KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

FFS: A cryptographic cloud-based deniable filesystem through exploitation of online web services

Store your sensitive data in plain sight

GLENN OLSSON

Stockholm, Sweden, 2022

FFS: A cryptographic cloud-based deniable filesystem through exploitation of online web services

Store your sensitive data in plain sight

GLENN OLSSON

Master's Programme, Computer Science, 120 credits
Date: November 10, 2022

Supervisors: Hamid Ghasemirahni, Zachory Peterson
Examiner: Gerald Quentin Maguire Jr

Host organization: Cal Poly
Swedish title: FFS: Ett kryptografiskt molnbaserat förnekningsbart filsystem genom utnyttjande av onlinebaserade webtjänster
Swedish subtitle: Lagra din känsliga data öppet

Abstract

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 using encoded and encrypted images. More data can usually be stored in image posts than in a 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 these online web services. The thesis compares the performance of FFS against two other filesystems. It can be concluded that FFS has limitations in mainly speed, making it unviable as a general-purpose usage, such as a substitute to the local filesystem on a computer. However, its portability and security makes it relevant for certain scenarios.

Keywords

Filesystem, Fejk FileSystem, Cloud-based filesystem, Steganograhpic filesystem

Sammanfattning

Sammanfattning på svenska

Nyckelord

Filsystem, Fejk FileSystem, Molnbaserat filsystem, Steganografiskt filsystem

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.

Amsterdam, The Netherlands November 2022
Glenn Olsson

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

ADD	Additional authentication data
AES	Advanced Encryption Standard
APFS	Apple Filesystem
API	Application Programming Interfaces
CED	Complete Encrypted Data
DES	Data Encryption Standard
EASCII	Extended ASCII
FFFS	Fejk Fejk Filesystem
FFS	Fejk Filesystem
FIFO	First In First Out
FOWS	Fake Online Web Service
FTP	File Transfer Protocol
FUSE	Filesystem in Userspace
GCM	Galois/Counter Mode
GCSF	Google Conduce Sistem de Fișiere
HKDF	Hashed Message Authentication Code based Key Derivation Functions
I/O	In- and output
IV	Initialization Vector
LCED	Length of Complete Encrypted Data
LRU	Least Recently Used
NIST	U.S. National Institute of Standards and Technology
OSN	Open Social Networks
OWS	Online Web Service
PBKDF	Password-Based Key Derivation Function

PCD	Pixel Color Data
RGB	Red Green Blue
RSA	Rivest-Shamir-Adleman
SHA	Secrure Hash Algorithms
SSD	Solid-State drive
VFS	Virtual Filesystem

Chapter 1

Introduction

To keep files and data secure we often use encrypted filesystems. However, while these filesystems hide the content of the data, they often do not conceal the existence of data. For instance, using snapshots of the filesystems from different moments in time, it could be possible to notice a difference in the data stored and therefore that data exists and where it is located. Snapshots could even reveal user passwords [1].

Deniable filesystems are intended to make the data deniable, meaning that the user is supposed to be able to plausibly deny the existence of data. This is often accomplished through the use of digital steganography. There are many reasons why this is important. For instance, in 2011, a Syrian man recorded videos of attacks on civilians carried out by Syrian security forces, which he wanted to share with the world [2]. By cutting his arm, he was able to hide a memory card inside the wound and smuggled it out of the country. However, if he would have used methods such as an encrypted deniable filesystem, the border control may not have been able to discover even the existence of data, even if they would have found the memory card. By only encrypting the data, the border control would have been able to see that he was trying to hide data and make him reveal the decryption key, either by legal measures or by force, which is why he smuggled it out.

There exist multiple deniable filesystems that are designed to combat this problem on physical devices, such as memory cards. However, even just carrying a memory card might subject you to suspicion of hiding data, no matter how the filesystem is designed. Another solution to hiding the data is therefore to hide it somewhere else, for instance online through the use of a cloud-based filesystem service, such as Google Drive. Someone searching your body and devices, at for instance an airport or border control, might not realize that you are using a cloud-based filesystem service to hide your data. Although, more thorough investigations of a person might reveal user accounts used on the service, leading to legal processes where the service is forced to disclose your data. Even if you encrypt the data you upload to such a service, you can still be forced to reveal the decryption keys. What we want to achieve is a combination of a deniable filesystem and a cloud-based filesystem, where the data is stored using digital cryptographic and steganographic methods but without any company or person other than the user controlling the actual data. To accomplish this, we can store the data on online social media platforms.

Social media platforms such as Twitter and Flickr have many millions of daily users that post texts and images (for example, of their cats or funny videos). According to Henna Kermani at Twitter, they processed 200 GB of image data every second in 2016 [3]. The photos posted on Twitter, as opposed to the ones stored on cloud services such as Google Drive, are stored for free on the service for the user, for what seems to be an indefinite period. There is also no specified limit on how many images or tweets one can make. Although, as stated in their terms of service, such limits can be imposed on specific users whenever Twitter wishes, and tweets can be removed at any point in time [4].

This project created a cryptographic and deniable cloud-based filesystem called The Fejk Filesystem (FFS) which takes advantage of free online web services, such as Twitter and Flickr, for the actual storage. The idea was to save the user's files by posting an encrypted version of the file as images and text posts on these web services. The intention was not to create a revolutionary fast and usable filesystem but instead to explore how well it is possible to utilize the storage that Twitter and similar services provide their users for free, as a cryptographic and deniable cloud-based filesystem. Additionally, the performance and limits of this filesystem is analyzed and compared to alternative filesystems, such as Google Drive, to compare the advantages and disadvantages of the developed filesystem compared to professional filesystems. The security of the filesystem is discussed and an analysis of the steganographic capability of the developed filesystem is presented.

1.1 Problem

Current cryptographic filesystems are mainly based on local-disk solutions, and while services such as Google Drive might encrypt your data, it can be considered unsafe storage as they might give out your data. A cryptographic and deniable decentralized cloud-based filesystem where the data is not controlled by any entity other than the user can be of importance, for instance for journalists in unsafe countries. Social media services often provide free storage which makes it a potentially good host of the data in such a filesystem as they would not be able to access the unencrypted data nor have any idea how the posts are connected, and it might even go unnoticed due to their constant heavy load of data from regular users of the services. Is it possible to exploit the storage on various social media services to create a cryptographic and deniable filesystem where the data is stored on these online web services through the use of free user accounts? What are the drawbacks of such a filesystem compared to similar filesystem solutions with regard to write and read speed, storage capacity, and reliability? Are there advantages to such a filesystem in regard to security and deniability?

1.2 Purpose and motivation

The purpose of this research is to explore the possibility to create a secure, steganographic cloud-based filesystem that stores data on Online Web Service (OWS)s and to compare the performance, benefits, and disadvantages of such a filesystem to existing steganographic filesystems and distributed filesystem services. A distributed filesystem service, such as Google Drive, provide data storage for users which can be both free and cost money. Even though Google Drive encrypts the user's data, they control the encryption and decryption keys, and the method of encryption [5]. This means that they can give out the user's files and data if faced with legal actions such as subpoenas. It also opens up the possibility of hackers gaining access to the files without the user having any way to control them.

The idea behind 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. As the file data is stored in the open but only accessible by the user, and as FFS can be unmounted to hide its existence, it is steganographic.

There are several steganographic 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 latency, they lack accessibility as you need to have the device to access the content on it. It also means that when you want to share or transport the data, you must physically move the device which can mean problems as it could for instance be taken from you or be destroyed. Cloud-based solutions counter this by being available from any location that has internet access to the services used. However, existing cloud-based solutions introduce other disadvantages. One example is CovertFS [6] where data is stored in images posted on web services. The images are actual images representing something, meaning that there is a limit on how much steganographic data can be stored. CovertFS limit this to $4\,\mathrm{kB}$ 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 less images are needed to store a file bigger than $4\,\mathrm{kB}$. It also means that the images produced by FFS do not look like a normal image, but instead has seemingly randomly colored pixels. More examples of similar filesystems will be presented in Chapter 3.

1.3 Goals

The project aims to create a secure, deniable filesystem that stores its data on online web services by taking advantage of the storage provided to its users. This can be split into the following subgoals:

1. to create a mountable filesystem where files and directories can be stored, read, and deleted,
2. for the filesystem to store all the data on online web services rather than on the local disk,
3. for the system to be secure in the sense that even with access to the uploaded files and the software, the plain-text data is unreadable without the correct decryption key,
4. to provide the user of the filesystem with plausible deniability of its data in the sense that it is not possible to associate the user with 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 Filesystem in Userspace (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 Delimitations

Due to limitations in time and as the system is only a prototype for a working filesystem and not a production filesystem, some features found in other filesystems are not implemented in 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 secure steganographic filesystem with a storage medium based on online web services and thus FFS will only aim to implement a minimal filesystem.

1.6 Structure of the thesis

Chapter 2 presents theoretical background information of filesystems and the basis of 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 filesystems

A Unix filesystem uses a data structure called an *inode*. The inodes are found in an inode table and each inode keeps track of the size, blocks used for the file's data, and metadata for the files in the filesystem. A directory simply contains the filenames and each file or directory's inode id. The system can with an inode id find information about the file or directory using the inode table. Each inode can contain any metadata that might be relevant for the system, such as creation time and last update time.

Figure 2.1 shows an example inode filesystem and how it can be visualized. The blocks of an inode entry are where in the storage device the data is stored, each block is often defined as a certain amount of bytes. Listing 2.1 describes a simple implementation of an inode, an inode table, and directory entries.

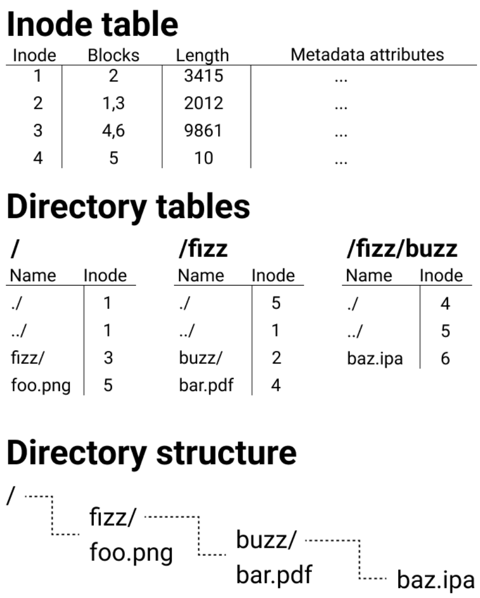

Figure 2.1: Basic structure of inode-based filesystem

Listing 2.1: Pseudocode of a minimalistic inode filesystem structure

```
struct inode_entry {
        int      length
        int[]    blocks
        // Metadata attributes are defined here
}

struct directory_entry {
        char*    filename
        int      inode
}

// Maps inode_id to an inode_entry
map<int, inode_entry> inode_table
```

Different filesystems provide different features and limitations. The Extended Filesystem (ext) exists in four different versions: ext, ext2, ext3, and ext4. This filesystem is often used on Unix systems. Each iteration brings new features and changes the limitations. For instance, comparing the two latest iterations, ext3 and ext4, ext4 can theoretically store files up to $16\,\mathrm{TiB}$ while ext3 can store files up to $2\,\mathrm{TiB}$ [7]. Additionally, ext4 supports timestamps in units of nanoseconds while et3 only supports timestamps with a resolution of one second. Additionally. ext4 natively supports encryption at the directory level through the use of the fscrypt Application Programming Interfaces (API) [8].

The Apple Filesystem (APFS) is a modern filesystem that is used on iPhones and Mac and can store files with a size up to $9\,\mathrm{EB}$ [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 $15\,\mathrm{GB}$ for free [12] using Google's clusters of distributed storage devices, meaning that the data is saved on Google's servers which can be located wherever they have data centers [13]. Paying customers can have a greater amount of storage using the service. Apple's iCloud and Microsoft's OneDrive are two additional examples of distributed filesystems where users have the option of free-tier and paid-tier storage.

Cloud-based filesystems, as opposed to a filesystem on a physical disk, are accessible from multiple computers and devices without requiring the user to connect a physical disk to the computer. Instead, as the filesystem is accessible through the internet, it can be accessed regardless of the user's location and on multiple devices, as long as a connection to the filesystem can be established. Thus, even if the user would lose their computer or if it would malfunction, the data on the cloud-based filesystem can still be accessed which means that the data could still be recovered. These filesystems are often owned by companies, such as Google Drive and Apple's iCloud, as they are big companies that can provide reliable storage. This also means that they have their agenda and policies, and as they are hosting the data they have the possibility of accessing your data. The data is often encrypted, but in the case of Google Drive, they have access and control of the encryption and decryption keys which in turn means that they have access and control of the data stored [5]. While they mention in their Terms of Service that the user retains ownership of the data [14], they also mention that they can disclose your data for legal reasons and that they retain the right to review the content uploaded by users [15]. Controlling the encryption and decryption keys also enables the possibility of hackers gaining access to your data by attacking Google. iCloud uses end-to-end encryption for some parts of the service, but not for the whole suite [16]. For instance, backup data and iCloud drive are not end-to-end encrypted while the Keychain and Memoji data are.

2.1.3 Data storage and encoding

Different file types have different protocols and definitions of how they should be encoded and decoded, for instance, a JPEG and a PNG file can be used to display similar content but the data they store is different. At the lowest level, storage devices often represent files as a string of binary digits no matter the file type (however there are non-binary storage devices [17], but this is outside the scope of this thesis). If one would represent an arbitrary file of X bytes, each byte (0x00 - 0xFF) can be represented as a character such as the Extended ASCII (EASCII) keyset and we can therefore decode this file as X different characters. Using the same set of characters for encoding and decoding we can get a symmetric relation for representing a file as a string of characters. EASCII is only one example of such a set of characters, any set of strings with 256 unique symbols can be used to create such a symmetric relation, for instance, 256 different emojis or a list of 256 different words. However, if we are using a set of words we would also have to introduce a unique separator so that the words can be distinguished. If we would use a single space character as the separator, we could make the encoded text look like a text document; however, random words one after another lead to a high probability of creating an unstructured text document. Further, if punctuation is introduced, for instance as part of some words, the text document could look like it contains random and unstructured sentences.

This string of X bytes can also be used as the data in an image. An image can be abstracted as a $h * w$ matrix, where each element is a pixel of a certain color. In an image with 16-bit Red Green Blue (RGB) color depth, each pixel consists of three 16-bit values, i.e. three pairs of bytes. One can therefore imagine that we can use this string of X bytes to assign colors in this pixel matrix by assigning the first two bytes as the first pixel's red color, the next two bytes as the same pixel's color green color, and so forth. The seventh and eighth bytes would represent the second pixel's red color. This means that X

bytes of data can be represented as

$$ceil(\frac{X}{2*3})$$

pixels, where $ceil$ rounds a float to the closest larger integer. For a file of $1\,\mathrm{MB}$, i.e. $X = 1\,000\,000$ we need $166\,667$ pixels in an image with 16-bit RGB color depth. The values of h and w are arbitrary but if we for instance want a square image we can set $h = w = 409$ which means that there will be $167\,281$ pixels in total, and the remaining 614 pixels will just be fillers to make the image a reasonable size. Using filler pixels requires us to keep track of the number of bytes that we store in the image so that we do not read the filler bytes when the image is decoded. However, we could choose $h = 1$ and $w = 166\,667$ which would mean a very wide image but would not require filler pixels. The string of bytes X is referred to as the Pixel Color Data (PCD).

This means that we can represent any file as a string of bytes which can then be encoded into text or as an image, which can be posted on for instance social media. However, there is a possibility that the social media services compress the images uploaded which could lead to data loss in the image, which would mean that the decoded data would be different from the encoded data. In this case, we would not be able to retrieve the original data that was stored unless we would use methods such as error-correcting codes. The error-correcting codes would have to be stored in a ensured lossless format, for instance, as a text post on an OWS.

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 written with 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.

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.

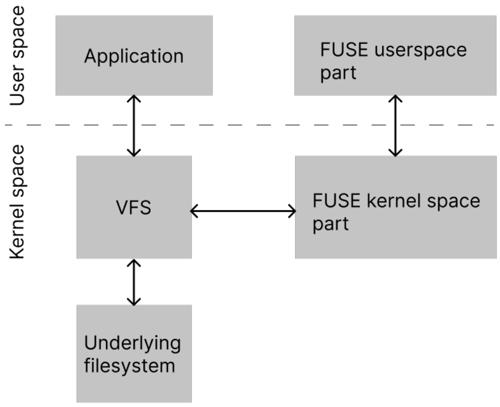

Figure 2.2: Simple visualization of how FUSE operations are executed

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 unique id associated with it [23]. Text posts are limited to 280 characters while images can be up to 5 MB and videos up to 512 MB [24]. A post with images can contain up to 4 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 [25]. 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 [26]. 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 [27]. Size restrictions are set for each post, depending on the media type. Images uploaded to Flickr can be a maximum of 200 MB and a video can be a maximum of 1 GB. Further, free-tier accounts can only have a total of 1 000 photos or videos on their account. A Flickr Pro account has unlimited storage on Flickr but is still subject to the per-item limit of 200 MB and 1 GB for images and videos, respectively [28]. 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 65535 characters according to Shhexy Corin [29]. 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[30]. 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. The original video can only be downloaded by the user [30]. Flickr does not state if it will always be possible to download the original versions of the file. Further, Flickr states that it retains the right to remove user content from the service at any time [31].

The Flickr API defines a query limit of 3 600 requests per hour, per application, across all API calls [32]. However, according to Sam Judson in 2013, this is not a hard limit [33]. 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 [32]. If abuse is detected, Flickr reserves the right to revoke API keys.

2.4 Cryptography

The Advanced Encryption Standard (AES) is an encryption standard established by the The U.S. National Institute of Standards and Technology (NIST), more specificity specifying the Rijndael block cipher [34]. 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 [35].

While AES is a good standard for the confidentiality of the data, confidentiality is often not enough to secure the data [36]. 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 [37]. Galois/Counter Mode (GCM) is a block cipher mode of operation which provides authenticated encryption [38]. 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 is often derived from a password that the user provides. Password-Based Key Derivation Function (PBKDF)s are functions that can be used to derive a key used for, for instance, AES. The input to a PBKDF is a secret, such as a password [39]. An example of a PBKDF schema is the Hashed Message Authentication Code based Key Derivation Functions (HKDF) presented by Krawczyk [40][41] 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 [42]. 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 [43]. The salt does not have to be secret and is sometimes stored with the produced cipher so that the decryption function easily can re-use the salt when deriving the decryption key. If the key used for encryption and the key used for decryption are derived using different salts, the keys will differ and the cipher cannot be decrypted.

Alternative encryption solutions are, among others, Rivest-Shamir-Adleman (RSA) and Data Encryption Standard (DES). 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 [44]. Mahajan and Sachdeva found that AES is the fastest algorithm for encryption and decryption between RSA, DES, and AES while maintaining very good security. This further makes AES a good choice as the cryptography technique for FFS.

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 [45]. 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.

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 [46]. 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 contain any type of file easily [47]. 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 [48]. He presents multiple examples of this technique on his Twitter profile*. The fact that the images are available for the public's eye might be evidence that Twitter's steganography detection software is not perfect. However, it is also possible that Twitter has chosen to not remove these posts.

* https://twitter.com/David3141593

Other examples of steganographic data storage on Open Social Networks (OSN)s include the paper presented by Ning, Singh, Madhyastha, Krishnamurthy, Cao, and Mohapatra where the authors build a system for private communication on public photo-sharing web services [49]. Due to the web services processing of uploaded multimedia, they first researched how the integrity of steganographic data could be maintained after being uploaded to these services. Following this, they presented an approach that ensured the integrity of the hidden messages in the uploaded images, while also maintaining a low likelihood of discovery from the steganographic analysis. Beato, De Cristofaro, and Rasmussen also explores the idea of undetectable communications over OSNs in another paper [50]. While implementation is not carried out, they present an idea where messages are encoded together with a cover object and a cryptographic key to produce a steganographic message which is then posted to the OSN. A web-based user interface client with a PHP server backend is presented as the method the users would use to create and share their secret messages.

A steganographic, or deniable, filesystem is a system that does not expose files stored on this system without credentials - neither how many files are stored, their sizes, their content, or even if there exist any files in the filesystem [51]. This is also known as a rubber hose filesystem because of the characteristic that the data only can be proven to exist with the correct encryption key which only is accessible if the person is tortured and beaten with a rubber hose because of its simplicity and immediacy compared to the complexity of breaking the key by computational techniques.

3.2 Cryptography

Some papers choose to invent their encryption methods rather than using established standards. Chuman, Sirichotedumrong, and Kiya proposes a scrambling-based encryption scheme for images that splits the picture into multiple rectangular blocks that are randomly rotated and inverted, both horizontally and vertically, along with shuffling of the color components [52]. This is used to demonstrate the security and integrity of images sent over insecure channels. The paper uses Twitter and Facebook to exhibit this. Despite its improvement and compatibility of a common image format, such as bitstream compliance, due to its well-proven security FFS will use AES as its encryption method.

3.3 Related filesystems

Multiple steganographic filesystems have been presented previously but many of these are focused on filesystems for physical storage disks to that the user has access. For instance, Timothy Peters created DEFY, a deniable filesystem using a log-based structure in 2014 [53]. DEFY was built to be used exclusively on SSD found in mobile devices to provide a steganographic filesystem that could be used on Android phones. Further examples of local disk-based filesystems can be found in [1, 51, 54, 55], among other papers. However, this paper aims to create a filesystem that is not based on a physical disk but rather a cloud-based steganographic filesystem that uses online web services as its storage medium.

In 2007, Baliga, Kilian, and Iftode presented an idea of a covert filesystem that hides the file data in images and uploads them to web services, named CovertFS [6]. The paper lacks implementation of the filesystem but they present an implementation plan which includes using 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 [56].

In 2016, Szczypiorski introduced the idea of StegHash - a way to hide steganographic data on OSN by connecting multimedia files, such as images and videos, with hashtags [57]. 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 [58]. 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 [59], created in 2011. It was created as a proof of concept to show that it is possible to store file data on Twitter. The filesystem uses sequential text posts to store the data. The filesystem is not mounted to the operating system, instead, the user interacts with a Python script through the command line. This makes the filesystem less convenient from a user perspective, compared to a mounted filesystem where the files can be browsed using a user interface or command line. There are two commands available: `upload` and `download` which upload and download files or directories, respectively. Names and permissions of files and directories are maintained throughout the upload and download process. The tweets are not encrypted but are enciphered into English words which makes them look like nonsense paragraphs, similar to what we mentioned in Section 2.1.3 about how arbitrary data can be encoded as plain text. This makes the filesystem less secure than an encrypted version as it can be read by anyone with access to the decoder. However, it does introduce a steganographic element to the filesystem.

In 2006, Jones created GmailFS - a mountable filesystem that uses Google's Gmail to store the data [60, 61]. 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 [62]. 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 [63, 64]. On the other hand, Google Drive provides a desktop application [65] 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. On the other hand, the duration of a filesystem operation on the mountable volume provided by the Google Drive Desktop application does not always include the time it takes to upload the file, this can occur at a later time. One difference between 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 [64], and the repository in GitHub has around 2 300 starts as of writing.

Another Google Drive-based filesystem is google-drive-ocamlfuse [66], developed for Linux using FUSE. The project is well received online. The repository has around 6 700 stars on GitHub at the time of writing and there are multiple articles online about the project [67–69]. The filesystem is well developed and, as of writing, well maintained. The filesystem supports filesystem operations such as symbolic links, Unix ownership, and multiple account support. According to the author of GCSF, GCSF tends to be faster than google-drive-ocamlfuse for certain operations, including reading cached files [70, 71]. google-drive-ocamlfuse has no native support of macOS but is focused on Linux.

Zadok, Badulescu, and Shender created Cryptfs, a stackable Vnode filesystem that encrypted the underlying, potentially unencrypted, filesystem [72]. 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 [73]. 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 [74]. 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, Bhanage, Zadok, and Seltzer presents a paper where they criticize several papers due to their lack of scientific and honest filesystem benchmarking [75]. 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 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 about such factors must be present when analyzing the results of the benchmarking.

Tarasov, Bhanage, Zadok, and Seltzer 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 [75]. IOZone is listed as being compatible with multiple different benchmarking types and as it is simpler to use [76] and still maintained. Due to these factors, IOZone was chosen as the benchmarking tool for FFS.

3.5 Summary

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

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
Steganographic	-	-	X	X	X
Cloud-based	-	X	-	X	X

*As mentioned, the user has no control over this encryption

Chapter 4

Method

This chapter presents the methodology of implementing FFS and the specifications of the development environment. We also present the benchmarking tools and methodology used to acquire the quantitative data for the evaluation of the filesystem.

