though they might not know that the service is a VPN provider. If the VPN 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 [111]. 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 FFS further.

FFS 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 removed shortly after being created, there are possibilities to recover removed data from APFS [90, 91, 92]. Furthermore, if FFS would crash for some reason while uploading a file, the temporary file could remain readable in the APFS filesystem. This could be solved by storing the temporary file in a temporary filesystem. For instance, a secure in-memory FUSE filesystem could be mounted on the computer and be used as the temporary file storage. This is part of future work for FFS.

Flickr state in their community guidelines [112] that they do not allow non-photographic elements in their service. FFS uploads images with seemingly random noise, not photographs. It is therefore probable that Flickr would remove the images uploaded by FFS if they were discovered or reported by the community. Another adversary of FFS 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 FFS at a time which is far from their 1 000 image limit. No FFS images has been removed from Flicker as of writing 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 FFS was running at the same time as the inode table is cached in memory. While the images could still be decrypted by FFS, 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 FFS 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 OWS for the data stored in FFS to be safe. However, this guarantee is hard to achieve as the Flickr terms of service state that Flickr retains the right to remove user content from the service at any time [33]. If Flickr would have knowledge about FFS it is possible that they would implement detection techniques of the FFS images, and remove them. Combating detection by the OWS is part of future work.

6.3 Impact

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

6.3.1 Societal impacts

Secure and hidden data is not only for the better good. As the data stored on FFS 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 only have 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 [113]. FFS could potentially provide secure storage for illegal groups such as terrorist organizations and child pornography rings. It is not possible to limit who uses FFS, by other means than not publishing the source code of the filesystem. However, this does not prevent criminal organizations from 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 [114].

An ethical consideration of FFS is that it is breaking the terms of services of Flickr by exploiting its free storage. Flickr and many other OWSs provide users with a lot of free storage. By exploiting this storage, the costs for the OWSs 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 OWS would implement detection techniques to combat the FFS images, these could falsely mark legit images for removal which again could affect honest users of the service. Fewer users of the OWS could eventually require the company to reduce their staff due to loss of revenue.

FFS 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 FFS 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

FFS 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 [115] and emit roughly the same amount of carbon dioxide as the global airline industry emits burning aircraft fuel [116]. 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 [117, 118]. The data centers Flickr are using and where they are located have 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 FFS, 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 FFS to a cloud-based filesystem, such as Google Drive. While both Google Drive and FFS store their data in data centers, the data that Google Drive stores can be less than the same file stored in FFS 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 FFS 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 API than Flickr does with its API. FFS 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. FFS 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. FFS 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, they can promote single-use of the devices which in turn could increase littering. However, with the low-performance of FFS compared to a local filesystem, FFS can often not be used as a substitute to a physical storage device. While a filesystem on a portable storage device was not included in the analysis of the thesis, FFS 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

FFS is a cryptographic and deniable cloud-based filesystem with free storage through exploiting online web services. Compared to other filesystems, FFS 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, GCSF. However, one key difference between these two filesystems is that FFS manages the cryptography of the filesystem, while GCSF delegates this task to Google Drive. This provides security benefits for FFS, but might also contribute to the slower computation time. The results also show that even when removing the dependency of an internet connection, FFS performs poorly, especially for the read operations compared to GCSF and APFS. The write operations of FFFS perform better than GCSF and FFS. The read operations of FFS and FFFS are more similar than the write operations, however, FFFS outperforms FFS for every read operation test leading to the conclusion that the internet connection and the OWS influence the file operations significantly. With better read performance than write performance, FFS 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 FFS, or even prove that data is stored on FFS. 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, FFS provides the user with proof of the authenticity of the data it stores.

7.2 Future work

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

Even though the files are encrypted so that the data is confidential, further research could include hiding the user's online activity through the use of for instance Tor. Currently, the anonymity of the user is not considered but for FFS 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 and ISP.

To improve the dependability and increase the storage capacity of FFS, 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, FFS could achieve even more storage. Future work includes comparing such a filesystem with the current state of FFS.

To increase the storage capacity further, FFS could take advantage of storing videos on the OWS as well. Flickr allows videos up to 1 GB on its service. 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 OWS would pursue identifying FFS images stored on their service to remove them, it would be a problem for FFS. 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 FFS. Part of future work is to explore if more data could be stored in the images, or if multiple OWSs 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 that 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, 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* (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*. 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. 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, 2007.

[7] Jim Salter. *Understanding Linux Filesystems: Ext4 and Beyond*. Opensource.com. 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. 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 Data Security Overview*. Apple Support. URL: https://support.apple.com/en-us/HT202303 (visited on 01/04/2023).

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

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

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

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

[26] *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).

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

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

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

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

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

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

[33] Flickr, Inc. *Flickr Terms & Conditions of Use*. Flickr. 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. 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 (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 (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. 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. 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). 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, 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, 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. 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* (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*. 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*. 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. 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. 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 (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, 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). 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. 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*. 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*. 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". Diploma thesis. Romainia: Babes-Bolyai University Cluj-Napoca, 2018. URL: https://harababurel.com/thesis.pdf (visited on 08/27/2022).

[66] Puşcaş Sergiu Dan. *Harababurel/Gcsf*. 2021. 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*. 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. 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! 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. 2021. URL: https://dev.to/yawaramin/use-google-drive-as-a-local-directory-on-linux-1b9 (visited on 09/08/2022).

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

[73] Puşcaş, Sergiu Dan. *Show HN: Google Drive as a File System | Hacker News*. Hacker News. 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] Erez Zadok. *FiST: Stackable File System Language and Templates*. URL: https://www.filesystems.org/ (visited on 02/02/2022).

[76] *Iozone Filesystem Benchmark*. 2016. 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, 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. 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. 2016. URL: https://9to5mac.com/2016/11/01/2016-macbook-pro-ssd/ (visited on 10/16/2022).

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

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

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

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

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

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

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

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

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

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

[95] Krypted. *Maximum Files in Mac OS X*. krypted. 2009. URL: https://krypted.com/mac-os-x/maximum-files-in-mac-os-x/ (visited on 12/03/2022).

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

[97] Puşcaş Sergiu Dan. *Gcsf::Config - Rust*. URL: https://docs.rs/gcsf/latest/gcsf/struct.Config.html (visited on 12/03/2022).

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

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

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

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

[102] Frederic Lardinois. *Google Updates Drive with a Focus on Its Business Users*. TechCrunch. 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).

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

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

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

[106] Post- och telestyrelsen. *Frågor och svar om datalagring | PTS*. pts.se. 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).

[107] *European Court of Justice/Sweden: Invalidation of Data Retention Obligations*. Library of Congress, Washington, D.C. 20540 USA. 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).

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

[109] 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).

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

[111] E. Ramadhani. "Anonymity Communication VPN and Tor: A Comparative Study". In: *Journal of Physics: Conference Series* 983.1 (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).

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

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

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

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

[116] 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).

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

[118] 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 FFS. Listing A.1 presents the attributes each C++ class stores in-memory, that is also encoded into the binary representation of the object when it is serialized before being uploaded to the OWS.

