FFS: A CRYPTOGRAPHIC CLOUD-BASED DENIABLE FILESYSTEM THROUGH EXPLOITATION OF ONLINE WEB SERVICES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Glenn Olsson

June 2022

COMMITTEE MEMBERSHIP

TITLE:	FFS: A cryptographic cloud-based deniable filesystem through exploitation of online web services
AUTHOR:	Glenn Olsson
DATE SUBMITTED:	June 2022
COMMITTEE CHAIR:	Zachary Peterson, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science

ABSTRACT

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

Glenn Olsson

Many online web services 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 filesystem which stores its data on an online web service using encoded and encrypted images. The filesystem, named , provides users with free, deniable, and cryptographic storage by exploiting the storage provided by these online web services. The thesis compares the performance of against two filesystems available. It can be concluded that has limitations in mainly speed, making it unviable for multi-purpose usage. However, it's portability and security makes it relevant for certain scenarios.

ACKNOWLEDGMENTS

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

To the people who said it could not be done.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

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

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

There exist multiple deniable filesystems that are designed to combat this problem on physical devices, such as memory cards. However, even just carrying a memory card might subject you to suspicion of hiding data, no matter how the filesystem is designed. Another solution to hiding the data is therefore to hide it somewhere else, for instance online through the use of a cloud-based filesystem service, such as Google Drive. Someone searching your body and devices, at for instance an airport or border control, might not realize that you are using a cloud-based filesystem service to hide your data. Although, more thorough investigations of a person might reveal user accounts used on the service, leading to legal processes where the service is forced

to disclose your data. Even if you encrypt the data you upload to such a service, you can still be forced to reveal the decryption keys. What we want to achieve is a combination of a deniable filesystem and a cloud-based filesystem, where the data is stored using digital cryptographic and steganographic methods but without any company or person other than the user controlling the actual data. To accomplish this, we can store the data on online social media platforms.

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

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

1.1 Problem

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

1.2 Purpose and motivation

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

The idea behind is to have a decentralized cloud-based filesystem where only the user has access to the unencrypted data. By encrypting and decrypting the files locally before uploading and after downloading them to these services (end-to-end encryption), it is possible to ensure that the user is the only one who has access to the encryption and decryption keys and therefore the unencrypted data. Even if the web service would look at the data uploaded by the user, it is unreadable without the decryption key. An interesting aspect of this is that online web services, such as social media, provide users with essentially an unbounded amount of storage for free. Anyone can create any number of accounts on Twitter and Facebook without cost, and with enough accounts, one could potentially store all their data using such a filesystem. We aim to exploit the storage web services give their users for free. As the file data is stored in the open but only accessible by the user, and as can be unmounted to hide its existence, it is steganographic.

There are several steganographic filesystems available but these lack certain aspects that aims to solve. Some filesystems are based on the local disk of the device in use, such as the physical storage device on a computer or phone, or an external storage device connected to a computer or phone. While these filesystems have advantages compared to cloud-based solutions, such as latency, they lack accessibility as you need to have the device to access the content on it. It also means that when you want to share or transport the data, you must physically move the device which can mean problems as it could for instance be taken from you or be destroyed. Cloud-based solutions counter this by being available from any location that has internet access to the services used. However, existing cloud-based solutions introduce other disadvantages. One example is CovertFS [6] where data is stored in images posted on web services. The images are actual images representing something, meaning that there is a limit on how much steganographic data can be stored. CovertFS limit this to 4 kB which means that such a filesystem with a lot of data will require many images which could lead to suspicion from the owners of the web services. stores as much data as possible in the images, meaning that less images are needed to store a file bigger than 4 kB. It also means that the images produced by do not look like a normal image, but instead has seemingly randomly colored pixels. More examples of similar filesystems will be presented in Chapter 3.

1.3 Goals

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

1. to create a mountable filesystem where files and directories can be stored, read, and deleted,
2. for the filesystem to store all the data on online web services rather than on the local disk,
3. for the system to be secure in the sense that even with access to the uploaded files and the software, the plain-text data is unreadable without the correct decryption key,
4. to provide the user of the filesystem with plausible deniability of its data in the sense that it is not possible to associate the user with if the filesystem is not mounted,
5. to analyze the write and read speed, storage capacity, and reliability of the filesystem and compare it to commercial cloud-based filesystems and local filesystems, and,
6. to analyze and discuss environmental and ethical aspects of the filesystem.

1.4 Research Methodology

The filesystem created through this thesis will be developed on a Macbook laptop running macOS Monterey, version 12.3.1. It will be written in C++20 and use the macOS library [7] which enables the writing of a filesystem in userspace rather than in kernel space. is available on other platforms too, such as Linux, but the filesystem will be developed on a Macbook laptop thus macFUSE is chosen. C++ is chosen because the API is available in C, and C++ version 20 is well established and used. Further details about the development environment will be found in Section 4.1.

The resulting filesystem will be evaluated against other filesystems, both commercial distributed systems, such as Google Drive, and an instance of [8] on the Macbook laptop referenced above. Quantitative data will be gathered from the different filesystems through the use of experiments with the filesystem benchmarking software IOzone [9]. IOzone was chosen because it is, compared to tools such as Fio and Bonnie++, simpler to use while still powerful [10]. We will look at attributes such as the differences in read and write speeds between different filesystems, as well as the speed of random read and random write. However, according to Tarasov et al., benchmarking filesystems using benchmarking tools is difficult to perform in a standardized way [11] which will be taken into consideration during the evaluation and when concluding the thesis. Further discussion about this will be found in Section 3.4.

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 going to be implemented in . The focus will be to implement a subset of the POSIX standard functions, containing only crucial functions for a simple filesystem, specifically, the functions *open*, *read*, *write*, *mkdir*, *rmdir*, *readdir*, and *rename*. However, file access control is not a necessity and will therefore not be implemented, thus functions such as *chown* and *chmod* are not going to be implemented. The reason is that the goal is to present and evaluate the possibility of creating a secure steganographic filesystem with a storage medium based on online web services and thus will only aim to implement a minimal filesystem.

There is also an argument that could be made that should support multiple users so that anyone can mount but only browse their files. However, as this project is only a proof-of-concept of the filesystem, this will not be implemented. Instead, will be built for single-user support where only a password will unlock everything is storing. This means that anyone who mounts with the password will access everything that other users might have stored.

1.6 Structure of the thesis

Chapter 2 presents theoretical background information of filesystems and the basis of while Chapter 3 mentions and analyzes related work. Chapter 4 describes the implementation and the design choices made for the system, along with the analysis methodology. Chapter 5 presents the results of the analysis and Chapter 6 discusses the findings and other aspects of the work. Lastly, Chapter 7 will finalize the conclusion of the thesis and discuss potential future work.

Chapter 2

BACKGROUND

This chapter presents concepts and information that is relevant for understanding, implementing, and evaluating . We first present the idea of inode-based filesystems and how data is stored in a filesystem. Following is the introduction of which will be used to implement . Later sections present background information about Twitter and the potential threat adversaries of .

2.1 Filesystems and data storage

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

2.1.1 Unix filesystems

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

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

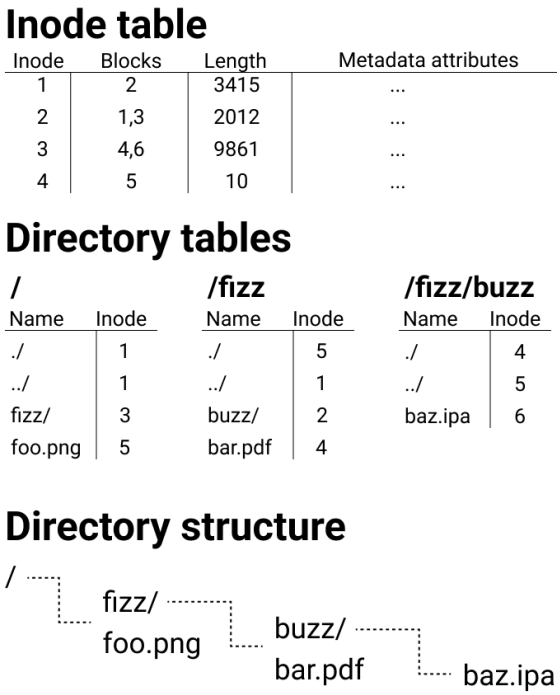

Figure 2.1: Basic structure of inode-based filesystem

Listing 2.1: Pseudocode of a minimalistic inode filesystem structure

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

struct directory_entry {
        char*   filename
        int     inode
}

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

Different filesystems provide different features and limitations. The Extended Filesystem (ext) exists in four different versions: ext, ext2, ext3, and ext4. This filesystem is often used on Unix systems. Each iteration brings new features and changes the limitations. For instance, comparing the two latest iterations, ext3 and ext4, ext4 can theoretically store files up to 16 TiB while ext3 can store files up to 2 TiB [12]. 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 API [13].

is a modern filesystem that is used on iPhones and Mac and can store files with a size up to 9 EB [14]. It supports timestamps in units of nanoseconds and is built to be used on [15]. It also supports modern features that its predecessor Mac OS Extended (HFS+) does not support, such as Snapshots and Space Sharing. natively supports encryption of the filesystem volume [16].

2.1.2 Distributed filesystems

Filesystems are used to store data, for instance locally on a hard drive of a computer, or in the cloud. Google Drive is an example of a filesystem that enables users to save their data online with up to 15 GB for free [17] 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 [18]. 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 [19], 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 [20]. 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 [21]. 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 [22], but this is outside the scope of this thesis). If one would represent an arbitrary file of X bytes, each byte (0x00 - 0xFF) can be represented as a character such as the keyset and we can therefore decode this file as X different characters. Using the same set of characters for encoding and decoding we can get a symmetric relation for representing a file as a string of characters. SCII is only one example of such a set of characters, any set of strings with 256 unique symbols can be used to create such a symmetric relation, for instance, 256 different emojis or a list of 256 different words. However, if we are using a set of words we would also have to introduce a unique separator so that the words can be distinguished. If we would use a single space character as the separator, we could make the encoded text look like a text document; however, random words one after another lead to a high probability of creating an unstructured text document. Further, if punctuation

is introduced, for instance as part of some words, the text document could look like it contains random and unstructured sentences.

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

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

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

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

2.2 FUSE

is a library that provides an interface to create filesystems in userspace rather than in kernel space which is otherwise often considered the standard when writing commercial filesystems [23]. The reason to implement a filesystem in kernel space is that it leads to faster system calls than when writing a filesystem in userspace. However, while filesystems written with are generally slower than kernel-based filesystems, using simplifies the process of creating filesystems. macFUSE is a port of that operates on Apple's macOS operating system and it extends the API [7]. macFUSE provides an API for C and Objective C.

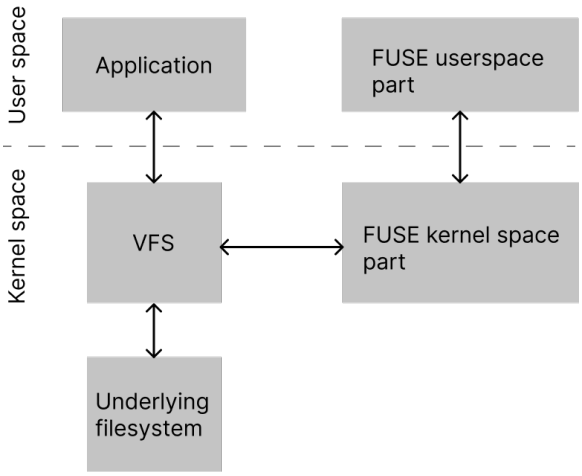

Figure 2.2: Simple visualization of how operations are executed

Figure 2.2 shows an overview how works. consists of a kernel space part and a userspace part that perform different tasks [24]. The kernel part of operates with the which is a layer in both the Linux kernel and the macOS kernel that exposes a filesystem interface for userspace applications [25, 26]. The interface is independent of the underlying filesystem and is an abstraction of the underlying filesystem operations which can be used on any filesystem the supports. The userspace part of communicates with the kernel space part through a block device. Operations on a mounted filesystem are sent to the from the user application, which is then sent to the kernel part of . If needed, the operations are transmitted to the userspace part of where the operation is handled and a response is sent back to the and the user

application through the kernel module. However, some actions can be handled by the kernel module directly, such as if the file is cached in the kernel part of [24]. The response is then sent back to the user application from the kernel module through the .

2.3 Online web services

This section presents two online web services (s), Twitter and Flickr, where one can create free-tier accounts. On both of these s, free-tier accounts can make numerous posts for free. The s provide free-to-use Application Programming Interfaces (APIs) 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 [27]. Text posts are limited to 280 characters while images can be up to 5 MB and videos up to 512 MB [28]. 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 [29]. 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 [30]. 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 [31]. 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 [32]. Flickr Pro costs between 7.49€ to 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 [33]. This has been verified through testing. The title of a post has also been discovered to have a limit of 255 characters through testing.

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[34]. Flickr also stores other formats of the file, such as thumbnails. User accounts can restrict who, other than themselves, can download the original image. The original video can only be downloaded by the user [34]. 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 [35].

The Flickr API defines a query limit of 3 600 requests per hour, per application, across all API calls [36]. However, according to Sam Judson in 2013, this is not a hard limit [37]. 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 [36]. If abuse is detected, Flickr reserves the right to revoke API keys.

2.4 Cryptography

is an encryption standard established by the , more specificity specifying the Rijndael block cipher [38]. is a symmetrical cipher, meaning that the same key is used for encryption and decryption. is used to make the data confidential so that no one except the person with the key can access the unencrypted data. produces 128-bit encrypted cipher blocks and supports key sizes of 128 bits, 192 bits, or 256 bits. The security of has been heavily researched since its introduction in the early 2000s, and literature has found it is well resistant to quantum attacks as well [39].

While is a good standard for the confidentiality of the data, confidentiality is often not enough to secure the data [40]. 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 [41]. is a block cipher mode of operation which provides authenticated encryption [42]. can be used with to provide secure, authenticated encryption of data. To encrypt using , the encryption function requires a key, a randomized and the data to encrypt. The output is the encrypted cipher text and an authentication tag. The decryption function of requires the same key and as was used as input in the encryption function, as well as the authentication tag and the cipher text received as output by the encrypting function. Further, both the encryption function and the decryption support to be provided. is data that should be authenticated, but not encrypted. If is provided to the encryption function, it must also be provided to the decryption function.

The key used when encrypting using is often derived from a password that the user provides. s are functions that can be used to derive a key used for, for instance, . The input to a is a secret, such as a password [43]. An example of a schema is the presented by Krawczyk [44][45] which utilizes a hashing algorithm that provide a pseudo-random key. supports multiple hashing algorithms. The security of is partially dependent on the security of the hashing algorithm used. A well-defined suit of hashing algorithms is the , which covers, among other hash functions, -256 [46]. -256 is a cryptographic hash function that outputs a 256-bit pseudo-random

cipher from its input, which can, for instance, be a password. Further, uses a salt to improve the security of the provided secret. The salt is random data used to further diffuse the produced key, making two keys with the same secret but different salts, different [47]. The salt does not have to be secret and is sometimes stored with the produced cipher so that the decryption function easily can re-use the salt when deriving the decryption key. If the key used for encryption and the key used for decryption are derived using different salts, the keys will differ and the cipher cannot be decrypted.

Alternative encryption solutions are, among others, and . is an asymmetrical cipher, meaning that it uses a public key and a private key for encryption and decryption. According to Mahajan and Sachdeva, asymmetric encryption techniques are more computationally intensive than symmetrical encryption techniques and are almost 1 000 times slower than symmetrical techniques [48]. Mahajan and Sachdeva found that is the fastest algorithm for encryption and decryption between , , and while maintaining very good security. This further proves to be a good choice as the cryptography technique for .

2.5 Threats

To consider a filesystem secure it is important to imagine different potential adversaries who might attack the system. Considering that has no real control of the data stored on the different services, all the data must be considered to be stored in an insecure system. Even if we could hide the posts made on the online web service, for instance, Twitter, by making the profile private, we must still consider that Twitter themselves could be an adversary or that they could potentially give out information, such as tweets or direct messages, to entities such as the police. Twitter's privacy policy mentions that they may share, disclose, and preserve personal information and content posted on the service, even after account deletion for up to 18 months [49]. Therefore, to achieve security the data stored must always be encrypted. We assume that an adversary has access to all knowledge about , including how the data is converted, encrypted, and posted. We also assume they know which websites and

accounts could post 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 compromised which can expose the user's identity. The problem of these other sources of information external to is not addressed in but remains for future work.