4.1 Development environment specification

Development of 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. The computer used to develop FFS was running macOS.

Table 4.1 presents the version of the libraries, APIs and tools used by FFS. FFS was developed using C++ and compiled using Apple clang. FFS uses the ImageMagick Magick++ library [77] for image processing. macFUSE [19] is used for FFS to use the FUSE API. cURLpp [78] is a cURL [79] C++ wrapper used by FFS to make HTTP requests. libOauth [80] is used by FFS to sign and encode HTTP requests according to the OAuth [78] standard. Flickcurl [81] is a C library used by FFS to communicate with parts of the Flickr API. Crypto++ [82] 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 version used for the operating system, libraries, and tools were the most recent up-to-date versions when the development of the filesystem started. To avoid

Table 4.1: The versions of the libraries, APIs, 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

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

4.2 FFS

The artifact that was developed as a result of this thesis is the Fejk Filesystem (FFS). It uses an OWS to store the data but behaved as a mountable filesystem for the users. As mentioned in Section 1.5 the filesystem is a proof-of-concept and does not support all functionalities that other filesystems do, such as links or access permissions. The reasoning is that these behaviors are not required for a useable system. Additionally, when comparing FFS to distributed filesystems such as Google Drive, many of these other filesystems also do not support functionality such as links.

4.2.1 Design overview

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 of the size of these posts for certain OWSs. To support file sizes bigger than these limitations, bigger 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 will instead keep track of the id numbers of the posts 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.

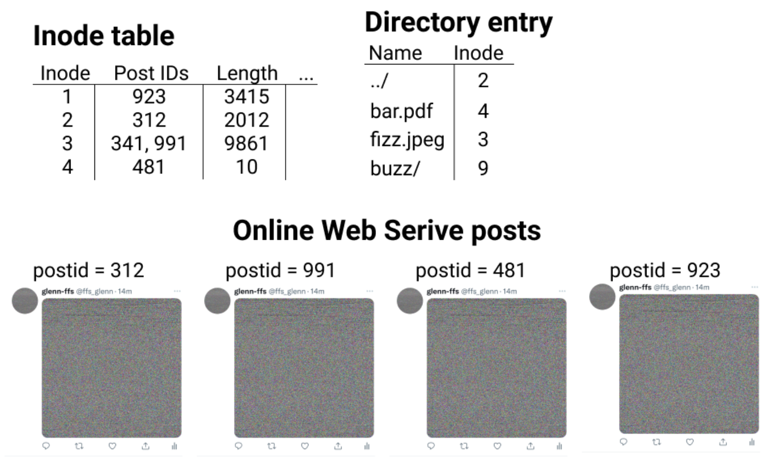

Figure 4.1: Basic structure of FFS inode-based structure

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 which is the root directory. This entry has a constant inode value of 0 for simplicity to look up the root directory. With the help of the root directory, all the files lower in the directory hierarchy can be found. The inode of all files and directories other than the root directory has a unique inode greater than 0. The `InodeTable` is always the most recent image saved on the OWS, making it easy to find it on the OWS.

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 Subsection 4.2.3), and concatenating them, can be read as a file or represented as a `Directory` object, depending on whether the `InodeEntry` is marked as a file or a directory.

As directories only know the filename's inode, the `Directory` object does not have to be updated (and thus uploaded) when a file or directory in it is edited, for instance adding data. Only the `InodeEntry`, and thus the `InodeTable`, needs to be updated with the new post IDs of the new file or directory. This saves computation time as every request to the OWS takes time. However, if the filename is edited or the file or directory is moved to another location, the parent directory of the file or directory would have to be edited, and thus its corresponding `Directory` object has to be updated.

When a new file or directory is created, it is saved in its parent directory with its filename and an inode. The same inode is used in the inode table to keep track of the file's or directory's inode entry. As shown in Appendix A, the inode is represented as an unsigned 32-bit integer. The inode is calculated by adding one to the currently greatest inode. This means that new files and directories will always receive a greater inode value than the ones currently in the inode table. This naïve approach to inode generation does not take into account that there might be an available inode less than the greatest inode in the inode table (for instance, due to the deletion of a previously created file). However, this inode generation approach is fast and will not be a problem until the integer overflows. As the inode is represented using a 32-bit integer, FFS would need to have saved over four billion files 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 a subset of them as shown in Table 4.2. The implemented operations are the most essential operations required for a working filesystem [83]. 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 Subsection 4.2.5.

Table 4.2: Filesystem operations implementable trough 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 data required. Therefore, only the directory or file that is affected by a change is uploaded to the system, while those unaffected can remain the same. The inode table has to be updated with every change of a file or directory as it contains the location of the file or directory.

FFS can be mounted to the local filesystem using FUSE, similar to how you can mount 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 $5\,\mathrm{MB}$ are not cached. Each time an image is uploaded or downloaded, it is added to the Cache Map with its post ID as the key. The post ID is also added to the beginning of the Cache Queue. If the Cache Queue exceeds 20 elements, the last element of the queue is removed, and the corresponding entry in the Cache Map is erased, thus the entry is fully erased from the cache. The queue ensures that the cache is limited to 20 entries, and by using the 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 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 the open file's parent directory. The open file's data is also cached in memory if it has have been read or written to while it is open. A file is opened with the use of the `open` or `create` filesystem operation, as described further down in Section 4.2.5. When a file is opened it is associated with a file handle identifier which is used for subsequent filesystem operations to refer to the file rather than using the path to the file in the filesystem. When a user is reading or writing data to a file, multiple `read` or `write` file operations might be executed. For instance, when writing a 100 B file two `write` operations might be executed:

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

The amount of `read` or `write` operations required to read or write data depends on the amount of data to read or write, the buffer sizes used by the file operation which depends on the buffer sizes supported by the filesystem. macFUSE can be mounted with a maximum buffer size of 32 MB [84]. To save computation time by not having to download the file from the OWS, or even decrypt the image data found in FFSs 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 it is checking the regular cache for the post ID. If there is data associated with the file handle, then this data is used for the file operation. If the file operation was a modifying operation, such as a `write` operation, the new data is associated with the file handle and stored in memory. When the file is closed with a subsequent `close` file operation, the modified data is encoded, encrypted and uploaded to the OWS. There is no limit to how many files can be open in the filesystem at the same time, nor how much modified data can be associated with a file handle. Further, if a file is not closed, the associated data is not disassociated with the file handle and is kept in the memory until the filesystem is shut down.

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 RGB color depth. The inode table and the directories are represented as C++ objects in memory during runtime but are serialized into a binary representation before they are encoded into images. The files saved to 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 do 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 $64\,\mathrm{B}$ 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 amount 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 4 bytes. For an image of $X = ceil(\frac{4+LCED}{6})$ pixels, FFS will set the width w of the image as $w = ceil(\sqrt{X})$. Further, the height h of the image is set as $h = ceil(\frac{X}{w})$. This will require $(w * h) - X$ filler bytes and will create an image with similar height and width. For certain values of X, h will be equal to w. For other values of X, $h = w - 1$. The resulting data encoded as pixels in the image is, in order:

- 4 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 4 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.

The encryption and decryption methods used are state-of-the-art solutions

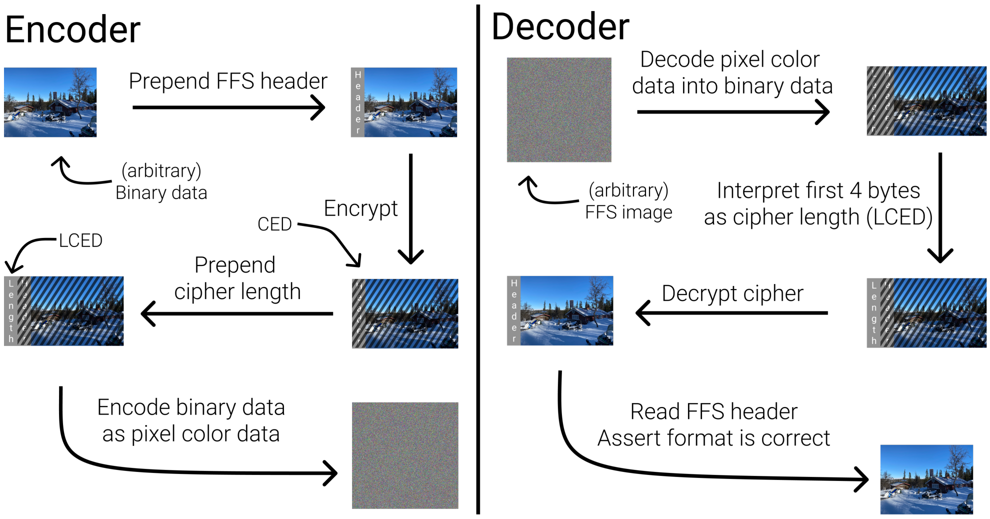

Figure 4.2: Simple visualization of the encoder and decoder of FFS. The input of the encoder is the binary data to store in FFS, eg. a file, and the output is the FFS image to upload to the OWS. The input to the decoder is an FFS image, and the output is the binary data stored on FFS, eg. a file

as defined and implemented by Crypto++ [82]. 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 [85].

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 are encoded in. 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 of the inode table, limiting the filesystem. More details about the limits of FFS are found in Subsection 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, multiple OWSs would be beneficial. This would enable features such as redundancy by using replication over multiple OWSs, for instance 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 relies on a specific binary representation of the image, this meant that the images could not be decoded into the previously uploaded data. Twitter has previously publicly announced changes to the way they store images [86] and even suggested workarounds [87] for users who are concerned about the potential data loss. However, during research for the thesis, it was concluded that the workarounds mentioned in [87] no longer work on Twitter. For instance, some PNG images less than 900x900px that have been uploaded to Twitter, have not been able to be downloaded as the same image, which contradicts the workaround mentioned by the Twitter employee. Further changes may have been made to the data management of images on Twitter since the initial research for the thesis; however, an official announcement has not been found.

Flickr saves the original version of the uploaded image and thus it can be used to download the same image as was uploaded. This also means that data that is encoded into an 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 also do not indicate that this would change. Therefore, Flickr can be used at this moment for the proof-of-concept filesystem that FFS is. A free-tier Flickr account is therefore used for FFS. However, as was noted in Section 2.3.2, only the user can download the original file - other users might get another file when downloading the image post.

Flickr provides an extensive free REST 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 the 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, computer memory cache misses/hits and process scheduling, among other factors, can fluctuate the value of t_{ffs}. In contrast, t_{ows} is the total time required to complete all requests to the 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 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 [88–90]. 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 [83].

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 following 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.

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.

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 \gls{OWS}
        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 cache
        curr_dir = cache.get_root_dir()

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

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 a number of bytes, starting from a set offset, from the file specified by the file handle. The data is read into a provided buffer. The full file is downloaded and read into memory, even if just a small part of the file is requested. The file is also cached so that subsequential requests for the same file are faster.

4.2.5.8 readdir

This operation reads the filenames inside the directory specified by a file handle. The result includes all filenames in the directory, and the special `"."` and `".."` directories.

4.2.5.9 write

This operation writes s bytes from a data array a, starting at the provided offset o, to the existing file at the provided file handle. All the data of the current file is read into memory. Starting from the offset, the new data from a overwrites the current data of the file, until s bytes have been written. If $o + s$ is greater than the file's size, the file size is set to $o+s$. If $o+s$ is less than the file's size, the data from position $o + s$ and forward remains the same, and the file size is not modified. See Figure 4.3 for a visualization of the result of a `write` operation given different offsets. The parent directory does not have to be modified.

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

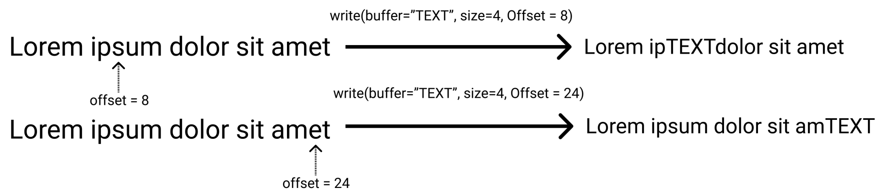

Figure 4.3: Visualization of how the write operation handles different offsets.

4.2.5.10 rename

This operation renames a file or directory to a new path. Both the old path and the new path have to be traversed to locate the parent directories and the file or directory to rename. The file or directory entry in the old parent directory is removed, and the old parent directory is updated to the 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.

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 uses this new file data.

4.2.5.13 unlink

This operation removes a file given the file path. The file is removed from the parent directory, and the parent directory is updated to the 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 sizes, this step could be omitted to save time.

4.2.5.14 rmdir

Similar to `unlink`, this operation removes the directory at the path. The directory and all its subdirectories are traversed, and the post IDs of these files and directories are recorded for deletion later. Following this, the entry of the removed directory is removed from the parent directory. The inode entry for the removed directory is removed. The parent directory is updated to the 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 on the main thread and would not be accessed there. This also means that this thread can be run with a lower priority.

4.2.5.15 getattr

This operation returns attributes about a file or directory given a path. This includes permissions, the number of entries (if the provided path points to a directory), timestamps of creation, timestamps of last access, and timestamps of last modification. However, as mentioned previously, 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 which 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 $200\,\mathrm{MB}$ per image, this allows storage of up to $200\,\mathrm{GB}$ 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 $199.8\,\mathrm{GB}$.

While Flicker supports each image to be up to $200\,\mathrm{MB}$, it is not possible to use the full $200\,\mathrm{MB}$ to store the file or directory data. The 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 does not exceed the limit of $200\,\mathrm{MB}$, FFS limits the PCD size to allow at least $10\,\mathrm{MB}$ for the PNG header and other PNG attributes, meaning that the PCD can be a maximum of $190\,\mathrm{MB}$. 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 $190\,\mathrm{MB}$, meaning that the encrypted cipher text cannot exceed $190\,000\,000 - 92 = 189\,999\,908\,\mathrm{B}$. 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 $floor(\frac{189\,999\,906}{16}) * 16 = 189\,999\,904$ bytes. Due to plain text padding, the unencrypted plain text can be a maximum of one byte less than this value [91], meaning that the plain text can be a maximum of $189\,999\,903\,\mathrm{B}$. For simplicity, this is rounded down to $189\,\mathrm{MB}$, leaving almost $11\,\mathrm{MB}$ in total for the PNG header and other PNG attributes. Therefore, $189\,\mathrm{MB}$ is set as the maximum amount of data FFS will store per image. Data greater than $189\,\mathrm{MB}$ in size is split into multiple encoded images. For instance, a file of $200\,\mathrm{MB}$ will be stored as $189\,\mathrm{MB}$ in one image, and $11\,\mathrm{MB}$ in another.

$189\,\mathrm{MB}$ of usable data per image gives FFS a maximum storage capacity of $188.811\,\mathrm{GB}$ 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 $188.811\,\mathrm{GB}$ stored in the filesystem, which would have to represent the root directory.

The inode table keeps the information about empty files and directories even though they store no data on the OWS. The inode of a file or directory is an unsigned 32-bit integer, meaning that the inode table could theoretically store up to over 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 $37\,\mathrm{B}$ 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 $189\,\mathrm{MB}$. Further, the size of the inode table is $4\,\mathrm{B}$ 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 directory in the inode table requires approximately (depending on the post ID length generated by the OWS) $12\,\mathrm{B}$ per file or directory it contains. The maximum number of empty files and directories X that the inode table can store is therefore, approximately:

$$X = ceil(\frac{189\,000\,000 - 4 - (12 * X)}{37}) + 1, X = 3\,857\,143$$

The additional directory is the root directory. Thus, the maximum number of files and directories that the inode table can store is close to four million; however, this requires all files and directories, except the root directory, to be empty. 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.

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 $5\,\mathrm{MB}$ in memory, requiring $100\,\mathrm{MB}$ of memory. However, even if the computer has less memory available, more memory can often be provided through swap on the hard disk. Apple ensures that the swapped data is securely encrypted on the hard disk [92]. However, using a swap puts a constraint on the available storage of the computer to be sufficient to store this data. Further, as FFS temporarily saves the data on the local filesystem before it is uploaded to Flickr, the storage device must have sufficient storage available. For instance, a file larger than the available storage on the local filesystem cannot be saved to FFS. If the local filesystem has no available storage, very 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 also stores data associated with file handles of modified and open files in memory. The amount of open files in FFS is unbounded, and the amount of data FFS can store per file handle is also unbounded. Further, the data associated with the file handle is not removed from memory until the file is closed. This presents a potential memory overflow scenario if, for instance, many files are opened and modified without being closed afterwards.

Another limitation of FFS is the rate limits presented 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. Further, some requests are sent concurrently to Flickr which means that FFS could reach $3\,600$ API calls faster. However, as long as FFS is not constantly serving filesystem calls, the API limit should be of little concern.

A limitation of FFS that is not possible to quantify is the bandwidth and latency of the network connection from the user to Flickr. The connection can vary significantly depending on for instance the network load at a given moment and the geographic location of the user. A slow network connection is not something FFS can solve, but is left as an exercise for the reader.

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 is 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 the required internet connection. It is the local filesystem of the development environment for FFS. It was selected as it will give the analysis an example of a modern, well-used, and fast filesystem, and how the benchmark data of FFS and other filesystems compare to this local filesystem.

GCSF was selected to compare FFS against another network-based filesystem. While GCSF is not a steganographic 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 was because it is a recent filesystem compared to other related filesystems. Some of the other filesystems discussed in Section 3.3 were developed many years before 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 it will require the execution time of the APFS file operation as well as the internal FFS computation time. Both FFFS and FFS are mounted with a maximum buffer size of 32 MB.

4.3.2 Tools

IOZone [74] is a filesystem benchmarking tool used to analyze the performance of filesystem file operations using different tests on a file [93]. 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 used for the read- or write operation. Normally, multiple buffer sizes are used for each test, for each file size tested. The buffer size starts at 4 kB and increases by a multiple of two up to a buffer size equal to the file size. Multiple file sizes are often used for benchmarking tests as well. 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

When IOZone reads from a file it has written to, it asserts that the file content is what it wrote previously to verify that the filesystem stores the data properly. This is not documented in the IOZone documentation [93] but was discovered during testing. However, while it asserts that file operations function correctly, it does not verify all aspects of the filesystem functionality. Further, as IOZone does not state that the file operations are tested, it cannot be assumed that the file operations are correct. Additionally, IOZone does not test if directory hierarchies work as expected, nor if multiple files can be stored at the same time. IOZone is a benchmarking tool used for evaluating the performance of the file operations of a filesystem, not testing the functionality. However, certain cases of the functionality of both FFS and GCSF was tested, and to support directory hierarchies and multiple files as expected. Future work could research the functionality of these filesystems utilizing online storage systems. 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, the thesis only uses 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 [93] 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 IOZone documentation [93] 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, is unknown. The documentation states that when the cache is unknown, it should be set to greater than the physical memory of the system. However, as the memory of the computer where the benchmarking is run is 16 GB, this is bigger than reasonable for testing FFS and GCSF. Each doubled file size takes exponentially much more time as both the file size and the buffer size are doubled. Further, it has been found that both GCSF and FFS occasionally crash during benchmarking due to, among other factors, unstable internet connections, meaning that a benchmarking test of 16 GB might never be complete due to the filesystem crashing first. The file sizes used for the IOZone tests are therefore set as:

1. 1024 kB,
2. 2048 kB,
3. 4096 kB,
4. 8192 kB, and
5. 16 384 kB

The buffer sizes tested are:

1. 4 kB,
2. 8 kB,
3. 16 kB,
4. 32 kB,
5. 64 kB,
6. 128 kB,
7. 256 kB,
8. 512 kB,
9. 1024 kB,
10. 2048 kB,

11. 4096 kB,
12. 8192 kB, and
13. 16 384 kB

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

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 (file operation) with performance data for each file size, and for each buffer size for each file size benchmarked for the test. The performance of the filesystem is measured in kilobytes per second.

The benchmarking of FFS and GCSF were both started simultaneously as they both depend on an internet connection. For a fair comparison of the two filesystems, they should be run with similar internet connection constraints. During the benchmarking of the two filesystems, an automatic speed test was conducted every five minutes to survey the current internet connection. The speed test uses Bredbandskollen's command line interface tool [94] which measures the latency, upload-, and download speed of an internet connection to a measurement server in Sweden, Norway, or Denmark [95]. The benchmarking tests of FFS and GCSF were carried out in Amsterdam in The Netherlands using an ethernet connection to a fiber-connected router. While the internet connection to the measurement server is not sure to be equal to the internet connection to the servers of Flickr or Google Drive, it is used as a reference point of the internet connection.

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 `https://github.com/GlennOlsson/FFS` [96]. The filesystem provides free cloud-based cryptographic and deniable storage as a mountable volume. 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 output result is divided into a table for each test for each filesystem. Each table presents the benchmark performance of the test for each file size and each buffer size. Each table has five rows and 13 columns, where each cell is the performance of the test with the specific file size and buffer size. The complete data tables and graphs presenting the performance of each file system for the different file sizes can be found in Appendix C.

The IOZone benchmarking for FFS ran for 41 minutes, and the IOZone benchmarking for GCSF took 20 minutes. They were started at the same time and the internet speed tests ran every five minutes until the benchmarking of FFS was completed. In total, eight speed tests were conducted with an average latency of $15.22\,\mathrm{ms}$. The average download speed was $90.96\,\mathrm{Mbit/s}$ and the average upload speed was $92.95\,\mathrm{Mbit/s}$.

Combining the 55 data points in one table, we get the overall performance of a test. Using this data, we can plot a box plot presenting the spread of the values in the table. Figure 5.1 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 FFFS are in general worse than the performance for GCSF and APFS. FFS has by far the biggest spread of values, and FFFS has also a significant spread. The median performance of FFS and FFFS are similar. GCSF and APFS have less spread. The median performance of GCSF and APFS are significantly higher than the median performance of FFS and FFFS, and APFS has the highest read performance of the four filesystems.

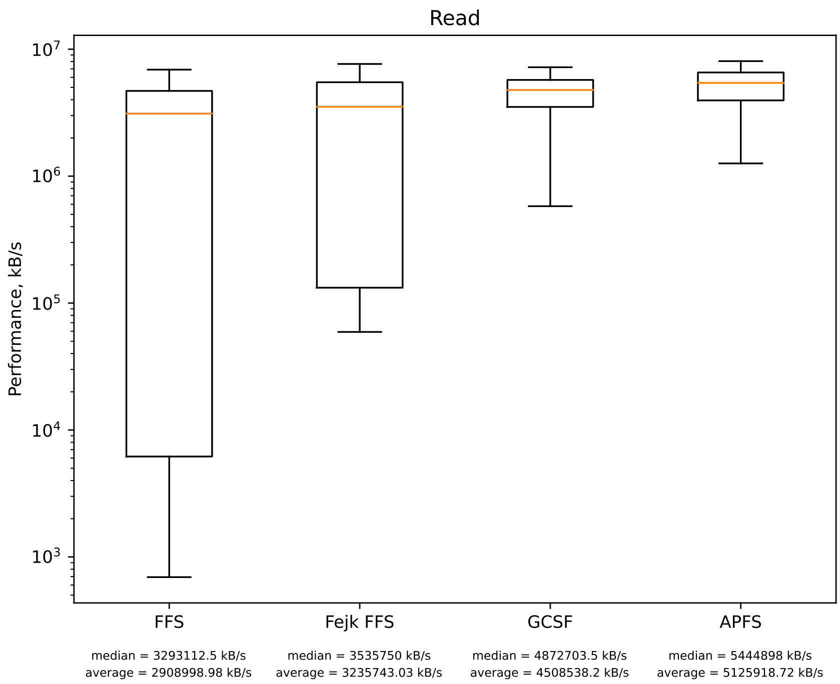

Figure 5.1: Box plot of the IOZone output for the Read test on the different filesystems

Figure 5.2 presents a box plot of the benchmarking results of the filesystems of the Write test. APFS has the best median write performance of the four filesystems, followed by FFFS. The cloud-based filesystems FFS and GCSF have the worst median write performance, where FFS has the worst performance of the filesystems. Comparing with the results of the Read test as presented in Figure 5.1, it can also be noted that the write performance is significantly worse for the write operations than the read operations for all filesystems. The average performance of the Write test for APFS is 11% of the Read performance. For FFFS, GCSF, and FFS this percentage is 0.32%, 0.05%, and 0.02%.