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

```
# 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 FFS IMAGES AND CLASSES

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

B.1 Serialized C++ objects

The `InodeTable`, `InodeEntry`, and the `Directory` class all have one `serialize` and one `deserialize` method each. The `serialize` method converts the 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 FFS image consists of multiple binary structures, including the FFS header and the encrypted data. This section visualizes these binary structures. Figure B.2 visualizes binary format of the FFS header. Figure B.2 visualizes the PCD of FFS images stored on the OWS.

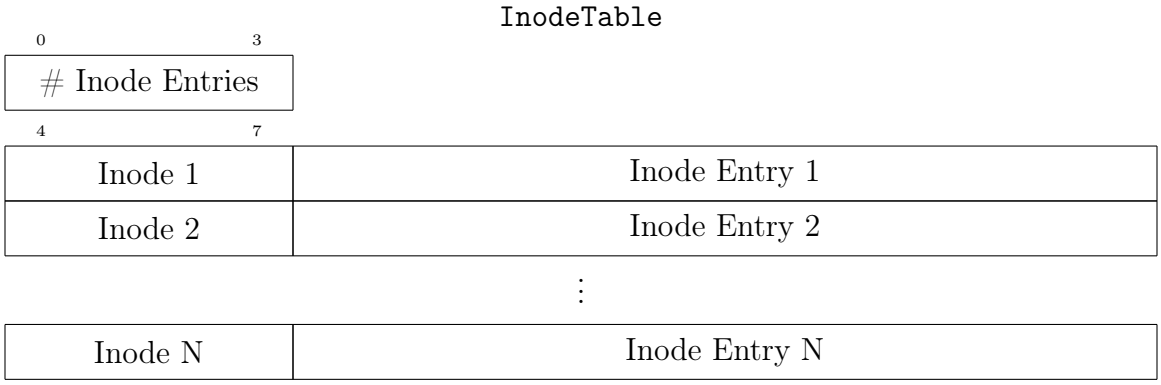

Figure B.1: `# Inode Entries` **is an unsigned integer representing the amount of inode entries the inode table contains. Following are** `# Inode Entries` **entries of an unsigned integer representing the inode of the inode entry, and the serialization of the corresponding** `InodeEntry` **object**

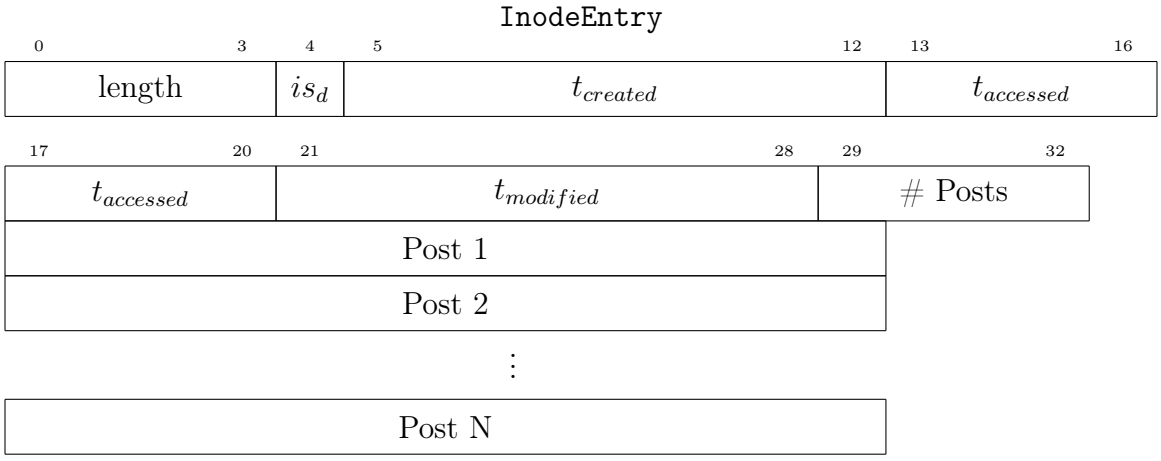

Figure B.2: Byte representation of a serialized `InodeEntry`**, representing a file or directory stored in FFS.** `length` **is an unsigned integer representing the amount of data stored on FFS by the file or directory, for instance the size of the file.** is_d **is a boolean with the value true** ($\neq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

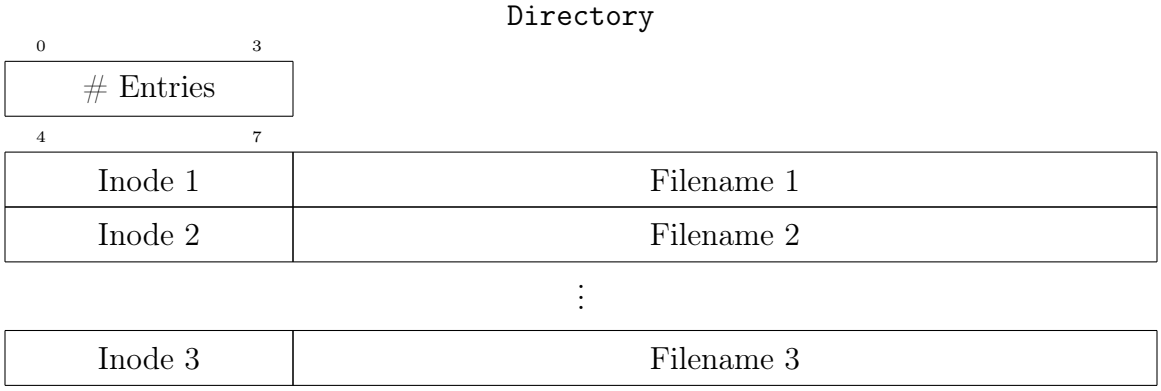

Figure B.3: Byte representation of a serialized `Directory`. `# Entries` is an unsigned integer representing the amount of entries in the directory. Following are `# Entries` inode-filename pairs. The `Inode` is an integer representing the inode of the file or directory, corresponding to the file's or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

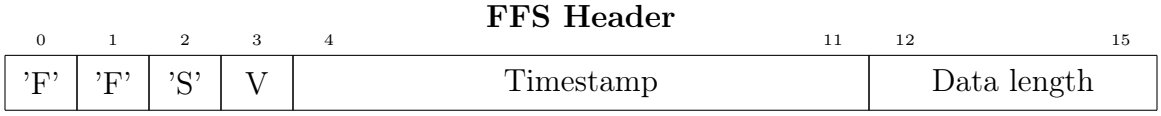

Figure B.4: `'F'` and `'S'` are the literal letters F and S in ASCII code. `V` is an integer representing the version of the FFS image produced. `Timestamp` is an unsigned integer representing the number of milliseconds since Unix epoch when the image was encoded. `Data length` is an unsigned integer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

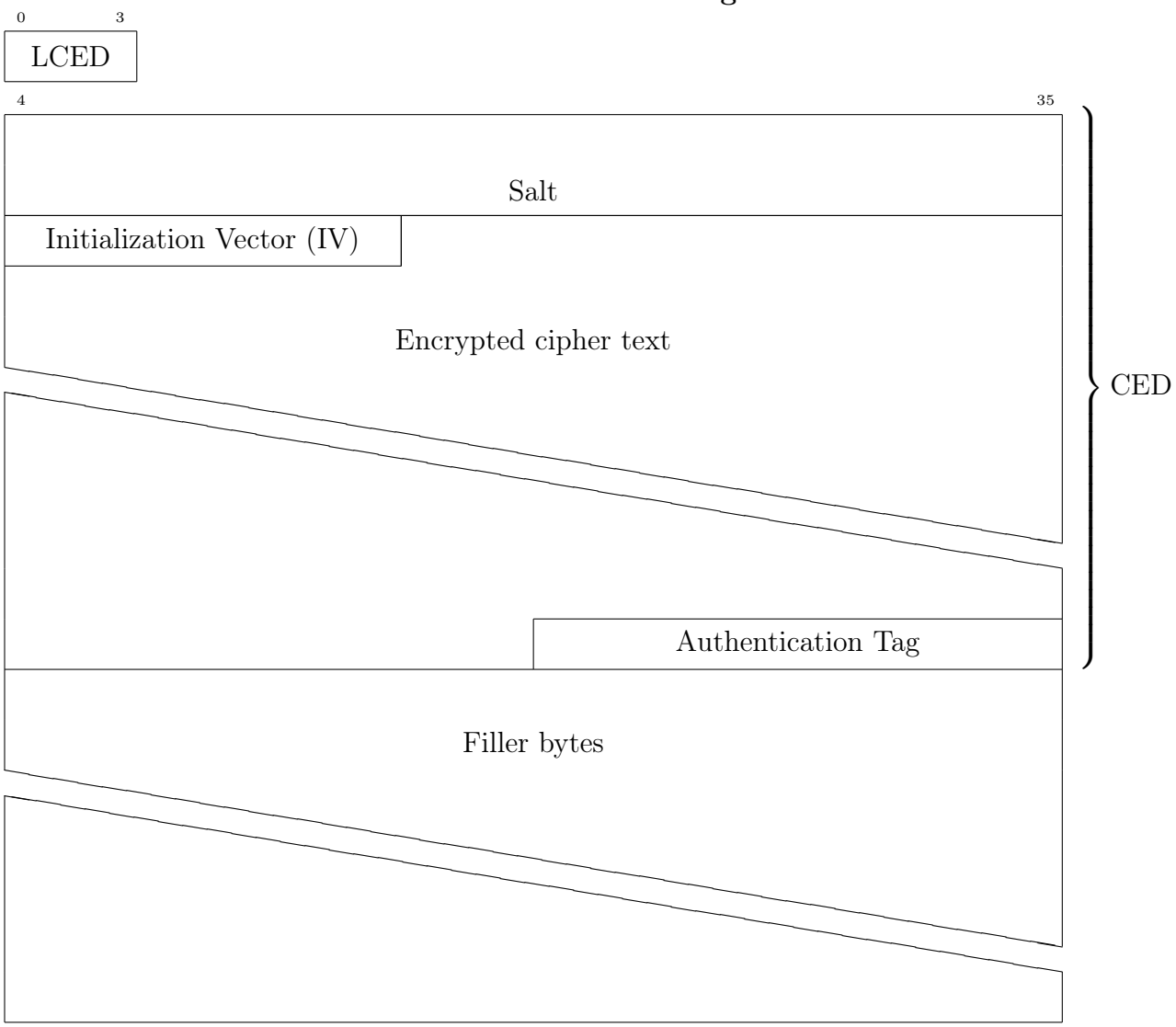

Figure B.5: Byte representation of the data stored as PCD in FFS images. `LCED` is an unsigned integer representing the Length of the CED. The `Salt` is a 64-byte randomized vector used to derive the encryption and decryption key. The `IV` is a 12-byte randomized vector used as the initial state of the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

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: Average IOZone result for the Read (UBC Enabled) test on FFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1627198.75	2155825.10	2593570.10	2867205.75	3145213.35	3409297.80	3107244.40	3511014.60	3384576.70				
2048	1941860.15	2682642.10	3311808.00	3863611.65	4032429.65	4232633.30	4404393.95	4367221.50	4550743.10	4703806.15			
4096	2281104.05	3259274.60	3900249.75	4862889.15	5101384.95	5570962.65	5467164.25	5933009.40	6288984.50	6030372.25	5306311.70		
8192	2593124.95	3739156.05	4850079.95	5731526.80	6084289.60	6443276.30	6298102.85	6411956.95	6530270.35	6285210.55	5446264.15	5318551.55	
16384	2677835.70	3914647.10	5131625.60	6139179.10	6613625.25	6619402.05	6796136.60	6872386.15	6760522.30	6716756.40	5724145.75	5467843.55	5393591.40
32768	2628859.55	3646973.85	4749944.75	5102768.50	3734117.00	6307173.85	2822104.30	2223186.70	3140540.80	5813694.00	4398699.40	1994172.40	779638.75
65536	1726512.55	2667808.70	4198703.65	4816868.10	4512971.20	7209483.35	4920580.30	5584274.30	5282001.65	5316163.05	5544862.10	3762952.45	4626342.65
131072	2102567.70	2135876.05	2703554.10	3942999.20	4387054.60	4610641.25	3957470.10	4686673.80	5460370.40	5330447.90	4784452.80	5054039.60	4774826.25
262144	89333.37	87974.84	105553.32	182866.79	196992.53	177883.79	1493786.16	299857.32	996932.37	884192.00	1615347.32	517825.63	445333.84

Table C.2: Average IOZone result for the Write (UBC Enabled) test on FFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	240.95	2778.75	260.30	254.20	251.25	4505.90	243.10	267.00	257.75				
2048	462.80	481.35	485.45	475.85	18150.20	22846.60	482.75	10263.45	15971.60	467.85			
4096	789.25	740.90	756.55	808.55	788.35	781.45	753.05	15743.95	47838.85	789.25	794.45		
8192	1065.85	1142.15	1157.40	1099.15	1110.65	1171.25	1133.25	1104.65	1176.35	48266.10	6712.40	1163.40	
16384	1309.90	1329.35	1299.75	1369.25	1346.50	1359.10	1347.90	1376.75	1340.00	1360.05	28226.20	6530.30	1310.80
32768	1644.25	1655.65	1682.45	1702.20	1674.45	1694.35	1717.10	7004.85	1726.05	1676.60	1607.65	29528.75	1651.85
65536	2118.05	2169.50	2137.15	2225.40	2171.25	2231.90	2208.20	2137.85	2176.80	2219.20	2202.00	2228.50	13818.65
131072	2531.00	2504.95	2569.05	2316.50	2516.10	2540.00	2480.40	2508.45	2554.35	2601.45	3636.90	2596.45	5450.55
262144	2606.63	2672.58	2526.42	2537.37	2550.16	2315.58	2727.84	2732.37	2620.89	2657.05	2674.47	2703.58	2700.47

Table C.3: Average IOZone result for the Re-Read (UBC Enabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2057360.25	2571045.65	3076165.95	3697323.30	3832558.25	4008510.55	4118144.85	4484216.95	4726955.95				
2048	2395597.70	3160307.25	3655987.40	4330109.10	5103747.55	5265719.75	5566002.75	5569603.00	5704020.25	5289910.25			
4096	2665388.85	3885116.90	4703424.50	5667872.15	6432146.10	6767835.90	6716755.60	6999068.20	7119333.45	6149584.80	5352278.50		
8192	2766453.95	4101797.20	5367533.70	6361934.70	6933285.10	7501853.30	7467824.65	7429500.95	7694379.75	7101050.50	6192737.20	5037291.85	
16384	2732339.80	3869876.55	5012185.25	6206738.35	7052390.75	7010675.95	7642184.65	7335238.05	6957081.40	7066031.50	5542587.80	5723808.30	5683227.20
32768	2779738.70	4062711.30	5255268.75	6504587.00	7048718.05	7129360.50	7563740.70	8107130.55	7651903.15	7101138.70	6265163.40	5923027.65	6027564.05
65536	2844560.50	4177246.15	5487978.50	6691498.40	7362385.35	7677975.50	7494739.35	7792490.95	7702126.55	7529112.25	6827889.35	5968258.95	6088153.70
131072	2876264.95	4268846.35	5518903.20	6630325.30	7484493.45	7877139.60	7751912.40	7977896.50	7930006.70	7769258.55	6817794.85	6069115.40	6051354.85
262144	2919507.00	4191963.37	5465054.16	6658825.53	7572039.47	7752727.84	7724718.21	7951701.89	8074120.05	7656738.11	7013852.37	6237473.00	6138923.16

Table C.4: Average IOZone result for the Re-Write (UBC Enabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	242.50	249.45	267.60	261.05	249.60	273.80	5321.95	3915.30	3728.65				
2048	487.40	476.95	490.60	479.75	493.10	453.65	461.70	15236.70	2262.35	2226.85			
4096	775.70	749.35	682.70	797.45	737.30	784.80	708.50	776.55	6504.90	2568.20	770.85		
8192	937.35	936.45	956.80	912.65	957.70	955.05	967.95	951.55	940.55	2158.50	982.70	955.30	
16384	1095.25	1123.85	1132.00	1129.00	1129.85	1146.10	1126.65	1139.45	1125.00	1118.25	1694.00	1090.70	1072.85
32768	1384.40	1397.15	1401.40	1391.10	1365.95	1381.90	1394.05	1386.35	1406.00	1383.15	1398.85	1683.35	1382.70
65536	1699.70	1710.90	1684.20	1710.70	1710.70	1702.75	1732.95	1729.65	1717.35	1716.55	1720.65	1717.75	1720.75
131072	1934.90	1942.90	1943.25	1960.65	1976.55	1996.40	1953.00	1975.30	1967.95	1958.50	1981.50	1966.10	1963.50
262144	1985.47	2020.05	2011.89	2041.21	2035.11	2037.53	2024.11	2059.32	2047.95	2040.58	2037.05	2042.26	2025.16

Table C.5: Average IOZone result for the Random read (UBC Enabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1720655.05	2451216.25	2994930.80	3686554.95	3822280.10	4683813.20	4586560.55	4716021.75	4572095.05				
2048	1859076.75	2809722.40	3631037.15	4601501.80	5124641.10	5364713.90	5848261.75	6039710.05	5830624.00	5178701.50			
4096	1978036.70	3144133.10	4135475.75	5246835.80	6034298.90	6497557.85	6659414.20	7022728.50	6907875.05	6302026.65	5323568.75		
8192	1858815.40	3163316.00	4452272.25	5599382.35	6470938.25	7239312.50	7504030.30	7286136.35	7466193.75	7107503.25	6102323.55	5430099.40	
16384	1933202.20	3039613.75	4212601.85	5317927.15	6463764.00	6576026.55	7091016.95	6944905.35	6964497.35	6633791.45	5992078.65	5666283.90	5845939.40