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

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

Chapter 3

RELATED WORK

The research area of creating filesystems to improve security, reliability, and deniability is not new and has been well worked on previously. This chapter presents previous work that is related to this thesis. This includes other filesystems that share similarities with the idea of , for instance within the idea of unconventional storage media and the area of steganography.

3.1 Steganography and deniable filesystems

Steganography is the art of hiding information in plain sight and has been around for ages. Today, a major part of steganography is hiding malicious code in for instance images, called stegomalware or stegoware. Stegomalware is an increasing problem and in a sample set of examined real-life stegomalware, over 40% of the cases used images to store the malicious code [50]. While will not include malicious code in its images, this stegomalware problem has fostered the development of detection techniques of steganography in for instance social media, and it is well-researched.

Twitter has been exposed to allowing steganographic images that contain any type of file easily [51]. 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 [52]. He presents multiple examples of this technique on his Twitter profile*. The fact that the images are available for the public's eye might be evidence that Twitter's steganography detection software is not perfect. However, it is also possible that Twitter has chosen to not remove these posts.

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

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

lic photo-sharing web services [53]. Due to the web services processing of uploaded multimedia, they first researched how the integrity of steganographic data could be maintained after being uploaded to these services. Following this, they presented an approach that ensured the integrity of the hidden messages in the uploaded images, while also maintaining a low likelihood of discovery from the steganographic analysis. Beato, De Cristofaro, and Rasmussen also explores the idea of undetectable communications over s in another paper [54]. While implementation is not carried out, they present an idea where messages are encoded together with a cover object and a cryptographic key to produce a steganographic message which is then posted to the . A web-based user interface client with a PHP server backend is presented as the method the users would use to create and share their secret messages.

A steganographic, or deniable, filesystem is a system that does not expose files stored on this system without credentials - neither how many files are stored, their sizes, their content, or even if there exist any files in the filesystem [55]. 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 [56]. This is used to demonstrate the security and integrity of images sent over insecure channels. The paper uses Twitter and Facebook to exhibit this. Despite its improvement and compatibility of a common image format, such as bitstream compliance, due to its well-proven security will use as its encryption method.

3.3 Related filesystems

Multiple steganographic filesystems have been presented previously but many of these are focused on filesystems for physical storage disks to that the user has access. For instance, Timothy Peters created DEFY, a deniable filesystem using a log-based structure in 2014 [57]. DEFY was built to be used exclusively on found in mobile devices to provide a steganographic filesystem that could be used on Android phones. Further examples of local disk-based filesystems can be found in [55, 58, 59, 1], among other papers. However, this paper aims to create a filesystem that is not based on a physical disk but rather a cloud-based steganographic filesystem that uses online web services as its storage medium.

In 2007, Baliga, Kilian, and Iftode presented an idea of a covert filesystem that hides the file data in images and uploads them to web services, named CovertFS [6]. The paper lacks implementation of the filesystem but they present an implementation plan which includes using . They limit the filesystem such that each image posted will only store a maximum of 4 kB of steganographic file data and the images posted on the web services will be actual images. This is different from the idea of where the images will be purely the encrypted file data and will therefore not be an image that represents anything but will instead look like random color noise. An implementation of CovertFS has been attempted by Sosa, Sutton, and Huang which also used Tor to further anonymize the users [60].

In 2016, Szczypiorski introduced the idea of StegHash - a way to hide steganographic data on by connecting multimedia files, such as images and videos, with hashtags [61]. 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 [62]. Multiple posts could be required to store a single file and each post referenced the next post like a linked list, which means that you only need the

root post to read all the data. This is unlike the idea of where a table will be kept to keep track of which posts store a certain file, and in what order they should be concatenated, similar to the idea of an inode table. SocialStegDisc lacks actual implementation of the filesystem but similar to CovertFS presents the idea of a social media-based filesystem.

TweetFS is a filesystem created by Robert Winslow that stores the data on Twitter [63], 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 [64, 65]. The filesystem was written in Python using and was presented well before the introduction of Google Drive in 2012. It does not support encryption as the plain file data is stored in emails. Today, Gmail and Google Drive share their storage quota and GmailFS has since become redundant as Google Drive is an easier filesystem to use. GMail Drive is another example of a Gmail-based filesystem and it was influenced by GmailFS [66]. GMail Drive has been declared dead by its author since 2015.

is a filesystem that stores its data on Google Drive, built using [67, 68]. On the other hand, Google Drive provides a desktop application [69] that presents a mounted volume in the local filesystem, representing the user's Google Drive filesystem. The

mountable volume provided by the desktop application does not always sync the stored data directly, but might instead store it locally until a later time. To enable direct synchronization of the data to Google Drive, interacts with the Google Drive REST 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 will not complete before the new file data has been completely stored on Google Drive. Therefore, the duration from the start of the filesystem operation until its end includes the time it takes to upload the file. On the other hand, the duration of a filesystem operation on the mountable volume provided by the Google Drive Desktop application does not always include the time it takes to upload the file, this can occur at a later time. One difference between and the idea of is that does not encrypt the data stored in the filesystem. While the data is, as mentioned previously, encrypted by Google Drive, the encryption keys are controlled by Google Drive, not the user of . The data stored on is also stored as its original files in Google Drive, not as images as intends to store the data. The Google Drive filesystem architecture is utilized by , for instance by using its directory hierarchy structure. This allows to avoid creating its own inode table and directory structures, as Google Drive provides the functionality these structures similarly provide , through the Google Drive API. The development of started in 2018 [68], and the repository in GitHub has around 2 300 starts as of writing.

Another Google Drive-based filesystem is google-drive-ocamlfuse [70], developed for Linux using . The project is well received online. The repository has around 6 700 stars on GitHub at the time of writing and there are multiple articles online about the project [71, 72, 73]. The filesystem is well developed and, as of writing, well maintained. The filesystem supports filesystem operations such as symbolic links, Unix ownership, and multiple account support. According to the author of , tends to be faster than google-drive-ocamlfuse for certain operations, including reading cached files [74, 75]. 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 [76]. 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 [77]. This approach enables one to create not only an encrypted filesystem but also to provide redundancy by replicating data to different underlying filesystems. If these filesystems are independent, then this potentially increases availability and reliability. aims to achieve stackability through the use of .

3.4 Filesystem benchmarking

IOzone is a filesystem benchmarking tool that is used to measure performance and analyze a filesystem [9]. It is built for, among other platforms, Apple's macOS where will be built, run, and tested. However, as mentioned previously, filesystem benchmarking is more complicated than one might imagine. Different filesystems might perform differently on small and big file sizes among other things, which means that we can never compare benchmarking outputs as just single numbers. We must instead compare different aspects of the filesystems. In 2011 Tarasov et al. presents a paper where they criticize several papers due to their lack of scientific and honest filesystem benchmarking [11]. The problem with benchmarking a filesystem is all the different components that are involved when interacting with a filesystem. For instance, they mention how benchmarking the of the filesystem, such as bandwidth and latency, is different from benchmarking on-disk operations, such as the performance of file read and write operations. The benchmarking tools can for instance rarely affect or determine how the filesystem handles caching and pre-fetching. This means that benchmarking the read and write performance of different filesystems can be misleading as they might handle this differently, meaning that the result could be different depending on for instance the distance between the files on the disk. Two files could be adjacent on the disk on one filesystem and therefore one could be pre-fetched into the cache when the other one is read. Considerations about such factors must be present when analyzing the results of the benchmarking.

Tarasov et al. also lists several different filesystem benchmarking tools available and used by the papers they reviewed, and how well the tools can analyze certain aspects of a filesystem [11]. IOZone is listed as being compatible with multiple different benchmarking types and as it is simpler to use [10] and still maintained. Due to these factors, IOZone was chosen as the benchmarking tool for .

3.5 Summary

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

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

	ext4	**Google Drive**	**DEFY**	**TweetFS**	**FFS**
Mountable	X	X	X	-	X
Read/Write/Remove file	X	X	X	X	X
Read/Write/Remove directory	X	X	X	X	X
Hard links	X	-	X	-	-
Soft links	X	-	X	-	-
File and directory access control	X	X	-	X	-
Encrypted	X	X*	X	-	X
Steganographic	-	-	X	X	X
Cloud-based	-	X	-	X	X

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

Chapter 4

METHOD

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

4.1 Development environment specification

Development of was done on a 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 , and the filesystem used was an encrypted partition. The computer was running macOS Monterey 12.5.

FFS was developed using C++20 and compiled using Apple clang version 13.0.0, using target x86_64appledarwin21.4.0. uses the ImageMagick Magick++ library [78] for image processing. Version 7.1.029 of Magick++ is used by . macFUSE [7] version 4.2.5 is used for to use the API. API version 26 is used. cURLpp [79] is a cURL [80] C++ wrapper used by to make HTTP requests. Version 0.8.1 of cURLpp is used by . libOauth [81] version 1.0.3 is used by to sign and encode HTTP requests according to the OAuth [79] standard. Flickcurl [82] version 1.26 is a C library used by to communicate with parts of the Flickr API. Crypto++ [83] is a C++ library providing cryptographic schemes. uses Crypto++ to encrypt end decrypt the data stored in , and to derive the keys used in the encryption and decryption algorithm. Crypto++ version 8.6 is used by .

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 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 (). It uses an to store the data but behaved as a mountable filesystem for the users. 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, and when comparing to distributed filesystems such as Google Drive, many of these other filesystems also often 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 , such as Flickr, as image posts. As mentioned in Section 2.3, there can be limitations of the size of these posts for certain s. To support file sizes bigger than these limitations, bigger files will be split into multiple posts, requiring to keep track of a list of posts. Figure 4.1 presents the basic outline of and an example content of the filesystem. is based on the idea of inode filesystems and uses an inode table to store information about the files and directories in the filesystem. However, instead of an inode pointing to specific blocks in a disk, the inode table of will instead keep track of the id numbers of the posts on the where the file or directory is located. The inode table entry for each file or directory will also contain metadata about the entry, such as its size and a boolean indicating if the entry is a directory or not.

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

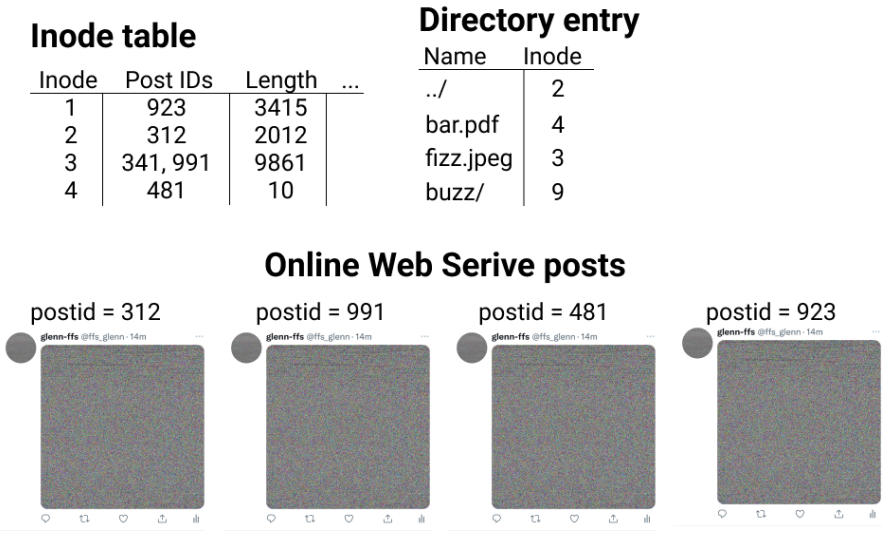

Figure 4.1: Basic structure of inode-based structure

`Directory` class is a data structure that stores mappings between filenames and the files' and directories' inode, for all files and directories stored in that directory. The `InodeEntry` is a data structure that keeps track of a file's or directory's information, such as where the data is stored and its metadata, such as size and creation timestamp. The `InodeTable` stores a mapping between an inode and the file's `InodeEntry`. The `InodeTable` has always at least one entry which is the root directory. This entry has a constant inode value of 0 for simplicity to look up the root directory. With the help of the root directory, all the files lower in the directory hierarchy can be found. The inode of all files and directories other than the root directory has a unique inode greater than 0. The `InodeTable` is always the most recent image saved on the , making it easy to find it on the .

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

1. The `Directory` object of the directory provides the inode of the given filename.
2. The inode is used to get the `InodeEntry` from the `InodeTable`.
3. Using the inode entry, the file can be located.

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

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

When a new file or directory is created, it is saved in its parent directory with its filename and an inode. The same inode is used in the inode table to keep track of the file's or directory's inode entry. As shown in Appendix A, the inode is represented as an unsigned 32-bit integer. The inode is calculated by adding one to the currently greatest inode. This means that new files and directories will always receive a higher greater inode than the ones currently in the inode table. This naïve approach to inode generation does not take into account that there might be an available inode less than the greatest inode in the inode table (for instance, due to the deletion of a previously created file). However, this inode generation approach is fast and will not be a problem until the integer overflows. As the inode is represented using a 32-bit integer, would need to have saved over four billion files before the inode value would overflow. This scenario is not in the scope of this proof-of-concept filesystem.

FFS does not support all filesystem operations that are implementable through , instead, implements a subset of them. The implemented functions are shown in Table 4.1. The implemented operations are the most vital operations required for a working filesystem [84]. Operations such as `chown` provide extended capabilities of the filesystem but these are not required for a proof-of-concept filesystem. The functionality of the filesystem operations implemented by and their implementation details are described in Subsection 4.2.5.

Table 4.1: Filesystem operations implementable trough the API, and wether or not implements them

Filesystem operation	Implemented by
`open`	Yes
`opendir`	Yes
`release`	Yes
`releasedir`	Yes
`create`	Yes
`mkdir`	Yes
`read`	Yes
`readdir`	Yes
`write`	Yes
`rename`	Yes
`truncate`	Yes
`ftruncate`	Yes
`unlink`	Yes
`rmdir`	Yes
`getattr`	Yes
`fgetattr`	Yes
`statfs`	Yes
`access`	Yes
`utimens`	Yes
`readlink`	No
`symlink`	No
`link`	No
`chmod`	No
`chown`	No
`fsync`	No
`fsyncdir`	No
`lock`	No
`bmap`	No
`setxattr`	No
`getxattr`	No
`listxatt`	No
`ioctl`	No
`flush`	No
`poll`	No

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

only the directory or file that is affected by a change is uploaded to the system, while the ones 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 , similar to how you can mount a network drive like a server. The mounted volume operates similarly to any other drive and can be accessed using, for instance, Mac's Finder or a shell terminal.

4.2.2 Cache

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

- a Cache Map - a mapping between a post ID and its image data, and
- a Cache Queue - a queue keeping track of the cached post IDs.

The cache stores a maximum of 20 image posts. The data stored in the cache is the encrypted image data. To avoid using too much memory, the cache is configured so that images greater than 5 MB are not cached. Each time an image is uploaded or downloaded, it is added to the Cache Map with its post ID as the key. The post ID is also added to the beginning of the Cache Queue. If the Cache Queue exceeds 20 elements, the last element of the queue is removed, and the corresponding entry in the Cache Map is erased, thus the entry is fully erased from the cache. The queue ensures that the cache is limited to 20 entries, and by using the valuation method, the queue ensures that the oldest element in the cache is removed when the cache exceeds the limit. When a file or directory is removed from the filesystem, all its data is also removed from the cache, if it is stored there.

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

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 .

4.2.3 Encoding and decoding objects

Entities that stores on the , and therefore also encodes and decodes, are: files, `Directory` objects, and the `Inode Table` object. All of these entities are stored on the using PNG images with 16-bit color depth. The inode table and the directories are represented as C++ objects in memory during runtime but are serialized into a binary representation before they are encoded into images. The files saved to are read into memory in a binary format before being encoded into images. All the data encoded into images are encoded similarly, and a detailed description of the binary structures can be found in Appendix B.