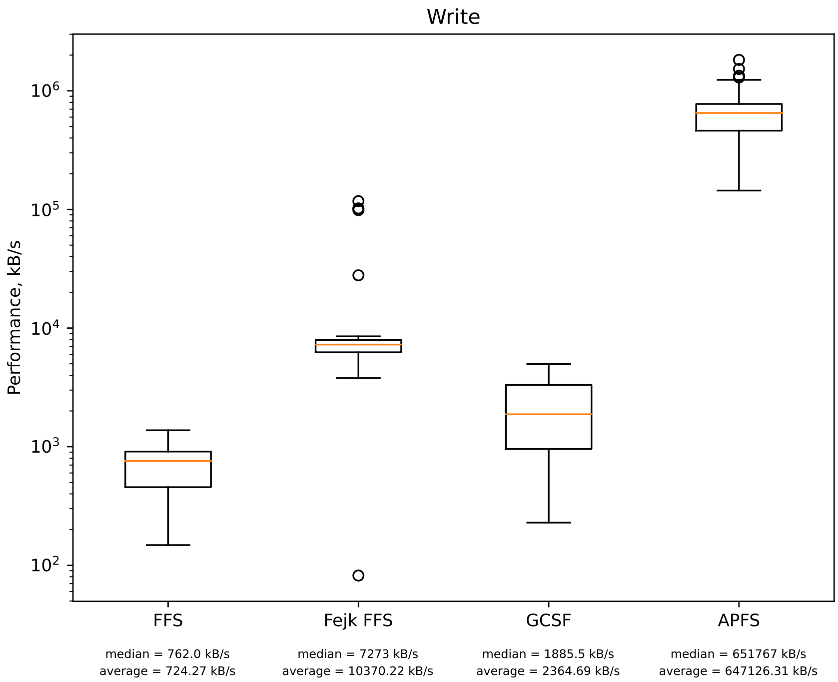

Figure 5.2: Box plot of the IOZone output for the Write test on the different filesystems

Figure 5.3 presents the result of the Re-Read test for the filesystems. The median performance of the filesystems are similar, with the performance of FFS being lowest. Furthermore, the spread of the values for FFS is greater than for the other filesystems. While the average performance for the Re-Read test of FFS is around $3\,\mathrm{GB/s}$, the lowest performance was at $468\,\mathrm{kB/s}$, namely for `file size = 1024, buffer size = 1024`. The spread of values is higher for APFS than for FFFS and GCSF. The average and median performance of FFFS and GCSF are both higher than for APFS. FFFS has a higher average performance than GCSF, while GCSF has the highest median performance.

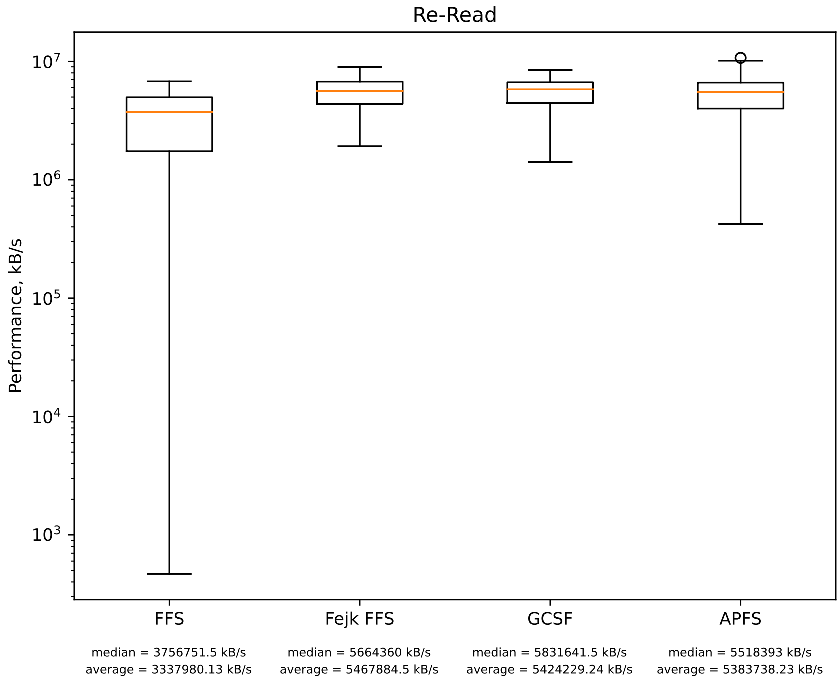

Figure 5.3: Box plot of the IOZone output for the Re-Read test on the different filesystems

Figure 5.4 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, followed by FFFS and finally the two cloud-based filesystems. The performance of the filesystems are overall very similar to the results for the Write test.

Figure 5.5 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 5.3. FFFS has the best average and median performance of the filesystems, and GCSF has also better median and average performance than APFS. The median and average performance of FFS is lower than the other filesystems. FFS also has a wide spread of values.

Figure 5.6 presents a box plot for the Random read test for the different filesystems. APFS has the highest performance, followed by FFFS and the cloud-based filesystems. GCSF has better performance than FFS. The

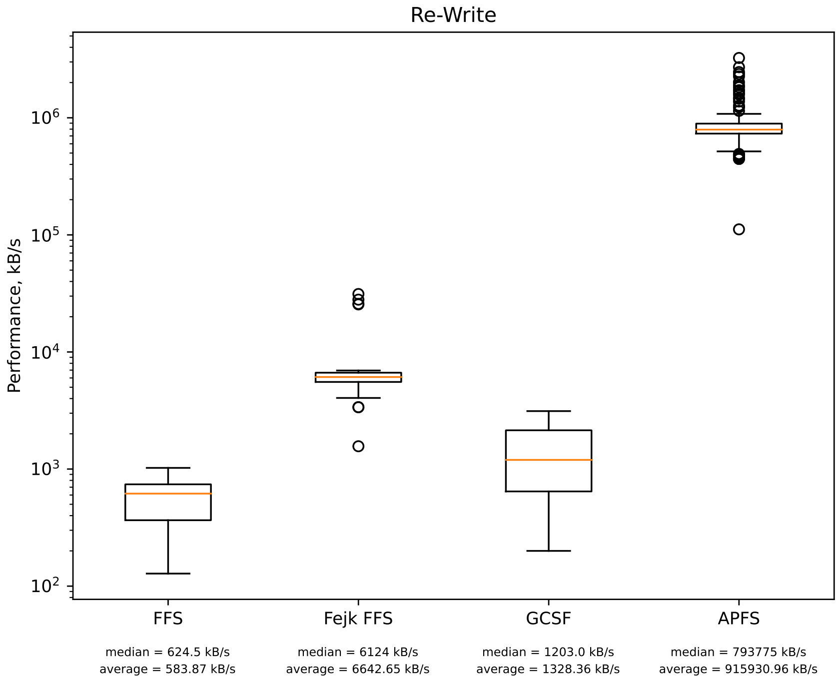

Figure 5.4: Box plot of the IOZone output for the Re-Write test on the different filesystems

difference between the Random read test and the Re-Read test is small for all filesystems. FFS does not have the best performance in any of the box plots presented. GCSF has better performance than FFS for all tests run with the benchmarking.

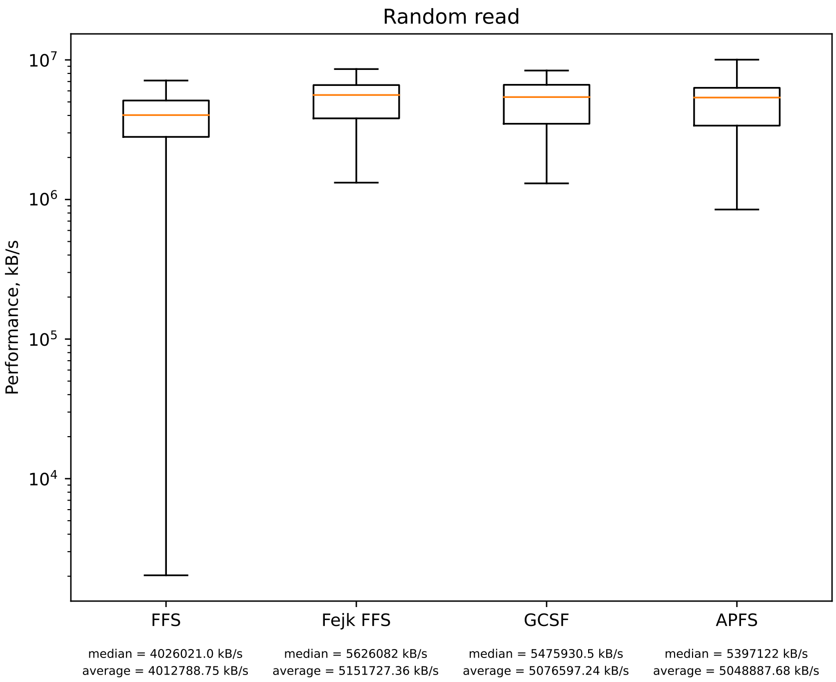

Figure 5.5: Box plot of the IOZone output for the Random read test on the different filesystems

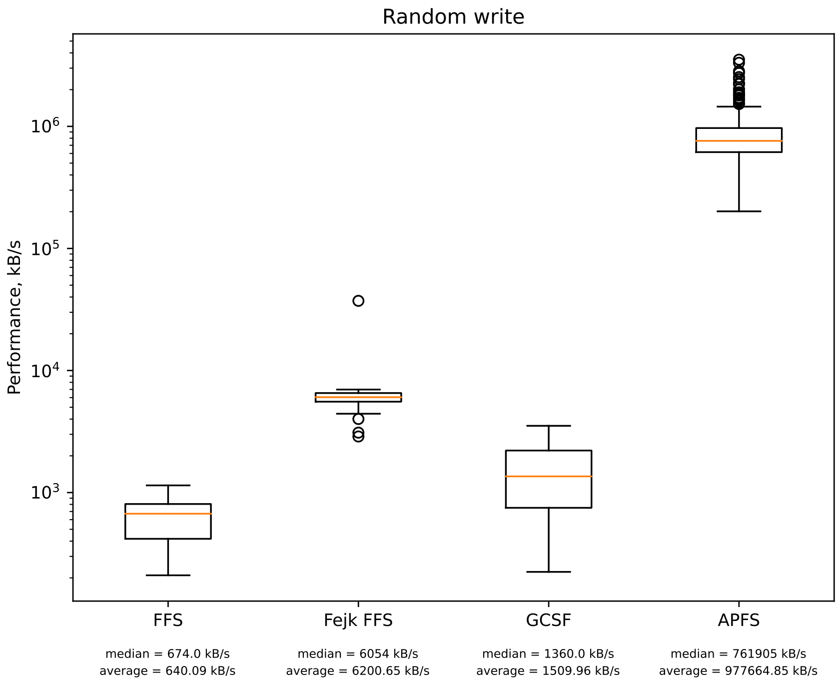

Figure 5.6: Box plot of the IOZone output for the Random write test on the different filesystems

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 impacts of FFS are introduced in Section 6.3.

6.1 Filesystems

Figure C.1, C.3, and C.5 show that FFS performs poorly for Read operations with a small buffer size. Beginning at 4 kB buffer size, the performance in general increases with the first few buffer sizes. This indicates that the overhead of the FFS read operation is high as the performance gets better when it reads fewer buffers. Overhead of the read operation includes, among other things, the time to fetch the image from Flickr if it is not in the cache, and decrypting the image which is required even if the image is cached. Further, it is expected that Re-Read should perform better than Read when the file size is small enough to fit in the cache. However, this is not an obvious conclusion that can be drawn from the result. Looking at table C.3, it shows that there is no significant drop in performance for file sizes bigger than the 5 MB cache file size limit. The performance is even better for the file sizes bigger than the cache file size limit than for the file sizes smaller than the cache file size limit for certain buffer sizes, such as `buffer size = 512kB` where the performance only increases for bigger file sizes. However, as Figure **??** shows, the average performance of Re-Read is better than the average performance of Read for FFS. It is expected that the cache will increase the performance of the filesystem.

One interesting comparison is between the benchmark results of FFS and

GCSF. Both filesystems are cloud-based FUSE filesystems dependent on an internet connection to their respective storage servers. Knowing that the benchmarking of the two filesystems was started at the same time, and the FFS benchmarking took 41 minutes while the GCSF benchmarking took 20 minutes, it is clear that GCSF is overall much faster than FFS for the specified benchmarking test, using the defined file sizes and buffer sizes. The data presented in Section 5.2 and Appendix C further confirms the conclusion that FFS is mich slower than GCSF. For instance, the Read test of GCSF performed in general better than the Read test of FFS. The median performance of the GCSF Read test is significantly better than the median performance of the FFS Read test. Looking at Figure **??** and Figure **??** we can see that FFS had much bigger spread of the values than GCSF did, especially for the Write, Re-Write, and Random Write tests. One outlier of the Write test of FFS performed at 88 671 kB/s, namely for `file size = 8192kB, buffer size = 2048kB` as can be seen in Table C.1. This is better performance than any of the performance data points of GCSF for the Write, Re-Write, and Random Write tests.

Comparing FFFS benchmarking results against the APFS benchmarking results, we can compare the theoretical best performance of FFS against a general-purpose widely-used filesystem. Furthermore, we can compare FFFS against the underlying filesystem in which it is storing its data. In Table C.13 and Table C.19 we can see that the read operation perform similarly for FFFS and APFS, where APFS is in general faster than FFFS. However, for certain data points, such as `file size = 4096kB, buffer size = 4kB`, FFFS has higher throughput than APFS with 2 866 270 kB/s for FFFS and 2 402 508 kB/s for APFS. The cache of the filesystems can greatly influence the performance of the read operation. In the case of FFFS, the filesystem will cache the data written as long as its size is less the limit of 5 MB. However, there is no significant difference between the FFFS performance when reading a file that fits in the FFFS cache, and one that does not. All files that are read in FFFS that are not in the FFFS cache are read from disk, which invokes at least one APFS read operation. While the APFS read operation called might not be called with the same buffer size as the read operation called by IOZone on FFFS, the performance of the FFFS read operation cannot exceed the APFS read operation. However, the similarity of the performance between the filesystem indicates that FFS implements fast read operations, and that the read operation performance of FFS depends to a great extent on the internet bandwidth and latency to the OWS, as well as the OWS's data processing performance.

While the values of the read operation for FFFS and APFS are comparable to each other, this is not the case for all tests. For instance the write operation of FFFS is much slower than the write operation of APFS, as can be seen in Table C.14 and Table C.20. The write operation of FFFS has about 2-3% the performance of the write operation of APFS. The reason for this could be the fact that FFFS has to encrypt the data stored, including creating all the cryptographic variables such as the salt and the IV. While APFS is also an encrypted filesystem, it is possible that the cryptographic functions are much faster than for FFFS as they for instance can be run in kernel space, while FFFS is running in user space.

FFFS and GCSF are comparable in some tests, which is interesting as GCSF is dependent on an internet connection while FFFS is not. The median performance of the Read test on GCSF is slightly worse than the medium performance of the Read test on FFFS. Meanwhile, the median Re-Read performance of GCSF is better than the median Re-Read performance of FFFS. This indicates that GCSF implements a faster cache than FFFS. One reason might be that the FFFS cache stores the encrypted version of the image, meaning that before the data is read, the image must first be decrypted and decoded. As Google Drive provides the raw data of the file stored, GCSF can store the raw data in its cache meaning that the data in the read operation can be returned faster. Re-Write and Random Write tests on FFFS outperform the same tests on GCSF. This is reasonable as the data written to GCSF must be uploaded to Google Drive, while the data written to FFFS is stored on the local disk. Uploading $16\,\mathrm{MB}$ of data with the average (reference point) upload speed of $92.95\,\mathrm{Mbit/s}$ would take about $1.4\,\mathrm{s}$. Meanwhile, we can see in Table C.20 that APFS can write $16\,\mathrm{MB}$ of data as fast as $1\,539\,559\,\mathrm{kB/s} = 1549.559\,\mathrm{MB/s}$, meaning it would take about $10\,\mathrm{ms}$ to write the data. However, the data written by FFFS is larger than $16\,\mathrm{MB}$ as the saved data by FFFS is inflated by encryption and PNG attributes.

It is easy to see, and it is not unexpected, that APFS outperforms FFS in performance. As a professional local filesystem, APFS will always have better performance than FFS. Further, like FFFS, the performance of FFS depends on the performance of APFS as the file which is uploaded to Flickr first needs to be saved on disk. This dependency could be removed, for instance by providing the temporary file to the FlickCURL library via a FUSE filesystem. Further, the median performance of the Re-Read test on FFS is about 72% of the performance of the Re-Read test on APFS. With higher bandwidth and with another OWS, it is possible that FFS could increase its performance. In contrast, the median performance of the Re-Read test on FFS is about 76% of the median performance of the same test on GCSF.

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 [97]. 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 encryption 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 the choice of password.

FFS puts a lot of trust in the open-source library Crypto++ [82]. 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 [85], 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. 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
- 1 directory and 12 different files, or
- 13 copies of the same file, et cetera.

It is also not possible to know if an image stored on Flickr has been uploaded by FFS or by the user manually to further inflate 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 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. One drawback of storing images on Flickr that are not stored in FFS is that it decreases the storage capacity of FFS.

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 [91]. 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 their maximum possible value of L_p, it is possible to know the largest possible amount of data which is

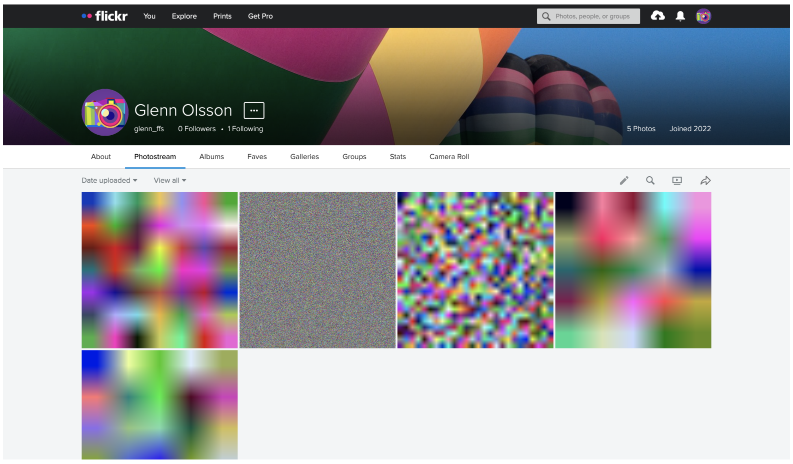

Figure 6.1: Screenshot of the Flickr profile used for FFS. At the moment of the screenshot, the filesystem is storing a previous version of this thesis in a directory inside the root directory. The images seen are the inode table, the thesis data, the root directory data, the subdirectory (containing 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* [98]).

stored by FFS on Flickr at a certain time. 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.

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*, 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

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

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.

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 wants 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 nor 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 wants 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 store 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. For instance, the data could be given to authorities who are requesting it with a subpoena. 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. A subpoena by authorities will not help more than possibly providing them with the IP addresses of origin of the uploaded content. 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.

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 environment impacts that 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 no 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 also known to be using it to spread their messages across the internet [99]. 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 to use 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 [100].

6.3.2 Environmental impact

FFS uses Flickr's data centers to store its data. Globally, data centers have a huge environmental impact. It has been estimated that they use over 2% of the world's electricity [101] and emit roughly the same amount of carbon dioxide as the global airline industry emits burning aircraft fuel [102]. These data centers are always on and are always consuming energy. When storing the data on a local filesystem instead, the device can be powered off while the filesystem is not in use, such as by detaching an external storage drive.

Further, 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 on 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 the overhead of FFS. Further, Google Drive can be optimized as a filesystem as it is the intention of Google Drive. Meanwhile, FFS is exploiting Flickr's image storage and is not optimized as filesystem storage. For instance, Google Drive can optimize the cache of the filesystem to better reflect a filesystem cache, reducing power consumption. However, as Flickr handles large amounts of data, Flickr has probably implemented energy-efficient solutions for retrieving images.

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 is FFS performing 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 at every read operation as well 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. 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 integrity 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.

To improve the dependability 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.

References

[1] J. Han, M. Pan, D. Gao, and H. Pang, "A multi-user steganographic file system on untrusted shared storage", in *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC '10*, Austin, Texas: ACM Press, Dec. 6, 2010, p. 317, ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920309. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1920261.1920309 (visited on 01/27/2022).

[2] R. Westhead, "How a Syrian refugee risked his life to bear witness to atrocities", *The Toronto StarWorld*, Mar. 14, 2012, ISSN: 0319-0781. [Online]. Available: 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". (), [Online]. Available: https://engineering.fb.com/2016/03/29/android/mobile-scale-london-recap/.

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

[5] D. Johnson. "Is Google Drive secure? How Google uses encryption to protect your files and documents, and the risks that remain", Business Insider. (Feb. 25, 2021), [Online]. Available: https://www.businessinsider.com/is-google-drive-secure (visited on 04/13/2022).

[6] A. Baliga, J. Kilian, and L. Iftode, "A web based covert file system", in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07, USA: USENIX Association, May 7, 2007.

[7] J. Salter. "Understanding Linux filesystems: ext4 and beyond", Opensource.com. (Apr. 2, 2018), [Online]. Available: https://opensource.com/article/18/4/ext4-filesystem (visited on 03/09/2022).

[8] "Fscrypt - ArchWiki". (), [Online]. Available: https://wiki.archlinux.org/title/Fscrypt (visited on 04/25/2022).

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

[10] T. Nelson. "What Is APFS and Does My Mac Support the New File System?", Lifewire. (), [Online]. Available: 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. (), [Online]. Available: 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". (), [Online]. Available: https://www.google.com/intl/sv/drive/ (visited on 10/26/2021).

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

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

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

[16] Apple Inc. "iCloud security overview", Apple Support. (), [Online]. Available: https://support.apple.com/en-us/HT202303 (visited on 04/25/2022).

[17] "Multi-state data storage leaving binary behind: Stepping 'beyond binary' to store data in more than just 0s and 1s", ScienceDaily. (Oct. 12, 2020), [Online]. Available: https://www.sciencedaily.com/releases/2020/10/201012115937.htm (visited on 03/10/2022).

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

[19] "Home - macFUSE". (), [Online]. Available: https://osxfuse.github.io/ (visited on 03/07/2022).

[20] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or Not to {FUSE}: Performance of {User-Space} File Systems", presented at the 15th USENIX Conference on File and Storage Technologies (FAST 17), Feb. 27–Mar. 2, 2017, pp. 59–72, ISBN: 978-1-931971-36-2. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor (visited on 04/06/2022).

[21] R. Gooch. "Overview of the Linux Virtual File System — The Linux Kernel documentation". (), [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/vfs.html (visited on 04/12/2022).

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

[23] Twitter. "Twitter IDs". (), [Online]. Available: https://developer.twitter.com/en/docs/twitter-ids (visited on 07/15/2022).

[24] "Media Best Practices - Twitter". (), [Online]. Available: https://developer.twitter.com/en/docs/twitter-api/v1/media/upload-media/uploading-media/media-best-practices (visited on 10/26/2021).

[25] "Retrieving older than 30 days Direct Messages (direct_messages/events/list) - Twitter API / Standard APIs v1.1", Twitter Developers. (Apr. 27, 2018), [Online]. Available: https://twittercommunity.com/t/retrieving-older-than-30-days-direct-messages-direct-messages-events-list/104901 (visited on 03/11/2022).

[26] "Understanding Twitter limits | Twitter Help". (), [Online]. Available: https://help.twitter.com/en/rules-and-policies/twitter-limits (visited on 03/11/2022).

[27] "Flickr upload requirements", Flickr. (Jul. 26, 2022), [Online]. Available: https://www.flickrhelp.com/hc/en-us/articles/4404079649300-Flickr-upload-requirements (visited on 07/31/2022).

[28] Flickr, Inc. "Upgrade everything you do with Flickr", Flickr. (), [Online]. Available: https://www.flickr.com/account/upgrade/pro (visited on 07/31/2022).

[29] "Flickr: The Help Forum: Captions/text In Flickr", Flickr Help Forum. (Jan. 2, 2009), [Online]. Available: https://www.flickr.com/help/forum/en-us/88316/ (visited on 07/31/2022).

[30] Flickr, Inc. "Download permissions", Flickr. (), [Online]. Available: https://www.flickrhelp.com/hc/en-us/articles/4404079715220-Download-permissions (visited on 07/31/2022).

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

[32] Flickr, Inc. "Flickr: The Flickr Developer Guide - API". (), [Online]. Available: https://www.flickr.com/services/developer/api/ (visited on 07/31/2022).

[33] "What are the API limits, actually? | Flickr API | Flickr", Flickr Help Forum. (Oct. 2, 2013), [Online]. Available: https://www.flickr.com/groups/51035612836@N01/discuss/72157636113830065/72157636114473386 (visited on 07/31/2022).