32768	1991384.65	3126429.40	4290030.85	5709699.15	6438786.10	6691771.55	7571015.90	7959091.00	7640640.65	7236454.85	6470738.40	5958053.95	6020133.65
65536	1966981.05	3122469.00	4342328.85	5794269.20	6796452.60	7424269.25	7394214.85	7683912.40	7830415.85	7540039.05	6803044.90	5973037.80	6054542.45
131072	1944334.20	3113164.60	4386140.50	5725380.00	6853835.30	7550000.20	7595633.35	7909279.85	7817854.00	7825986.80	6823176.00	6035397.35	5982038.45
262144	1950691.95	2996691.79	4392928.16	5735366.95	6765935.84	7338845.63	7540994.47	7851986.21	7982726.68	7518419.16	6987378.47	6249313.37	6064863.58

Table C.6: Average IOZone result for the Random write (UBC Enabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	251.55	271.05	259.80	257.40	251.45	256.60	6320.65	2328.30	252.80				
2048	454.55	437.30	472.75	488.30	491.20	469.85	459.70	7696.40	474.60	481.75			
4096	731.70	752.15	715.80	782.85	781.75	791.50	770.35	759.30	4526.20	787.65	785.25		
8192	957.00	950.95	947.50	968.75	964.10	966.00	918.60	943.20	972.00	2125.35	975.60	958.95	
16384	1122.60	1135.25	1116.45	1137.05	1159.50	1138.35	1148.60	1132.85	1129.50	1154.95	1127.20	1095.70	1116.85
32768	1369.40	1383.45	1376.35	1428.10	1381.70	1380.15	1398.00	1402.95	1406.15	1389.90	1304.10	1394.15	1385.25
65536	1660.65	1697.15	1700.45	1718.90	1739.50	1721.05	1714.85	1740.75	1704.15	1737.85	1701.00	1741.95	1733.95
131072	1908.65	1937.70	1938.85	1967.55	1987.55	1968.35	1996.55	1994.60	1991.10	1976.65	1952.05	1973.75	1967.10
262144	2004.47	2009.11	2041.89	2031.63	2040.42	2063.42	2046.53	2059.53	2054.11	2038.74	2046.58	2027.84	2063.47

Table C.7: Average IOZone result for the Read (UBC Disabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	26644.45	15003.90	46199.85	50959.15	57235.35	62203.85	63526.75	65286.40	67512.65				
2048	28685.75	38243.95	46685.85	51132.15	56223.80	60590.65	62240.55	63313.15	66371.05	67894.45			
4096	30080.45	41760.60	52369.95	56647.75	61847.55	64979.70	68014.40	71642.10	72109.35	74387.10	77311.05		
8192	29903.95	41169.40	52174.00	58512.30	62182.60	63209.70	67154.50	68814.00	71824.20	75095.35	74822.60	76791.85	
16384	30388.05	43508.50	51574.95	61688.15	62749.30	67736.60	70910.80	71814.25	72170.20	75554.75	75265.30	76864.35	77438.20
32768	32159.15	44563.50	55979.60	63546.80	67660.30	70442.55	72702.55	72845.90	73435.30	74950.45	76320.90	76886.10	76276.45
65536	32202.00	45084.35	55783.35	63573.85	69085.80	71802.45	71842.15	75269.80	74660.50	77273.10	77662.70	78666.25	77679.75
131072	32731.20	45214.05	55922.15	65013.65	64799.50	72846.10	74642.80	73510.80	77038.50	77342.20	78362.15	76450.20	77028.45

Table C.8: Average IOZone result for the Write (UBC Disabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	264.50	264.00	248.45	254.90	267.50	266.20	260.35	260.00	256.35				
2048	426.20	422.70	433.80	423.80	452.40	424.80	444.95	430.40	437.85	429.20			
4096	632.45	647.70	634.45	652.80	662.75	645.30	621.85	634.35	660.40	656.75	659.20		
8192	847.25	880.60	854.40	883.60	873.35	882.15	881.85	878.25	837.60	879.25	888.90	879.45	
16384	971.55	981.90	972.70	968.60	968.20	982.40	1000.00	985.90	990.45	976.25	1000.45	988.75	992.70
32768	1147.55	1162.25	1152.20	1171.50	1175.25	1166.75	1173.60	1164.40	1172.45	1166.10	1152.75	1170.15	1170.90
65536	1324.70	1325.05	1338.40	1325.20	1341.45	1338.90	1317.55	1361.80	1348.90	1345.65	1356.75	1349.50	1352.55
131072	1307.35	1306.25	1409.65	1279.70	1304.85	1327.55	1343.05	1361.75	1407.35	1456.80	1452.30	1469.00	1472.85

Table C.9: Average IOZone result for the Re-Read (UBC Disabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	30077.80	38635.70	48216.00	52432.90	57774.55	59267.65	64612.90	64644.05	65753.95				
2048	33607.70	46030.75	54206.55	60511.00	64559.40	66607.20	68586.50	71589.50	75854.50	77539.55			
4096	31487.70	43959.75	56426.15	65374.25	70090.90	74142.80	76334.25	76027.05	76291.65	75284.45	77392.40		
8192	30666.60	42517.30	53378.20	59748.05	63287.85	67483.85	69208.80	71024.70	71643.30	75827.85	75154.50	78180.30	
16384	32507.15	44301.45	54694.80	61928.00	64907.75	68322.45	71119.90	72907.05	74215.80	75859.45	75520.35	77460.70	77769.95
32768	31737.65	44606.70	56775.25	64062.75	67300.40	70836.35	72494.70	72740.95	73283.65	75995.40	76335.75	77711.70	76398.85
65536	33209.30	45768.10	57221.35	65051.70	69067.70	72830.60	74797.90	75748.65	75896.15	77256.90	77808.35	78355.45	77112.60
131072	33051.35	45731.10	56405.85	64453.00	68084.70	72849.95	73713.60	75095.65	76510.35	77742.30	76946.05	76403.95	77941.45

Table C.10: Average IOZone result for the Re-Write (UBC Disabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	259.40	266.65	262.00	262.35	256.25	262.10	254.90	263.90	268.55				
2048	443.65	438.65	431.70	426.70	442.15	439.20	430.55	419.85	429.30	440.60			
4096	658.00	659.95	611.15	646.15	626.50	611.45	634.00	624.50	651.15	640.20	660.85		
8192	758.10	754.50	742.15	738.00	744.60	732.70	749.60	741.10	746.70	748.50	714.10	753.05	
16384	830.15	847.15	849.45	856.90	839.50	840.05	869.95	842.95	852.40	859.80	856.50	835.20	865.60
32768	976.40	984.15	1001.05	985.00	996.70	994.95	1012.80	1007.25	1003.80	992.70	997.60	1002.30	978.20
65536	1134.50	1141.10	1144.65	1164.10	1142.50	1157.20	1136.05	1153.70	1146.00	1154.65	1147.45	1155.00	1150.65
131072	1230.50	1246.90	1250.55	1257.50	1260.75	1256.20	1267.20	1262.80	1265.15	1263.05	1255.10	1250.80	1248.45

Table C.11: Average IOZone result for the Random read (UBC Disabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	21904.50	23850.30	27169.00	28526.00	29102.90	30155.10	30058.50	30696.45	31907.50				
2048	22079.10	26995.95	30101.15	32012.80	33152.65	34302.25	33802.40	34033.15	34274.15	35091.45			
4096	20946.60	26283.55	30436.05	32332.50	33224.20	34478.05	35071.25	34666.75	34756.40	34679.55	34736.30		
8192	5328.20	5663.00	5778.35	5451.75	5804.85	5990.05	5978.25	5712.00	6106.80	6107.25	5944.25	6011.70	
16384	6227.45	6576.95	6629.70	6823.50	6792.60	6861.10	6950.30	6701.35	6881.15	6918.50	6885.05	6909.05	6987.95
32768	6590.45	7059.00	7187.25	7427.60	7389.65	7456.90	7446.60	7478.10	7440.85	7424.50	7550.00	7418.15	7432.90
65536	6791.00	7183.00	7451.55	7554.70	7615.55	7646.15	7513.65	7668.80	7535.35	7668.85	7708.10	7732.20	7601.75
131072	6843.55	7243.90	7546.75	7653.70	7699.70	7773.95	7766.55	7732.00	7793.95	7728.00	7800.75	7798.70	7698.60

Table C.12: Average IOZone result for the Random write (UBC Disabled) test on FFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	262.00	255.30	255.95	266.35	259.75	255.45	262.75	255.05	263.55				
2048	439.10	423.75	408.70	430.00	424.05	435.30	443.10	413.30	432.00	435.05			
4096	634.85	637.20	631.25	655.80	637.60	640.15	625.70	657.70	663.25	642.90	651.80		
8192	769.95	769.95	773.00	779.30	778.10	775.55	758.20	763.45	769.90	753.10	784.20	780.95	
16384	876.35	853.40	879.30	879.10	887.95	867.15	880.40	865.10	876.05	857.55	869.65	882.70	869.05
32768	1008.00	1011.80	1011.95	1018.90	1025.05	1022.80	1034.40	1012.20	1035.55	1028.05	1026.30	1020.90	1020.55
65536	1131.30	1167.60	1164.50	1144.35	1174.95	1162.95	1164.80	1174.80	1150.70	1167.60	1165.80	1151.60	1168.75
131072	1239.10	1257.15	1265.35	1271.70	1267.25	1272.10	1263.45	1273.00	1266.95	1270.15	1268.85	1268.60	1267.70

C.2 GCSF

Table C.13: Average IOZone result for the Read (UBC Enabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1267344.55	2123676.55	2549776.65	3054221.30	3290418.40	3445131.70	3303573.80	3579027.40	4411927.05				
2048	1498881.35	2371486.00	3451162.00	3552185.95	4064775.70	4139759.20	4507029.50	4334591.60	5387339.35	5090614.70			
4096	1843560.85	2242537.35	3458999.55	3893807.80	4475929.25	4316776.00	4474515.70	5106369.00	5488712.70	5606806.80	5329149.50		
8192	2076682.20	3039127.90	3515673.90	4454268.10	4597684.55	5175612.90	5142105.20	5768888.75	6079129.10	5886666.05	5385142.45	5130196.35	
16384	2414077.40	3216632.35	4031683.90	4396101.45	4710400.85	5052579.20	5086774.15	5808154.90	5327470.15	4895669.25	5360113.30	5040014.75	5477455.55
32768	1934526.75	2866364.90	2676321.05	3425737.00	5408573.35	5555417.80	2533685.15	2257479.85	2456851.15	4011663.10	1986283.80	1605345.40	2368419.45
65536	1826433.90	1120267.75	2026416.75	1777668.90	2828084.95	4070503.15	1151287.30	1865697.25	1166374.55	1395755.20	1997674.95	1198648.00	2278744.25