The input to the image encoder is the binary data do encode as an image. A header (header) is prepended to the binary data, containing among other things, the size of the data and a timestamp of when the data was encoded. The header and the input data are encrypted using authenticated encryption, utilizing and . The key used for the encryption is derived using the function utilizing the -256 hashing algorithm, along with a random 64 B salt vector, re-generated every time new data is encrypted. The salt is stored with the cipher to ensure that the decryption algorithm uses the same salt to derive the decryption key. The secret used in the is a password provided by the user. also uses a random , re-generated every time new data is encrypted. The length of the is set to 12 bytes. The resulting data from the encryption is the salt, the , and the encrypted cipher (including the authentication tag). These three data points are concatenated into a string of bytes. This string of bytes is referred to as the .

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

the image as $w = ceil(\sqrt{X})$. Further, the height h of the image is set as $h = ceil(\frac{X}{w})$. This will require $(w * h) - X$ filler bytes and will create an image with similar height and width. For certain values of X, h will be equal to w. For other values of X, $h = w - 1$. The resulting data encoded as pixels in the image is, in order:

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

The filler bytes are randomized.

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

To decode an image, the decoder first interprets the 4 first bytes as the . The salt and are retrieved from the as they are of known length. The decryption key is derived using the and salt and results in the same key as the encryption key because is a symmetric cipher algorithm. The remaining bytes of the $(-len(IV) - len(salt)$ bytes) are decrypted using the decryption key. The decrypted data consists of the header concatenated with the original stored data. The header is asserted to be in the correct format before the original binary data is returned from the decryption function. Figure 4.2 visualizes the encoder and decoder for all data saved in .

The encryption and decryption methods used are state-of-the-art solutions as defined and implemented by Crypto++ [83]. Crypto++ is a well-used and well-maintained C++ library for cryptography, and as of writing has no reported CVE security vulnerabilities for the functionality used by [85].

An image has an upper size limit, defined by the used. If the data to be stored in , such as a file, exceeds this limit, it is split into multiple data arrays of sizes less than this limit. Each data array is encrypted and encoded as images independently of each other, and will be encrypted using different salts and s. Only the inode table stores

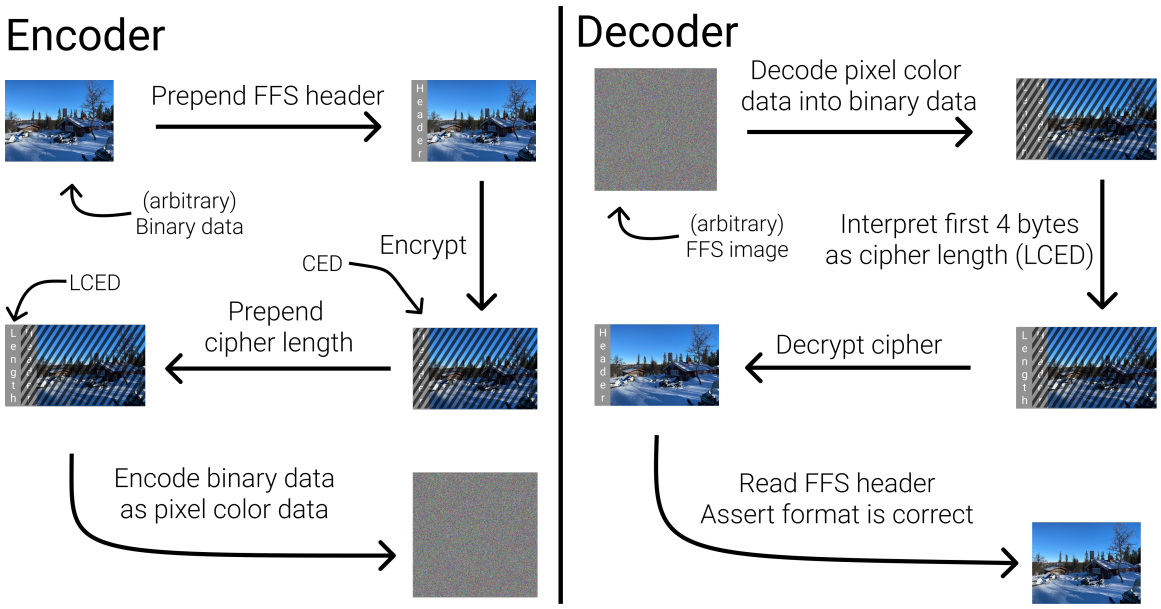

Figure 4.2: Simple visualization of the encoder and decoder of . The input of the encoder is the binary data to store in , eg. a file, and the output is the image to upload to the . The input to the decoder is an image, and the output is the binary data stored on , eg. a file

the different post IDs in the order they are encoded in. While files and directories stored in can be separated into multiple images, the inode table is limited to only one image for simplicity when interacting with the . This introduces a size limit of the inode table, limiting the filesystem. More details about the limits of are found in Subsection 4.2.6.

4.2.4 Online web services

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

The initial intention of was to use Twitter as the . Initial research for the thesis found that it was possible to upload a file and download the same file without any data loss. However, it was later found that this was not a reliable conclusion. Some images

uploaded to Twitter were converted to another image format when they were stored by Twitter, which meant that the decoder could not decode the data as it expected another image format. Other images were compressed or re-coded which led to data loss when downloading the image. As the decoder of images relies on a specific binary representation of the image, this meant that the images could not be decoded into the previously uploaded data. Twitter has previously publicly announced changes to the way they store images [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 to the same image, which contradicts the workaround mentioned by the Twitter employee. Further changes may have been made to the data management of images on Twitter since the initial research for the thesis, however, an official announcement has not been found.

Flickr saves the original version of the uploaded image and thus it can be used to download the same image as was uploaded. This also means that data that is encoded into an -encoded image can be uploaded, downloaded, and decoded into the same data as before. While they do not assure that they will always support original images, they also do not indicate that this would change. Therefore, Flickr can be used at this moment for the proof-of-concept filesystem that is. A free-tier Flickr account is therefore used for .

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. Every time an image is uploaded to Flickr, it is due to some modification in the filesystem, for instance, a write operation to a file or a creation of a new directory. For every modification in the filesystem, the inode

table will have to be updated. Therefore, we can ensure that the inode table is always the most recently uploaded image to Flickr by configuring to upload all other images first, for instance, the newly written file. This provides with a simple way of querying the inode table from Flickr - by simply requesting the most recently uploaded image on the Flickr account.

While the Flickr API is extensive in its functionality, only uses a few of the provided capabilities. The Flickr API capabilities that uses are:

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

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

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

For benchmarking purposes, a fake variant of , , has also been developed. uses a , which stores the data on the local filesystem. The is used by just like how Flickr is used by , by storing encoded images on it. By storing the images on the local filesystem, the filesystem operations duration is shorter as the local filesystem operations are in general faster than the network requests. This allows us to analyze the theoretical performance limit of , and how it could perform if the used had very low latency and the network connection to the had very high bandwidth. By analyzing , we can also estimate how much of the filesystem operation time is affected by the time of the network requests. The time T of an filesystem operation can be modeled like:

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

where t_{ffs} is the time that takes to, for example for a file read operation;

- 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}. t_{ows} is the total time required to complete all requests to the for a filesystem operation. For instance, for a similar read operation as above;

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

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

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

in the `/tmp` directory of the local filesystem and is removed by directly 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, 89, 90]. Although, these methods require you to decrypt the volume, requiring the decryption password. Without this password, the data cannot be recovered. Even with the decryption password, it is not certain that the data is recoverable. If an adversary obtains proof that an image has been present in the `/tmp` directory, they could conclude that has been used to store data, reducing the deniability of the filesystem.

4.2.5 Implemented filesystem operations

Following is a detailed description of all the operations implemented by , and how they are implemented by . Further explanations about the intended functionality of the operations can be found in [84].