[34] H. Kumar Verma and R. Singh, "Performance Analysis of RC6, Twofish and Rijndael Block Cipher Algorithms", *International Journal of Computer Applications*, vol. 42, pp. 1–7, Mar. 1, 2012. DOI: 10.5120/5773-6002.

[35] X. Bonnetain, M. Naya-Plasencia, and A. Schrottenloher, "Quantum Security Analysis of AES", *IACR Transactions on Symmetric Cryptology*, vol. 2019, no. 2, p. 55, Dec. 6, 2019. DOI: 10.13154/tosc.v2019.i2.55-93. [Online]. Available: https://hal.inria.fr/hal-02397049 (visited on 08/24/2022).

[36] J. Ross Wallrabenstein. “When it Comes to Data Integrity, Can We Just Encrypt the Data? - EngineerZone Spotlight - EZ Blogs - EngineerZone”, ADI EngineerZone. (Nov. 17, 2021), [Online]. Available: https://ez.analog.com/ez-blogs/b/engineerzone-spotlight/posts/data-integrity-encrypt-data (visited on 08/24/2022).

[37] D. Khovratovich. “Answer to "Why should I use Authenticated Encryption instead of just encryption?"”, Cryptography Stack Exchange. (Dec. 7, 2013), [Online]. Available: https://crypto.stackexchange.com/a/12192 (visited on 08/24/2022).

[38] D. McGrew and J. Viega, “The Galois/counter mode of operation (GCM)”, *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[39] G. Kodwani, S. Arora, and P. K. Atrey, “On Security of Key Derivation Functions in Password-based Cryptography”, in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, Jul. 2021, pp. 109–114. DOI: 10.1109/CSR51186.2021.9527961.

[40] H. Krawczyk, “Cryptographic Extraction and Key Derivation: The HKDF Scheme”, 264, 2010. [Online]. Available: https://eprint.iacr.org/2010/264 (visited on 08/24/2022).

[41] H. Krawczyk and P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function (HKDF)”, Internet Engineering Task Force, Request for Comments RFC 5869, May 2010, 14 pp. DOI: 10.17487/RFC5869. [Online]. Available: https://datatracker.ietf.org/doc/rfc5869 (visited on 08/24/2022).

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

[43] D. Arias. “Adding Salt to Hashing: A Better Way to Store Passwords”, Auth0 - Blog. (Feb. 25, 2021), [Online]. Available: https://auth0.com/blog/adding-salt-to-hashing-

a-better-way-to-store-passwords/ (visited on 08/27/2022).

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

[45] Twitter. "Privacy Policy". (), [Online]. Available: https://twitter.com/en/privacy (visited on 02/15/2022).

[46] S. C. Foundation. "SIMARGL: Stegware primer, part 1". (Feb. 14, 2020), [Online]. Available: https://cuing.eu/blog/technical/simargl-stegware-primer-part-1 (visited on 02/09/2022).

[47] "Twitter images can be abused to hide ZIP, MP3 files — here's how". (), [Online]. Available: https://www.bleepingcomputer.com/news/security/twitter-images-can-be-abused-to-hide-zip-mp3-files-heres-how/ (visited on 02/09/2022).

[48] D. Buchanan, *Tweetable-polyglot-png*, Feb. 9, 2022. [Online]. Available: https://github.com/DavidBuchanan314/tweetable-polyglot-png (visited on 02/09/2022).

[49] J. Ning, I. Singh, H. V. Madhyastha, S. V. Krishnamurthy, G. Cao, and P. Mohapatra, "Secret message sharing using online social media", in *2014 IEEE Conference on Communications and Network Security*, Oct. 2014, pp. 319–327. DOI: 10.1109/CNS.2014.6997500.

[50] F. Beato, E. De Cristofaro, and K. B. Rasmussen, "Undetectable communication: The Online Social Networks case", in *2014 Twelfth Annual International Conference on Privacy, Security and Trust*, Jul. 2014, pp. 19–26. DOI: 10.1109/PST.2014.6890919.

[51] R. Anderson, R. Needham, and A. Shamir, "The Steganographic File System", in *Information Hiding*, D. Aucsmith, Ed., ser. 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.

[52] T. Chuman, W. Sirichotedumrong, and H. Kiya, "Encryption-Then-Compression Systems Using Grayscale-Based Image Encryption for JPEG Images", *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1515–1525, Jun. 2019, ISSN: 1556-6021. DOI: 10.1109/TIFS.2018.2881677.

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

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

[55] J. Domingo-Ferrer and M. Bras-Amorós, "A Shared Steganographic File System with Error Correction", in *Modeling Decisions for Artificial Intelligence*, V. Torra and Y. Narukawa, Eds., ser. 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.

[56] C. Sosa, B. Sutton, and H. Huang, "The Super Secret File System", 2007. [Online]. Available: https://www.cs.virginia.edu/~evans/wass/projects/ssfs.pdf (visited on 03/09/2022).

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

[58] J. Bieniasz and K. Szczypiorski, "SocialStegDisc: Application of steganography in social networks to create a file system", in *2017 3rd International Conference on Frontiers of Signal Processing (ICFSP)*, Sep. 2017, pp. 76–80. DOI: 10.1109/ICFSP.2017.8097145.

[59] R. Winslow. "Tweetfs/tweetfs at master · rw/tweetfs", GitHub. (), [Online]. Available: https://github.com/rw/tweetfs (visited on 04/06/2022).

[60] R. Jones. “Google hack: Use Gmail as a Linux filesystem”, Computerworld. (Sep. 15, 2006), [Online]. Available: https://www.computerworld.com/article/2547891/google-hack--use-gmail-as-a-linux-filesystem.html (visited on 03/09/2022).

[61] R. Jones. “Gmail Filesystem Implementation Overview”. (Apr. 11, 2006), [Online]. Available: https://web.archive.org/web/20060411085901/http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem-implementation.html (visited on 03/09/2022).

[62] B. Viksoe. “Viksoe.dk - GMail Drive shell extension”. (Apr. 10, 2004), [Online]. Available: http://www.viksoe.dk/code/gmail.htm (visited on 03/09/2022).

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

[64] S. Pușcaș, *Harababurel/gcsf*, Aug. 24, 2022. [Online]. Available: https://github.com/harababurel/gcsf (visited on 08/27/2022).

[65] Google. “Install and set up Google Drive for desktop - Google Workspace Learning Center”. (), [Online]. Available: https://support.google.com/a/users/answer/9965580?hl=en (visited on 08/27/2022).

[66] A. Strada, *Google-drive-ocamlfuse*, Jun. 17, 2022. [Online]. Available: https://github.com/astrada/google-drive-ocamlfuse (visited on 09/08/2022).

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

[68] J. Sneddon. “Mount Your Google Drive on Linux with google-drive-ocamlfuse”, OMG! Ubuntu! (May 10, 2017), [Online]. Available: http://www.omgubuntu.co.uk/2017/04/mount-google-drive-ocamlfuse-linux (visited on 09/08/2022).

[69] Y. Amin. “Use Google Drive as a local directory on Linux”, DEV Community. (Feb. 18, 2021), [Online]. Available: https://dev.to/yawaramin/use-google-drive-as-a-local-directory-on-linux-1b9 (visited on 09/08/2022).

[70] shubhamharnal. “In short, GCSF tends…”, r/DataHoarder. (Jul. 2, 2018), [Online]. Available: www.reddit.com/r/DataHoarder/comments/8vlb2v/google_drive_as_a_file_system/e1oh9q9/ (visited on 09/08/2022).

[71] harababurel. “Show HN: Google Drive as a file system | Hacker News”, Hacker News. (Jul. 1, 2018), [Online]. Available: https://news.ycombinator.com/item?id=17430397 (visited on 09/08/2022).

[72] E. Zadok, I. Badulescu, and A. Shender, “Cryptfs: A Stackable Vnode Level Encryption File System”, 1998. DOI: 10.7916/D82N5935. [Online]. Available: https://doi.org/10.7916/D82N5935 (visited on 03/04/2022).

[73] “FiST: Stackable File System Language and Templates”. (), [Online]. Available: https://www.filesystems.org/ (visited on 02/02/2022).

[74] “Iozone Filesystem Benchmark”. (), [Online]. Available: https://www.iozone.org/ (visited on 03/07/2022).

[75] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, “Benchmarking file system benchmarking: It *IS* rocket science”, in *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA: USENIX Association, May 2011. [Online]. Available: https://www.usenix.org/conference/hotosxiii/benchmarking-file-system-benchmarking-it-rocket-science.

[76] U. K. Agarwal. “Comparing IO benchmarks: FIO, IOZONE and BONNIE++”, FuzzyWare. (May 19, 2018), [Online]. Available: https://uditagarwal.in/comparing-io-benchmarks-fio-iozone-and-bonnie/ (visited on 03/13/2022).

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

[78] J.-P. Barrette-LaPierre, *cURLpp*, Aug. 23, 2022. [Online]. Available: https://github.com/jpbarrette/curlpp (visited on 08/27/2022).

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

[80] *Liboauth*. [Online]. Available: https://sourceforge.net/projects/liboauth/ (visited on 08/27/2022).

[81] D. Beckett. "Flickcurl: C library for the Flickr API". (), [Online]. Available: https://librdf.org/flickcurl/ (visited on 08/27/2022).

[82] "Crypto++ Library 8.7 | Free C++ Class Library of Cryptographic Schemes". (), [Online]. Available: https://www.cryptopp.com/ (visited on 08/27/2022).

[83] G. Kuenning. "CS135 FUSE Documentation". (2010), [Online]. Available: https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html (visited on 07/30/2022).

[84] B. Fleischer. "Mount options · osxfuse/osxfuse Wiki", GitHub. (Nov. 30, 2020), [Online]. Available: https://github.com/osxfuse/osxfuse (visited on 11/07/2022).

[85] "Cryptopp : Security vulnerabilities". (), [Online]. Available: https://www.cvedetails.com/vulnerability-list/vendor_id-15519/Cryptopp.html (visited on 08/27/2022).

[86] NolanOBrien. "Upcoming changes to PNG image support", Twitter Developers. (Dec. 20, 2018), [Online]. Available: https://twittercommunity.com/t/upcoming-changes-to-png-image-support/118695 (visited on 09/06/2022).

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

[88] LLC SysDev Laboratories. "How to recover data from encrypted Apple APFS", UFS Explorer. (Aug. 19, 2022), [Online]. Available: https://www.ufsexplorer.com/articles/how-to/recover-data-apfs-encryption/ (visited on 09/11/2022).

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

[90] A. Santos. “How to Recover Data from an APFS Hard Drive on Mac [a Full Guide]”, Macgasm. (Dec. 10, 2021), [Online]. Available: https://www.macgasm.net/data-recovery/apfs-data-recovery/ (visited on 09/11/2022).

[91] Z. Z. Coder. “Answer to "Size of data after AES/CBC and AES/ECB encryption"”, Stack Overflow. (Jul. 19, 2010), [Online]. Available: https://stackoverflow.com/a/3284136/8138631 (visited on 09/11/2022).

[92] Apple Inc. “What is secure virtual memory on Mac?”, Apple Support. (), [Online]. Available: https://support.apple.com/en-gb/guide/mac-help/mh11852/mac (visited on 09/11/2022).

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

[94] Internetstiftelsen. “Bredbandskollen CLI | Bredbandskollen”. (), [Online]. Available: http://www.bredbandskollen.se/om/mer-om-bbk/bredbandskollen-cli/ (visited on 10/11/2022).

[95] Internetstiftelsen. “Mer om Bredbandskollen | Bredbandskollen”. (), [Online]. Available: http://www.bredbandskollen.se/om/mer-om-bbk/ (visited on 10/11/2022).

[96] G. Olsson, *Fejk File System*, Jul. 28, 2022. [Online]. Available: https://github.com/GlennOlsson/FFS (visited on 10/09/2022).

[97] “RandomNumberGenerator - Crypto++ Wiki”. (Apr. 13, 2021), [Online]. Available: https://cryptopp.com/wiki/RandomNumberGenerator (visited on 09/11/2022).

[98] geekosaur. “Answer to "Why are dot underscore ._ files created, and how can I avoid them?"”, Ask Different. (May 29, 2011), [Online]. Available: https://apple.stackexchange.com/a/14981 (visited on 09/11/2022).

[99] A. Rudd. “Encryption and counter-terrorism: getting the balance right”, GOV.UK. (Jul. 31, 2017), [Online]. Available: https://www.gov.uk/government/speeches/encryption-and-counter-terrorism-getting-the-balance-right (visited on 09/11/2022).

[100] D. Berreby, “Engineering Terror”, *The New York TimesMagazine*, Sep. 10, 2010, ISSN: 0362-4331. [Online]. Available: https://www.nytimes.com/2010/09/12/magazine/12FOB-IdeaLab-t.html (visited on 09/11/2022).

[101] J. McLean. “Data centers generate the same amount of carbon emissions as global airlines”, TNW | Syndication. (Feb. 15, 2020), [Online]. Available: https://thenextweb.com/news/data-centers-generate-the-same-amount-of-carbon-emissions-as-global-airlines (visited on 10/09/2022).

[102] F. Pearce. “Energy Hogs: Can World’s Huge Data Centers Be Made More Efficient?”, Yale E360. (), [Online]. Available: https://e360.yale.edu/features/energy-hogs-can-huge-data-centers-be-made-more-efficient (visited on 10/09/2022).

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, which is also encoded into the binary representation of the object when it is serialized before 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.2 visualizes the serialized format of the `InodeEntry`. Figure B.3 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.4 visualizes binary format of the FFS header. Figure B.5 visualizes the PCD of FFS images stored on the OWS.

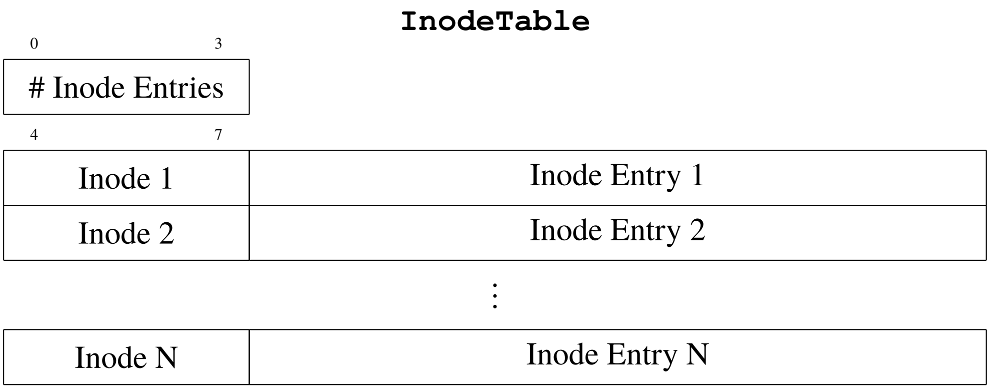

Figure B.1: `# Inode Entries` is an unsigned integer representing the amount of inode entries the inode table contains. Following are `# Inode Entries` entries of an unsigned integer representing the inode of the inode entry, and the serialization of the corresponding `InodeEntry` object

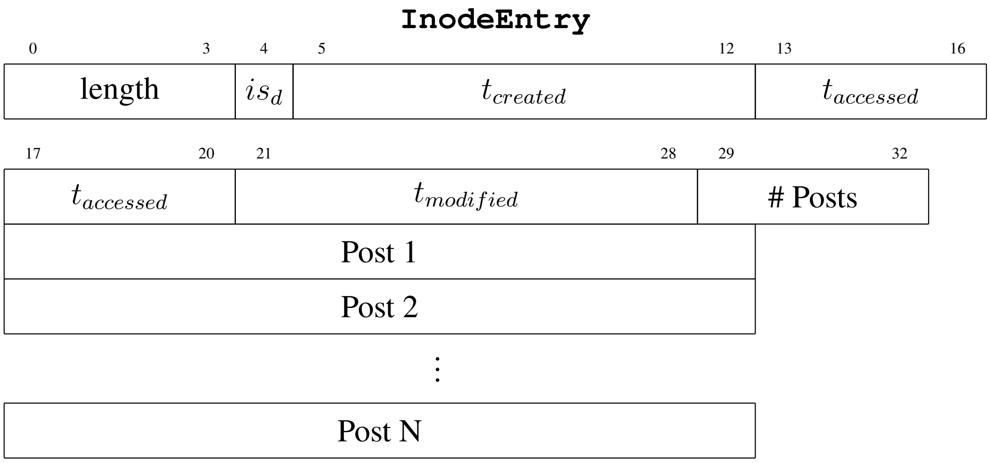

Figure B.2: Byte representation of a serialized `InodeEntry`, representing a file or directory stored in FFS. `length` is an unsigned integer representing the amount of data stored on FFS by the file or directory, for instance the size of the file. is_d is a boolean with the value true ($\neq 0$) if 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

`Directory`

0–3
Entries

4–7	
Inode 1	Filename 1
Inode 2	Filename 2
⋮	
Inode 3	Filename 3

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. However, as the strings are null-terminated, the deserializer can read the bytes until the null-character is found

FFS Header

0	1	2	3	4–11	12–15
'F'	'F'	'S'	V	Timestamp	Data length

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.

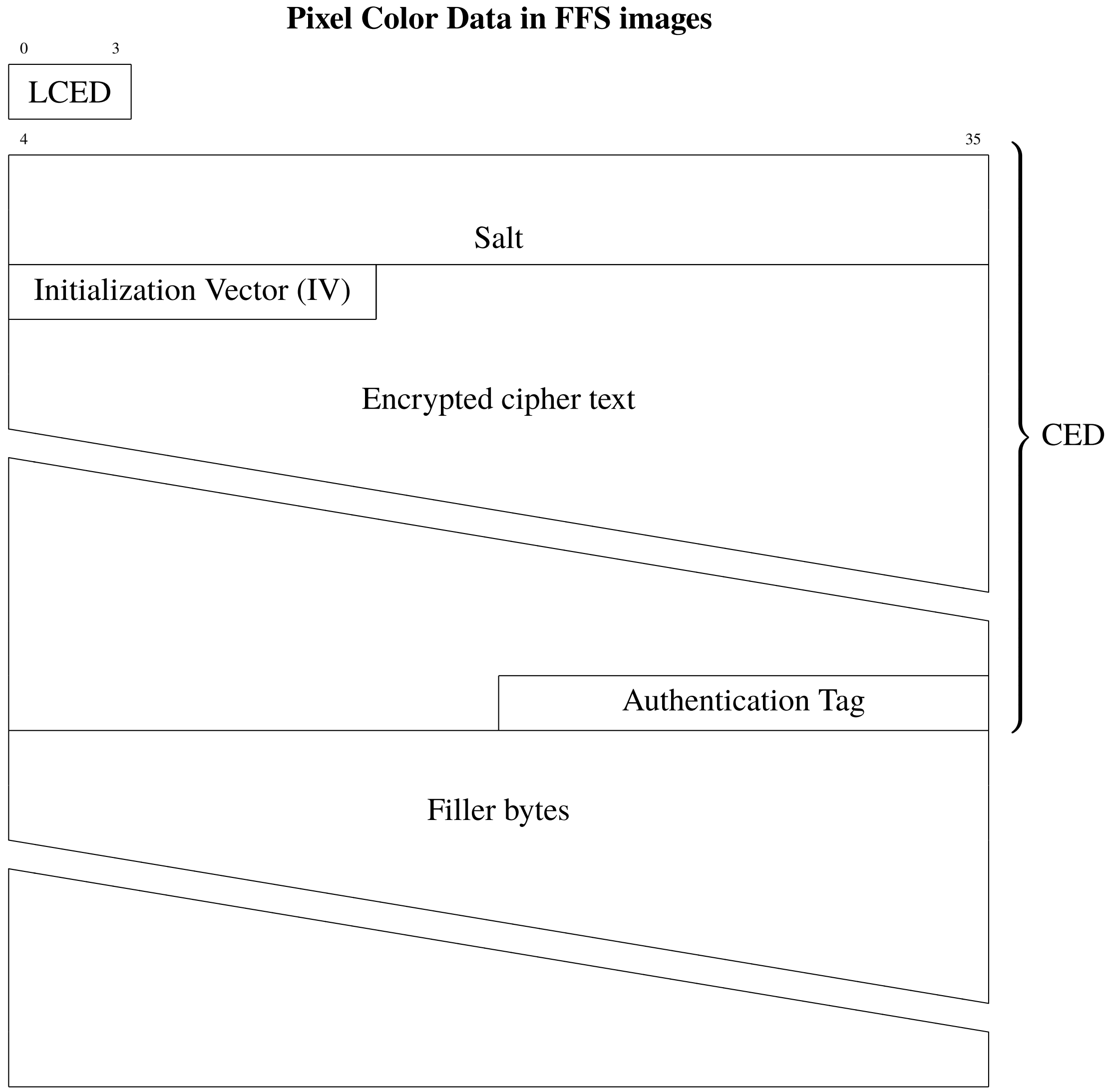

Figure B.5: Byte representation of the data stored as PCD in FFS images. `LCED` is an unsigned integer representing the Length of the CED. The `Salt` is a 64-byte randomized vector used to derive the encryption and decryption key. The `IV` is a 12-byte randomized vector used as the initial state of the 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 and figures of the IOZone benchmarking outputs produced. Each section contains the tables and figures for the benchmarking results of the specific filesystem. Each table and each figure represents one IOZone test. Each table presents the throughput in kilobytes per second for a test, for the different file sizes and buffer sizes, both presented in kB. Each graph in the figures visualizes the performance of the filesystem for different file sizes. The x-axis shows the buffer size in kilobytes, and the y-axis is the throughput in kilobytes per second.