131072	712407.20	837332.10	1003494.55	1793592.85	1649745.10	1143195.00	1799118.95	1224216.15	1548870.55	1435073.50	1130272.35	1037200.30	1160534.75

Table C.14: Average IOZone result for the Write (UBC Enabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	359.60	360.55	427.35	383.50	375.95	411.95	445.80	432.30	437.05				
2048	608.80	672.60	717.25	722.00	709.80	648.30	663.75	688.25	745.30	760.50			
4096	1022.25	1034.10	1021.65	1173.55	1096.35	1109.70	1065.20	1100.05	1259.00	1217.35	1298.30		
8192	1478.35	1530.60	1601.70	1490.55	1638.90	1646.90	1680.80	1535.65	1884.50	1727.55	1943.90	1974.80	
16384	1985.35	2011.45	1940.75	2131.00	2146.65	1785.15	2115.35	1956.00	2259.55	2501.00	2698.95	2687.30	2669.75
32768	2593.90	2663.65	2761.50	2276.40	2561.55	2469.10	2581.35	2490.70	2441.65	2615.30	3155.75	3351.90	3298.95
65536	3084.50	3219.15	3213.40	3061.80	3320.85	2916.40	2851.95	3248.90	3060.55	3331.60	3235.80	3299.85	3825.65
131072	3510.45	3628.90	3388.40	3580.45	3479.30	3609.05	3478.00	3685.40	3491.50	4021.85	3652.60	3244.95	3612.70

Table C.15: Average IOZone result for the Re-Read (UBC Enabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2004270.80	2683580.95	3302549.70	4140315.65	3988612.95	4691353.70	4639306.65	4649954.55	5004788.90				
2048	2197064.10	3193953.00	4007998.90	4733307.35	5248819.95	5668424.20	5422453.45	5846002.40	6112072.90	5771411.50			
4096	2549764.40	3670209.65	4619723.10	5288168.20	5693166.25	5790226.55	5592740.65	6589324.00	6708710.30	6523389.00	5313230.05		
8192	2757692.90	3982320.50	4855313.40	5717581.25	6189045.90	6467479.60	6637779.60	6659396.55	7023367.75	6552331.75	6006216.55	5166804.65	
16384	2737236.85	4062410.30	5231692.50	6146922.00	6552101.30	6817410.35	7090989.15	7166254.25	7182545.70	6657063.50	6208745.95	5266711.40	5457537.45
32768	2827179.00	4025799.35	5205423.60	6016812.60	6959253.95	6800253.10	6893397.90	6893282.45	6963088.70	6718356.75	5864796.05	5553095.25	5372772.10
65536	2761188.60	3909760.90	5048851.90	5762187.15	6343481.30	6692136.90	6619623.10	6955564.90	6614407.70	6152815.50	5410914.60	4865559.55	4859855.80
131072	2752390.95	3800832.90	4845101.45	5733016.40	6385719.50	6432983.65	6321504.60	6531308.65	6625174.55	6296019.60	5411879.80	4870521.30	5028281.90

Table C.16: Average IOZone result for the Re-Write (UBC Enabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	297.30	351.80	316.40	340.10	345.50	341.30	364.60	361.35	367.85				
2048	393.90	577.10	603.60	529.85	611.80	614.85	610.15	604.25	635.95	623.90			
4096	670.30	796.45	1032.10	962.05	979.25	1016.60	1012.90	924.65	1001.95	1037.70	1013.70		
8192	790.80	1129.80	1045.95	1255.30	1352.10	1407.10	1297.95	1487.15	1408.75	1396.45	1512.75	1511.65	
16384	950.55	1576.60	1485.75	1478.00	1307.25	1701.70	1583.95	1861.30	1823.75	1852.15	1975.60	1944.00	1927.80
32768	1813.35	1801.70	1681.15	1889.25	2018.10	1878.30	1555.00	1463.60	1785.20	1973.35	2244.30	2246.35	2241.95
65536	2161.80	2238.35	2222.40	2270.45	2151.90	2217.10	2067.20	2132.10	2458.45	2100.35	2089.00	2252.80	2468.05
131072	2431.10	2374.45	2644.10	2516.40	2446.20	2640.50	2580.90	2403.10	2707.40	2685.05	2668.35	2546.30	2749.80

Table C.17: Average IOZone result for the Random read (UBC Enabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1498219.35	2344720.10	3080303.00	3874251.80	4546407.40	4778661.85	4611515.90	4243269.90	4508479.25				
2048	1800314.95	2812802.40	3729273.65	4378568.75	5075371.95	5407484.80	5551934.35	5725692.15	5984930.55	5573431.05			
4096	1971236.55	2903321.05	4094762.95	5078558.45	5839001.05	5855868.35	6184707.15	6630274.45	6446722.35	6375918.70	5733987.10		
8192	2040968.60	3170625.25	4221714.50	5324730.40	5978322.80	6486254.55	6729310.35	6793057.35	6910282.85	6789926.60	6124171.80	5207470.20	
16384	2010636.15	3066886.05	4316947.00	5425928.75	6122569.50	6671971.70	6981686.60	7187780.90	7101490.75	6487911.60	6119170.75	5490654.85	5417628.70
32768	2013556.95	3076798.35	4305443.75	5440444.15	6313165.10	6398931.50	6891871.60	6941467.90	6807632.00	6757647.20	5765157.05	5368672.90	5239131.60
65536	1883805.55	2953573.30	4187946.40	5012295.80	6032576.60	6489121.25	6400092.40	6779466.50	6232960.65	6026053.85	5579952.65	4715031.90	4734257.00
131072	1856943.80	2848758.25	3947984.90	4897471.50	5911764.15	5987880.05	6218885.00	6457690.20	6356603.70	6231786.35	5434549.30	4894964.70	4866498.50

Table C.18: Average IOZone result for the Random write (UBC Enabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	348.10	351.75	370.75	374.20	377.00	378.50	370.45	370.15	372.35				
2048	662.25	708.40	579.50	652.75	634.55	650.30	643.75	649.10	657.85	653.95			
4096	1186.00	1095.75	1204.55	1034.75	1043.30	1062.30	1019.70	1059.10	1026.25	1046.70	1010.25		
8192	1506.70	1840.30	1622.85	1770.50	1460.85	1517.70	1520.10	1451.50	1514.80	1499.35	1480.70	1455.85	
16384	1748.20	2067.60	2693.80	2193.50	2493.15	1830.85	1821.00	1942.95	1813.80	1950.70	1969.80	1961.90	1916.80
32768	1877.20	2153.90	2545.80	3341.40	2605.50	3078.20	2239.55	2230.65	2227.05	2256.85	2270.35	2234.70	2254.95
65536	2195.00	2014.25	2401.05	2830.90	3848.30	2809.55	3456.60	2479.70	2426.20	2444.55	2465.85	2452.35	2507.90
131072	1965.70	2408.40	2266.60	2666.20	3196.10	4480.90	3250.05	4070.15	2774.55	2770.90	2750.20	2767.95	2794.55

Table C.19: Average IOZone result for the Read (UBC Disabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	80743.95	151027.25	286239.80	506581.55	813810.35	1083700.05	1451416.80	1764857.50	1099304.00				
2048	84472.50	160897.35	295229.50	542695.30	878663.35	1259390.65	1767767.65	2111526.40	2177889.05	1122615.05			
4096	83218.95	155104.85	281463.95	483238.75	772460.15	1184896.00	1518398.95	1972338.80	2141372.80	1937105.95	1234557.25		
8192	82581.85	158449.00	283701.30	463949.35	677695.30	1143571.55	1462300.25	1678279.15	1915368.05	1644113.80	1358144.35	924246.15	
16384	72387.80	131468.20	244971.65	406282.55	674330.35	1004046.80	1536310.65	1776046.45	1915062.95	1529397.30	1322474.95	1154578.60	715355.80
32768	82302.90	161231.35	298478.50	519691.15	854422.10	1306319.90	1875572.30	2296575.75	2426731.50	2092502.15	1692319.65	1189270.55	1164936.00
65536	85614.55	162883.50	292600.45	508783.95	828805.60	1310862.00	1866527.55	2262690.90	2352691.25	2090679.30	1906822.80	1474047.30	1160148.95
131072	81031.50	151053.35	273604.95	463483.25	740790.20	1067333.50	1490251.40	1886744.90	1601649.00	1468598.60	1497675.80	1208364.15	1231383.40

Table C.20: Average IOZone result for the Write (UBC Disabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	395.65	410.40	402.80	416.80	410.25	408.65	412.25	402.85	399.80				
2048	689.00	684.80	692.80	676.95	682.55	706.05	685.50	707.25	712.85	702.45			
4096	1063.15	1092.45	1070.25	1071.55	1085.90	1072.25	1078.25	1079.25	1075.25	1099.05	1083.10		
8192	1452.10	1474.80	1484.05	1476.70	1498.90	1443.50	1491.35	1493.85	1502.20	1486.05	1503.15	1473.40	
16384	1790.90	1775.15	1798.45	1816.60	1827.75	1835.25	1825.05	1828.55	1806.45	1825.00	1828.55	1832.50	1832.20
32768	1985.15	2007.10	2033.80	2032.55	2014.20	2028.75	2011.95	2028.25	2059.25	2040.65	2039.15	2034.90	2035.45
65536	2077.45	2150.75	2139.30	2132.90	2165.40	2170.30	2173.50	2148.15	2138.10	2170.10	2180.60	2170.60	2173.85
131072	2162.60	2197.80	2204.40	2219.10	2231.90	2210.55	2234.85	2230.45	2201.80	2234.45	2232.20	2233.30	2187.40

Table C.21: Average IOZone result for the Re-Read (UBC Disabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	85268.40	159990.45	304544.20	538867.00	912389.35	1289930.65	1793401.35	2126515.55	2100925.10				
2048	86359.20	166486.30	303609.85	565188.90	951162.15	1396904.45	2035677.25	2420982.45	2436436.15	1868213.80			
4096	87407.95	168224.85	317814.30	525528.70	827677.60	1311306.70	1745219.65	2276016.85	2441605.50	2094812.40	1801952.85		
8192	87908.15	164235.80	311736.55	549546.65	852490.30	1313556.80	1893362.80	2114780.75	2339225.55	1729352.90	1415570.60	1301696.75	
16384	84526.75	162406.70	311393.50	519330.15	902256.35	1302242.95	1916211.80	2327608.80	2473838.30	1936682.70	1675515.35	1244362.75	1123252.65
32768	85160.25	163085.80	317564.50	556505.90	854359.60	1340222.30	1910694.15	2380576.25	2484048.20	2178840.15	1649978.60	1407143.40	1386830.95