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

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)
```

By traversing a path, has to fetch all parent directories in the hierarchy. The file or directory with the filename is not fetched while traversing the path, as it might not be necessary for the operation. All operations that rely on the path of a file or directory have to download all parent directories of the path. However, the directories in the path could be cached and therefore would not be required to be downloaded from the . Further, `open`, `opendir`, and `create` associates a file handle with a file or directory, so that certain other operations can 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 and cached. Therefore, it is assumed that the inode table is always up to date in memory and on the . This will be true as long as there are not multiple instances working with the same account at the same time. This scenario has undefined behavior as there is no locking implemented for .

All filesystem operations are synchronous unless specified. Further, is running in single-thread mode meaning that a filesystem operation call must complete before another can begin. This helps limit the risk of data races as two processes cannot call different operations that, for instance, modify the inode table at the same time.

4.2.5.1 open

Given a path to a file, the file is associated with a file handle. The file handle is used in subsequent operations to avoid traversing the file path multiple times. The file is not downloaded from the , only the parent directories are downloaded during the path traversing as explained above. An `open` call must, eventually, be followed by a `release` call. Although, multiple other operation calls can occur between these events.

4.2.5.2 create

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

4.2.5.3 release

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

Subsequent operations for the file will require path traversing as the file handle can no longer be used.

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

4.2.5.4 opendir

Given a path to a directory, the directory is associated with a file handle. The file handle is used in subsequent operations to avoid traversing the file path multiple times. The directory is not downloaded from the , only the parent directories are downloaded during the path traversing as explained above. An `opendir` call must, eventually, be followed by a `releasedir` call. Although, multiple other operation calls can occur between these events.

4.2.5.5 releasedir

Given a file handle, this operation closes the directory in the filesystem, disassociating the file handle from the directory. The current states of the directory and the inode table are saved to the , and the previous versions of the directory and inode table are deleted from the . Subsequent operations for the directory will require path traversing as the file handle can no longer be used.

The directory must have a file handle associated with it before `releasedir` is called. This requires a preceding `opendir` call.

4.2.5.6 mkdir

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

new posts, and a new entry is created for the new directory. The inode table is updated in the , and the old version of the table is removed from the .

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

4.2.5.7 read

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

4.2.5.8 readdir

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

4.2.5.9 write

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

The file and inode table are not updated on the , this occurs instead in the subsequent `release` call.

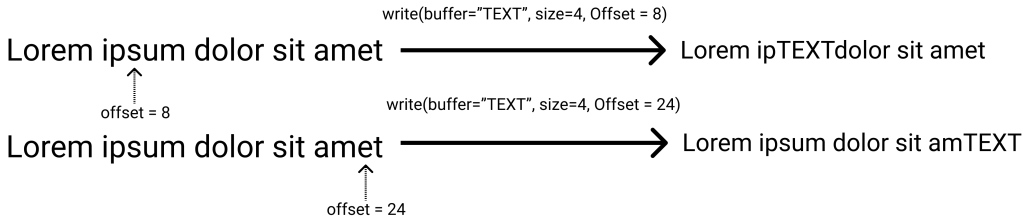

Figure 4.3: Visualization of how the write operation handles different offsets.

4.2.5.10 rename

This operation renames a file or directory to a new path. Both the old path and the new path have to be traversed to locate the parent directories and the file or directory to rename. The file or directory entry in the old parent directory is removed, and the old parent directory is updated to the . A new entry is created in the new parent directory, with the new filename. The new parent directory is updated to the . The inode entry of the renamed file or directory does not have to be modified. However, as both the old parent directories and the new parent directory are updated in the , their inode entries need to be updated with the new posts. The inode table is updated to the and the old table is removed from the . The old posts associated with the old parent directory and the new parent directory are removed from the .

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

4.2.5.11 truncate

This operation truncates or extends the file in the given path, to the provided size s. The full current file is downloaded into memory. The data of the current file is written to a new buffer until either the file is fully written, or until s bytes have been written. If the current file's size is smaller than s, the remaining bytes are written as the NULL character. The new file data is uploaded to the , and the old data is

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

4.2.5.12 ftruncate

This operation is similar to `truncate`, but is called from a user context which means it has a file handle associated with it. The operation truncates or extends the file in the given file handle, to the provided integer s. The full current file is read into memory, either from the or from the cache. The data of the current file is written to a new buffer until either the file is fully written, or until s bytes have been written. If the current file's size is smaller than s, the remaining bytes are written as the NULL character.

The file and inode table are not updated to the , this occurs instead in the subsequent `release` call.

4.2.5.13 unlink

This operation removes a file given the file path. The file is removed from the parent directory, and the parent directory is updated to the . The old parent directory data is removed on the . The removed file's entry in the inode table is also removed, and the inode table updates the entry for the parent directory with its new posts. The inode table is then updated on the and the old inode table is removed on the . Finally, the data of the removed file is removed from the . The last step is not necessary for a working filesystem; however, to save space on the , this is done. If the permits unlimited images and sizes, this step could be omitted to execution save time.

4.2.5.14 rmdir

Similar to `unlink`, this operation removes the directory at the path. The directory and all its subdirectories are traversed, and the post IDs of these files and directories are recorded for deletion later. Following this, the entry of the removed directory

is removed from the parent directory. The inode entry for the removed directory is removed. The parent directory is updated to the , and the inode table is updated with the new posts of the parent directory. Following this, the inode table is updated to the . The old parent directory and the old inode table are removed from the .

The operation also starts a new thread, where all the posts of files and subdirectories inside the removed directory, are removed from the . They are removed to save space on the , and a separate thread is used to save computation time 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.

4.2.5.15 getattr

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

4.2.5.16 fgetattr

This operation is similar to `getattr` but is called from a user program context meaning that the file has a file handle associated with it. Other than skipping the path traverse step, this operation returns the equivalent information as `getattr`.

4.2.5.17 statfs

This operation returns metadata information about . This includes, among other things, the maximum filename size and the filesystem ID. The operation has a short computation time as it does not have to download or upload any files. The only variable information is read from the inode table which is stored in memory and thus does not have to be downloaded from the .

4.2.5.18 access

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

4.2.5.19 utimens

This operation, provided new timestamps, updates the last access timestamp, the last modified timestamp, or both, of the file or directory at the given path. The file or directory does not have to be downloaded. However, the inode entry for the file's or directory's inode is updated with the new timestamps if they are newer than the previous timestamps but not greater than the current time since epoch. The new state of the inode table is updated to the , and the old version is removed from the .

4.2.6 FFS limitations

FFS has numerous of limitations due to both implementation decisions and limits. As Flickr allows a free-tier user account to store up to 1 000 images of up to 200 MB per image, this allows storage of up to 200 GB of images per account on Flickr. However, as the inode table is required to be stored on the filesystem, a maximum of 999 images can be used to save file and directory data. This limits the filesystem to a maximum of 999 files and directories when utilizing one free-tier account on Flickr.

While Flicker supports each image to be up to 200 MB, it is not possible to use the full 200 MB as the file data to store. The image includes, among other things, a PNG header, other PNG attributes, and the which in total is of greater size than the unencrypted data. To ensure that the along with the PNG header and other PNG attributes does not exceed the limit of 200 MB, limits the size to allow at least 10 MB for the PNG header and other PNG attributes, meaning that the can be a maximum of 190 MB. The cryptographic variables , salt, and the authentication tag are stored in the using 12, 16, and 64 bytes respectively, for a total of 92 bytes. The size limit means that these 92 bytes, along with the encrypted cipher text, cannot exceed 190 MB, meaning that the encrypted cipher text cannot exceed 190 000 000−92 =189 999 908 B. However, as is a block cipher producing cipher blocks of 16 bytes, the resulting cipher text must be divisible by 16. The largest encrypted cipher text that allows is therefore $floor(\frac{189\,999\,906}{16}) * 16 = 189\,999\,904$ bytes. Due to plain text padding, the unencrypted plain text can be a maximum of one byte less than this value [91], meaning that the plain text can be a maximum of 189 999 903 B. For simplicity, this is rounded down to 189 MB, leaving almost 11 MB in total for the PNG header and other PNG attributes. 189 MB is set as the maximum amount of data will store per image. Data greater than 189 MB is split into multiple encoded images. For instance, a file of 200 MB will be stored as 189 MB in one image, and 11 MB in another.

189 MB of usable data per image gives a maximum storage capacity of 188.811 GB using 999 files and directories on one free-tier account on Flickr. Each file with data requires at least one image, thus there can be a maximum of 998 non-empty files and directories in the filesystem, excluding the root directory. However, there could also be just one single entry of 188.811 GB stored in the filesystem, which would have to represent the root directory.

The inode table keeps the information about empty files and directories even though they store no data on the . The inode of a file or directory is an unsigned 32-bit integer, meaning that the inode table could theoretically store 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 B in the inode table, consisting of the inode, length,

and other variables that must exist for an inode entry. As the inode table is limited to one single image on the , the inode table is limited to a maximum size of 189 MB. Further, the size of the inode table is 4 B plus the size of each entry, and one of these entries is the root directory. Even if a file is empty, it is still stored with its filename and inode in its parent directory. A non-empty directory in the inode table requires approximately (depending on the post ID length generated by the) 12 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 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 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. However, even if the computer has less memory available, more memory can often be provided through a memory swap on the hard disk. Apple ensures that the swapped data is securely encrypted on the hard disk [92]. However, using a memory swap puts a constraint on the storage of the computer to be sufficient. Further, as 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 . If the local filesystem has no available storage, very few filesystem operations can be performed on as any operation that modifies the inode table requires the new inode table to be saved to the local filesystem before it is uploaded to Flickr.

Another limitation of 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 could reach 3 600 API calls faster. However, as long as is not constantly serving filesystem calls, the API limit should be of little concern.

A limitation of 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 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 are compared with the result of two different instances of ; one instance that uses Flickr as its , and one instance that uses a by storing the encoded images in the local filesystem on the test machine.

4.3.1 Filesystems

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

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

The encrypted filesystem was used as a reference for a local filesystem without the required internet connection. It is the local filesystem of the development environment

for . It was selected as it will give the analysis an example of a modern, well-used, and fast filesystem, and how the benchmark data of and other filesystems look compared to this local filesystem.

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

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

4.3.2 Tools

IOZone [9] 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 multiplies by 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 run 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 has been 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. IOZone does also 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. Although, the thesis has tested cases of the functionality aspect of both and , and found that they support directory hierarchies and multiple files as expected. Future work could research the functionality aspect of these filesystems utilizing online storage systems. is expected to have full functionality as 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 . Further, 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 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 of the benchmarking, the maximum file size of the tests should be set to a value bigger than the filesystem cache. While the cache limit is known to be 5 MB, the cache size limit or the existence of such a limit for the other filesystems, such as ,

is unknown. The 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 and . Each doubled file size takes exponentially much more time as both the file size and the buffer size is doubled. Further, it has been found that both and might occasionally crash during benchmarking due to numerous factors, 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 , and potentially other filesystems save the data to the storage medium only after the device is closed. In the case of , if the time of closing the file was not included, the performance of the filesystem would appear to be higher than it is.

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

The benchmarking of and 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 and 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 , which uses Flickr as its . The source code of can be found on GitHub [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 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. It is a table of 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 ran for 41 minutes, and the IOZone benchmarking for took 20 minutes. They were started at the same time and the internet speed tests ran

every five minutes until the benchmarking of was completed. In total, eight speed tests were conducted with an average latency of 15.22 ms. The average download speed was 90.96 Mbit/s and the average upload speed was 92.95 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 5.2 presents a box plot of the benchmarking results of . It can be observed that the Read, Re-Read, and Random Read operations are significantly faster than the Write, Re-Write, and Random Write operations. The Write, Re-Write, and Random Write operations all have a big spread of values. It can also be noted that the median performance of Re-Read and Random Read are better than the median performance of Read. The median performance of Re-Read and Random Read are comparable where Random Read has a bigger spread of values.

Figure 5.1: Box plot of the IOZone output for the different tests for