C.1 FFS

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1456510	1066594	2173780	2369237	2950233	2760606	898	3709575	789				
2048	2333076	2208887	2653011	1858	3598924	4031308	4937295	3561619	819	2077			
4096	693	3490	2960	4404	1956079	3661	4887857	4779081	4320017	3519162	3474326		
8192	2280643	3279	3342	5599873	5830794	4641448	3193	6434642	7953	3942233	3609	4620228	
16384	1874825	3111899	4509505	5586129	6124836	6901759	4695940	6551072	5746830	4964991	3992114	5107050	4687931

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	241	259	148	268	244	287	184	272	242				
2048	448	527	290	491	441	412	480	426	535	463			
4096	724	802	800	667	742	683	618	599	605	629	836		
8192	1001	758	766	788	814	790	780	970	1116	1148	982	1091	
16384	865	914	1273	1366	1273	1222	1360	1306	884	1375	878	905	847

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1577984	1200447	3160839	2201637	3506651	3172513	3778101	4317393	468				
2048	2455806	2162727	2705656	817	3931669	3969827	5489457	2131071	3476572	988			
4096	1795	1664	2058	1517	4597474	1775	5761565	1727	1783	3707507	1768		
8192	1625049	2614736	3670204	5191229	4307455	4691514	6173362	6511465	3795898	4586921	5295233	4759753	
16384	1856740	3735402	4984437	5134140	6360084	6778535	5349148	6773190	6194400	5444935	4963198	4898106	4572517

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	266	258	171	220	250	321	283	128	259				
2048	250	479	495	489	252	256	248	283	410	435			
4096	632	783	791	660	416	735	594	591	600	617	733		
8192	674	707	711	531	583	548	687	568	863	1006	938	928	
16384	640	684	943	942	855	819	915	908	680	1023	668	641	746

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1276817	1347311	2653170	2805690	3390391	3345496	2375790	4920874	3472628				
2048	1765367	2031	2464968	2810101	3390131	4301837	5507054	4023754	3864455	3806235			
4096	1520503	1958756	3474326	3063563	3857345	3686820	6321227	4227532	4541567	4729091	4028288		
8192	1512525	2613145	2697461	5247519	5780764	4816466	6496691	6220301	4352746	3665505	5221205	4628318	
16384	1580092	3033998	4045461	4921609	6377200	7114697	6585601	7001613	6215691	6440555	5531719	5013529	4655852

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	255	253	243	267	215	257	216	220	232				
2048	210	263	476	394	418	297	460	423	418	511			
4096	660	835	805	644	748	605	600	570	559	611	786		
8192	806	714	668	719	721	671	677	806	922	743	931	948	
16384	808	784	1145	1129	1076	1047	1107	756	1141	1077	806	760	792

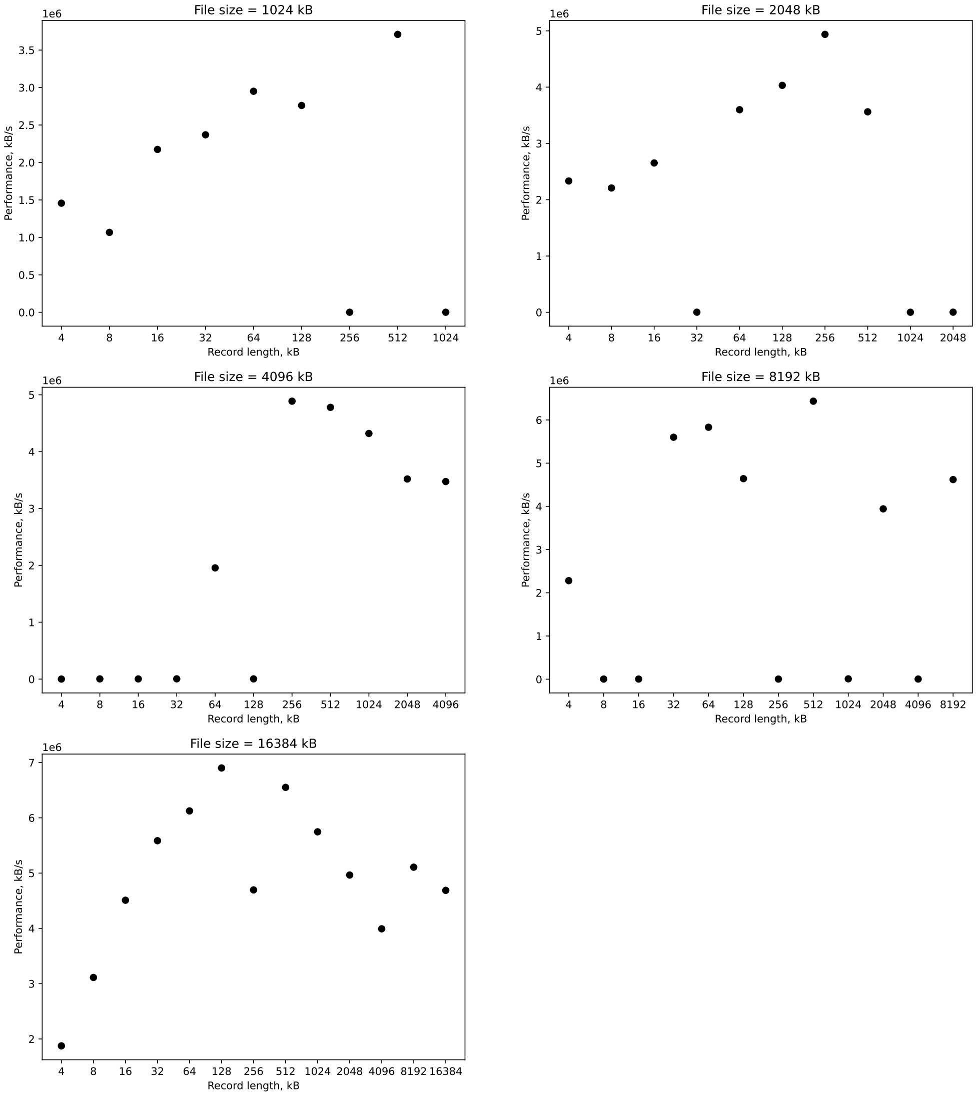

Figure C.1: IOZone output for FFS Forward Read

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	903860	1087111	3794792	2573677	3210456	3483896	3835457	3436508	5219904				
2048	1663811	2709070	3555722	5478953	3391470	3292669	5904523	7207495	5403134	4423683			
4096	1905950	2728780	4188367	3857345	5642353	6649056	6311938	6410863	5445618	5093643	4977067		
8192	579472	3532842	3713439	5244315	4768340	6643663	4282224	3957671	5494212	5159270	4659702	5155399	
16384	2691208	3620090	4594528	5150687	5910434	5935961	6951327	6052550	5719090	6875519	5964297	5789435	5736755

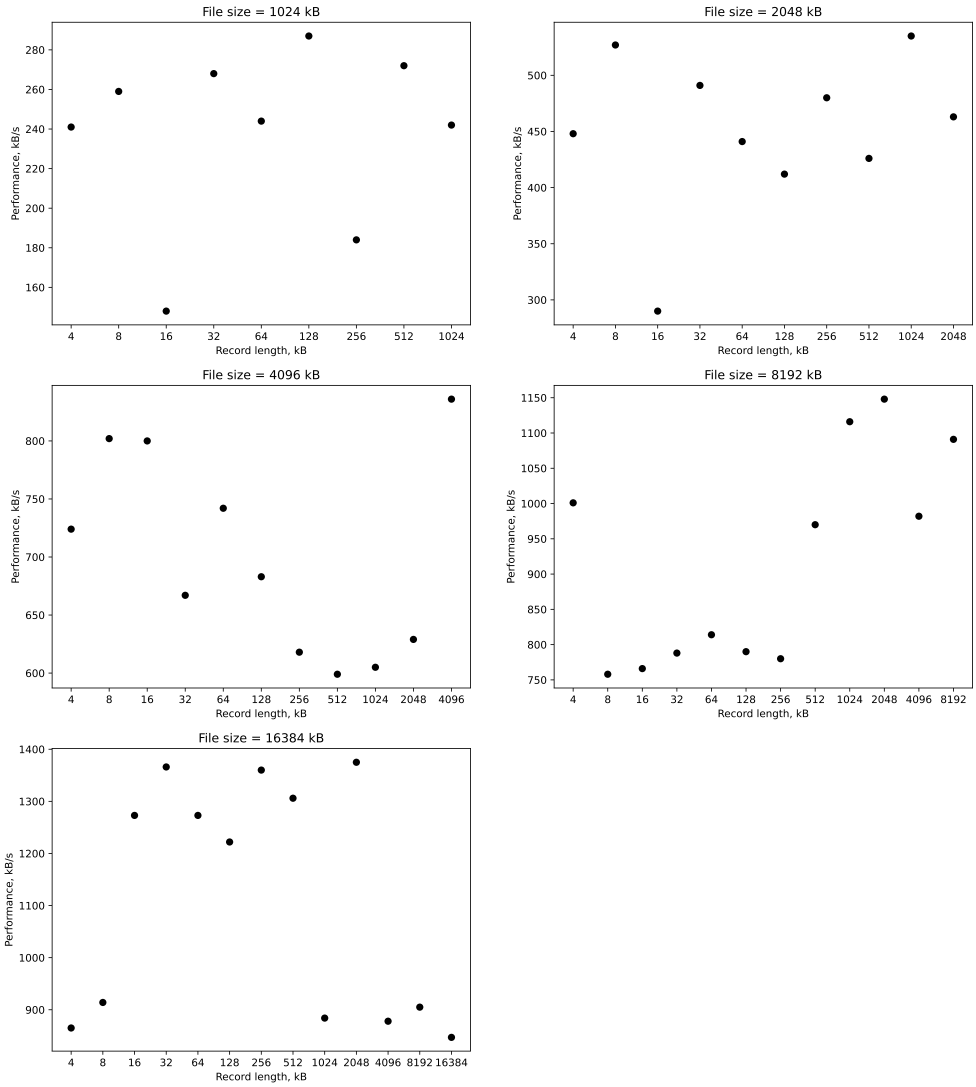

Figure C.2: IOZone output for FFS Forward Write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	229	505	413	505	479	515	535	541	523				
2048	447	1028	957	1003	938	957	965	422	955	999			
4096	1872	1808	1876	1947	1795	1730	1895	1779	519	1811	1703		
8192	3065	3250	3253	3328	2764	3214	3313	3014	2484	3165	3144	3234	
16384	4370	4705	4651	4518	4848	4855	4739	4624	4609	4786	4739	4752	4983

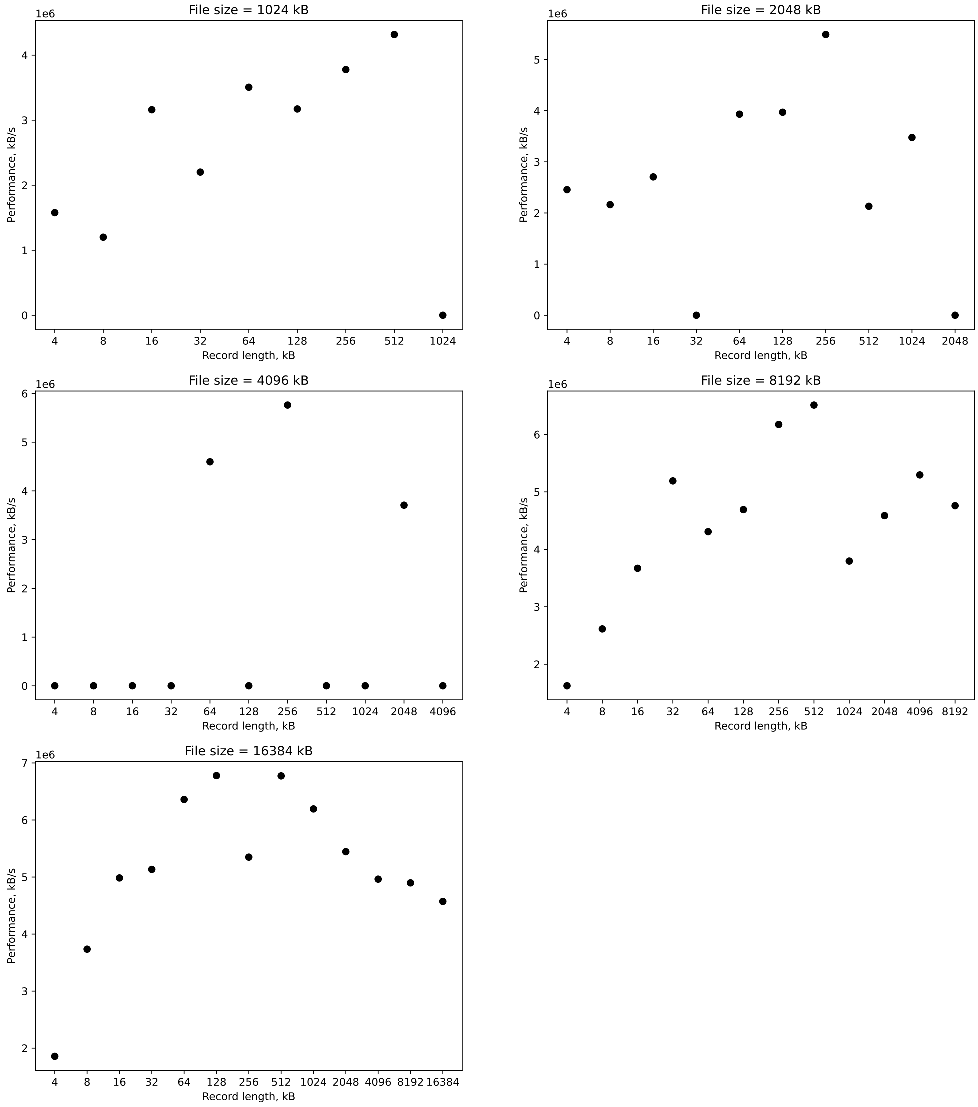

Figure C.3: IOZone output for FFS Re-Read

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1835608	1414304	4470172	3567824	4433259	3908759	4783849	2837198	5853003				
2048	2913021	4367454	5785224	6378005	5769681	3543986	6802261	8464610	6440169	5884299			
4096	2448388	3416292	4452120	5955295	6041154	7001316	6213770	5389246	6815229	6005255	5810280		
8192	3217865	4558321	4929091	6405850	6713763	6611703	7167528	7373641	6590145	6703285	4776295	5184180	
16384	2898845	4149775	5004037	6104162	7004468	7275131	7585931	7291341	6815511	7278984	6335457	5785536	5570732

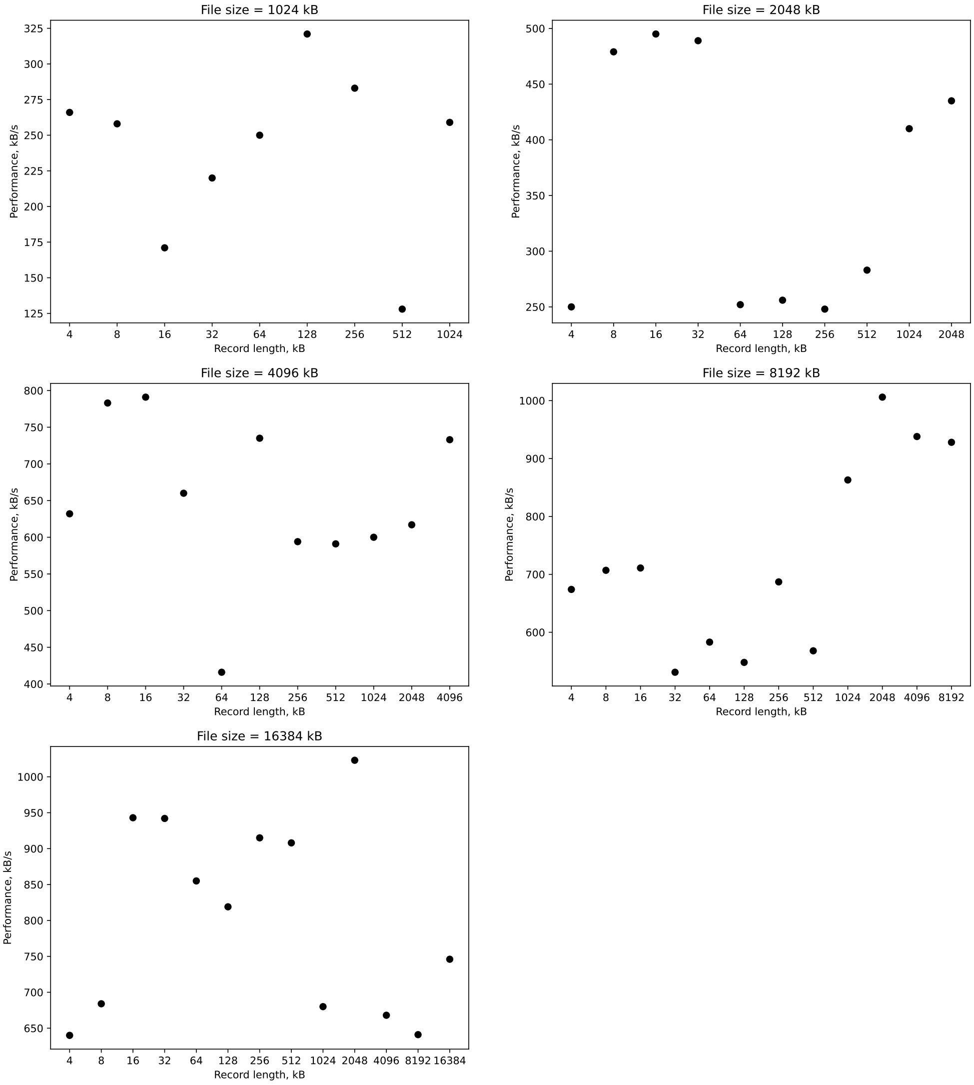

Figure C.4: IOZone output for FFS Re-Write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	352	201	409	412	487	420	200	442	438				
2048	600	765	625	759	731	722	663	742	703	718			
4096	934	1100	795	762	1295	1501	1421	1424	1444	1423	1209		
8192	293	1197	318	602	2306	2086	2254	2239	2247	2201	2252	2200	
16384	1420	1472	1264	1563	695	2986	2074	2996	3126	3033	2649	3000	2890

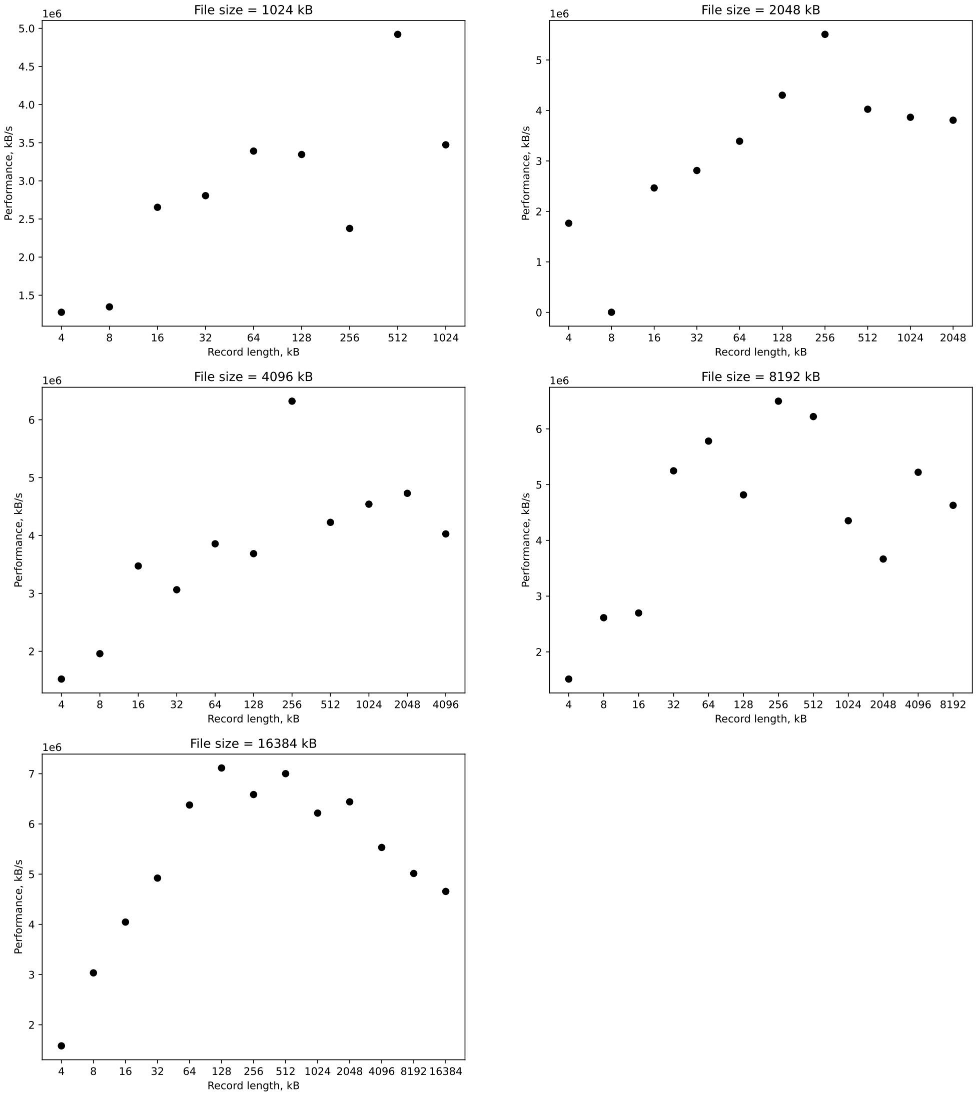

Figure C.5: IOZone output for FFS Random read

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1304348	1610528	4079544	3003881	4199201	3835457	4334822	3239514	4783849				
2048	2250555	3513544	5303064	7477273	5416763	3336148	6759439	8390200	3624742	5144271			
4096	1894600	2736168	3124291	5779008	5535098	7087972	6420447	5721272	7314299	6480999	6060333		
8192	2315532	3448109	3462355	5777847	6244039	5347148	7074554	7224798	6725591	6989645	5610847	5333039	
16384	2009294	3109505	4169920	5403406	6534875	6998048	7320858	6745267	6725463	7420464	5931862	5814418	5714334

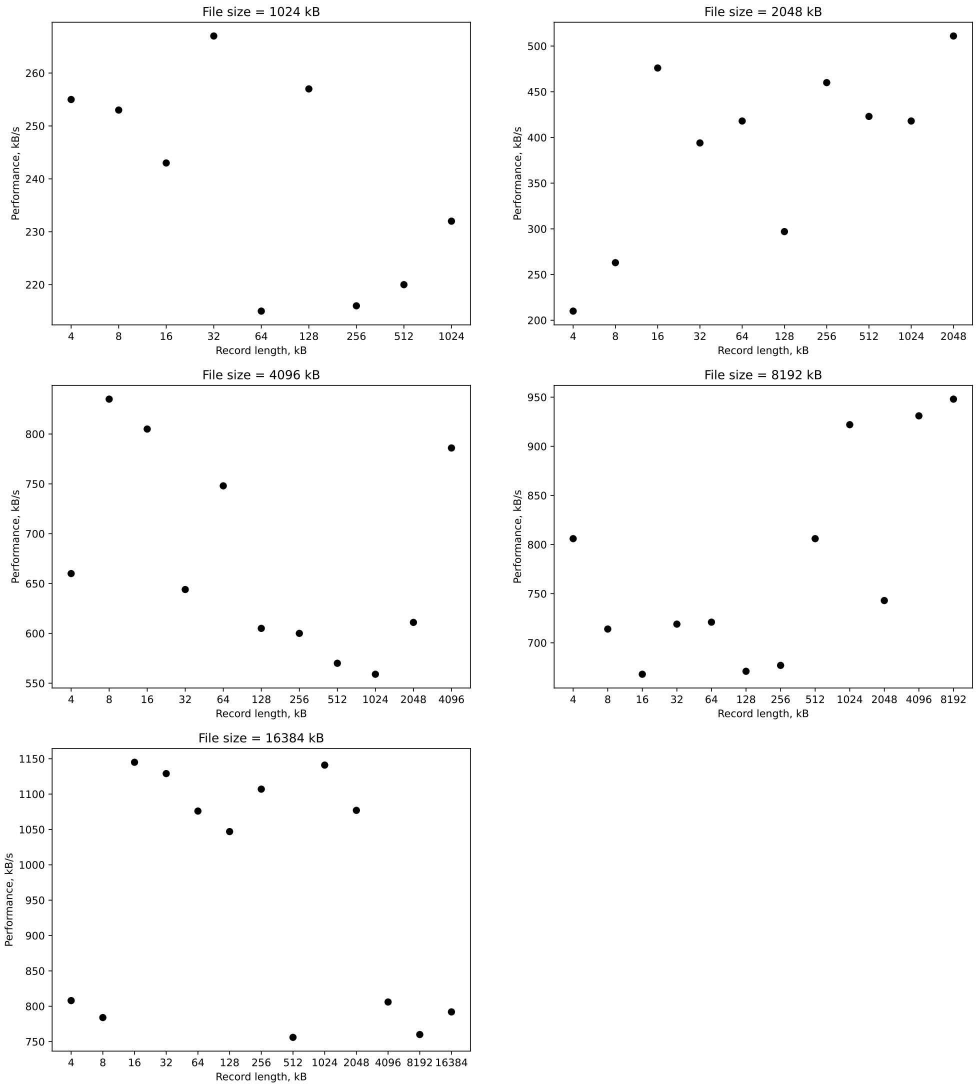

Figure C.6: IOZone output for FFS Random write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	451	261	394	439	383	383	372	224	466				
2048	687	811	785	808	715	632	712	798	805	564			
4096	888	1337	1459	1363	1440	1323	1403	1473	1099	1426	1231		
8192	2208	865	2220	2293	2177	2227	2213	2120	2151	2233	2207	2130	
16384	1282	1357	1560	3266	3523	2971	2604	2931	3001	3044	2973	3090	1270