65536	87085.15	167655.70	306860.25	531388.95	843007.55	1298604.80	1853069.60	2293608.55	2223202.40	1993357.75	1687091.70	1366902.50	1297829.20
131072	86411.35	166381.75	304276.60	557006.15	951134.50	1348090.50	1791949.65	2145707.65	2274332.25	1791739.75	1727913.90	1422605.10	1350273.60

Table C.22: Average IOZone result for the Re-Write (UBC Disabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	297.90	346.35	356.70	342.95	355.80	353.85	361.70	352.50	357.70				
2048	466.10	558.40	580.25	581.15	600.00	599.00	592.30	589.00	601.20	588.85			
4096	664.25	895.60	909.50	915.70	884.80	885.50	900.25	905.65	892.80	900.15	914.20		
8192	825.10	1234.85	1237.45	1215.05	1206.75	1210.45	1248.95	1239.00	1238.75	1226.55	1231.30	1220.15	
16384	963.00	1426.25	1494.80	1512.75	1514.45	1506.70	1480.75	1506.30	1501.20	1486.45	1504.95	1512.80	1472.85
32768	1500.35	1552.25	1582.80	1551.30	1587.75	1572.80	1557.65	1566.70	1572.45	1589.35	1583.85	1601.35	1607.15
65536	1646.85	1682.20	1693.80	1698.15	1701.45	1692.30	1706.55	1674.55	1669.05	1689.05	1672.90	1708.55	1733.20
131072	1799.95	1822.15	1818.00	1853.40	1828.05	1849.55	1846.45	1826.95	1832.65	1848.00	1843.30	1853.60	1860.50

Table C.23: Average IOZone result for the Random read (UBC Disabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	85857.10	161870.55	305391.85	543648.00	912460.60	1305448.30	1773824.70	2161719.35	2121298.10				
2048	85476.20	166593.60	300744.45	554584.40	922604.90	1406890.55	2058157.45	2412090.45	2372391.05	1778128.35			
4096	86711.70	167619.20	323935.15	543636.95	827724.20	1285692.35	1845602.45	2261682.65	2251374.75	1994850.25	1836260.05		
8192	83547.55	163015.40	317644.85	535604.45	865652.50	1304802.55	1903792.45	2192671.35	2345047.80	2016516.45	1688288.65	1500031.35	
16384	84254.65	150324.35	314212.95	538555.00	847866.50	1318477.45	1911800.55	2332491.00	2383809.15	1940528.15	1672793.10	1414664.55	1390557.80
32768	83372.45	163730.80	300180.40	562818.20	893262.25	1281677.85	1844281.15	2283345.05	2403701.30	1984256.85	1570061.95	1326187.50	1294625.55
65536	85684.95	166333.85	297298.60	548328.50	843988.95	1257574.75	1842464.80	2156979.55	2141843.10	1923485.75	1653894.90	1375425.55	1339246.05
131072	84897.40	165122.85	298283.80	545003.00	921692.85	1383487.00	1885756.60	2215676.60	2329623.85	1836027.00	1757456.15	1416842.25	1337229.35

Table C.24: Average IOZone result for the Random write (UBC Disabled) test on GCSF in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	364.50	352.25	340.90	349.10	356.85	357.50	358.35	354.70	357.80				
2048	595.40	688.45	600.95	586.35	598.00	577.65	590.25	596.20	591.45	586.00			
4096	1148.35	997.90	1107.20	906.95	895.75	897.10	901.70	914.25	900.40	916.45	926.05		
8192	1429.55	1784.20	1422.80	1683.35	1222.25	1258.55	1232.00	1257.45	1247.25	1261.00	1268.10	1245.00	
16384	1540.20	1797.65	2486.30	1732.45	2226.35	1518.95	1509.95	1512.50	1520.55	1493.70	1519.15	1515.20	1527.50
32768	1517.00	1707.80	2025.25	2861.50	1993.10	2525.70	1635.50	1660.75	1659.70	1646.75	1638.40	1647.90	1643.70
65536	2010.45	1619.25	1834.20	2171.25	3192.45	2131.90	2773.05	1740.50	1729.60	1763.25	1763.15	1748.45	1759.70
131072	2000.45	2205.50	1733.30	1990.40	2391.10	3666.95	2330.55	3092.75	1878.95	1881.40	1879.35	1883.00	1883.15

C.3 Fejk FFS

Table C.25: Average IOZone result for the Read (UBC Enabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2053905.80	2539302.10	3383393.80	3786161.80	3844458.45	4349421.95	4453170.05	4155352.50	4814200.70				
2048	2230740.45	3054248.10	4050795.65	4625475.00	4628500.45	4795772.40	5244160.90	5523933.40	5938402.55	5611716.65			
4096	2606528.85	3710212.85	4691790.25	5358551.70	5865082.20	6372662.90	6452395.15	6508255.30	6840249.85	6341445.45	5875249.60		
8192	2916795.60	4171994.60	5387036.25	6139707.45	6609355.80	7146409.45	6889117.80	7156088.70	7206355.05	6797752.75	5983075.05	5774237.00	
16384	2877072.95	4149844.45	5273774.65	6200641.00	6907744.05	7265547.15	7151939.85	7266053.65	7332501.90	7340984.35	6507728.45	6183113.05	6388056.55
32768	2868123.75	4059486.65	5608949.45	6525358.90	7300642.00	7287387.00	7321225.50	7418608.10	7670348.00	7711066.75	6728219.60	6016822.20	5961137.15
65536	2866776.05	4271157.95	5458080.60	6548850.90	7077384.80	7437444.85	7228858.50	7363499.75	7130706.90	7361942.70	6334794.30	6096895.40	5955488.40
131072	2164945.35	3363939.80	3442850.25	5718875.05	6598102.30	5567981.60	5836827.40	7171400.50	5879688.05	7187717.95	6014182.35	4619208.75	5459310.45
262144	240372.70	98406.00	95169.95	92830.15	103052.75	98062.30	98423.40	100730.50	97695.20	106475.45	88457.30	93433.75	92872.45

Table C.26: Average IOZone result for the Write (UBC Enabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5732.75	6047.50	6359.30	6080.10	6428.45	6483.90	6486.90	6465.70	6508.25				
2048	5963.75	6268.20	6386.25	6491.90	6517.25	6567.10	6572.20	6589.65	6554.30	6603.05			
4096	6331.60	6608.95	6756.50	6888.25	6946.05	6943.75	6955.00	6923.30	6932.95	6945.25	6961.95		
8192	6924.75	7261.15	7481.15	7534.85	7587.20	7612.45	7504.65	7561.20	7612.00	7623.25	7615.85	7675.90	
16384	7453.80	7901.75	8094.20	8201.60	8258.85	8303.95	8366.50	8371.35	8373.30	8380.70	8319.70	8330.60	8347.10
32768	7825.90	8258.75	8511.30	8597.50	8709.70	8744.95	8733.60	8765.55	8768.50	8808.95	8773.20	8650.15	8709.20
65536	7977.35	8325.70	8485.20	8571.30	8610.30	8661.55	8646.45	8665.35	8742.95	8667.05	8693.80	8697.60	8748.60
131072	7906.35	8276.65	8492.50	8565.25	8548.20	8700.90	8690.00	8761.20	8737.40	8751.55	8756.65	8706.35	8749.30
262144	7910.15	8341.45	8529.50	8628.40	8693.10	8708.80	8721.65	8597.05	8757.55	8741.70	8759.75	8746.95	8689.30

Table C.27: Average IOZone result for the Re-Read (UBC Enabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2305452.00	2932742.10	4058240.65	4307567.15	4312091.50	4936664.00	5168183.70	5185401.30	6047324.20				
2048	2485492.95	3525141.65	4486162.70	5332389.65	5468022.25	5651803.15	6013147.95	6226682.55	6608024.00	6091181.75			
4096	2806484.75	4103106.60	5299952.65	6322473.75	6835677.25	7413880.65	7237925.20	7535905.90	7619806.85	7026269.30	5855951.25		
8192	2973125.10	4278478.60	5652640.80	6585893.15	7183777.55	7610220.50	7527898.00	7791789.50	7475321.05	7100639.90	5980831.10	6168642.10	
16384	2954194.25	4274452.45	5625519.35	6769092.65	7337200.40	7535192.05	7627325.60	7865444.80	7884355.45	7860425.95	6952371.50	6329154.75	6328256.85
32768	2938234.30	4296752.90	5731316.85	6779793.45	7518051.15	7635633.05	7671104.10	7748317.35	7908527.25	7936942.80	6987757.45	6143942.50	6171004.05
65536	2946454.80	4455721.95	5653646.95	6758636.55	7224035.15	7700027.25	7509545.45	7616961.55	7785506.75	7710750.00	6780523.55	6474695.35	6134818.65
131072	2982629.20	4572166.70	5898049.05	7128817.35	7845719.25	8370901.25	8276651.10	8243353.30	8472134.70	8317606.30	7656660.05	6647098.20	6581188.20
262144	3158912.25	4682523.90	6197964.35	7669311.20	8294974.10	8732823.75	8468414.65	8629828.55	8839237.95	8620108.30	7529952.55	6523919.85	6446805.60

Table C.28: Average IOZone result for the Re-Write (UBC Enabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5046.35	5312.25	5598.40	5755.60	5720.95	5697.75	5826.10	5758.70	5866.70				
2048	5305.40	5529.15	5662.70	5748.00	5746.75	5791.10	5810.35	5823.40	5816.55	5820.75			
4096	5824.90	6018.20	6200.30	6264.15	6302.70	6316.40	6337.50	6322.25	6307.25	6353.65	6349.35		
8192	5906.65	6079.20	6183.65	6230.50	6356.25	6297.45	6228.20	6320.20	6382.15	6371.95	6418.30	6430.00	
16384	6396.70	6578.05	6702.20	6823.40	6853.30	6902.40	6909.00	6905.20	6904.75	6876.75	6907.75	6925.75	6908.25
32768	6670.85	6938.95	6991.30	7159.40	7199.85	7218.45	7218.75	7235.95	7258.20	7270.70	7194.50	7180.60	7266.95
65536	6613.05	6865.60	7008.95	7076.00	7123.40	7131.80	7118.25	7146.95	7134.15	7178.05	7166.95	7200.70	7144.15
131072	6583.25	6873.20	6976.40	7051.55	7083.15	7111.50	7088.25	7144.60	7162.75	7183.75	7185.40	7168.10	7199.15
262144	6665.25	6911.50	7044.35	7124.45	7141.95	7161.55	7206.25	7223.85	7216.75	7203.65	7159.80	7140.40	7156.60

Table C.29: Average IOZone result for the Random read (UBC Enabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1774803.30	2595069.55	3735685.20	4296393.40	4341152.15	4913053.50	5296167.30	5226629.50	5960617.40				