Figure 5.2 presents a box plot of the benchmarking results of . It shows that Write, Re-Write, and Random Write have significantly less performance than the results of Read, Re-Read, and Random Read. It can be noted that the median performance of Re-Write and Random Write are similar, and both are less than the median performance of Write. The median performance of Re-Read and Random Read are higher than the median performance of Read.

Figure 5.2 presents a box plot of the benchmarking results of . In this plot, one can see that the spread of the values from Write and Re-Write is high. Looking at Table C.14, we can see that certain values are indeed greater than the most, such as `file size = 2048, buffer size = 128` where the performance is 116 330 kB/s which is significantly more than most values in the table. Further, Figure 5.2 shows that Read, Re-Read and Random Read have similar performances, where the median and best performance of Re-Read and Random Read is better than the median and best performance of Read. Random Write has less of a spread than Write and Re-Write has, and the median performance of Random Write is better than the median performance of both Write and Re-Write.

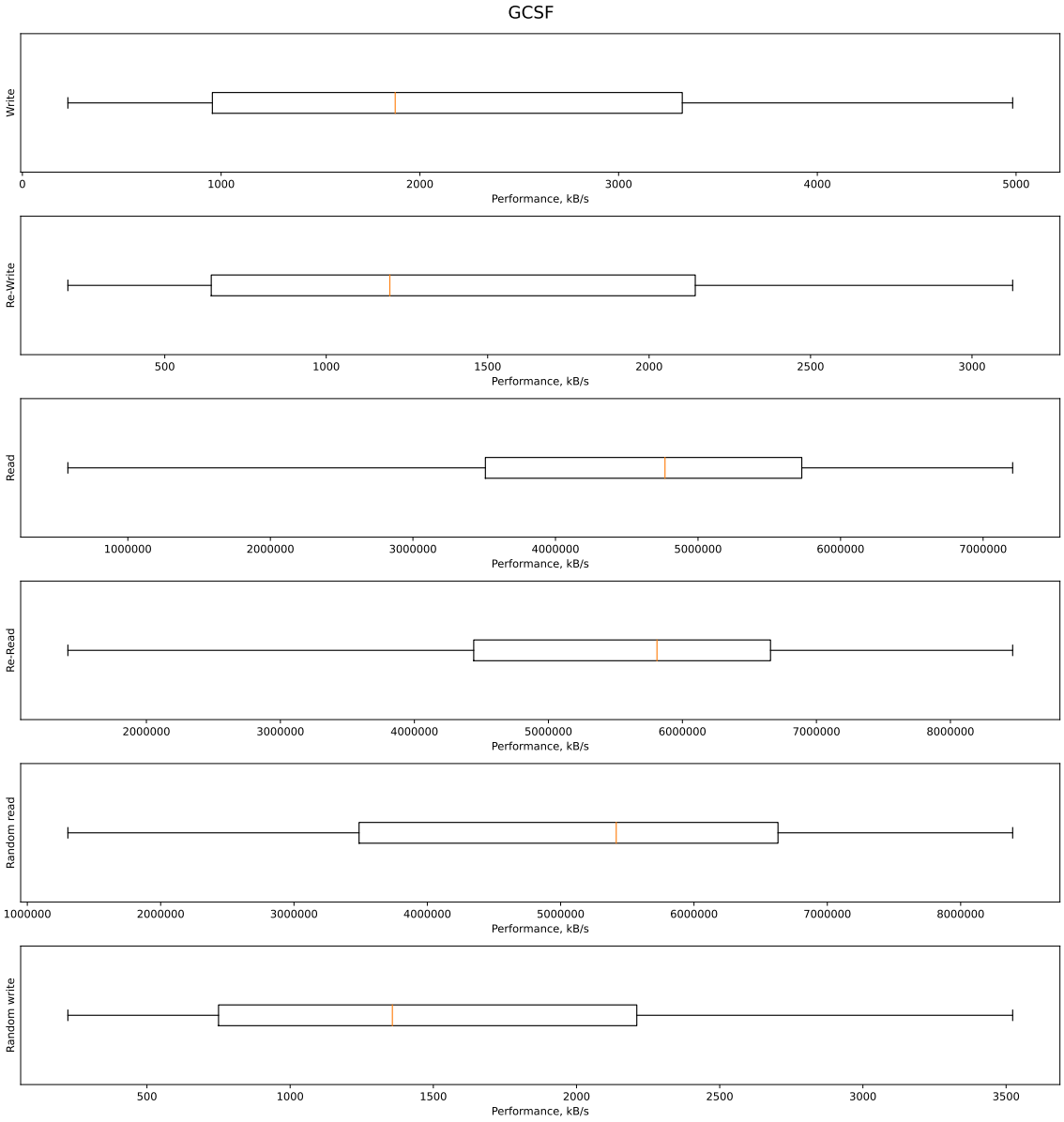

Figure 5.2: Box plot of the IOZone output for the different tests for

Figure 5.3: Box plot of the IOZone output for the different tests for

Figure 5.2 presents a box plot for . In this figure, it is possible to see that the performance of the different write operations is lower than the performance of the read operations. Further, it can be noted that the median performance of the Re-Read and the Re-Write tests is better than the performance of the standard Read

and Write tests. The median performance of Random Read is similar to the median performance of Read, but the median performance of Random Write is significantly lower than the median performance of Write.

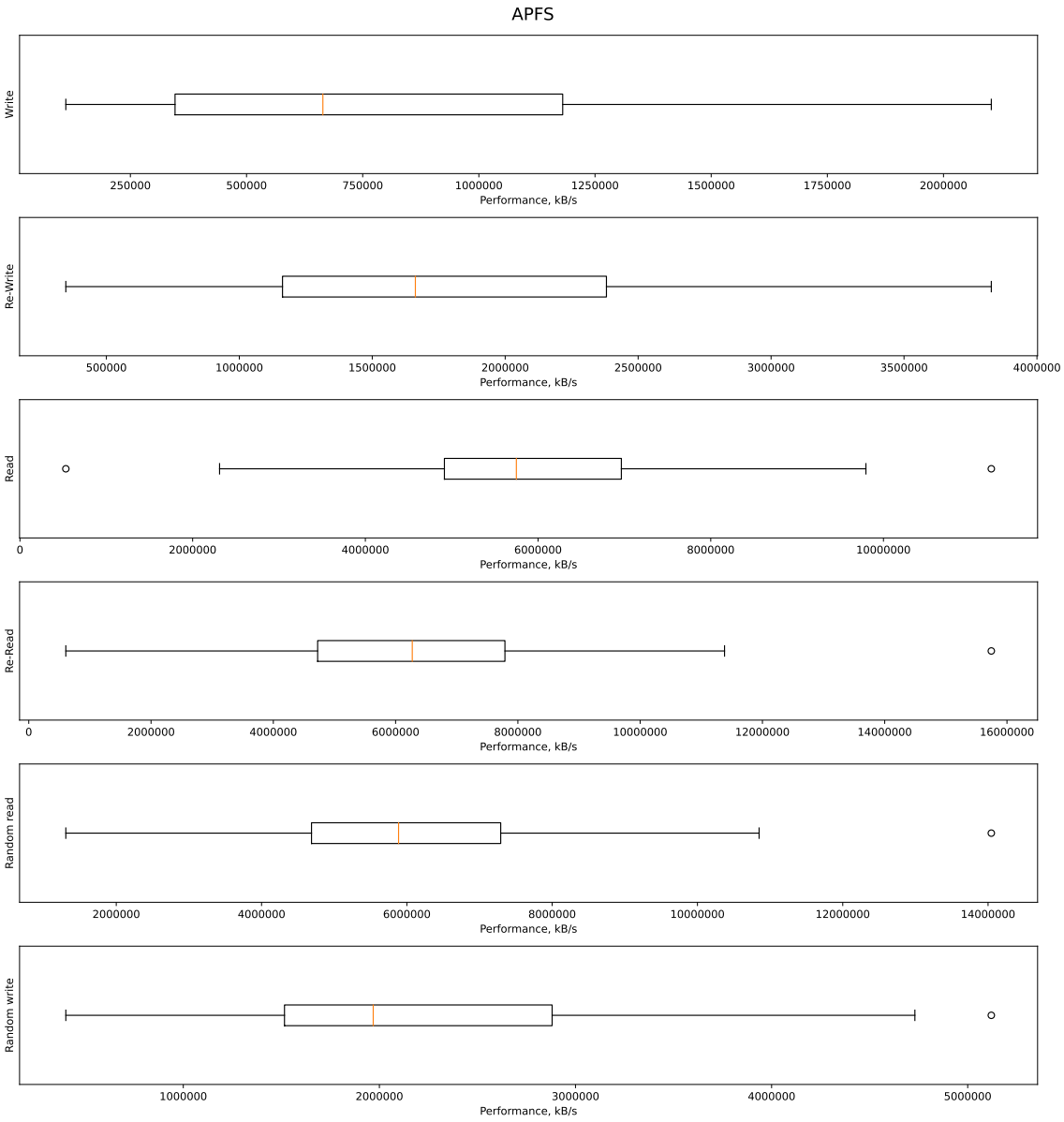

Figure 5.4: Box plot of the IOZone output for the different tests for

Chapter 6

DISCUSSION

This chapter presents discussions and analysis of the results. The presented benchmarking results are analyzed and discussed. Further, 's deniability and security are analyzed and discussed. Potential societal impacts of are also introduced.

6.1 Filesystems

Figure C.1, C.1, and C.1 show that performs poorly for Read operations with a small buffer size. Beginning at 4 kB buffer size, the performance in general goes up with the first few buffer sizes. This indicates that the overhead of the read operation is high as the performance gets better when it can read 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 5.2 shows, the average performance of Re-Read is better than the average performance of Read for . This supports the theory that the cache will increase the performance of the filesystem.

One interesting comparison is between the benchmark results of and . Both filesystems are cloud-based filesystems dependent on an internet connection to their respective storage servers. Knowing that the benchmarking of the two filesystems was started at the same time, and the benchmarking took 41 minutes while the bench-

marking took 20 minutes, it is already notable that is overall faster than 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 is slower than . For instance, the Read test of performed in general better than the Read test of . The median performance of the Read test is significantly better than the median performance of the Read test. Looking at Figure 5.2 and Figure 5.2 we can see that produced much bigger spread of the values than did, especially for the Write, Re-Write, and Random Write tests. One outlier of the Write test of 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 for the Write, Re-Write, and Random Write tests.

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

While the values of the read operation for and are comparable to each other, this is not the case for all tests. For instance is the write operation of much slower than the write operation of as can be seen in Table C.14 and Table C.20. The write

operation of has about 2-3% the performance of the write operation of . The reason for this can be the fact that has to encrypt the data stored, including creating all the cryptographic variables such as the salt and the . While is also an encrypted filesystem, it is possible that the cryptographic functions are much faster than for as they for instance can be run in kernel space, while is running in user space.

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

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

6.2 Security and Deniability

The data stored in images is encrypted with state-of-the-art encryption standards. Using -, does not only provide 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 is dependent on, among other things, the password the user chooses. A bad password, for instance, short or commonly used, is easily breakable for an adversary. An adversary who has access to an encrypted image could brute-force the bad password used to derive the encryption key much faster than they could brute-force the encryption key. does not put any constraints on the password used - as long as it is at least one byte it is acceptable for . This puts the responsibility on the user for the choice of password constraints.

FFS puts a lot of trust in the open-source library Crypto++ [83]. Crypto++ provides cryptographic functions that uses for, among other things, deriving the encryption key, encrypting the data, and verifying the authentication tag. While there are no reported CVE security vulnerabilities as of writing citeCryptoppSecurityVulnerabilities, 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 provides vulnerabilities, such as side channels, which could be exploited. is developed by a single author without a review from anyone else.

Anyone with access to Flickr.com can view and download the original images stored by , 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 who view the Flickr profile, they see different sizes of images with seemingly randomly generated pixel colors. However, for adversaries who know about the details of , more information can be retrieved. For instance, they could assume that the most recently uploaded image to Flickr is representing the inode table. However, as we assume the adversary does not have access to the decryption key, they cannot read the data of the image and thus cannot verify that this is indeed the inode table. The exact number of files

and directories in cannot be known precisely without access to the content of the inode table. Even if the Flickr account has, for instance, 15 images stored, and we know that one represents the inode table and one represents the root directory, it is not possible to conclude if other images stores file data or directory data. The remaining 13 images in the example could represent:

- one big single file split over 13 images, or
- one big single directory split over 13 images, or
- 13 different files, or
- 13 different directories, or
- 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 or by the user manually to further diffuse the amount of data stored on the service. For instance, by encrypting random data using 's encoder and uploading the images to Flickr, but without saving the posts in the inode table or in a directory of , the images will look indistinguishable from the other images on Flickr. Only with access to the decrypted inode table can one know if the image is stored in 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 to post encrypted random data at random time intervals to automatically diffuse the knowledge about the images on the . This would mean that even Flickr would not be able 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 is that it decreases the storage capacity of .

The size of data stored in an image is not completely hidden. While the exact number of bytes of unencrypted data that the image stores is not possible to know without the decryption key, it is possible to get an estimate. If you know the binary structure of the image (as presented in Appendix B), you can find out how many bytes the encrypted cipher is, the value of the data, the value of the salt used for the encryption key derivation, and the value of the authentication tag. By knowing the length of the cipher, the length of the unencrypted data can be placed in a range. The length of

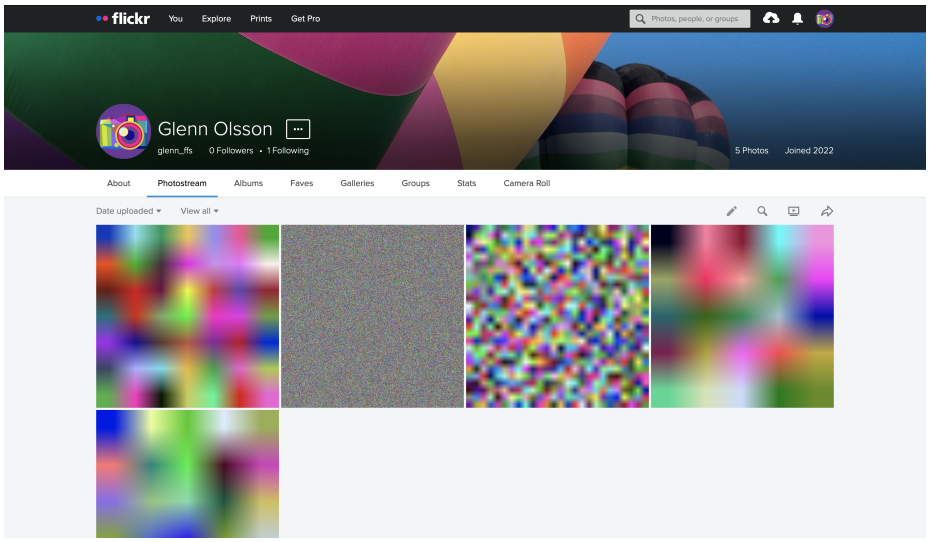

Figure 6.1: Screenshot of the Flickr profile used for . At the moment of the screenshot, the filesystem is storing a previous version of this thesis in a directory inside the root directory. The images seen are the inode table, the thesis data, the root directory data, the subdirectory (containing the thesis) data, and a temporary file containing extra attributes of the thesis document created by macOS while was mounted (this file is sometimes referred to as a *turd* [98]).

the cipher L_c in bytes is divisible by 16 (as is a 16-byte block cipher), and the length of the plain text must be less than L_c due to the requirement of at least one bit of padding [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 stored by on Flickr at a certain time. However, it is **not** possible to know if all this data is stored on through entries in the inode table. It is also **not** possible to know if the plain text represents a file or directory without the decrypted data of the inode table.

If a user supplies a different password when mounting than used previously, the images stored on Flickr cannot be decrypted. When tries to read the image it believes represents the inode table (the most recently uploaded image) and it fails,

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

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

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

inode table in another storage medium and upload it again when they want to access their actual instance.

One aspect where is better than is its security against the potential adversary of the store owners. stores the data in its original format on Google Drive, essentially providing an overlay filesystem for Google Drive. While this can be desired in certain situations, such as using on one machine and the Google Drive website on another, it gives Google Drive access to your data. As mentioned, Google Drive encrypts your data from outside agents, but as they control the encryption and decryption keys themselves, the data stored can be accessed by the company. For instance, the data could be given to authorities who are requesting it with a subpoena. on the other hand gives the user control of all its data. While Flickr can give out the images uploaded by , this data can be accessed by anyone with access to Flickr.com anyway. 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 with one aspect of better security than , but this might also be a factor why is slower than . By requiring the data to be encrypted when it is written and uploaded, and decrypted when it is downloaded and read, will need to compute new cryptographic variables every time a file or directory is written which requires a lot of computations. Further, every time an image is read it must be decrypted, even if it is in the cache of . Decrypting an image requires a lot of computations as well as, other than decrypting the data, the decryption key must first be derived from the password. Meanwhile, it is possible that Google Drive is caching the unencrypted files, or performing the cryptographic computations on high-performance computers requiring less computation time. So while gaining a security aspect of the filesystem, sacrifices the performance of the filesystem operations.

6.3 Impact

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

6.3.1 Societal impacts

Secure and hidden data is not only for the better good. As the data stored on cannot be decrypted by bad guys 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]. could potentially provide secure storage for illegal groups such as terrorist organizations and child pornography rings. It is not possible to limit who uses , by other means than not publishing the source code of the filesystem. However, this does not prevent criminal organizations 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 , 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 to a cloud-based filesystem, such as Google Drive. While both Google Drive and store their data in data centers, the data that Google Drive stores can be less than the same file stored in due to the overhead of . Further, Google Drive can be optimized as a

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

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

7.2 Future work

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

Even though the files are encrypted so that the data is confidential, further research could include hiding the user's online activity through the use of for instance Tor. Currently, the integrity of the user is not considered but for to be further plausibly deniable, this should be addressed as the user could otherwise be identified by its IP address and other online fingerprints that could be provided by the online web services.

To improve the dependability of , support for more online web services could be implemented. For instance, GitHub provides free user accounts with many gigabytes of data. Even free-tier distributed filesystems, such as Google Drive, could be utilized. If multiple user accounts are used in coordination over multiple services, could achieve even more storage.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[43] Gaurav Kodwani, Shashank Arora, and Pradeep K. Atrey. "On Security of Key Derivation Functions in Password-based Cryptography". In: *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2021 IEEE International Conference on Cyber Security and Resilience (CSR). July 2021, pp. 109–114. DOI: 10.1109/CSR51186.2021.9527961.

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

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

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

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

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

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

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

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

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

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

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

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

[56] Tatsuya Chuman, Warit Sirichotedumrong, and Hitoshi Kiya. "Encryption-Then-Compression Systems Using Grayscale-Based Image Encryption for JPEG Images". In: *IEEE Transactions on Information Forensics and Security* 14.6

(June 2019), pp. 1515–1525. ISSN: 1556-6021. DOI: 10.1109/TIFS.2018.2881677.

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

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

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

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

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

[62] Jędrzej Bieniasz and Krzysztof Szczypiorski. "SocialStegDisc: Application of Steganography in Social Networks to Create a File System". In: *2017 3rd International Conference on Frontiers of Signal Processing (ICFSP)*. 2017 3rd In-

ternational Conference on Frontiers of Signal Processing (ICFSP). Sept. 2017, pp. 76–80. DOI: 10.1109/ICFSP.2017.8097145.

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

[64] Richard Jones. *Google Hack: Use Gmail as a Linux Filesystem*. Computerworld. Sept. 15, 2006. URL: https://www.computerworld.com/article/2547891/google-hack--use-gmail-as-a-linux-filesystem.html (visited on 03/09/2022).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[85] *Cryptopp : Security Vulnerabilities*. URL: 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. URL: 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. URL: 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. URL: 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. URL: https://www.easeus.com/mac-file-recovery/recover-files-from-apfs-drive.html (visited on 09/11/2022).

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

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

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

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

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

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

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

[97] *RandomNumberGenerator - Crypto++ Wiki*. Apr. 13, 2021. URL: 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. URL: https://apple.stackexchange.com/a/14981 (visited on 09/11/2022).

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

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

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

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

APPENDICES

Appendix A

DIRECTORY, INODETABLE, AND INODEENTRY CLASS AND ATTRIBUTES REPRESENTATION

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

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