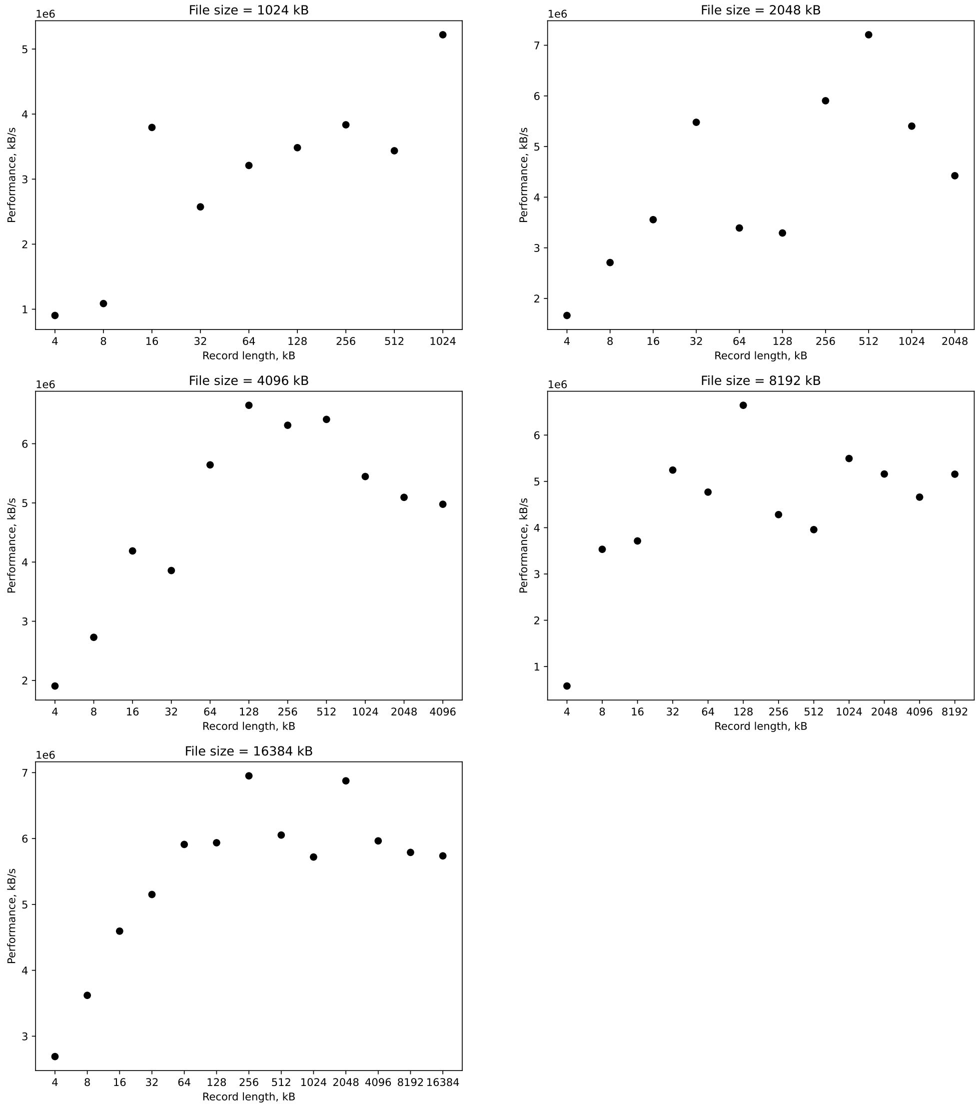

Figure C.7: IOZone output for GCSF Forward Read

C.2 GCSF

C.3 Fejk FFS

C.4 APFS

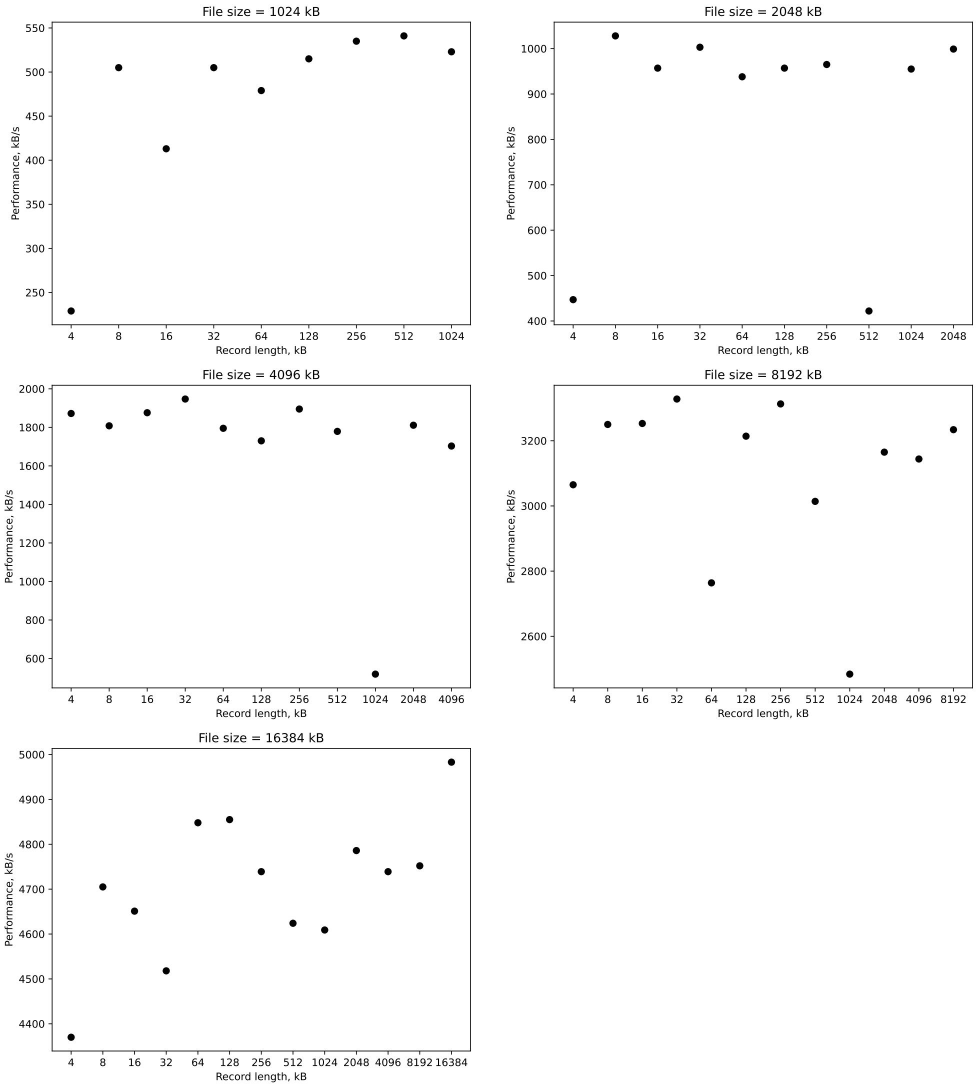

Figure C.8: IOZone output for GCSF Forward Write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1605111	2472911	889631	2715230	3618930	3001782	1032244	2578312	2467229				
2048	1921339	1930406	2194777	2821176	3200654	4130162	4154130	2649737	5032754	3286370			
4096	2426601	2784507	3271230	3574071	3670280	5542240	4386193	3771811	3995498	4763181	4000150		
8192	2895693	3535750	4173008	5378957	4899570	4548665	5914094	6558696	4271046	4104220	5475825	5888754	
16384	3094244	4152032	4908602	3512941	6234299	5855539	6313919	7272051	6212881	7111751	5507336	5387732	5486670
32768	2848900	3537882	4681672	6104381	6554760	5933093	6802626	6375072	6339199	6146697	6030191	5591994	5440583
65536	70116	3562495	5159913	5999237	7335592	7026566	7648924	6445904	77043	129764	5783761	5526649	4978066
131072	394463	2034461	454646	6311863	3782086	280028	7525973	6890568	75477	59220	2674224	238745	657556
262144	65486	75529	77854	97658	79916	72736	76276	71780	106689	72269	72176	79924	126614
524288	132857	78791	80051	80883	94834	82109	81531	81694	77350	79706	87932	79187	84020

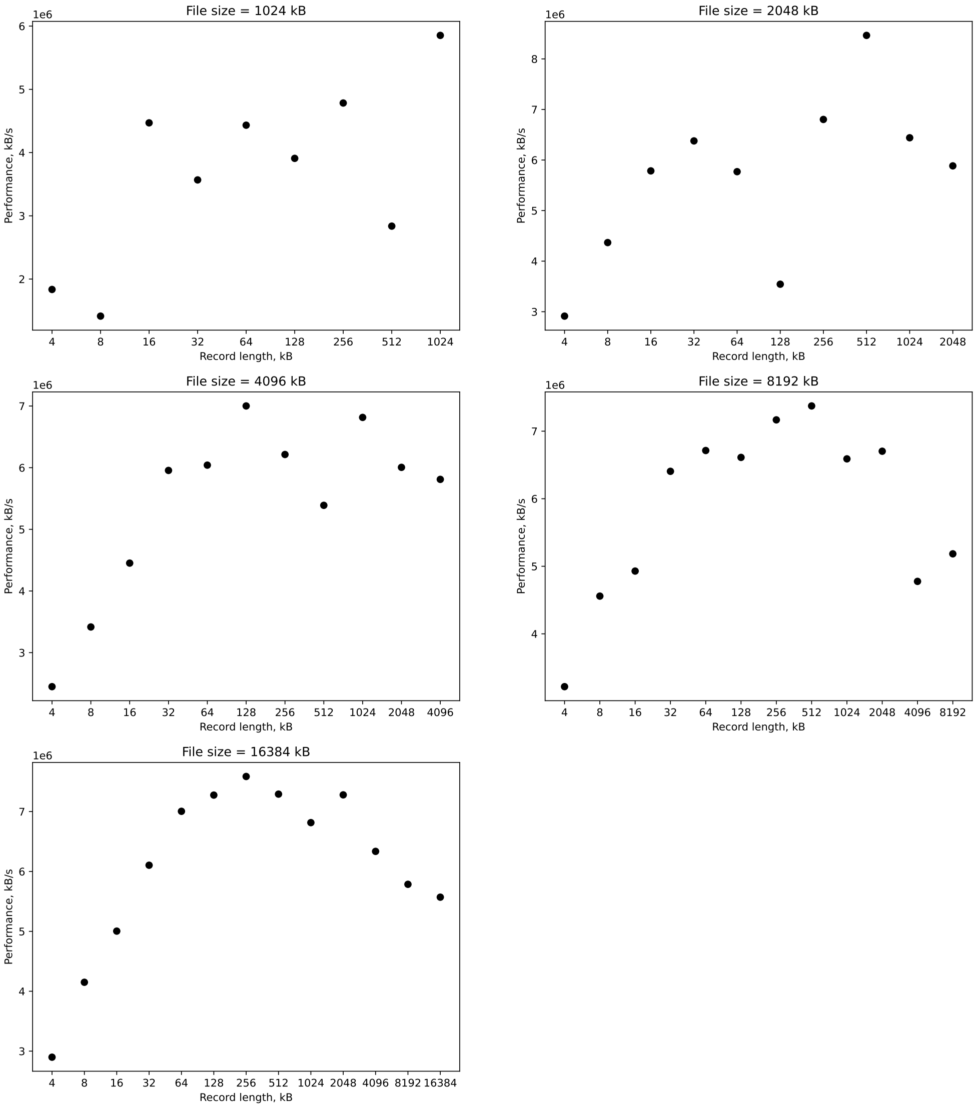

Figure C.9: IOZone output for GCSF Re-Read

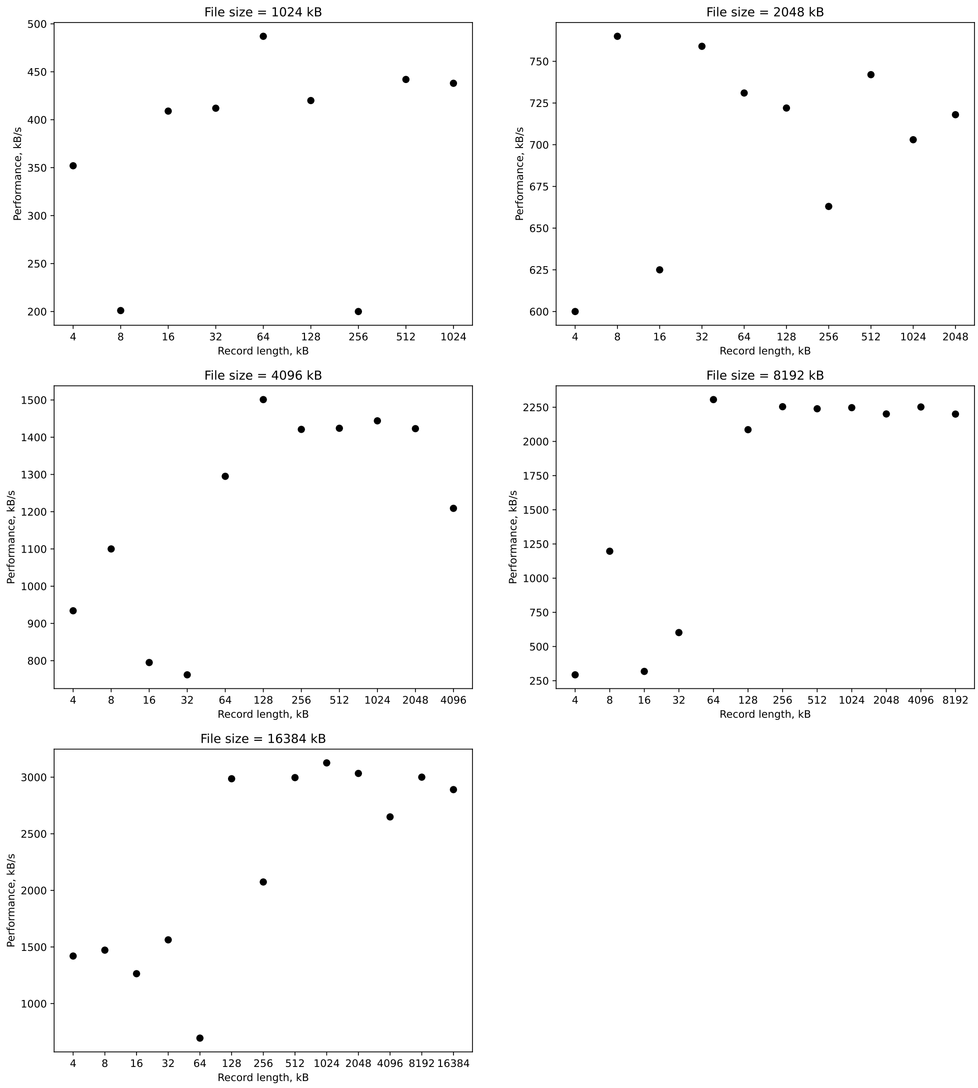

Figure C.10: IOZone output for GCSF Re-Write

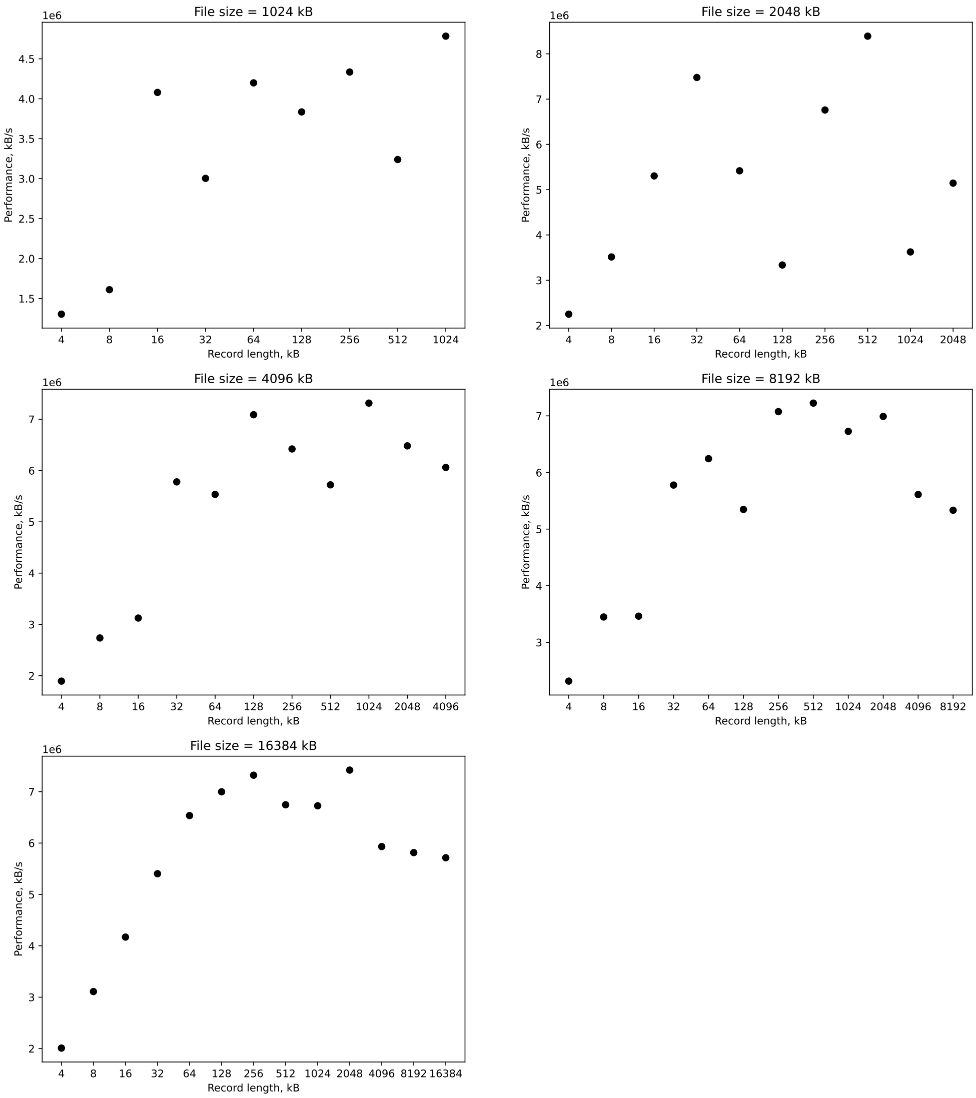

Figure C.11: IOZone output for GCSF Random read

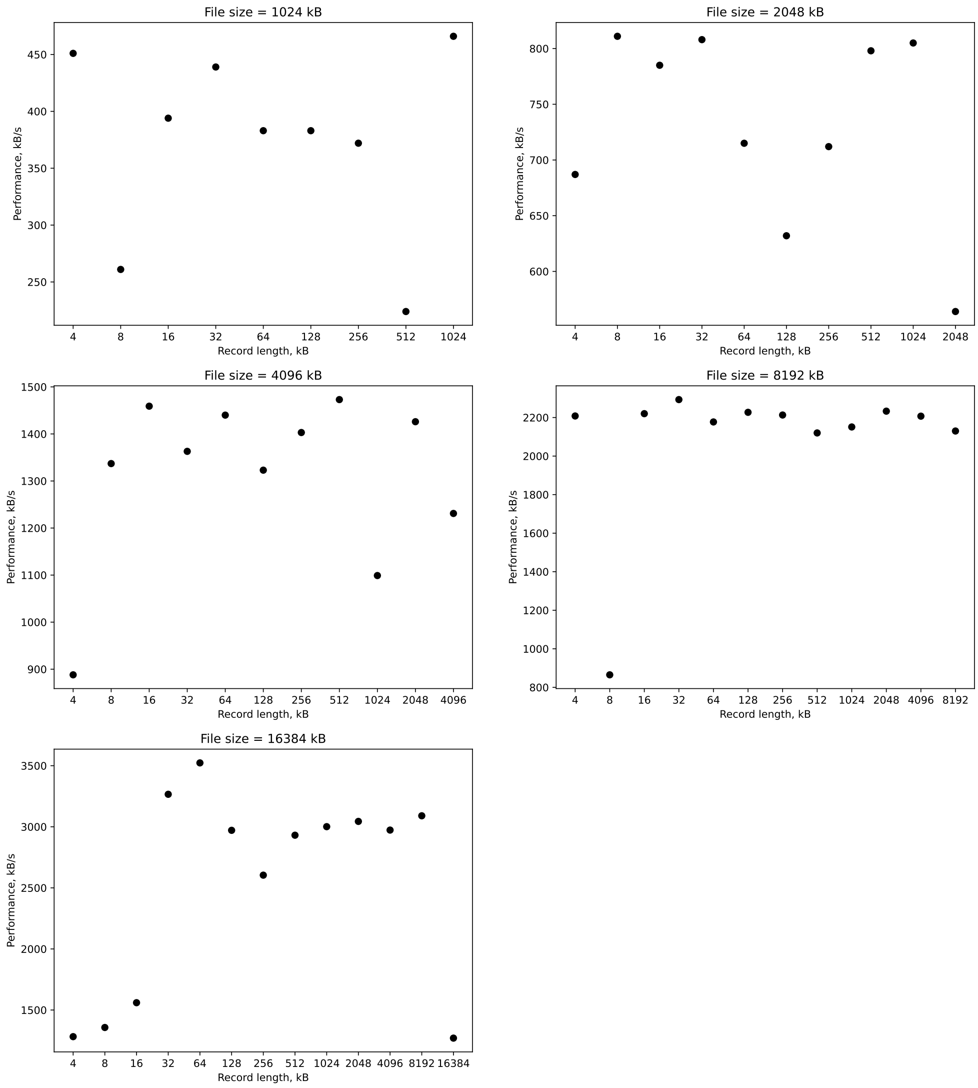

Figure C.12: IOZone output for GCSF Random write

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

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

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1984913	2820430	2353657	4674510	4232305	3198502	1920993	4943530	3142339				
2048	2107025	3011047	2367151	3915540	4633675	5209791	5119743	4663865	4829046	3736694			
4096	3075079	3614679	4462528	4302706	5971855	6503078	4906002	6471234	4132949	5184332	3442303		
8192	2826843	4114541	4678737	5261179	4158362	6182248	5094251	5357152	6537482	5382328	5756551	5915112	
16384	3115426	4517212	5147601	6343059	6609036	6262707	6753222	7984318	6787239	7246745	5728625	5310703	5452279
32768	2786858	3742299	5009656	7004755	7299786	7123475	7777741	6994062	6752162	7039557	6382473	5967613	5664360
65536	2205856	4396090	5428742	6393876	6889031	6089074	7811071	6860318	7461842	6801920	6440769	5750246	4823451
131072	3287977	3847707	5198793	6348527	6951999	8443258	7909267	7114111	7388515	5298401	3139222	5802490	6479087
262144	2974653	4452026	5376537	6145825	7115944	7217446	7116911	6533364	6027138	6506108	5360469	5598732	5819467
524288	2694624	4244212	5595026	6429351	8954377	7308594	7499700	7481075	8812173	7601901	7190208	5874969	6654414

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	4574	4975	1566	5157	4391	4913	3380	27982	5001				
2048	3371	4654	5206	4614	5631	5635	5680	31266	5577	5685			
4096	5548	5728	5588	5188	5330	5148	5077	5928	5449	4753	5515		
8192	5296	5167	5574	5177	5002	5540	5362	5527	5913	5893	5485	4895	
16384	5933	5866	6309	6263	5714	6230	6215	6563	6454	6412	5640	25482	6565
32768	6314	6369	6575	6687	6841	6852	6824	6819	6889	6768	6782	6699	6885
65536	5933	6327	5753	6290	6514	5615	6329	6632	5962	5815	5749	5935	25761
131072	5885	5619	6099	6598	6601	5990	6717	6763	5816	4048	6124	4737	6436
262144	5769	6203	6474	6722	6879	6865	6565	6140	6177	6033	6164	6586	6653
524288	6125	6552	6680	6653	6891	6797	6802	6883	6882	6789	6948	6789	6758

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1438461	2474336	3335105	4339202	4574926	3594699	1662264	4415030	3390391				
2048	1639674	2531808	2980747	3071337	4971585	5626082	5235193	4240255	3864455	4591569			
4096	2333965	2898672	4076076	4214051	3641495	6716643	3128843	6022095	4428023	4935601	5551194		
8192	2217071	1882745	4242044	6006101	3655755	6054786	6947247	6912307	6339662	5785631	5789530	5583494	
16384	2235520	2613523	4010753	5637917	5897753	5362924	6542963	7922644	6671272	7117644	4702688	5233056	5189585
32768	1953956	2985674	4118168	6358850	7157605	6732647	7552086	7015482	6541968	7126430	6545707	5985025	5650853
65536	1320279	3522910	4420978	5494832	6317898	6091233	7578280	6635907	7317626	6684480	6613236	5663404	4218907
131072	2179297	3011967	4211016	5669935	6324062	7899720	7685258	6951208	7292757	5564284	2867974	5863448	6536945
262144	2105727	3331986	4265958	5398262	6599796	7121152	6889437	6293359	6270783	5819868	5256751	5823875	5831010
524288	1839859	3104575	4555259	5698917	8056924	6869955	7439341	7032130	8596724	7549366	7260406	6177468	6877711

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	4619	4008	2880	4530	4425	4764	3097	5879	4465				
2048	4620	4691	5179	5665	5728	5660	4573	5554	5775	5519			
4096	5284	5681	5333	5194	5488	5822	5854	6127	37216	5145	5738		
8192	5442	4747	5737	5386	5800	5088	5881	5811	5414	5708	4852	5635	
16384	5553	6011	6191	6267	6188	5654	6348	6585	6428	6509	5831	6558	6496
32768	6253	6238	6434	6511	6810	6829	6984	6800	6877	6801	6853	6875	6857
65536	4967	6318	5379	5753	6616	6361	6528	6611	5854	5900	5980	6472	5881
131072	5754	5611	6114	6317	6102	6407	6469	6662	4629	4861	5661	5536	6443
262144	5741	6259	6439	6689	6907	6427	6463	6100	6309	5998	6035	6849	6857
524288	6054	6530	6737	6806	6715	6910	6903	6940	6916	6894	6893	6785	6981