2048	1901002.80	2954528.85	4033953.95	5044630.25	5343873.75	5827565.25	6137783.30	5931513.90	6226279.30	5784525.85			
4096	2104227.75	3361164.70	4689293.95	6022811.00	6991347.70	7603913.60	7488389.65	7234674.95	7260245.60	6886380.05	6148741.80		
8192	2138500.50	3336886.05	4619806.75	5782391.40	6791296.85	7348983.30	7440444.85	7651820.80	7371996.10	7115177.40	6440823.05	6232928.00	
16384	2115242.60	3326664.65	4670488.15	5781144.35	6881501.20	7445507.60	7599383.65	7723636.90	7849214.05	7987016.40	6970906.25	6302277.90	6415391.40
32768	2069329.85	3335832.25	4708856.40	5930084.25	6855779.70	7286918.70	7503921.90	7624453.45	7798985.95	7948039.90	6966240.20	6181383.95	6193804.45
65536	2041991.75	3352255.40	4606963.85	5910156.25	6799799.20	7392060.60	7454789.50	7619963.60	7813400.00	7687252.65	7071707.90	6472735.95	6173460.70
131072	2042864.50	3356982.65	4787906.60	6095148.70	7161643.90	7952374.20	7724725.55	8146952.95	8530152.30	8246080.30	7352557.20	6678971.45	6590929.65
262144	2120727.30	3435805.70	4905959.40	6604273.70	7590100.45	8251450.80	8333189.50	8570188.05	8717575.85	8641487.25	7451945.90	6473790.75	6423435.05

Table C.30: Average IOZone result for the Random write (UBC Enabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5053.90	5380.20	5430.05	5774.60	5710.95	5846.45	5791.35	5816.25	5820.00				
2048	5346.30	5563.65	5643.30	5780.95	5833.45	5828.05	5837.30	5837.55	5876.15	5865.00			
4096	5815.50	6062.35	6215.40	6303.15	6337.35	6319.55	6361.05	6338.35	6351.60	6357.70	6316.00		
8192	5954.60	6069.50	6161.10	6291.30	6337.40	6326.35	6338.60	6346.50	6367.90	6394.55	6382.65	6420.65	
16384	6411.35	6597.75	6739.20	6840.55	6837.95	6903.85	6881.45	6890.45	6942.15	6955.10	6914.10	6926.55	6913.00
32768	6635.65	6938.45	7087.85	7123.05	7140.35	7230.90	7231.95	7269.45	7257.30	7275.55	7210.85	7218.45	7255.65
65536	6628.55	6874.35	6973.95	7090.95	7092.65	7136.10	7116.85	7145.60	7114.35	7162.60	7108.05	7126.15	7134.45
131072	6603.65	6864.35	6971.80	7050.65	7089.45	7133.95	7126.45	7118.30	7171.15	7137.30	7199.20	7160.45	7114.90
262144	6630.05	6914.15	7039.85	7136.05	7149.45	7174.30	7185.90	7210.85	7195.25	7188.60	7158.10	7164.20	7175.40

Table C.31: Average IOZone result for the Read (UBC Disabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	30419.75	44672.00	51621.20	53212.60	53274.00	58568.75	59076.45	56677.95	59868.30				
2048	32825.70	40404.30	46086.25	51673.50	54435.80	56900.30	59390.45	64267.30	71501.70	83509.20			
4096	39425.65	51184.20	61897.60	69619.90	74458.30	75405.15	76497.45	78426.95	80623.85	82058.05	81435.65		
8192	41509.75	51924.40	61331.55	67262.95	73866.65	77051.10	80476.30	81639.40	83382.90	83765.70	84339.50	83527.20	
16384	41595.35	54558.90	62923.05	70040.55	74605.25	78634.15	80427.90	82097.15	81818.00	83879.30	85058.50	85970.60	83405.55
32768	42081.55	54954.15	64640.05	71847.95	73857.65	77852.15	77999.05	81063.50	82459.60	83983.30	84892.85	82659.20	84252.20
65536	42323.65	54661.65	64512.65	71136.85	74281.15	78263.50	82520.05	81725.05	83369.20	84817.70	85815.15	84726.60	84730.00
131072	42537.55	55126.15	66255.15	72610.50	76203.55	78849.80	81069.20	83536.90	84602.80	85896.00	85497.75	84892.25	86268.35
262144	42618.80	55843.50	65619.50	73275.80	75992.95	78603.15	83328.20	85174.25	86439.10	86450.55	86520.15	86281.00	86118.00

Table C.32: Average IOZone result for the Write (UBC Disabled) test on FFFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5778.95	5962.25	5672.55	6272.45	6254.95	6393.60	6494.25	6473.00	6372.30				
2048	5806.55	6184.95	6342.20	6455.55	6469.95	6441.80	6561.40	6558.55	6544.10	6596.20			
4096	6308.10	6622.20	6762.55	6876.10	6902.55	6930.80	6891.70	6897.20	6893.30	6903.90	6887.90		
8192	6907.05	7135.00	7393.90	7459.20	7532.60	7602.40	7615.65	7593.70	7576.40	7591.35	7503.60	7561.30	
16384	7435.95	7819.60	8052.75	8121.95	8240.40	8236.10	8306.80	8348.60	8360.50	8297.90	8308.95	8297.65	8275.70
32768	7781.30	8258.65	8413.35	8430.65	8569.75	8619.40	8656.40	8672.45	8657.05	8711.80	8768.50	8723.20	8711.50
65536	7928.55	8216.35	8424.45	8561.20	8634.05	8630.90	8666.25	8646.15	8636.30	8698.45	8682.90	8668.30	8703.30
131072	7923.60	8333.35	8474.95	8600.00	8628.80	8626.25	8748.60	8711.30	8778.30	8712.95	8762.95	8732.60	8753.60
262144	7927.75	8252.50	8475.85	8586.95	8584.65	8702.35	8665.95	8718.45	8734.00	8709.90	8716.90	8720.55	8747.75

Table C.33: Average IOZone result for the Re-Read (UBC Disabled) test on FFFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	30844.70	45244.20	54686.80	55328.10	53664.60	60251.20	56594.10	59996.35	60738.80				
2048	38955.05	46692.40	56222.40	67984.70	74293.00	74375.90	79505.75	82267.55	82098.60	82472.30			
4096	41713.40	53583.55	64252.20	71889.10	75817.70	77360.80	79319.70	80001.40	80210.40	82446.40	82895.05		
8192	41532.60	52523.55	59326.05	69463.35	74778.05	79911.80	82387.45	83909.30	84179.55	83896.30	85541.95	83349.65	
16384	42074.20	53867.45	64114.10	70651.90	75330.55	78378.25	82953.80	84549.50	84365.55	83599.35	86637.55	85451.70	84967.35
32768	42284.90	55255.95	63858.10	71707.35	74991.70	78019.25	78690.20	83317.40	83917.35	85329.85	86912.55	82363.60	85787.95
65536	42074.00	55796.85	64571.00	72881.55	75517.85	78523.35	80813.25	84419.75	84512.50	84694.40	84992.90	85112.35	85237.95
131072	42687.00	55229.15	65547.45	72619.60	75243.05	78705.75	82551.35	84600.35	84870.35	86345.75	86005.55	85820.45	84728.15
262144	42621.70	55880.90	66546.90	71967.80	75787.15	78842.30	83155.05	85087.35	85468.35	85807.45	85434.60	85062.10	84389.25

Table C.34: Average IOZone result for the Re-Write (UBC Disabled) test on FFFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	4979.95	5423.60	5721.00	5798.65	5659.25	5787.80	5803.40	5732.00	5820.65				
2048	5276.30	5456.45	5615.05	5715.20	5752.55	5758.70	5767.45	5773.60	5802.75	5799.45			
4096	5831.90	6002.25	6120.55	6227.00	6230.00	6172.60	6257.45	6269.90	6309.00	6302.20	6292.70		
8192	5877.95	6035.65	6168.25	6236.25	6275.20	6325.75	6296.70	6349.90	6367.40	6391.45	6388.55	6384.80	
16384	6269.50	6544.00	6613.60	6698.45	6799.45	6732.35	6861.80	6842.05	6860.35	6879.40	6822.50	6853.75	6878.50
32768	6579.00	6884.70	6999.60	7054.50	7128.35	7183.80	7112.20	7205.30	7204.05	7224.15	7234.30	7092.40	7261.05
65536	6619.20	6773.95	6988.05	7066.50	7048.75	7141.00	7167.25	7123.15	7081.70	7115.30	7133.70	7167.15	7142.15
131072	6577.90	6834.65	7001.30	7067.75	7090.20	7123.80	7120.55	7171.35	7147.15	7149.35	7138.80	7101.65	7112.45
262144	6615.45	6847.20	6978.25	7078.65	7092.95	7085.15	7150.40	7172.60	7160.70	7125.70	7168.50	7156.35	7175.35

Table C.35: Average IOZone result for the Random read (UBC Disabled) test on FFFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	20353.15	25004.40	26847.35	27381.60	27774.45	28546.50	28962.00	29045.90	31646.75				
2048	25476.25	29633.35	32122.50	35245.95	36146.25	37036.15	37981.80	38644.00	37934.20	38034.75			
4096	25942.95	29960.10	32952.20	34366.10	36099.45	36919.50	37457.55	37687.55	37821.45	38220.30	38526.35		
8192	19475.95	21361.10	22933.45	23740.75	24741.10	25350.60	25768.85	26203.15	25929.60	25451.15	26269.45	26009.30	
16384	20225.55	22927.40	24158.90	25240.00	25565.25	26373.05	26954.00	27306.95	27250.55	27206.40	27281.85	26994.60	27027.65
32768	20237.80	22734.50	24036.70	25158.55	25698.15	26242.50	26017.55	26837.70	26931.25	26937.75	27045.40	26809.20	27143.70
65536	19616.85	22213.30	23718.60	24750.00	25090.00	25268.05	25728.15	25936.95	25884.45	25801.60	25926.60	26141.45	26171.80
131072	19502.65	21605.15	23374.45	24093.10	24507.10	24661.25	25192.75	25566.30	25512.65	25615.30	25617.75	25598.40	25384.65
262144	19573.60	22179.75	23718.30	24585.50	24996.10	25299.80	25757.25	25814.85	25827.80	25916.50	25884.35	25996.50	25922.55

Table C.36: Average IOZone result for the Random write (UBC Disabled) test on FFFS in kilobytes per second

File size (kB)	Buffer size (kB) 4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5494.30	5746.25	5821.60	5960.00	5931.55	6100.70	6099.55	6177.95	6274.85				
2048	5630.00	5774.40	5948.15	5998.75	6077.35	6051.35	6114.90	6126.20	6140.15	6184.25			
4096	5937.95	6153.90	6294.45	6363.60	6371.30	6333.25	6451.05	6424.30	6443.50	6438.40	6459.50		
8192	5914.10	6151.35	6262.20	6274.65	6403.45	6439.05	6510.30	6505.40	6468.20	6365.65	6399.60	6468.95	