```
# typedef inode_id = uint32_t

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


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

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

```
        uint8_t is_dir

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

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

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

Appendix B

BINARY REPRESENTATION OF IMAGES AND CLASSES

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

B.1 Serialized C++ objects

The `InodeTable`, `InodeEntry`, and the `Directory` class all have one `serialize` and one `deserialize` method each. The `serialize` method converts the object's data into binary form, and the `deserialize` method converts the serialized data into an object. The deserializer expects the same format of its input data as the serializer produces. The figures in this section visualize the serialized output of the different classes. Figure B.1 visualizes the serialized format of the `InodeTable`. Figure B.1 visualizes the serialized format of the `InodeEntry`. Figure B.1 visualizes the serialized format of the `Directory`.

B.2 FFS Images

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

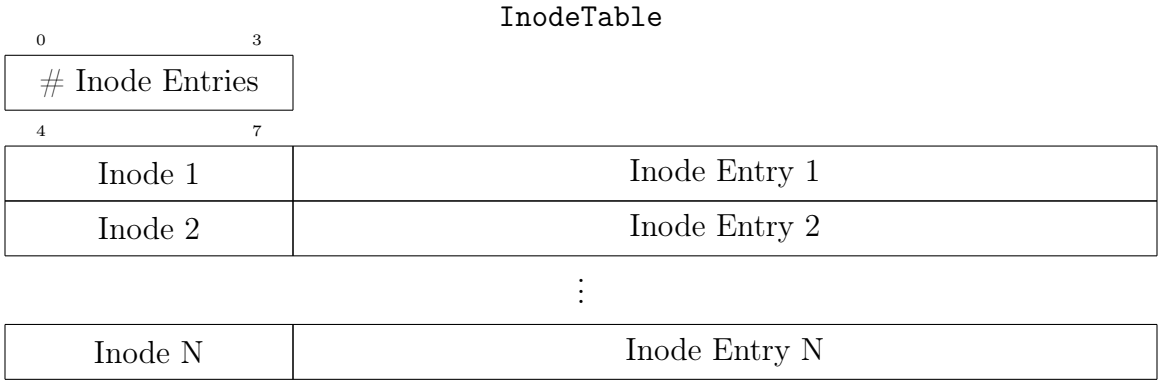

Figure B.1: `# Inode Entries` **is an unsigned integer representing the amount of inode entries the inode table contains. Following are** `# Inode Entries` **entries of an unsigned integer representing the inode of the inode entry, and the serialization of the corresponding** `InodeEntry` **object**

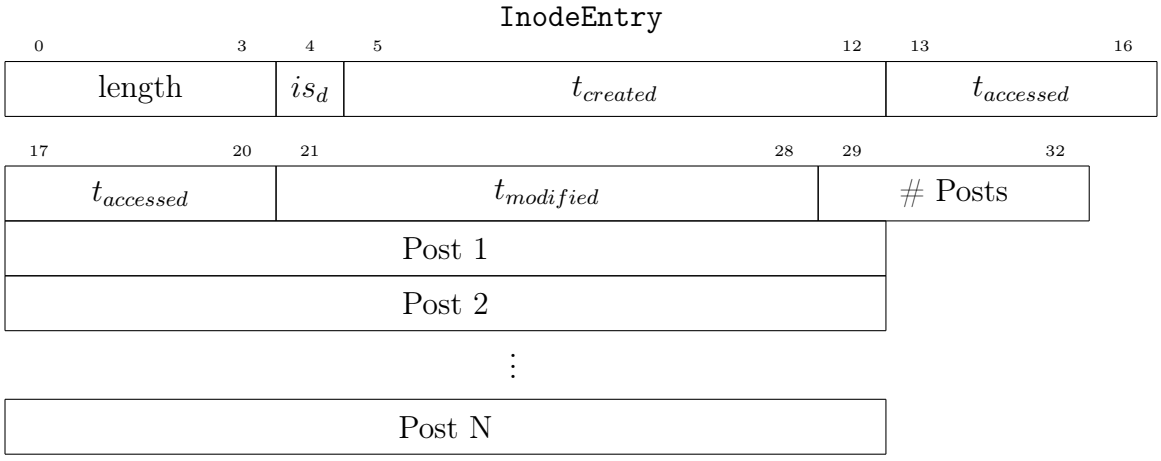

Figure B.2: Byte representation of a serialized `InodeEntry`**, representing a file or directory stored in . `length` is an unsigned integer representing the amount of data stored on by the file or directory, for instance the size of the file.** is_d **is a boolean with the value true** ($\neq 0$) **if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

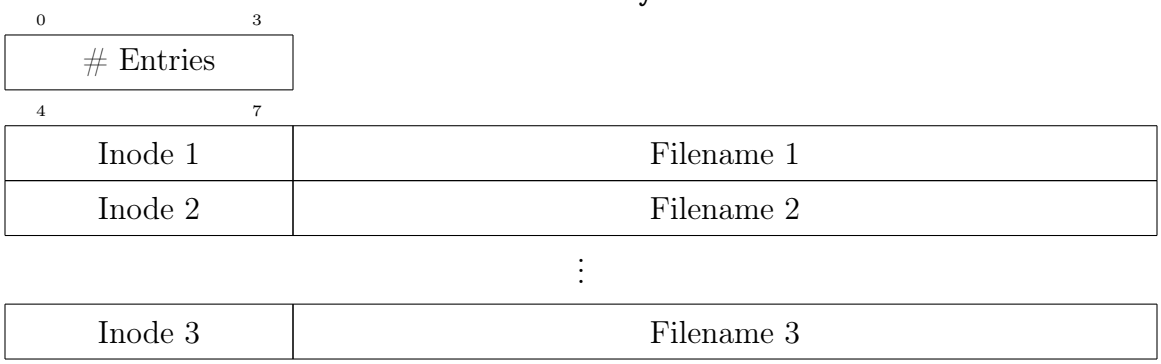

Figure B.3: Byte representation of a serialized `Directory`. `# Entries` is an unsigned integer representing the amount of entries in the directory. Following are `# Entries` inode-filename pairs. The `Inode` is an integer representing the inode of the file or directory, corresponding to the file's or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

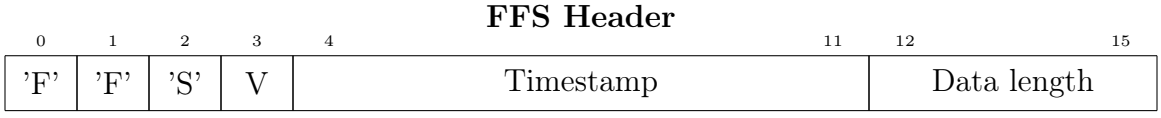

Figure B.4: `'F'` and `'S'` are the literal letters F and S in code. `V` is an integer representing the version of the image produced. `Timestamp` is an unsigned integer representing the number of milliseconds since Unix epoch when the image was encoded. `Data length` is an unsigned integer representing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

in images

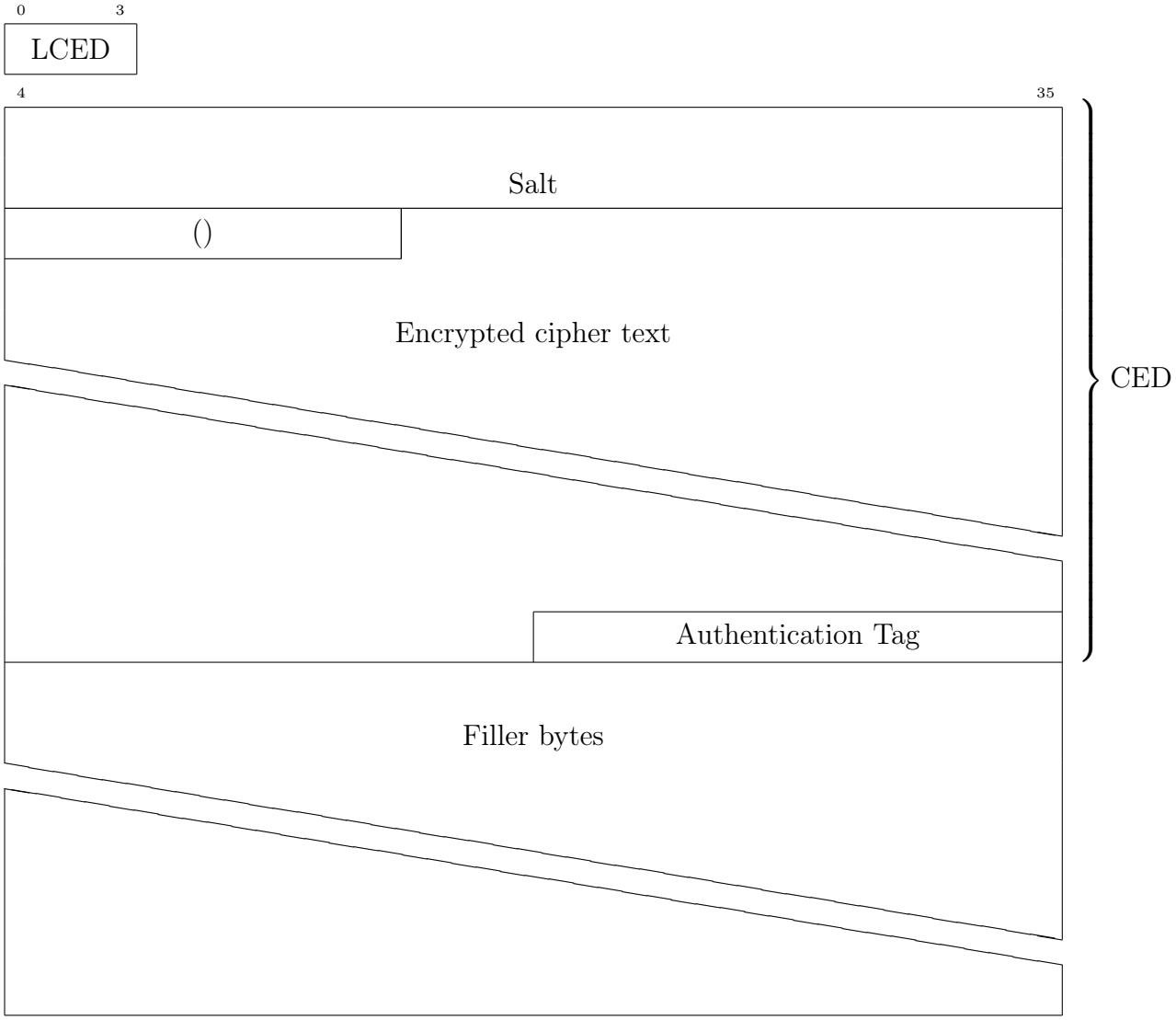

Figure B.5: Byte representation of the data stored as in images. `LCED` is an unsigned integer representing the Length of the . The `Salt` is a 64-byte randomized vector used to derive the encryption and decryption key. The `IV` is a 12-byte randomized vector used as the initial state of the encryption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

Appendix C

IOZONE BENCHMARKING DATA

This appendix contains tables 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 FFS Read

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	873705	1635675	2044438	2270300	3066069	2275110	2370545	3532609	3567824				
2048	1860176	2645657	3938881	3696494	3255235	2771120	4405533	3743207	2892423	3130664			
4096	2360583	1793342	2929815	3806915	4241099	4795088	3871253	5107271	3297602	4718699	3515562		
8192	2512998	1952390	3632950	3742562	5322300	4721816	4749227	6321001	6187815	3723500	4506310	5291970	
16384	2498271	3786446	4627011	3795229	5915013	6478199	6619859	6579926	6595715	5931862	4705264	4929022	5066012