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2300703	3472628	5444898	6025439	3425544	2353657	3521025	5479632	1259592				
2048	2429413	3548378	2324238	2288326	2392202	7817519	5277002	6939647	6867521	7529708			
4096	2597576	3931494	4922872	3456849	6151473	6908408	7064655	5894001	4648477	5402804	6281934		
8192	2693866	3716653	4555903	5487193	4143818	6405850	8054689	7585256	5708730	7537004	4594281	5576245	
16384	1887130	3339494	5330063	3537899	6635840	6409917	6430310	4554938	6921222	3453620	2166589	5158807	4187197
32768	2452479	2775321	4702013	6173478	5991547	6411354	6621073	5601338	7129387	6994417	5692749	5448563	4751758
65536	2673966	4183591	5032015	6032415	6770924	6720270	6932816	7149936	6714032	6609102	6538515	5680492	5542138
131072	2775537	4124081	5119016	5613831	5348765	6775497	4877838	5514443	4523778	5408171	2241853	2555460	2956665
262144	2612033	4088592	5074689	6218999	6861877	6715279	6735808	7042156	6728553	3955521	5947137	4522621	5227609
524288	2831691	4069862	5013312	5936378	6222708	5719104	6886219	7442614	6690431	6702136	6003539	5352820	5317805

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	159475	302147	204797	669331	770562	1528563	931702	371703	1340164				
2048	194475	310725	254254	368875	880855	1237420	1097081	1219155	1295263	1823453			
4096	199931	318406	640709	611714	1033076	1064440	1039326	931976	900421	1207570	1213627		
8192	204824	316817	385558	593660	651767	589312	784669	772167	831858	729741	672246	823307	
16384	204440	287428	473154	599882	656623	774480	800275	732212	736592	786598	815402	797276	729763
32768	144265	292913	473815	630616	713183	641806	729863	628379	622493	659873	643456	669350	747666
65536	251368	348690	492085	481153	751525	761160	762827	492373	784232	780599	465990	763618	781920
131072	214970	346696	490691	576345	707747	631216	649070	782725	785093	624839	724503	447659	787787
262144	204778	333219	414944	473988	524245	596270	688020	648681	713499	591206	417813	724537	704260
524288	302386	398100	352544	407122	735632	577917	752389	716243	777618	661744	640415	527875	613906

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2672984	3999762	7590887	8246774	3582705	2820430	5226256	7208671	4014717				
2048	2519185	3878414	2094183	2531808	3319389	9752360	3855782	10143926	8784909	10713237			
4096	2659914	4201683	5497899	4581535	6628532	7756829	8205092	7447471	6066754	4853336	5445618		
8192	2906963	3849914	3912159	6542461	6848927	5607184	8532752	7550253	7593638	8127085	5809106	5501249	
16384	2220998	2777913	5518393	2383791	6829736	6747917	6622411	2365574	7304517	4617684	2171450	5157259	3420444
32768	2515593	422414	4812321	6197698	6090584	6317346	6907241	6022528	7163948	7024087	5715476	5106444	5972021
65536	2746120	4150992	5069321	6282807	6731955	6501553	6983011	6811697	6463182	6476583	6588509	5450919	5876120
131072	2755173	4126310	5114587	5225974	4340138	6555887	5927107	5109264	4320967	5892231	2764011	3077523	4241949
262144	2857991	3711246	5355664	6016321	6473815	6511426	6046594	7033237	6847177	3413775	6368117	5259819	4680728
524288	2838776	3987466	5016823	5765649	6391807	5818309	6905530	7615063	7011905	5884007	5375391	4806809	4982732

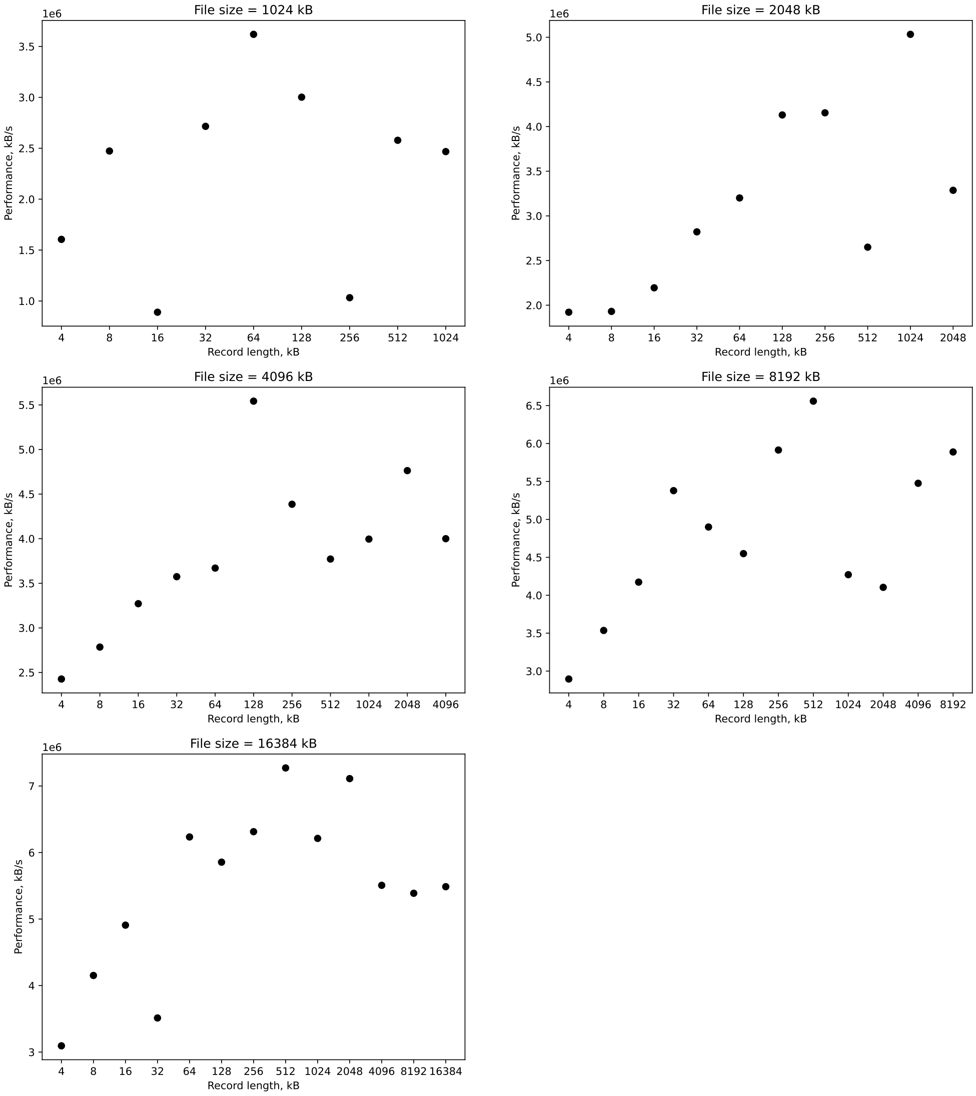

Figure C.13: IOZone output for Fejk FFS Forward Read

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	516668	888894	1378002	1271902	1232484	1896395	760735	2015654	793775				
2048	446281	1080657	1063926	1030239	111521	2704804	2258247	2365196	2458618	3245396			
4096	607259	784547	1234469	1146103	1474438	1693297	1823802	1577900	1624143	1721808	1454465		
8192	483301	617095	792195	853238	901716	999881	1028771	950560	977217	964787	934938	980703	
16384	451959	578063	779372	764570	849712	856906	877175	841410	446416	833672	847375	857398	848516
32768	445138	560634	769079	803450	817277	823935	819651	820120	825722	824999	819279	822821	810668
65536	486312	652780	774380	789258	798799	803976	802861	806467	800008	803372	801959	802725	797411
131072	476958	643974	758567	644890	762831	772610	770272	544649	776289	790748	743047	740269	785215
262144	462449	646059	524252	491535	784510	785636	730755	787167	786522	687147	754028	785252	785848
524288	471563	636068	673811	630183	778054	739432	758990	590300	782380	530680	733838	728177	775108

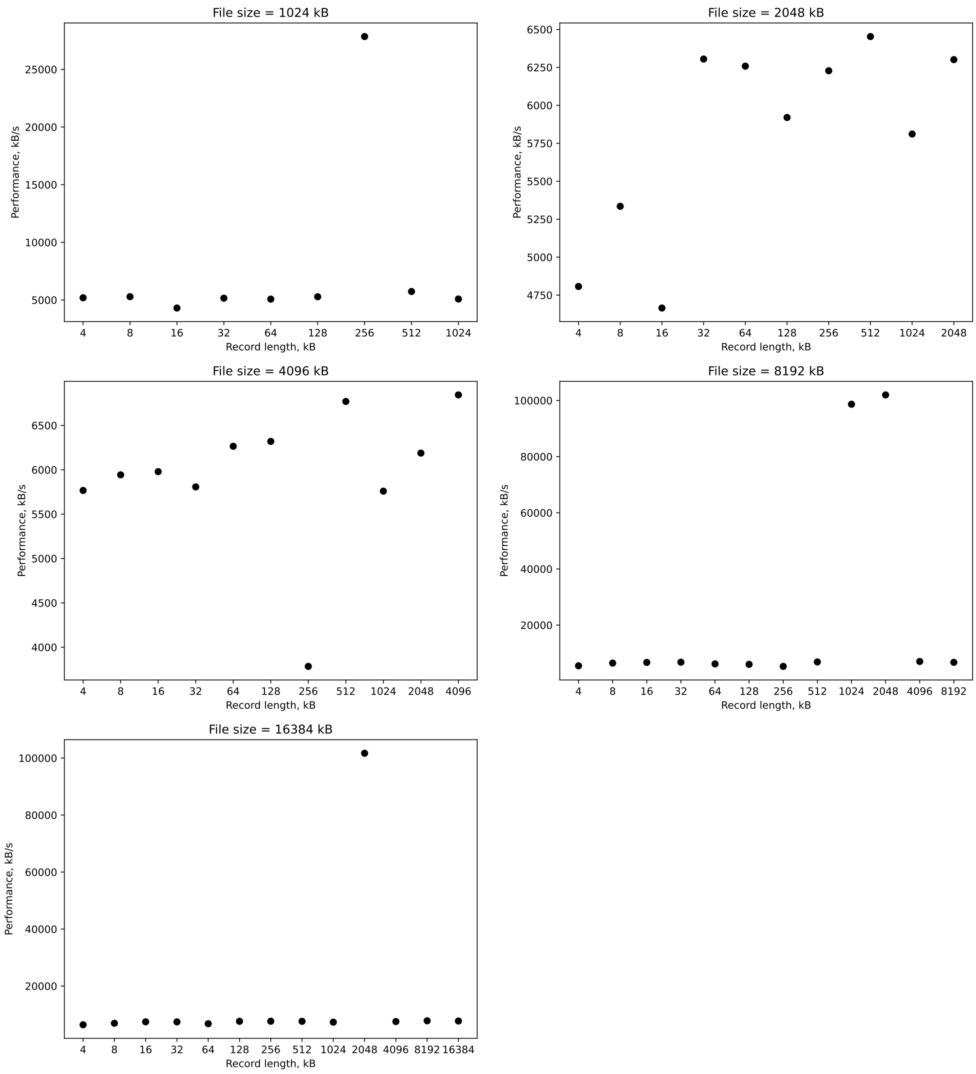

Figure C.14: IOZone output for Fejk FFS Forward Write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1899750	3074849	6059442	7485055	3941039	2572135	3377062	6441107	1299219				
2048	2076967	3287628	1943069	2801852	3287628	9752360	4204968	9944290	8261095	10037248			
4096	1978835	3335375	5236478	2703870	6941906	8944095	9524234	7558881	5996870	4875373	6103394		
8192	1819630	3044239	3383593	5888754	6670750	5776876	8642207	7779318	6924845	5905961	5572627	5237920	
16384	1554007	2421507	4341144	4405431	6212881	5181367	6039252	3657858	7139088	5282129	5080244	4895314	3495607
32768	1558975	2220991	4084269	5435204	6073896	5979556	6635778	5711676	7174793	7100291	6436274	5468725	5888102
65536	1901416	3064454	4018870	5162336	6343120	6161136	6712557	7024950	6344145	6591827	6297922	5434646	5464464
131072	1887775	3091541	4023453	5183353	3293670	5722700	5496414	5696785	3931719	6076610	1999695	847392	3620272
262144	1883367	2885714	4278225	5223883	5600786	5905358	6090977	6498034	7061150	4274183	6760118	5397122	4505903
524288	1860852	2993347	3873539	5135903	6126286	5680384	6422253	7510125	6504565	5663268	5166103	4755541	5349161

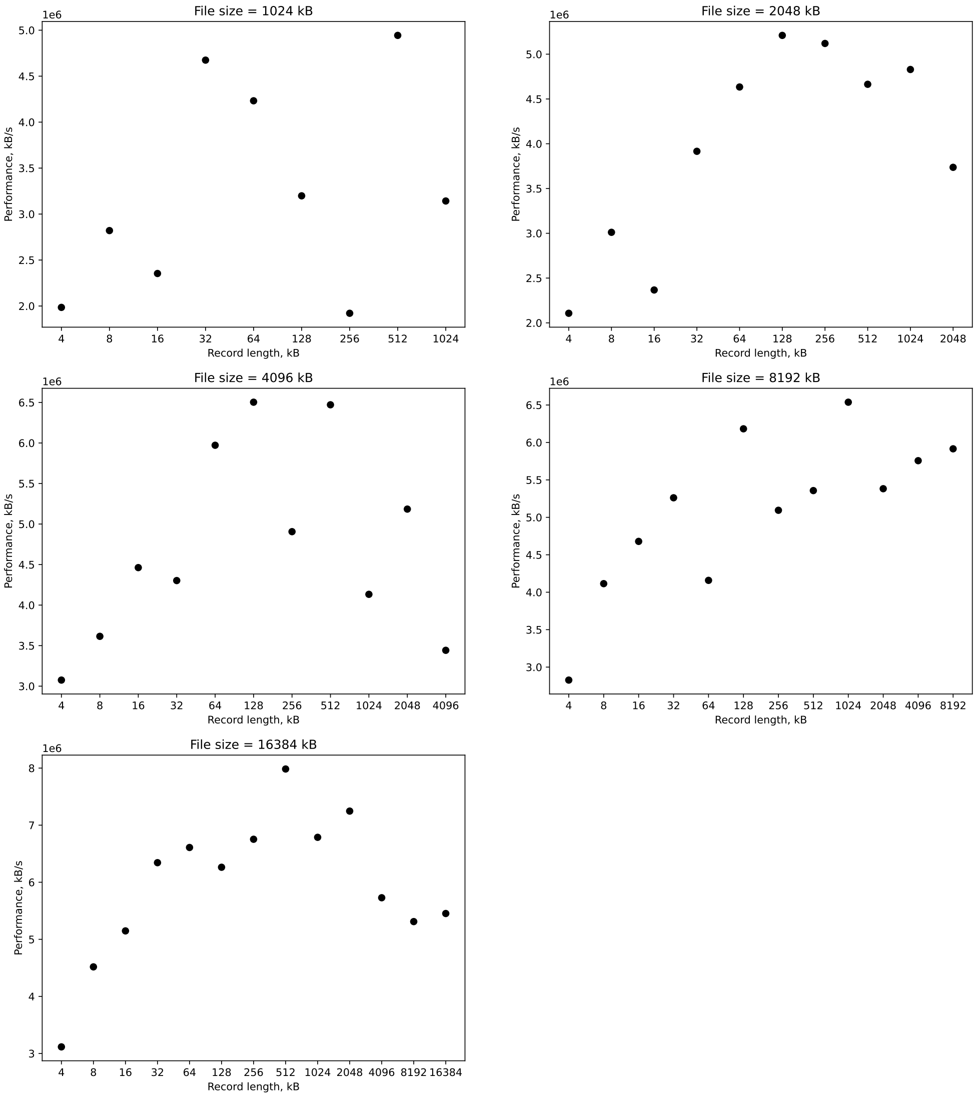

Figure C.15: IOZone output for Fejk FFS Re-Read

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	584523	755251	2020395	2534194	2835325	2437821	2008114	2255990	799241				
2048	649335	954776	474508	1523901	2223755	3519302	1798259	1793378	3318107	2738433			
4096	532497	886025	1069144	1136474	1860943	2239351	1813981	1903416	1447846	1672526	1405213		
8192	554530	810279	965247	1353555	1451443	1545660	1714146	1622824	983538	1018766	935600	955796	
16384	357261	559387	764001	760172	907993	931337	968783	913327	953102	781980	969179	837370	808209
32768	201371	413784	651295	733376	774802	808646	817647	795687	816427	815081	805050	822446	820184
65536	378858	501350	604459	663018	708627	728283	756312	761905	740855	735434	727814	779968	782116
131072	373346	506122	594740	650852	607028	667936	692831	608203	567929	692191	579639	581513	699404
262144	370122	450730	466516	573833	670580	691430	567649	705739	713318	566151	695148	772297	770628
524288	383438	393948	567035	494778	658265	617706	686779	670817	711736	438021	617581	741894	741605