16384	6373.60	6619.00	6738.20	6807.80	6752.90	6893.25	6949.00	6921.70	6944.65	6933.20	7008.30	6953.60	6958.35
32768	6554.80	6872.25	7020.20	7092.10	7130.10	7170.70	7111.75	7228.80	7236.70	7234.90	7222.95	7185.75	7231.25
65536	6599.70	6840.25	6982.75	7085.35	7092.70	7137.85	7127.25	7162.00	7193.50	7149.75	7146.80	7222.35	7187.25
131072	6587.90	6804.35	6963.40	7028.40	7080.50	7112.60	7142.20	7182.80	7151.40	7195.35	7163.50	7166.90	7154.20
262144	6560.20	6865.00	7015.75	7083.80	7087.75	7133.55	7155.10	7178.45	7153.80	7164.50	7146.35	7157.10	7175.45

C.4 APFS

Table C.37: Average IOZone result for the Read (UBC Enabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB) 4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2505142.75	3947423.75	5029464.05	6325459.70	6664831.75	6852046.75	7263616.85	7253148.45	6932969.05				
2048	2833344.60	3914270.05	5141140.60	5673971.90	6774919.00	6726154.25	6984071.85	7284814.80	7509658.40	7968662.75			
4096	2835075.80	4303843.35	5695634.50	6871394.20	7551409.55	7853567.70	7767625.75	7949528.95	8200171.85	8010943.40	6953058.80		
8192	2943388.50	4350073.30	5699950.60	6948211.40	7876780.45	8109326.55	8114319.00	8387961.55	8416383.10	8387703.70	7203638.30	6433162.80	
16384	2929742.95	4393358.80	5855787.25	7101022.45	8082983.50	8385611.90	8270546.30	8448977.10	8423945.65	8456770.05	7678318.45	6678421.65	6536073.00
32768	2973698.40	4498428.05	5980895.60	7367903.30	8199695.80	8574441.30	8452076.85	8623497.50	8757444.30	8660054.85	7745995.65	6758706.55	6586635.60
65536	2966873.60	4451561.10	5934394.45	7386644.90	8200989.95	8542560.60	8289522.50	8544421.50	8548165.90	8644598.05	7689359.80	6679236.95	6548192.15
131072	2928170.35	4452142.80	5952344.05	7309468.05	8141994.00	8394364.05	8364151.40	8391235.40	8578182.45	8544360.65	7754364.95	6727926.65	6640597.75
262144	3027381.25	4525555.15	5992235.60	7241097.80	8092614.55	8420552.15	8318017.55	8461557.95	8481746.85	8403300.50	7720025.60	6738606.30	6640925.55

Table C.38: Average IOZone result for the Write (UBC Enabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB) 4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	186099.20	310206.50	500742.85	691573.75	935191.05	1122654.65	1127029.10	1169234.30	1202556.30				
2048	184251.70	302676.50	508600.40	726714.35	934738.75	1162493.30	1311508.30	1279102.70	1352761.45	1362628.35			
4096	194132.25	306174.20	431809.60	598131.15	623853.35	671691.15	744617.55	739939.70	753265.30	738848.25	802041.90		
8192	196823.20	297478.25	413518.60	516152.35	609602.70	609721.65	607386.50	597566.30	560469.25	602355.45	647730.95	749220.30	
16384	206557.40	316786.30	441570.85	560214.85	624154.95	621822.10	629365.05	612058.60	617749.00	626769.00	642424.40	694424.80	759194.25
32768	215381.65	314492.80	451145.35	488510.40	622827.55	705854.65	627444.15	643092.10	728792.00	574343.10	724316.70	706725.60	689653.60
65536	225914.10	325867.55	443216.65	538706.60	645421.75	679784.70	649012.70	684130.70	659376.30	624667.85	649335.50	606638.50	694810.35
131072	212762.95	316470.50	409990.65	522267.10	580831.60	609476.00	596332.00	620530.80	608777.95	639222.15	626478.70	640536.45	605587.30
262144	237669.10	333715.15	425258.25	468231.85	537136.10	514724.30	563943.15	614177.95	572262.70	502432.65	645917.90	603768.75	548925.85

Table C.39: Average IOZone result for the Re-Read (UBC Enabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB) 4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2864913.00	4457254.90	5899133.70	7520212.00	8208251.10	8937225.85	9151623.90	8912770.45	7666827.95				
2048	3126268.60	4357993.25	5854421.60	7087637.45	8392397.95	8623002.85	8909731.45	9401073.50	9652936.30	10791022.25			
4096	3023676.90	4653007.35	6001682.65	7347650.75	8225070.90	8561534.95	8881377.35	9230862.50	9298187.95	9259088.15	7392544.85		
8192	2992085.60	4452976.05	5911533.00	7308326.65	8303622.00	8564211.10	8573562.90	8835664.65	8762633.05	8949920.30	7515508.55	6489744.00	
16384	2915062.65	4472376.30	5958909.70	7161980.70	8272757.10	8655521.70	8517193.25	8608849.00	8730536.60	8611689.35	7816617.00	6650463.50	6548628.15
32768	2989633.05	4495206.55	5987309.95	7445322.25	8353319.40	8788971.15	8591265.95	8809024.95	8909988.85	8718320.20	7601697.60	6682304.55	6631451.00
65536	2964321.65	4487189.10	5994709.00	7370460.75	8192883.45	8450360.45	8252873.05	8528504.40	8509413.90	8470845.25	7679268.90	6692979.75	6567885.35
131072	2952983.05	4463693.05	5940194.30	7165623.30	8122873.50	8412316.55	8376695.00	8436897.55	8562218.25	8423819.75	7875330.10	6790054.05	6622086.40
262144	3041684.50	4562805.85	5907206.85	7240804.75	8180966.95	8372326.40	8298822.60	8477387.85	8476310.90	8340749.05	7720895.10	6586778.55	6426962.35

Table C.40: Average IOZone result for the Re-Write (UBC Enabled) test on APFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	592265.75	964630.95	1241391.00	1655315.30	2196691.05	2485202.45	2685085.00	2443023.25	2354507.65				
2048	695877.30	1039077.90	1477724.90	1847796.10	2252878.40	2422239.00	2557725.30	2504708.10	2704999.15	2409399.50			
4096	521157.75	708622.25	864388.05	1039198.15	1106422.75	1134042.45	1113623.40	1135161.15	1172232.65	1030819.85	1132413.45		
8192	453490.30	588686.45	698455.55	777957.85	841099.70	867889.50	886578.65	892233.55	893457.45	878194.20	849966.00	852271.20	
16384	453524.55	597639.00	714710.30	773620.05	811933.20	830448.00	816822.30	832619.40	826439.85	819958.65	817466.05	807248.40	801283.60
32768	457456.80	592106.05	721793.95	781742.20	796979.20	804587.25	805568.50	804844.35	794026.85	801556.05	796794.25	792515.35	789547.95
65536	459039.25	595846.70	723756.85	779975.30	784512.80	790169.15	781677.80	771490.40	774523.50	766627.30	740691.95	748889.60	779504.60
131072	450188.50	588480.85	696290.75	750173.75	773099.70	756497.75	760731.85	759690.60	769162.60	764463.00	758248.65	760929.45	755230.30
262144	447285.75	581555.20	668141.10	651799.85	713612.80	744931.20	731429.95	732824.55	677986.50	715258.15	715275.60	690793.00	724471.65

Table C.41: Average IOZone result for the Random read (UBC Enabled) test on APFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2074232.85	3326397.00	4819859.70	6424618.10	7434418.40	8548853.85	8494377.30	8587693.75	8912411.80				
2048	2296427.45	3566392.10	5092423.85	6789020.95	8519752.00	8619856.40	9557681.60	10224323.10	10546242.60	10512351.85			
4096	2202085.60	3612580.40	5262798.25	7355692.55	8760353.00	9359305.85	9577690.80	9899693.90	9710620.95	9263448.40	7445681.65		
8192	2098551.40	3403879.10	4834552.15	6450319.15	7675319.40	8292786.70	8515363.85	8781027.50	8755866.60	8618139.00	7706597.60	6433761.95	
16384	2083004.15	3315526.95	4781180.90	6211346.70	7521180.15	8175518.25	8399519.10	8436323.05	8694983.70	8681158.35	7840220.05	6646525.90	6560844.55
32768	2095987.45	3384059.10	4866934.40	6319699.45	7471846.85	8105607.85	8136492.05	8380844.55	8503627.70	8358723.25	7674717.05	6698855.95	6624042.85
65536	2049908.55	3326376.75	4766198.35	6258631.85	7468234.35	8061158.70	8042111.85	8422183.00	8463943.70	8437504.75	7657744.60	6676710.75	6582147.55
131072	2022670.20	3296024.40	4603002.50	6118507.35	7374244.20	8020616.65	8189100.10	8366518.95	8526327.55	8335670.05	7843343.25	6675994.40	6492964.50
262144	2066259.35	3342684.30	4676294.00	6152717.05	7427353.85	7945244.95	8094863.35	8310994.75	8353960.70	8180888.85	7481651.75	6444912.20	6336362.60

Table C.42: Average IOZone result for the Random write (UBC Enabled) test on APFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	649922.10	1037171.95	1654836.35	2157718.95	2662083.25	3160374.95	3325716.40	3298368.65	3487067.10				
2048	651957.35	1083515.35	1654384.05	2215889.45	2812167.20	3066118.40	3814475.00	3583517.35	3728142.60	3564031.50			
4096	505476.95	680253.05	873951.05	983493.20	1343542.30	1513499.70	1394845.50	1194666.05	1260494.95	1136762.50	1152796.60		
8192	526225.60	762757.50	997478.45	1165187.20	1333584.00	1376043.65	1382063.10	1414744.85	871433.60	866443.60	839324.05	825390.35	
16384	424059.40	603453.55	730686.85	825685.40	887560.30	926906.20	932126.80	940152.25	937466.85	935266.90	987616.75	797618.40	792772.55
32768	414258.15	530101.60	650144.85	723884.05	766565.05	793231.50	800877.10	806011.00	807396.85	802958.55	788915.10	788193.40	785617.80