Table C.2: IOZone result for FFS Write

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	242	230	276	269	238	248	270	222	254				
2048	97	429	453	163	121	181	392	426	530	409			
4096	747	492	778	599	751	715	753	656	716	255	676		
8192	1018	1017	900	1130	1046	958	247	1193	1117	88671	992	1179	
16384	1053	1234	989	1335	1060	1338	765	761	1321	1278	1198	1258	1262

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	948157	2260740	3281592	2540189	1815435	2115949	3541348	3046495	3821805				
2048	2188069	3286370	4500161	4663865	4432815	4321315	5280246	4521480	5293261	5267294			
4096	2721431	1901520	3140855	6013663	6068897	5736555	5696610	5664678	4501111	6204794	3453374		
8192	2797840	2854319	4864885	4304757	5626466	6486879	5814021	6311712	7067279	4053382	4493932	5241116	
16384	2598600	3568766	4164613	5259892	5866036	6954141	7019493	7080244	7071501	6008099	5595225	5496324	5359160

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	209	273	226	239	281	159	100	182	31994				
2048	354	139	66	50	473	371	146	482	434	464			
4096	722	741	779	573	680	812	649	667	741	34591	639		
8192	719	863	117	840	976	953	872	956	805	1322	5831	1057	
16384	874	1008	1124	1097	968	1101	1082	1015	1193	1031	6722	1149	757

Table C.5: IOZone result for FFS Random read

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	942952	1774189	3103735	3029305	3001782	2300703	3937426	3821805	4079544				
2048	1846977	2794560	4240255	4462754	4741090	5719737	6092991	4312636	3169944	3210223			
4096	2116840	1650988	2850103	4099419	5551194	5461198	6158088	5497899	3146031	5572803	3968731		
8192	2050494	2428626	4029614	4686395	5343822	6266816	5708730	6575012	6703285	5516264	5531360	5191229	
16384	1796127	2954176	4046414	4309294	6079322	6357730	6079322	6442366	5702479	5936474	6122653	5148758	5608926

Table C.6: IOZone result for FFS Random write

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	200	274	243	130	273	210	242	220	183				
2048	148	382	404	463	219	489	420	516	418	482			
4096	265	730	757	753	572	612	765	461	592	33381	707		
8192	523	875	439	926	990	970	930	760	930	443	927	905	
16384	1126	1048	1036	734	1129	1136	855	966	976	1006	1187	1115	1201

Table C.7: IOZone result for GCSF Read

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

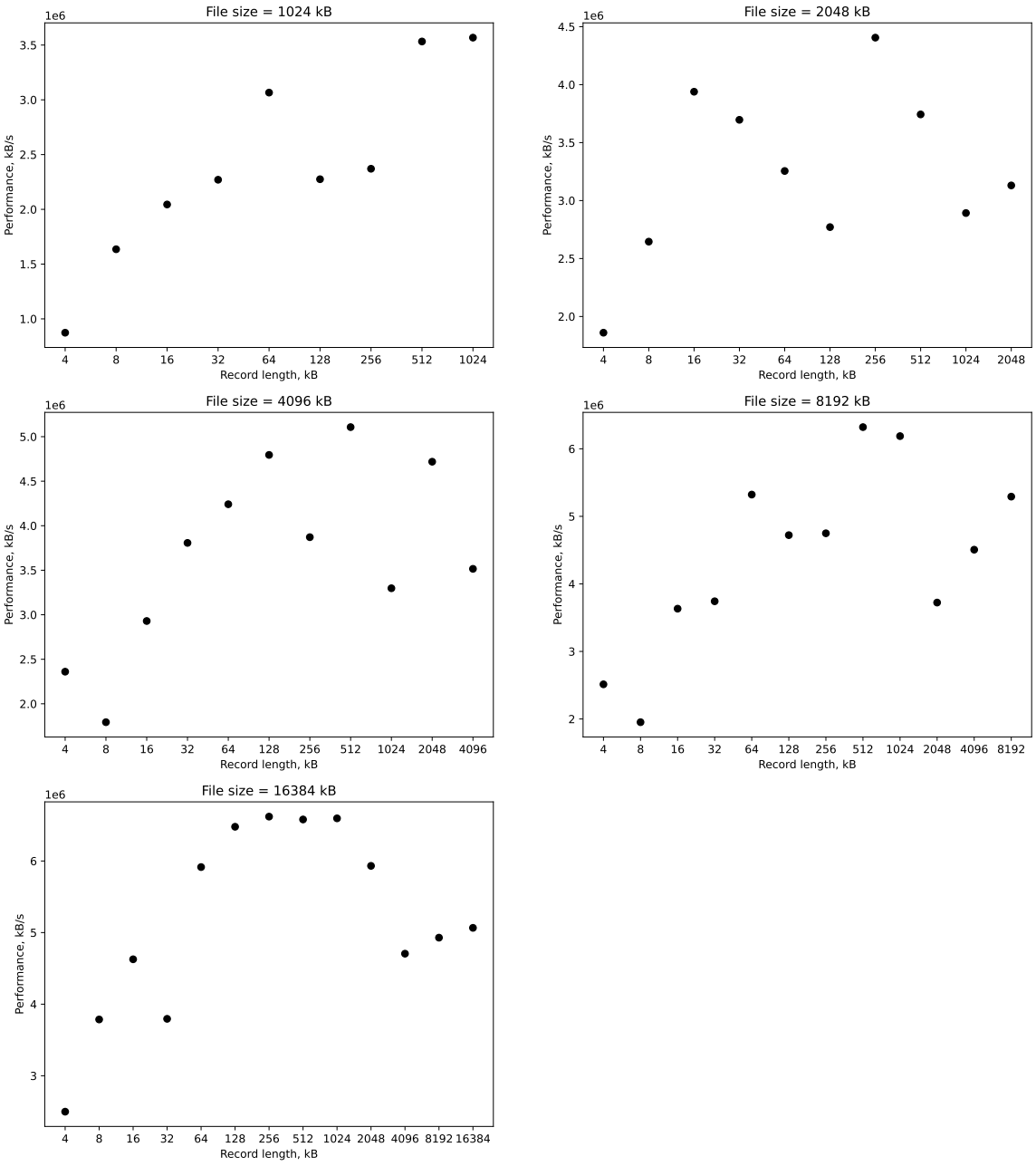

Figure C.1: IOZone output for Forward Read

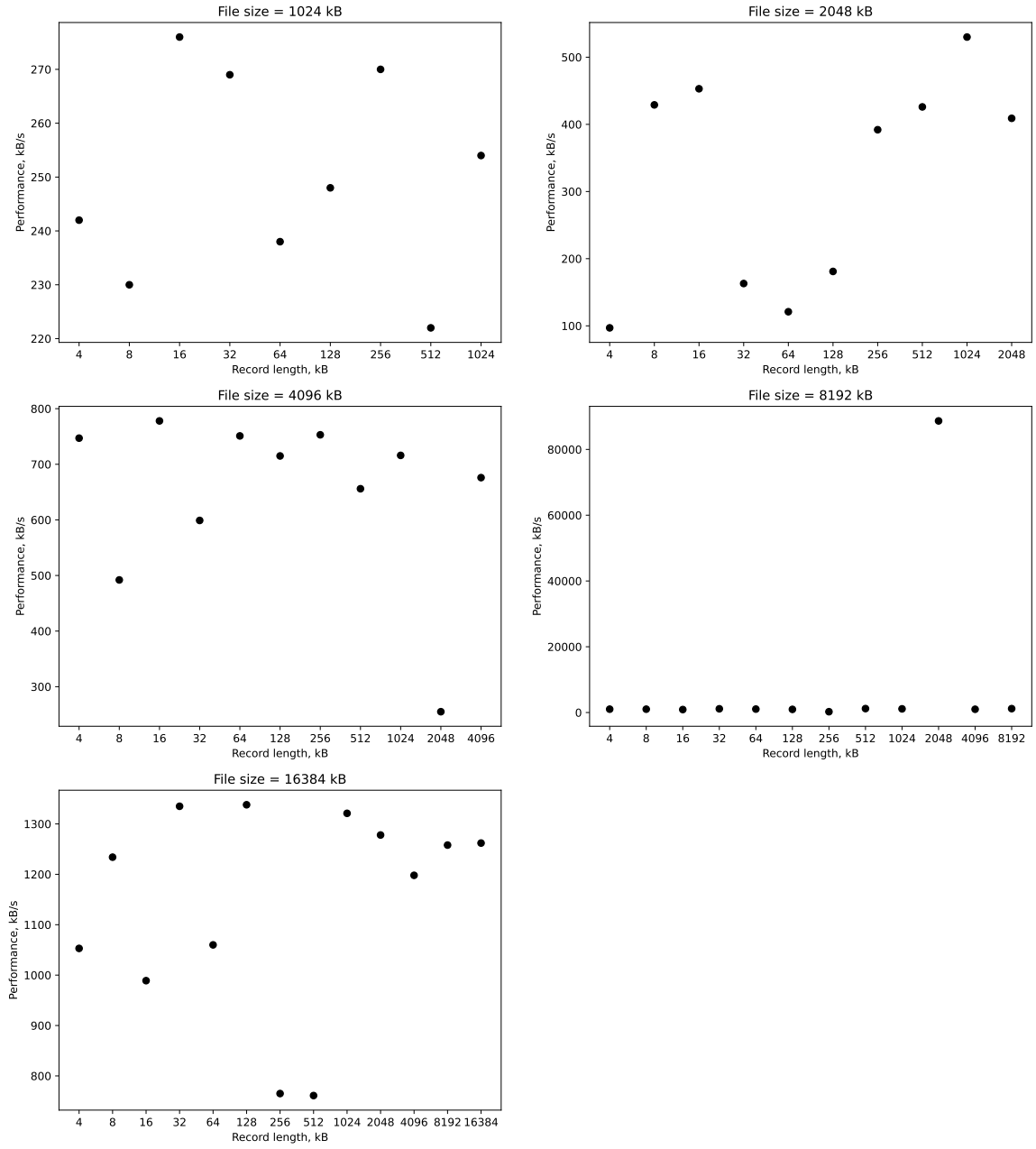

Figure C.2: IOZone output for Forward Write

C.2 GCSF

C.3 Fejk FFS

C.4 APFS

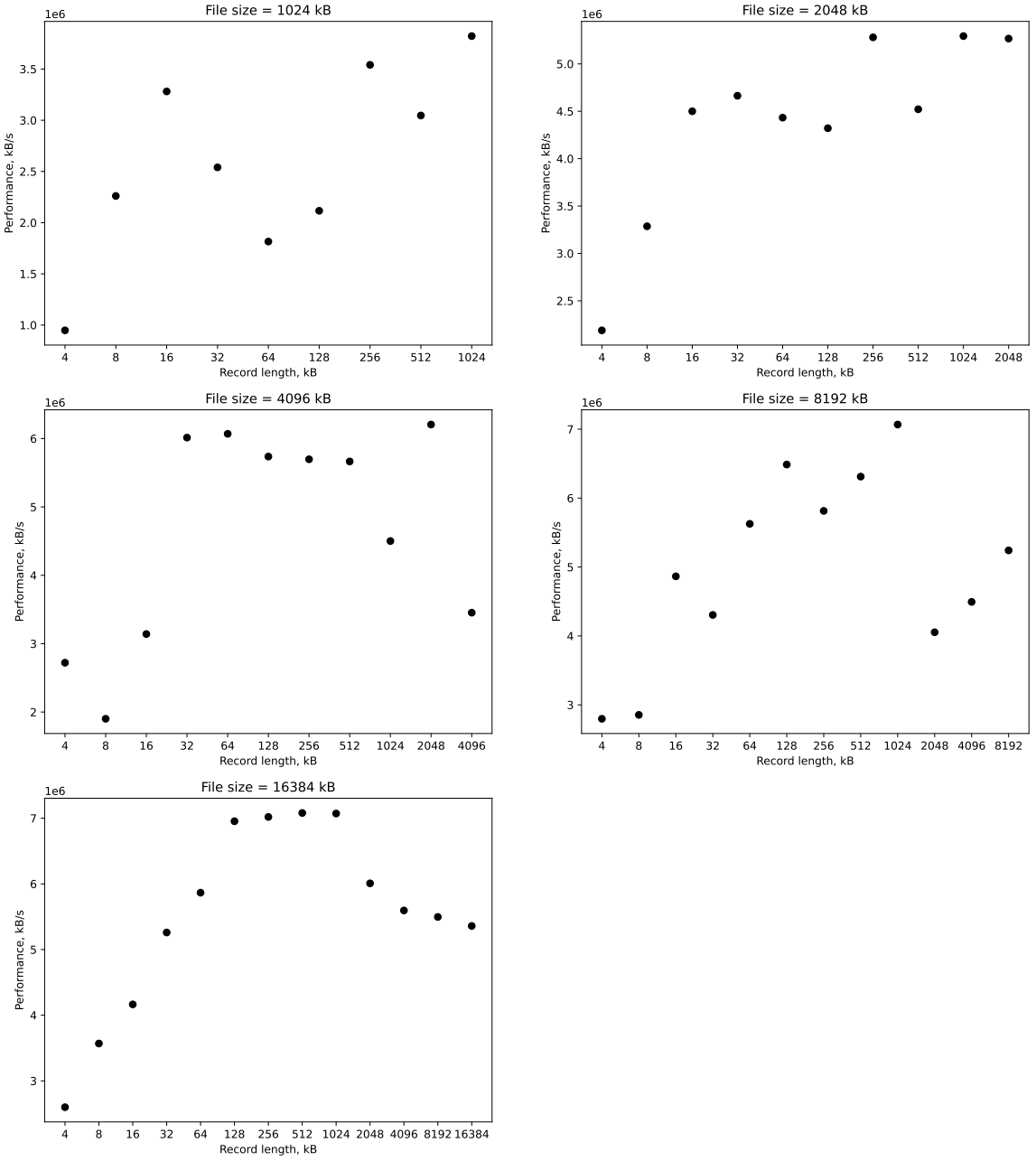

Figure C.3: IOZone output for Re-Read

Table C.8: IOZone result for GCSF Write

	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

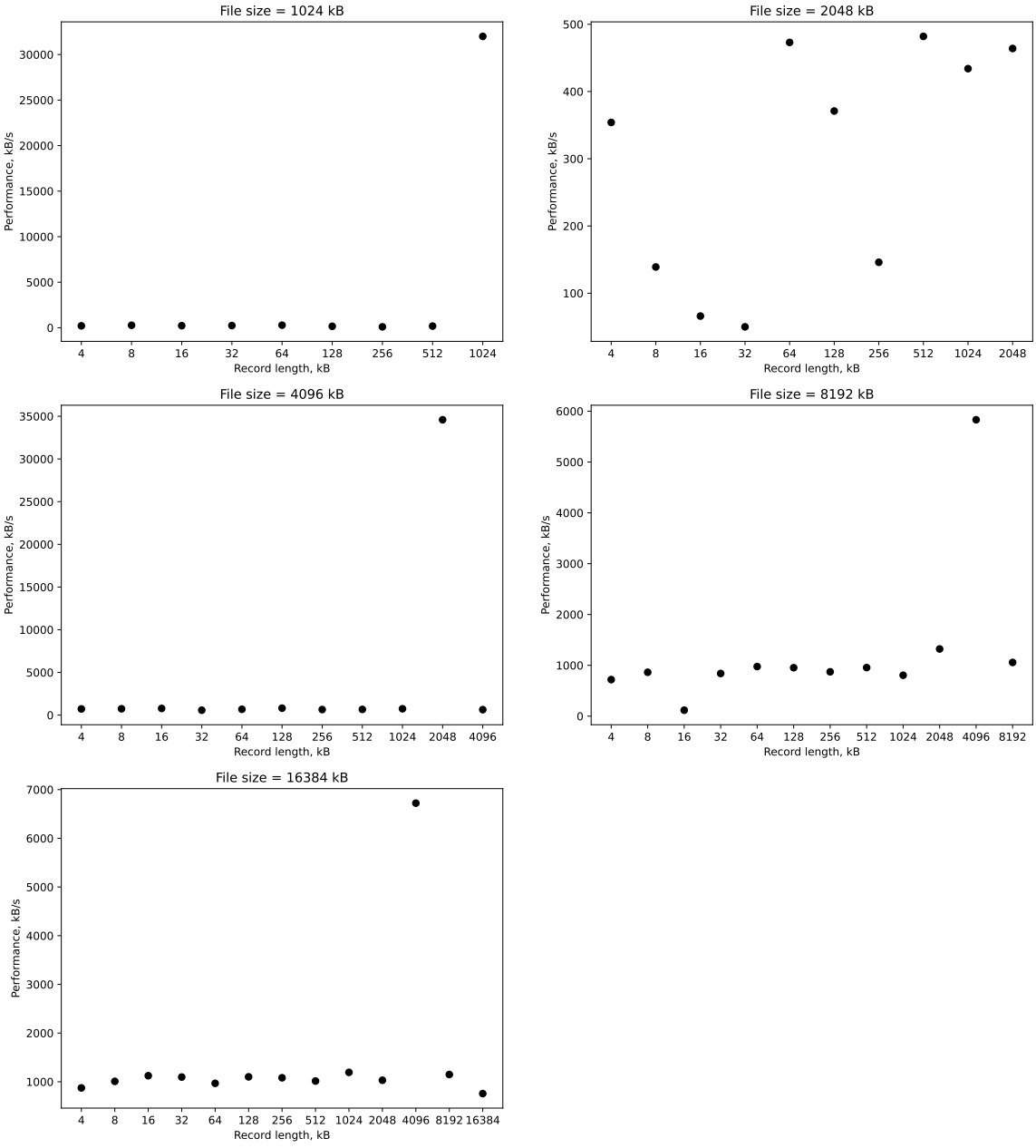

Figure C.4: IOZone output for Re-Write

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

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

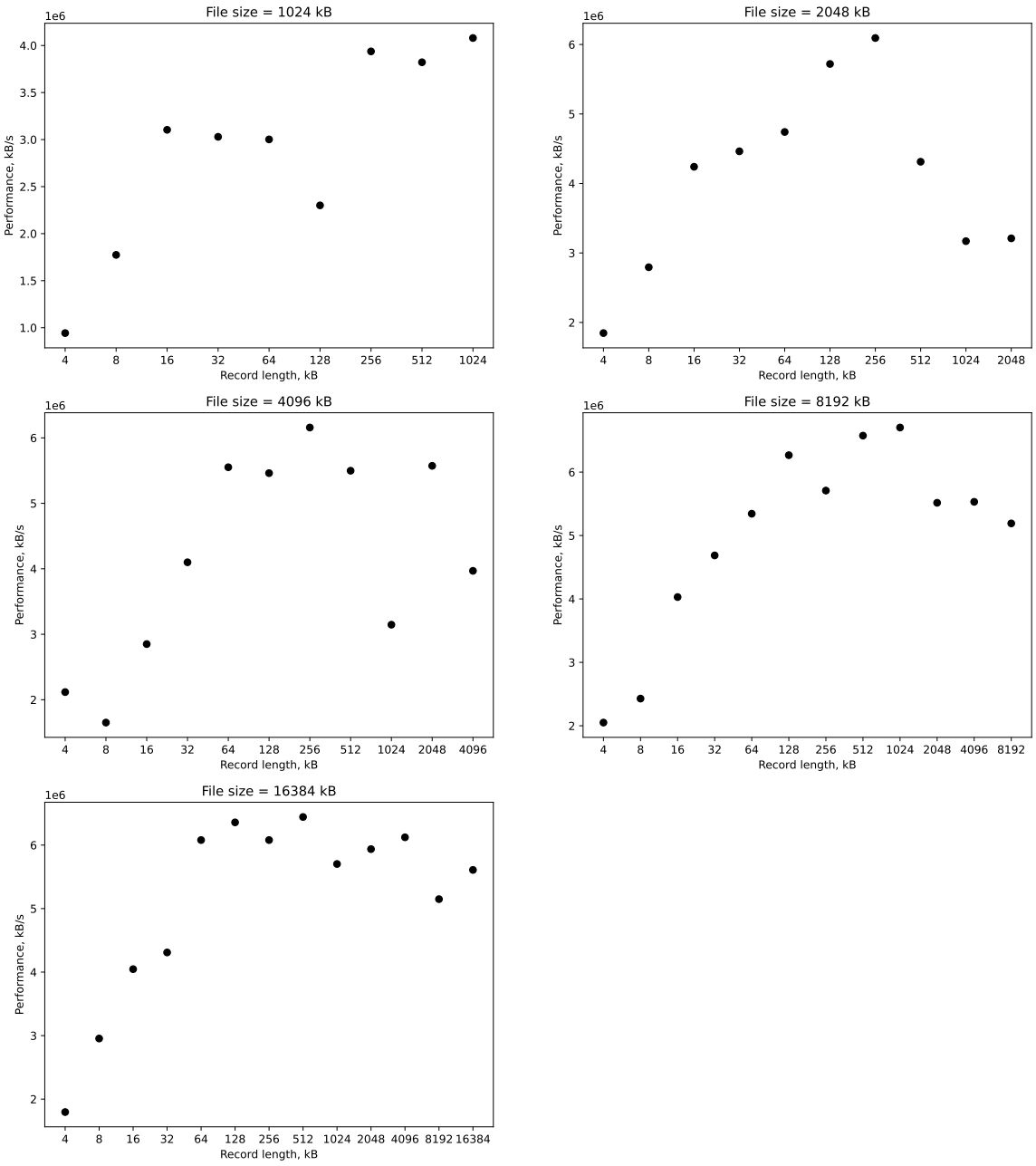

Figure C.5: IOZone output for Random read

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

	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

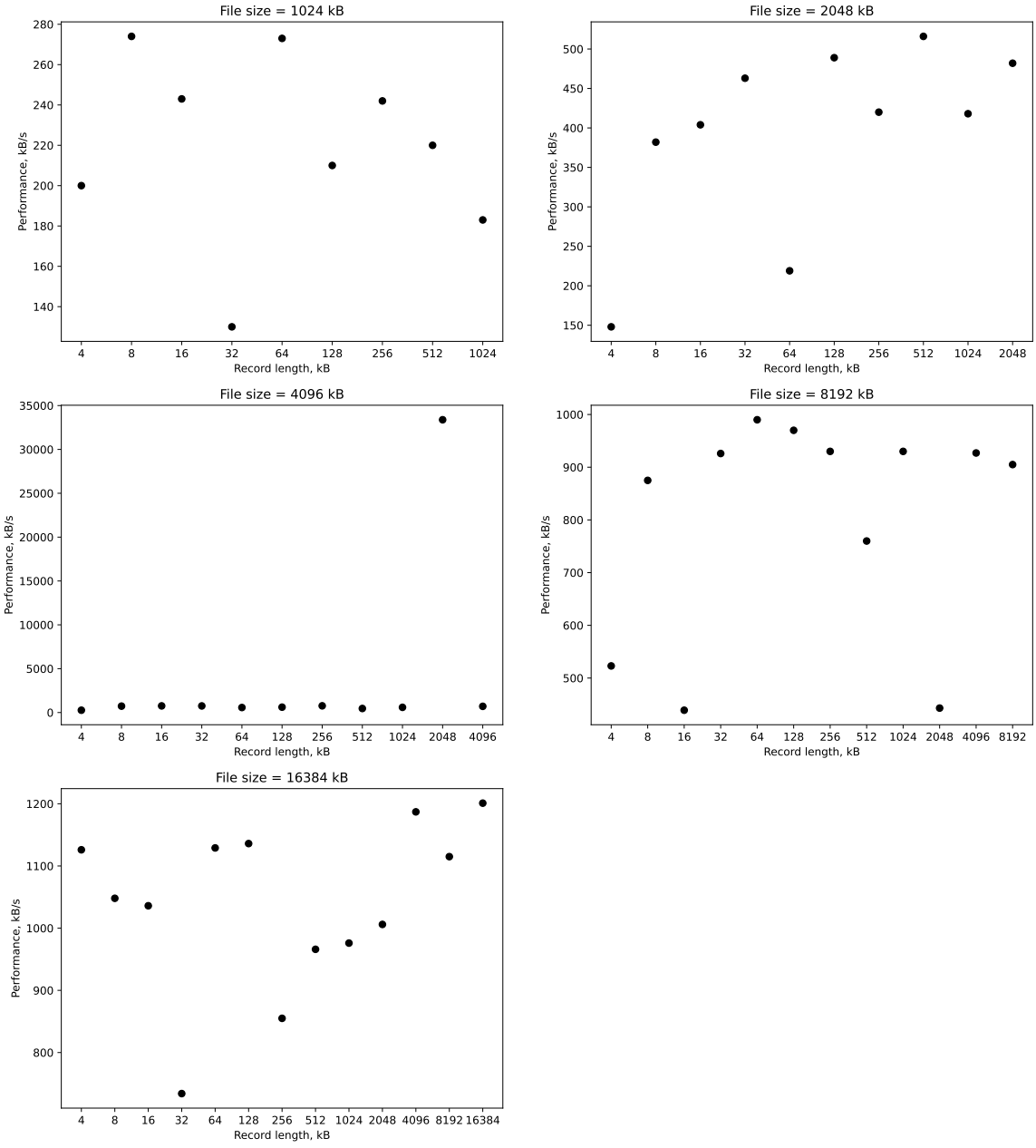

Figure C.6: IOZone output for Random write

Table C.11: IOZone result for GCSF Random read

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

Table C.12: IOZone result for GCSF Random write

File size (kB)	Buffer 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

Table C.13: IOZone result for Fejk FFS Read

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1992279	3314514	4114719	3791442	3241960	3239514	2008114	3160839	3291652				
2048	2243502	2913021	4796687	5280246	3835124	5900467	5238385	4796687	3931669	4697018			
4096	2866270	3096138	4501111	2375927	5841891	6449370	5581856	5810280	7301864	4404184	5382492		
8192	3010894	2750802	4196452	5665431	6616796	6838023	6410631	6211306	6643663	4571055	5765244	5553712	