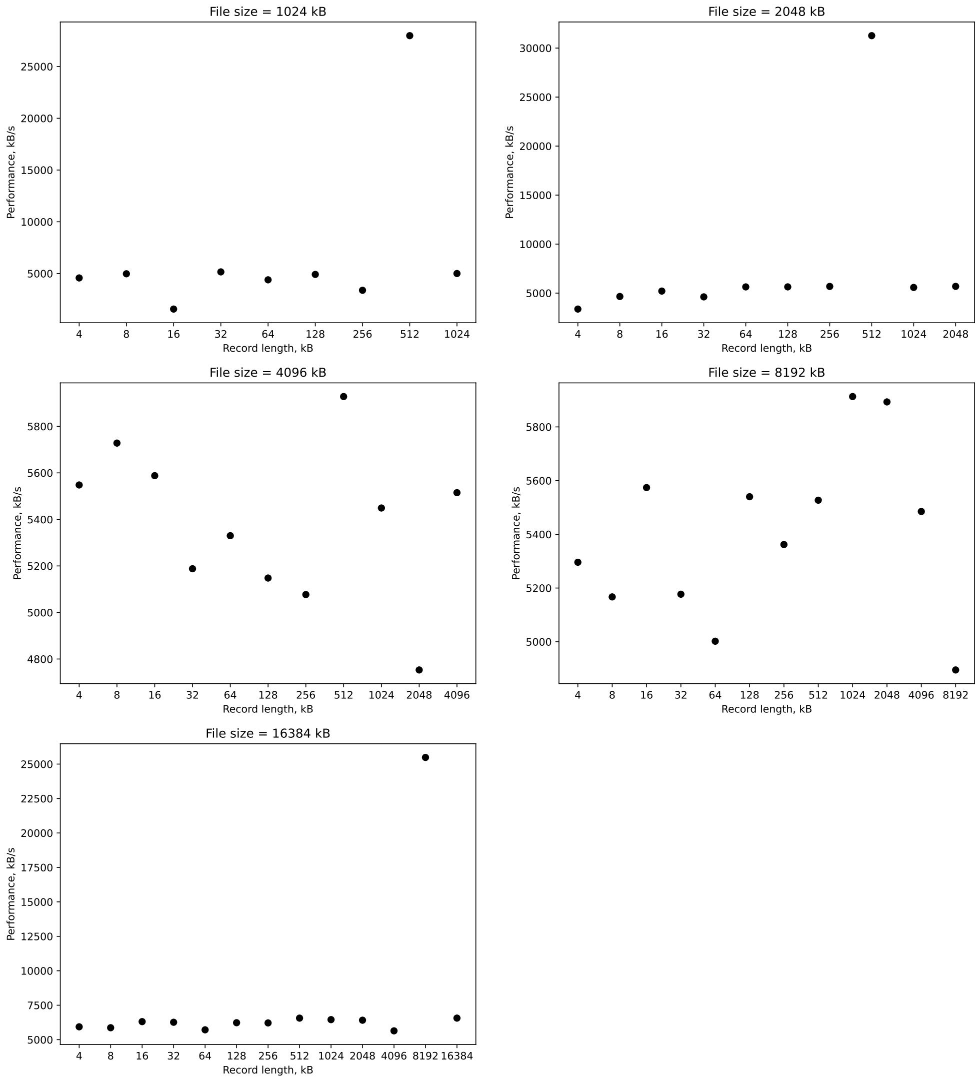

Figure C.16: IOZone output for Fejk FFS Re-Write

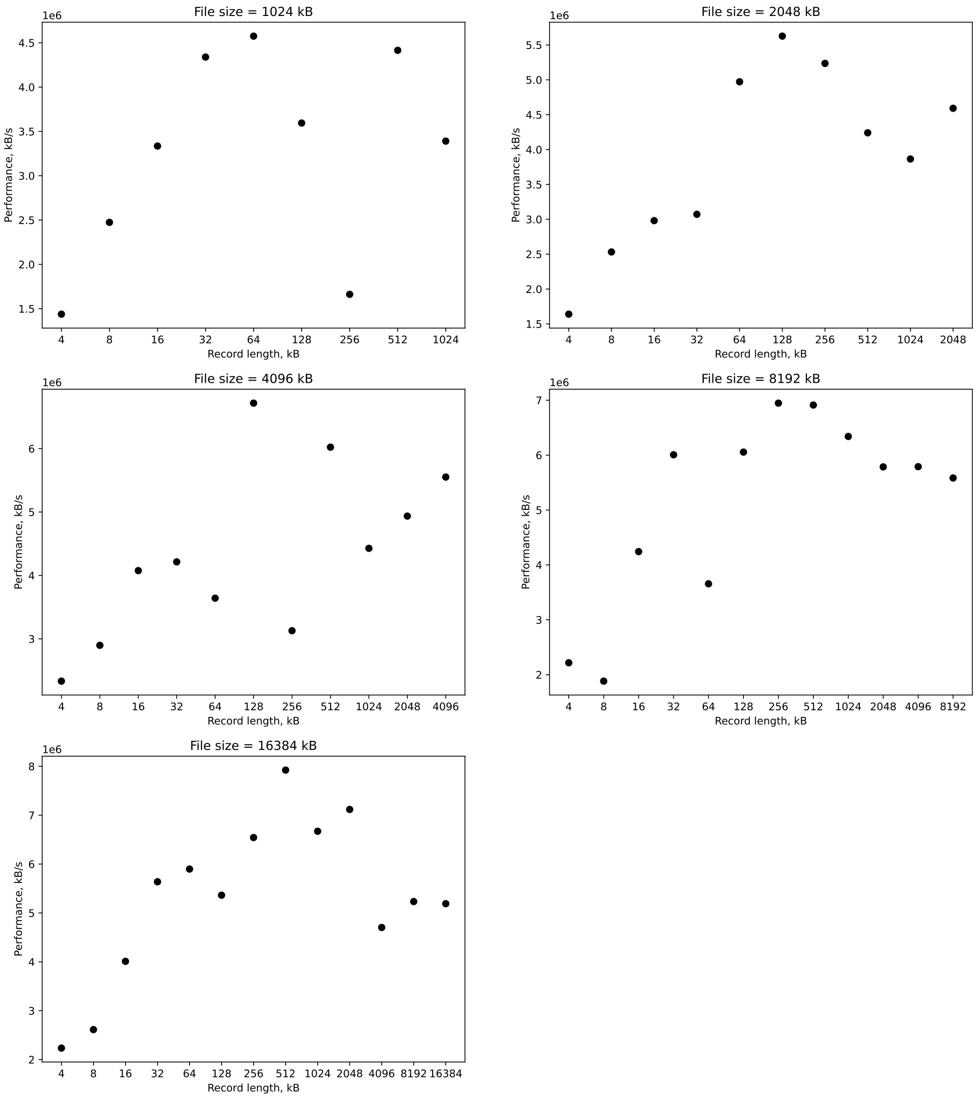

Figure C.17: IOZone output for Fejk FFS Random read

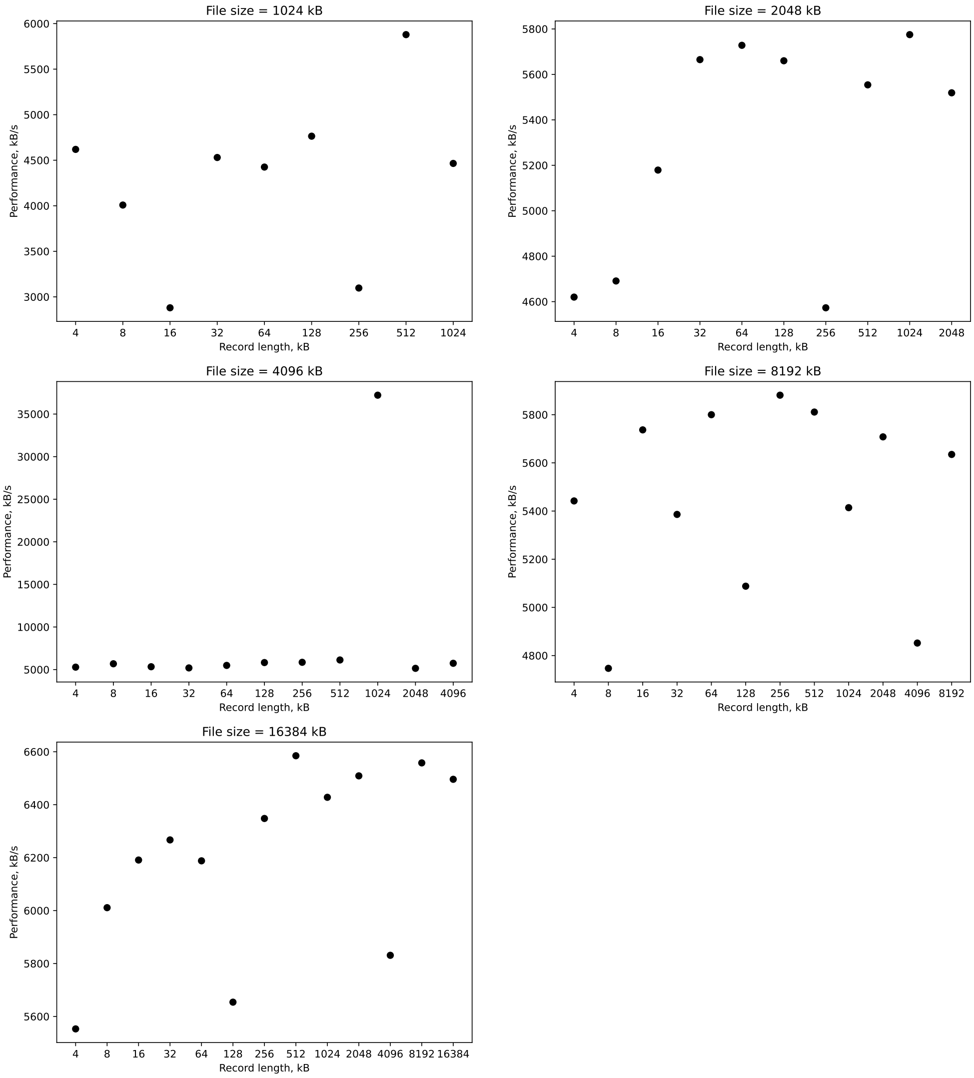

Figure C.18: IOZone output for Fejk FFS Random write

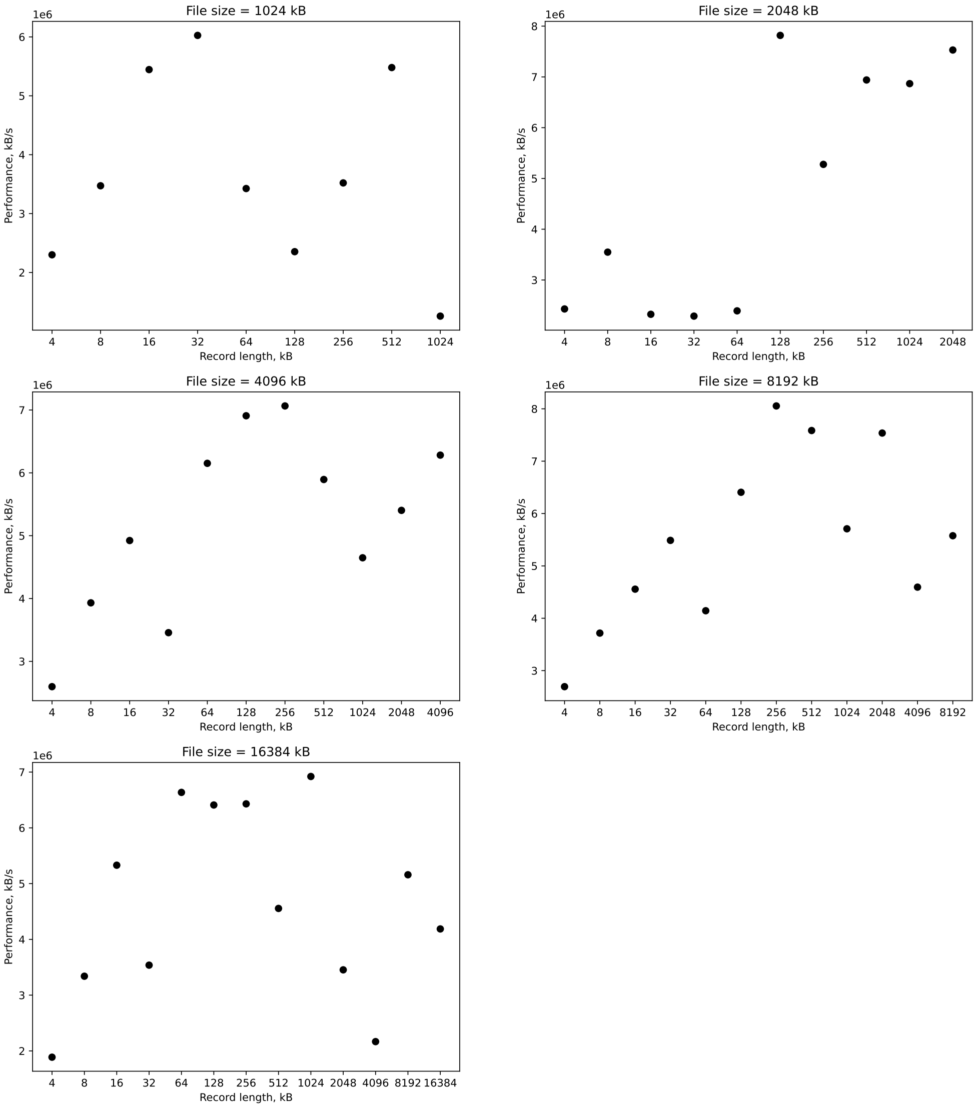

Figure C.19: IOZone output for APFS Forward Read

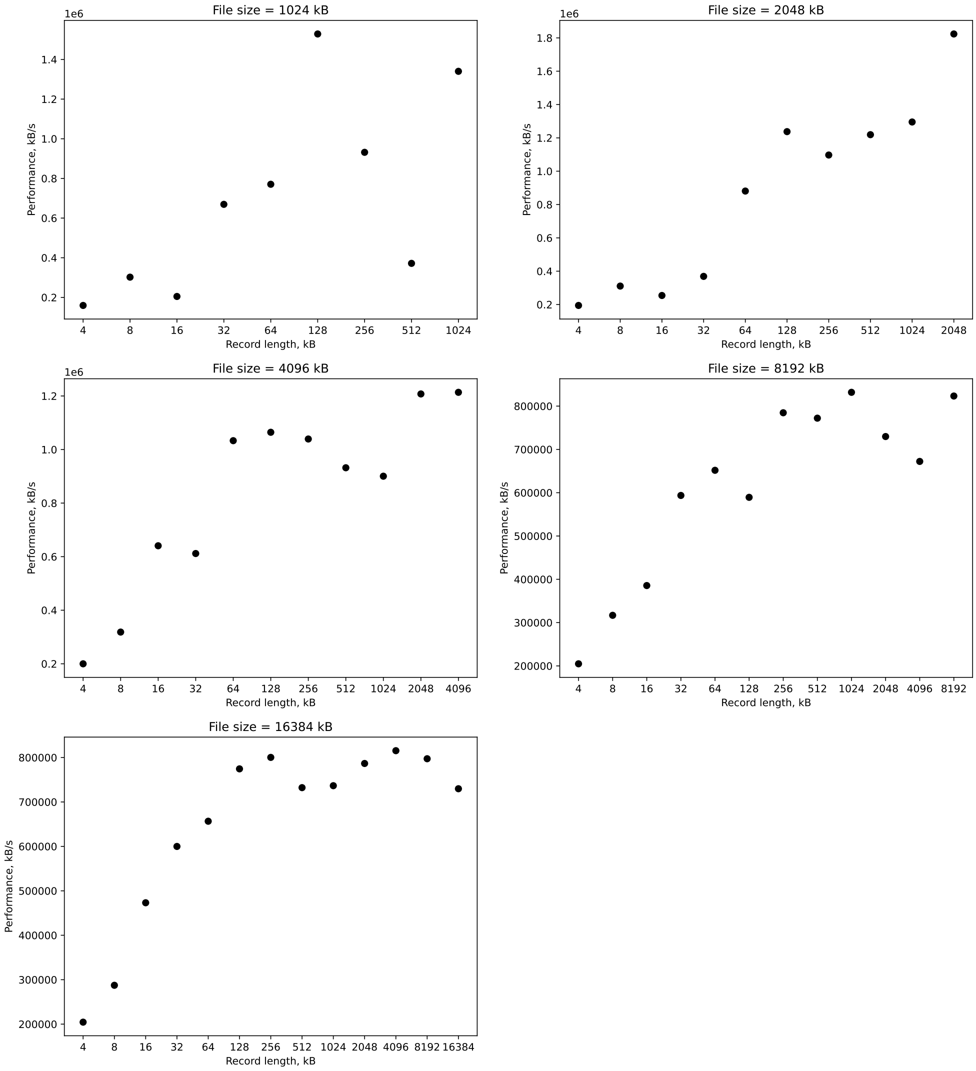

Figure C.20: IOZone output for APFS Forward Write

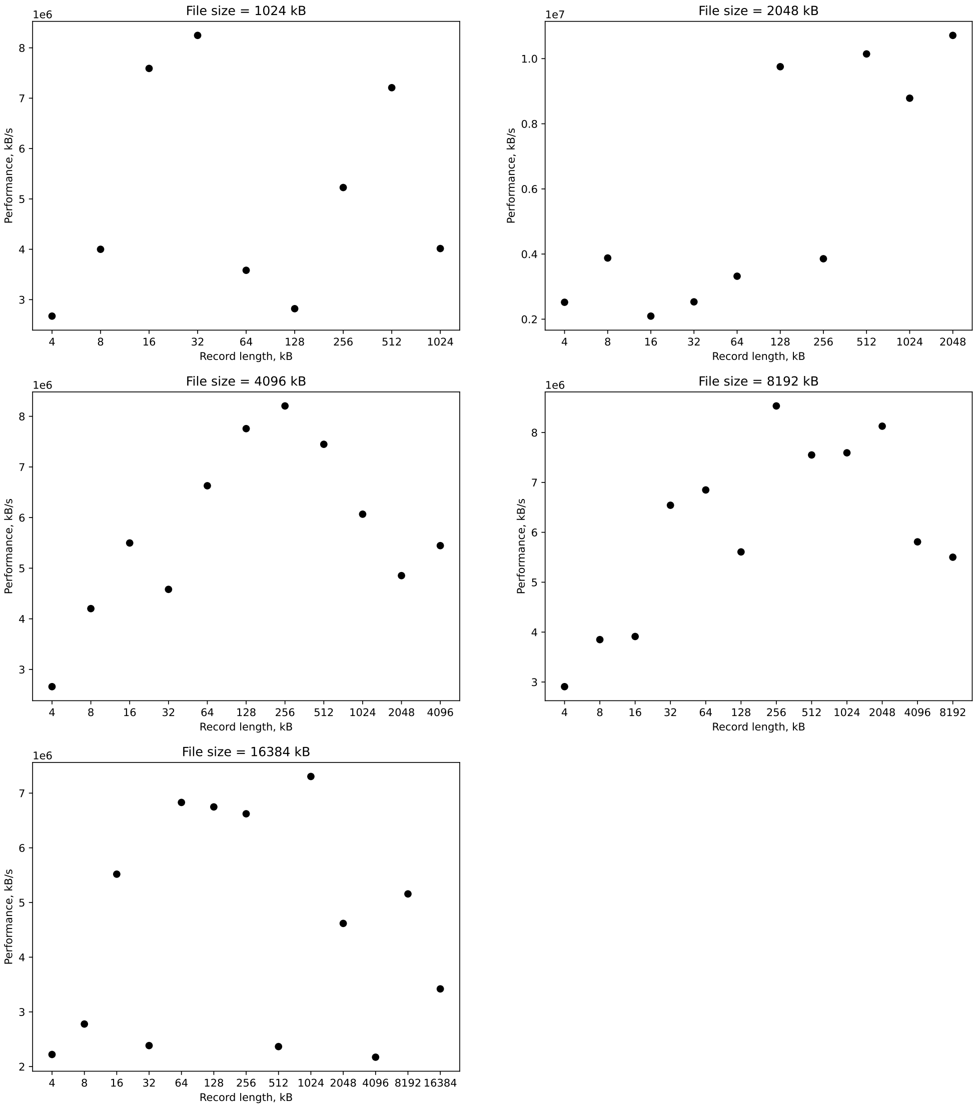

Figure C.21: IOZone output for APFS Re-Read

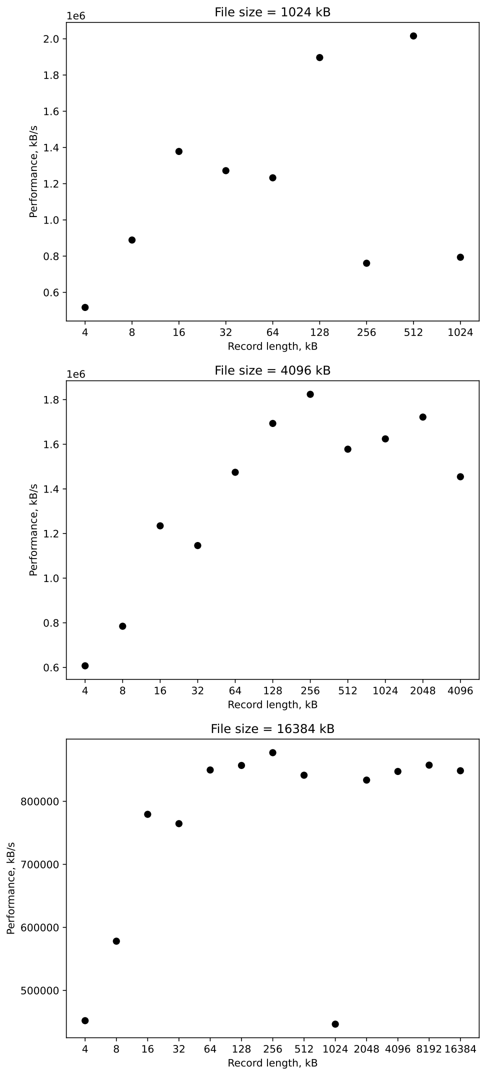

Figure C.22: IOZone output for APFS Re-Write

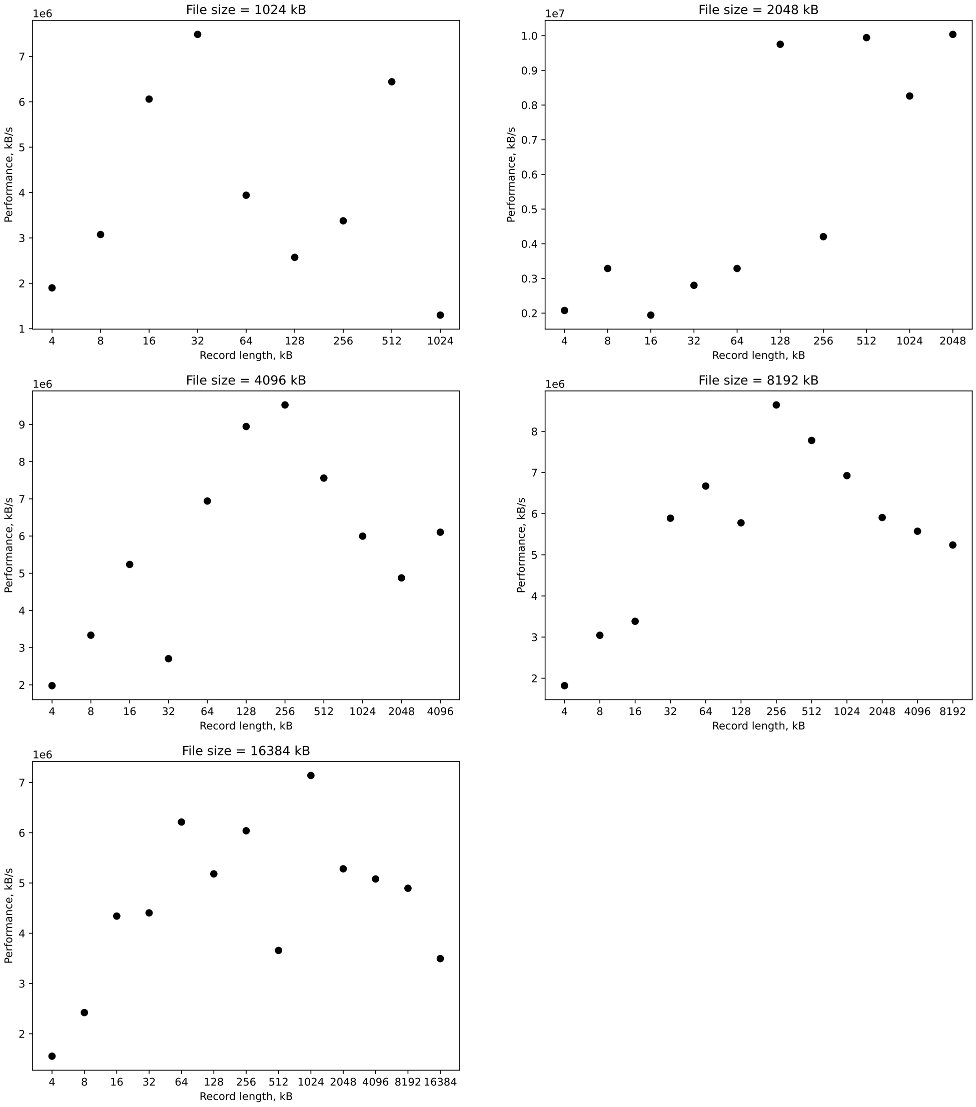

Figure C.23: IOZone output for APFS Random read

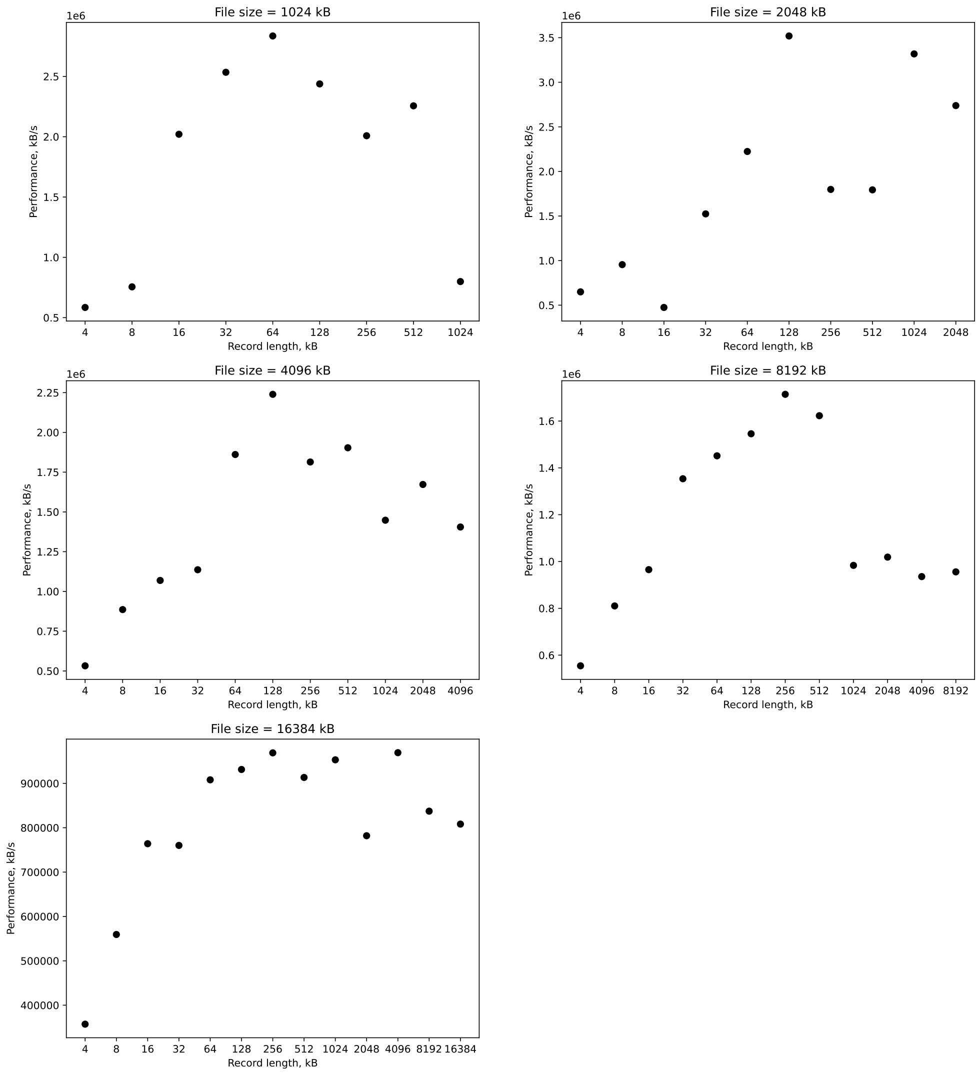

Figure C.24: IOZone output for APFS Random write

TRITA-EECS-EX- 2022:00

For DIVA

{
"Author1": { "Last name": "Olsson",
"First name": "Glenn",
"Local User Id": "u18orpa8",
"E-mail": "glennol@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Degree1": {"Educational program": "Master's Programme, Computer Science, 120 credits"
,"programcode": "TCSCM"
,"Degree": "Degree of Master (120 credits)"
,"subjectArea": "Computer Science and Engineering"
},
"Title": {
"Main title": "FFS: A cryptographic cloud-based deniable filesystem through exploitation of online web services",
"Subtitle": "Store your sensitive data in plain sight",
"Language": "eng" },
"Alternative title": {
"Main title": "FFS: Ett kryptografiskt molnbaserat förnekningsbart filsystem genom utnyttjande av onlinebaserade webtjänster",
"Subtitle": "Lagra din känsliga data öppet",
"Language": "swe"
},
"Supervisor1": { "Last name": "Ghasemirahni",
"First name": "Hamid",
"Local User Id": "u1fz5jtv",
"E-mail": "hamidgr@kth.se",
"organisation": {"L1": "",
"L2": "Computer Science" }
},
"Supervisor2": { "Last name": "Peterson",
"First name": "Zachory",
"E-mail": "znjpeterson@gmail.com",
"Other organisation": "Cal Poly"
},
"Examiner1": { "Last name": "Maguire Jr",
"First name": "Gerald Quentin",
"Local User Id": "u1d13i2c",
"E-mail": "maguire@kth.se",
"organisation": {"L1": "",
"L2": "Computer Science" }
},
"National Subject Categories": "10201",
"Other information": {"Year": "2022", "Number of pages": "xviii,100"},
"Series": { "Title of series": "TRITA-EECS-EX" , "No. in series": "2022:00" },
"Opponents": { "Name": "A. B. Normal & A. X. E. Normalè"},
"Presentation": { "Date": "2022-03-15 13:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/ddddddddddd"
,"Address": "Isafjordsgatan 22 (Kistagången 16)"
,"City": "Stockholm" },
"Number of lang instances": "2",
"Abstract[eng]": €€€€
Many OWSs today, such as Flickr and Twitter, provide users with the possibility to post images which are stored on the platform for free. This thesis explores the idea of creating a cryptographically secure filesystem which stores its data on an online web service using encoded and encrypted images. More data can usually be stored in image posts than in a text posts on OWSs. The filesystem, named FFS, provides users with free, deniable, and cryptographic storage by exploiting the storage provided by these online web services. The thesis compares the performance of FFS against two other filesystems. It can be concluded that FFS has limitations in mainly speed, making it unviable as a general-purpose usage, such as a substitute to the local filesystem on a computer. However, its portability and security makes it relevant for certain scenarios. €€€€,
"Keywords[eng]": €€€€
Filesystem, Fejk FileSystem, Cloud-based filesystem, Steganograhpic filesystem €€€€,
"Abstract[swe]": €€€€
Sammanfattning på svenska €€€€,
"Keywords[swe]": €€€€
Filsystem, Fejk FileSystem, Molnbaserat filsystem, Steganografiskt filsystem €€€€,
}