65536	403399.30	511380.00	601343.70	670271.70	715757.10	739347.70	744645.50	754563.00	754500.05	735796.55	727383.65	775893.40	769747.15
131072	410830.60	500099.50	571384.60	644571.95	673555.30	695946.55	700937.25	708369.50	712985.75	707357.10	708849.05	745366.15	752575.30
262144	405562.85	498175.85	558733.90	586705.45	631143.25	669489.80	687256.15	682657.65	658771.60	673667.05	661071.85	712207.10	729524.00

Table C.43: Average IOZone result for the Read (UBC Disabled) test on APFS in kilobytes per second

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	210162.80	204940.30	375178.00	365138.45	424292.90	477905.50	569549.00	888248.20	6737824.85				
2048	395919.85	402611.50	553508.50	506011.35	592675.40	583652.35	587972.80	653102.55	1009099.00	7849131.15			
4096	487559.50	556467.65	630018.60	661954.10	698762.45	667385.95	710769.25	726212.75	879588.55	1152974.55	6411991.45		
8192	546887.55	656527.65	779556.95	759872.55	777149.50	788052.25	824191.85	807671.60	909998.20	1091562.25	1411122.50	5726351.05	
16384	603024.85	747491.70	822830.30	815794.25	827174.95	852701.80	860714.45	862685.25	882009.00	959223.25	1100461.45	1469370.40	5832347.20
32768	660798.50	833326.15	856073.95	862990.55	860245.25	868789.90	876686.20	876089.60	913467.05	938946.50	994758.30	1110264.70	1494454.60
65536	648648.55	832282.95	872285.10	873565.50	856936.15	876632.35	882941.55	882018.10	899152.45	899622.10	937166.65	983578.00	1095303.75
131072	744496.55	840599.95	884439.15	883303.00	883343.15	880956.15	872050.20	860533.10	871110.50	867685.40	888025.80	915005.60	978307.30
262144	807401.45	856593.65	888093.65	886135.30	886924.05	866637.30	864613.80	867406.80	869593.00	878919.55	894023.30	905508.50	933751.05

Table C.44: Average IOZone result for the Write (UBC Disabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	47568.70	66769.55	29464.65	61808.25	98247.30	151795.10	151700.95	208201.20	313752.65				
2048	60086.05	78330.65	49778.10	63849.95	117504.10	173844.60	217935.05	267242.35	297741.30	466384.45			
4096	61802.95	78434.80	54520.25	78999.30	133031.80	199036.00	261045.75	324871.45	390262.95	430824.65	637078.85		
8192	60834.75	79027.45	64656.20	78220.35	141553.05	201778.55	297869.60	393207.20	442945.60	523357.65	584349.40	712604.30	
16384	61802.40	99205.45	68267.80	86443.25	145367.20	256190.70	379475.25	448257.90	506044.35	589824.10	638260.30	664499.05	745686.10
32768	66331.55	81487.15	73141.15	109369.20	182713.20	241762.55	369557.20	444235.15	513281.10	602202.05	676506.10	587290.40	666938.75
65536	66991.30	86726.80	73195.20	104151.35	189070.50	291313.50	396955.95	461863.35	529716.85	496979.60	679930.05	571764.65	673438.90
131072	111927.30	121063.45	97750.65	133631.15	206144.90	299445.95	378344.35	431569.25	491322.00	470867.60	594773.85	549299.15	579586.60
262144	173656.20	192284.10	124095.70	168635.40	233643.45	317196.40	395638.40	436893.80	488395.65	442663.10	497117.25	555677.35	565397.95

Table C.45: Average IOZone result for the Re-Read (UBC Disabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2847610.40	4620067.25	6607965.85	6953134.20	8472011.05	8184672.55	7050857.60	8604032.80	10839244.35				
2048	3004105.15	4616355.30	6368285.10	7062356.90	8911666.05	8701480.55	8797422.30	8674003.55	9936075.65	10604197.90			
4096	2851221.70	4509978.50	6116509.75	7500195.45	8843072.35	8479006.95	8766581.30	8990328.40	8657705.65	7972926.45	7066304.15		
8192	2763081.45	4204973.00	5491018.60	6604874.65	7353053.05	7556045.45	7939296.35	7934525.50	7746331.10	7599873.25	6541789.25	5762974.30	
16384	2786032.50	4030290.45	5265094.75	6382649.20	7289525.80	7316852.60	7572734.35	7690694.50	7547413.10	7317534.30	6661005.75	6011373.20	5813492.50
32768	2759381.05	4103193.60	5335380.65	6487595.15	6908663.95	7313190.65	7275915.60	7422447.55	7468387.75	7322302.35	6763725.85	5865869.55	5901481.40
65536	2744742.65	4011001.00	5259717.20	6406725.15	7095102.75	7245325.65	7244637.80	7340606.55	7457131.65	7308501.55	6671333.95	5992961.25	5823927.45
131072	2698561.50	4093971.00	5296503.45	6397058.40	6949727.70	7130463.30	7165734.10	7281714.20	7293742.50	7303007.60	6667841.70	5919628.30	5888190.30
262144	2766820.50	4069090.70	5298685.85	6395880.00	6936945.95	7198925.80	7194783.20	7293669.35	7456221.90	7454899.85	6588468.20	5785781.25	5867765.55

Table C.46: Average IOZone result for the Re-Write (UBC Disabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	245900.40	330657.40	118808.35	132746.70	208724.40	297512.30	368018.20	462185.60	516126.35				
2048	316826.50	424364.85	169094.80	117463.55	253776.60	327250.60	389710.35	474905.85	540097.80	623727.90			
4096	315595.15	438319.45	195872.65	198354.45	292248.25	370950.90	456799.15	545932.30	616472.20	674190.65	719018.80		
8192	309646.30	425562.15	202478.75	213011.95	322773.85	397713.55	522780.15	600654.45	659018.60	707259.35	737665.15	756790.45	
16384	318542.55	437192.20	204751.85	236537.20	358947.85	444657.85	535197.70	607098.40	658229.65	710417.70	741951.45	757908.70	765713.00
32768	321160.15	460764.65	204298.50	267114.45	363280.65	466395.10	532141.35	606749.40	662896.25	710791.20	742240.05	758405.45	766172.15
65536	318093.20	459221.15	203724.45	289560.30	348408.35	471241.75	537136.50	607085.85	664632.40	711611.85	711446.20	740760.80	717320.80
131072	346187.55	464512.65	202786.45	272076.35	359091.70	469873.60	533441.00	590505.75	633628.45	630096.55	607097.10	598912.65	598818.80
262144	350057.45	465233.75	198914.80	287256.40	358911.90	445592.65	498986.40	565630.30	545542.35	543569.15	622449.35	637520.95	649420.85

Table C.47: Average IOZone result for the Random read (UBC Disabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2095940.65	3580778.80	5686306.95	6756560.05	8162409.75	8261574.45	7343689.85	9352461.45	10622540.60				
2048	2203459.40	3673152.65	5437650.15	7154341.00	8709548.80	9576915.30	10085946.95	10018981.25	10566756.55	10468927.30			
4096	2097863.30	3570118.55	5186250.80	6913126.80	8586345.65	8940698.60	9345523.05	9363830.15	9009101.75	8587695.80	7052459.65		
8192	2010445.55	3256268.40	4597801.65	5937654.35	6932842.85	7351691.70	7898870.75	7887014.35	7796732.00	7750505.45	6705852.20	5942219.35	
16384	1978281.00	3140276.95	4385971.20	5647738.00	6728942.55	7216186.50	7496026.05	7722716.95	7753236.90	7611188.40	6829920.05	6023521.65	5877012.10
32768	1943124.90	3130991.10	4418271.85	5713335.70	6443227.95	7121104.20	7265523.75	7528713.80	7601741.90	7464638.45	6816827.55	5855760.90	5941920.45
65536	1906385.95	3054880.05	4290319.80	5657852.30	6581181.30	7057154.50	7186119.75	7309873.10	7472877.80	7404166.40	6677548.50	6005450.30	5823731.95
131072	1897866.85	3053580.60	4298070.80	5597836.65	6501995.90	6960278.65	7101578.20	7307253.75	7353158.25	7346077.30	6640796.30	5872827.05	5895048.00
262144	1895270.60	2999913.85	4285513.30	5569702.05	6465773.90	6910559.10	7058880.85	7224809.60	7409227.00	7390846.00	6530883.75	5725737.45	5892876.60

Table C.48: Average IOZone result for the Random write (UBC Disabled) test on APFS in kilobytes per second

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	305117.95	381572.20	543381.35	553658.05	578917.20	581548.30	540993.65	553117.40	558749.20				
2048	330652.60	459000.55	576773.90	609399.10	647791.10	654115.35	635629.00	642297.55	637652.65	644827.75			
4096	328303.25	458436.85	554958.20	635634.00	671795.05	686156.30	690473.20	681290.05	674131.75	656198.05	630915.60		
8192	250578.50	343877.50	422886.50	473009.00	465064.65	473593.05	458371.55	453413.00	682574.65	676555.80	665270.65	658438.75	
16384	256971.95	350790.35	439785.65	475387.85	486861.60	489750.30	474627.80	482215.90	489081.30	488511.15	519289.00	706805.35	706431.85
32768	259435.95	365012.55	457671.65	491803.45	483946.30	503973.50	496493.00	504519.10	502983.60	495386.90	498140.05	741637.40	733676.55
65536	254287.50	370203.50	438379.45	490600.10	491807.90	512342.25	514387.35	509011.75	512009.95	513358.30	508490.85	755238.50	755424.50
131072	272429.50	364065.45	425462.55	477877.30	482513.60	483911.35	508464.80	510998.85	512270.20	510560.20	503272.45	622105.85	618886.15
262144	285254.20	367971.50	414396.85	434964.25	449163.90	459265.90	483362.50	496486.30	484252.35	475187.10	486534.60	624676.20	674190.40