16384	3230846	3842338	4895314	5683144	6063230	6619859	6407526	7173368	6590654	6511962	6316241	5391113	5329650

Table C.14: IOZone result for Fejk FFS Write

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	6486	6595	6647	6667	6558	6559	85355	7342	6026				
2048	6499	6807	7030	7035	7233	116330	7192	6963	109349	7212			
4096	6614	7140	7266	6590	7044	7187	7103	7250	7226	6897	6983		
8192	4833	5677	6189	7044	6998	7219	7231	7167	110695	113346	7182	7269	
16384	7163	7174	7479	7516	7569	7787	7473	7774	7688	7628	7438	7294	7503

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2197132	4029784	4574926	5652717	5417427	4493556	5120336	3941039	4594502				
2048	3024831	2994254	4911886	5765808	4741090	7059413	5020987	5184636	4886737	6629029			
4096	3122019	3540193	5468151	4376138	7038605	7925004	6659365	6321227	7715028	5490871	5291314		
8192	2886449	2338541	4554091	6249718	5940679	7793434	7236972	8062248	6272536	7281448	5465373	5509188	
16384	3351874	3992114	5232657	6129753	7356911	7861915	7123547	8388806	7180114	6907309	6511962	4982992	5403831

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

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	4860	6161	6096	5938	6321	6364	6297	5894	6383				
2048	5942	6061	6376	6386	6418	6268	6373	34391	37349	6452			
4096	6084	6174	6233	6413	6370	6415	6422	6143	6434	6392	6433		
8192	4744	5055	5045	5912	5832	5625	5764	5798	5917	25475	5799	5912	
16384	6109	6150	5047	6336	6474	6574	6391	6498	6381	5934	6154	6245	6322

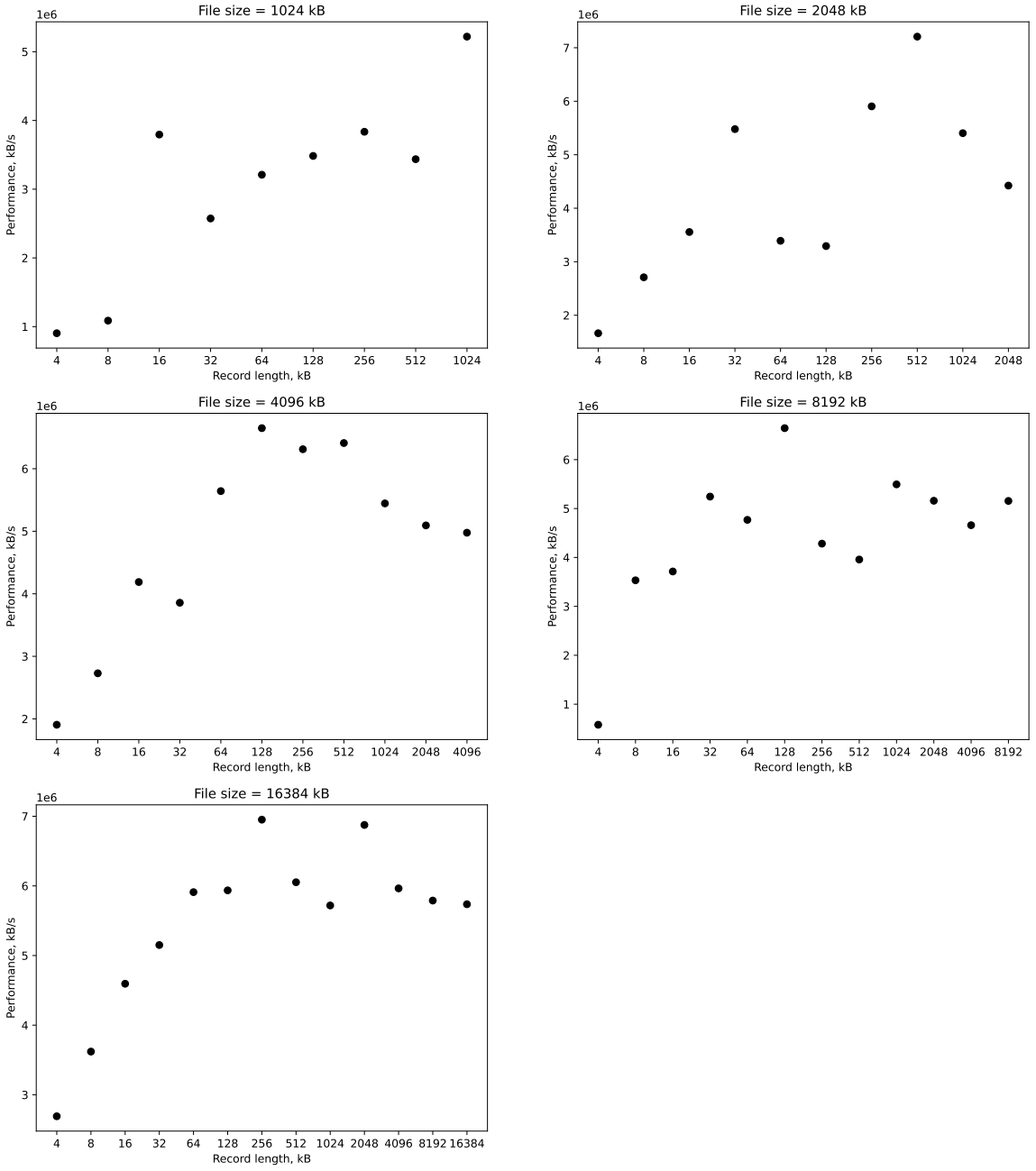

Figure C.7: IOZone output for Forward Read

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1549519	3085895	4805258	5751117	4232305	6569179	5630486	4805258	5564829				
2048	2455806	3020577	5593112	7237860	5251194	8261095	6024617	4829046	2348384	6479029			
4096	2299908	3129412	4853336	3834958	7286380	8625273	7367624	7197849	7225093	5565581	5944991		
8192	2271747	2173593	4003788	5599873	6720329	7340560	7261443	8030217	6191160	5504775	5979968	5872650	
16384	2310146	2878929	4302549	5402132	7053355	6860418	7005182	7533538	6318564	7040348	6878271	5483167	5457909

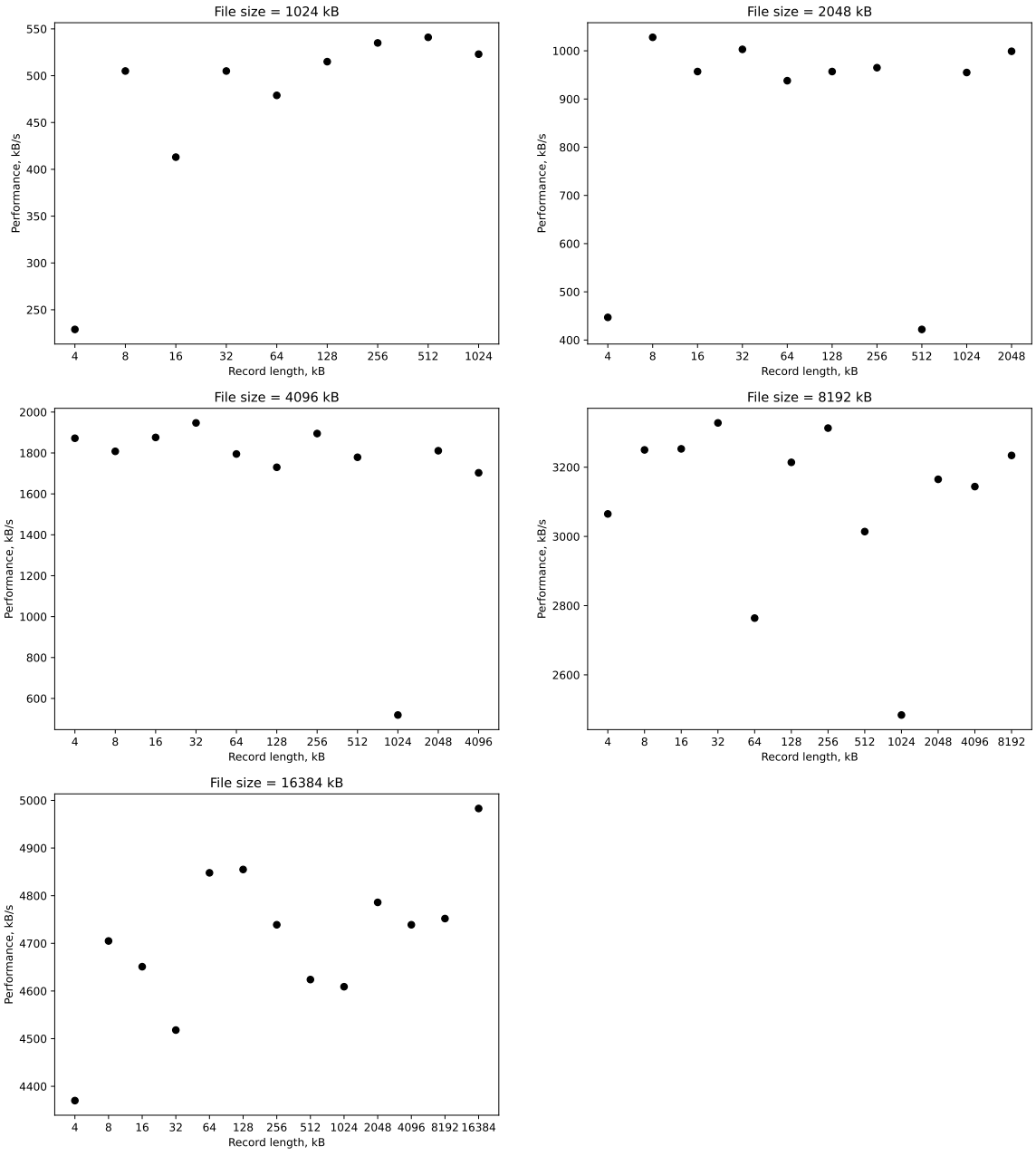

Figure C.8: IOZone output for Forward Write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	5864	6137	6251	6146	6362	6252	5775	6457	6352				
2048	5869	6190	6252	6124	6398	6533	6415	6331	6587	6546			
4096	5871	6067	6273	6436	6521	6467	6325	6507	6495	6433	6474		
8192	4923	4113	4917	5871	5843	5937	5954	5965	5961	5907	5954	6169	
16384	6151	6081	5323	6342	6399	6219	6462	6278	6355	5947	5092	6248	6404

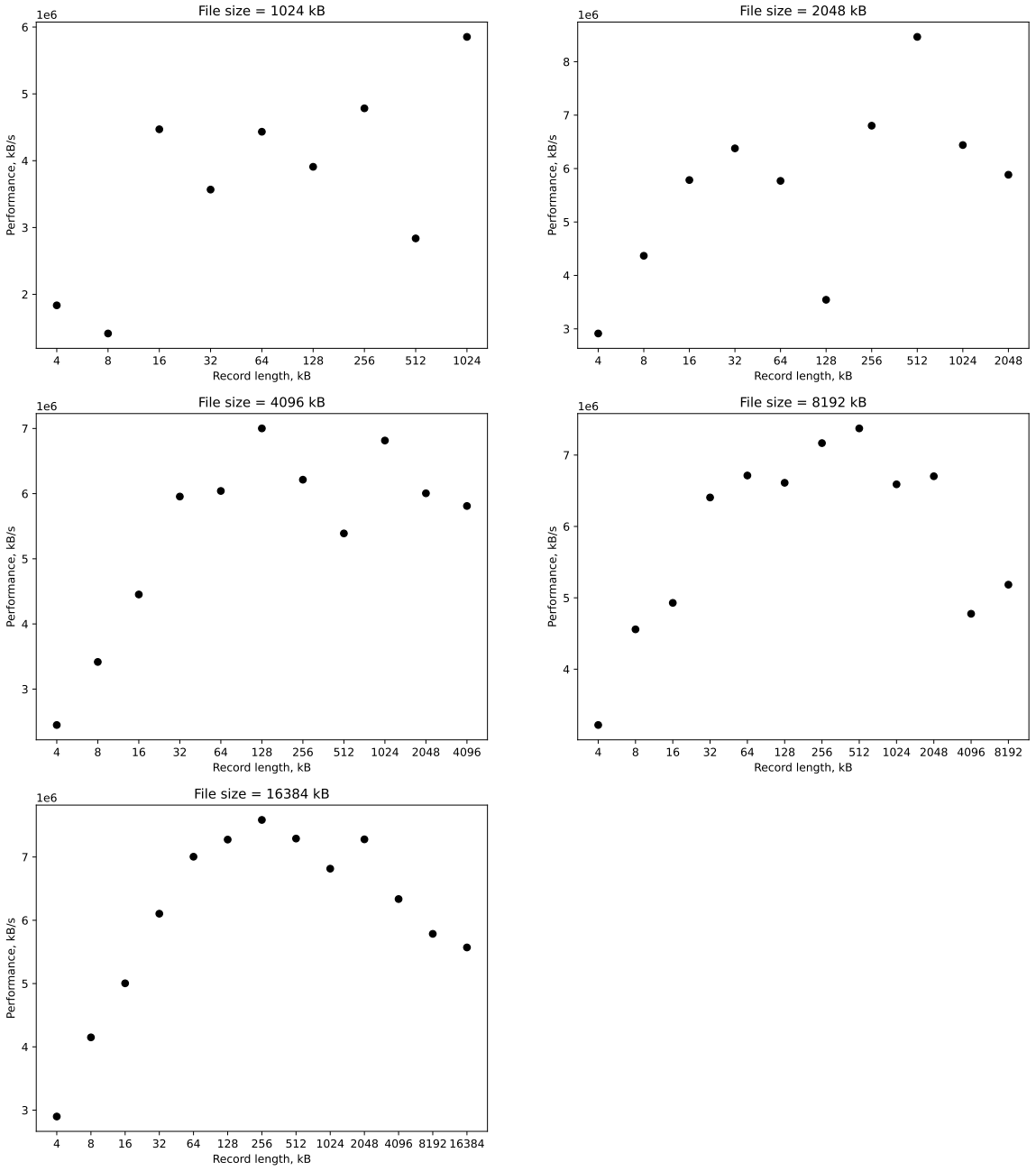

Figure C.9: IOZone output for Re-Read

Table C.19: IOZone result for APFS Read

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2311849	3821805	5065980	7369466	8542002	4876180	7869040	4114719	11249091				
2048	2354177	3430751	4994712	5769681	8398403	9796850	7065219	6939647	7789164	8221561			
4096	2402508	3491981	4876757	6022095	5475122	5953231	7301864	7774379	6989921	5411313	6094733		
8192	2474985	3314080	5653313	5588034	5748846	6720329	6045199	5851647	5966469	7891886	5716328	5645881	
16384	2773652	531358	2567147	4954252	6065906	6208391	5221128	7634813	5999183	6731392	5716711	5603438	5559465

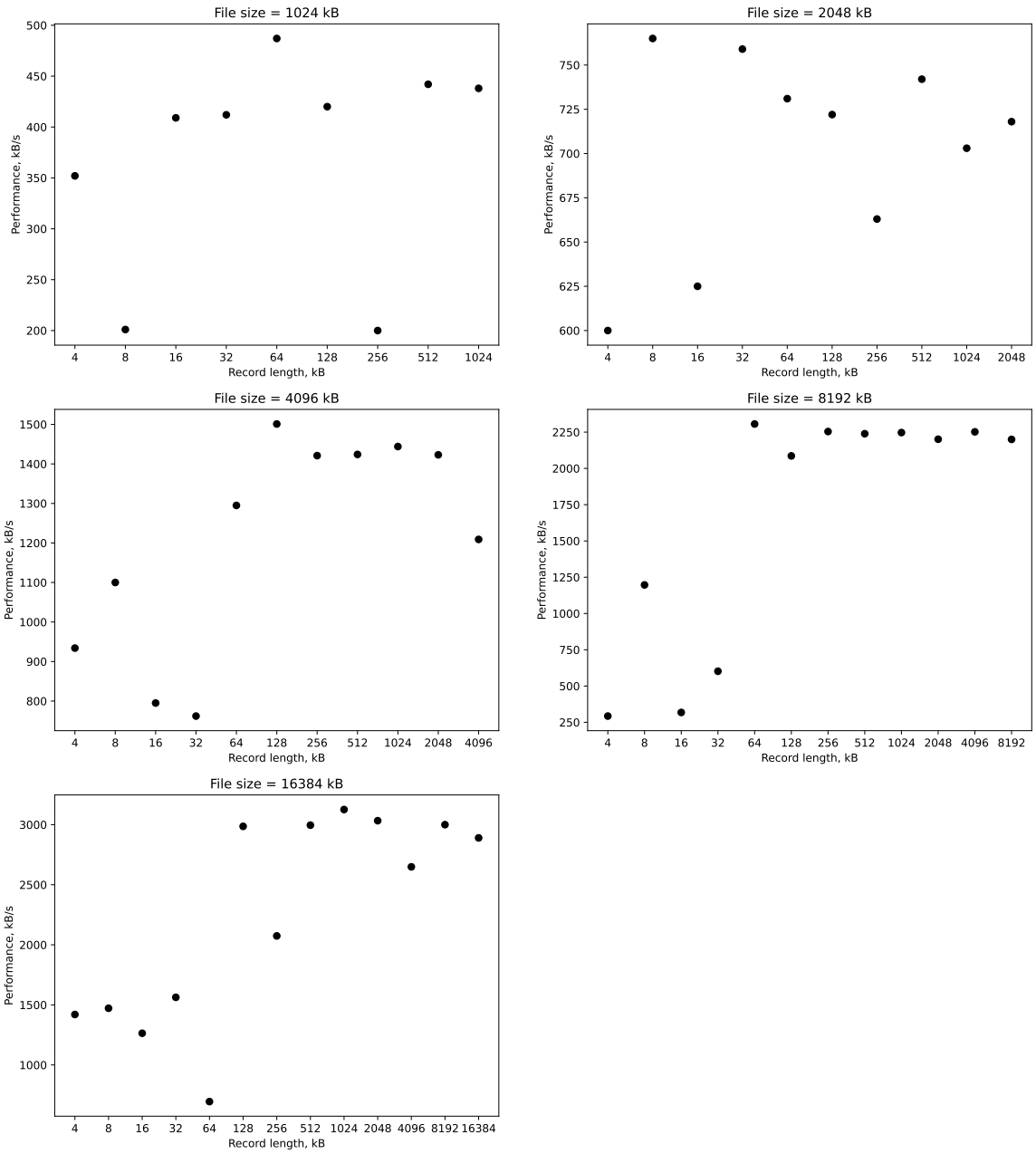

Figure C.10: IOZone output for Re-Write

Table C.20: IOZone result for APFS Write

File size (kB)	Buffer size (kB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	157562	280097	419520	220777	271116	1091532	854752	1687075	305782				
2048	180154	251044	414476	397980	425248	664300	379685	1344739	1421279	2102899			
4096	186197	340567	351651	934714	486051	1136174	771788	1353618	784547	735918	1231902		
8192	200062	251335	403762	604799	537746	1224736	745948	1595914	1381695	1063479	1439766	445727	
16384	207663	216404	111071	574597	729051	1050647	941932	630420	1114024	1460506	1495696	1490829	1539559

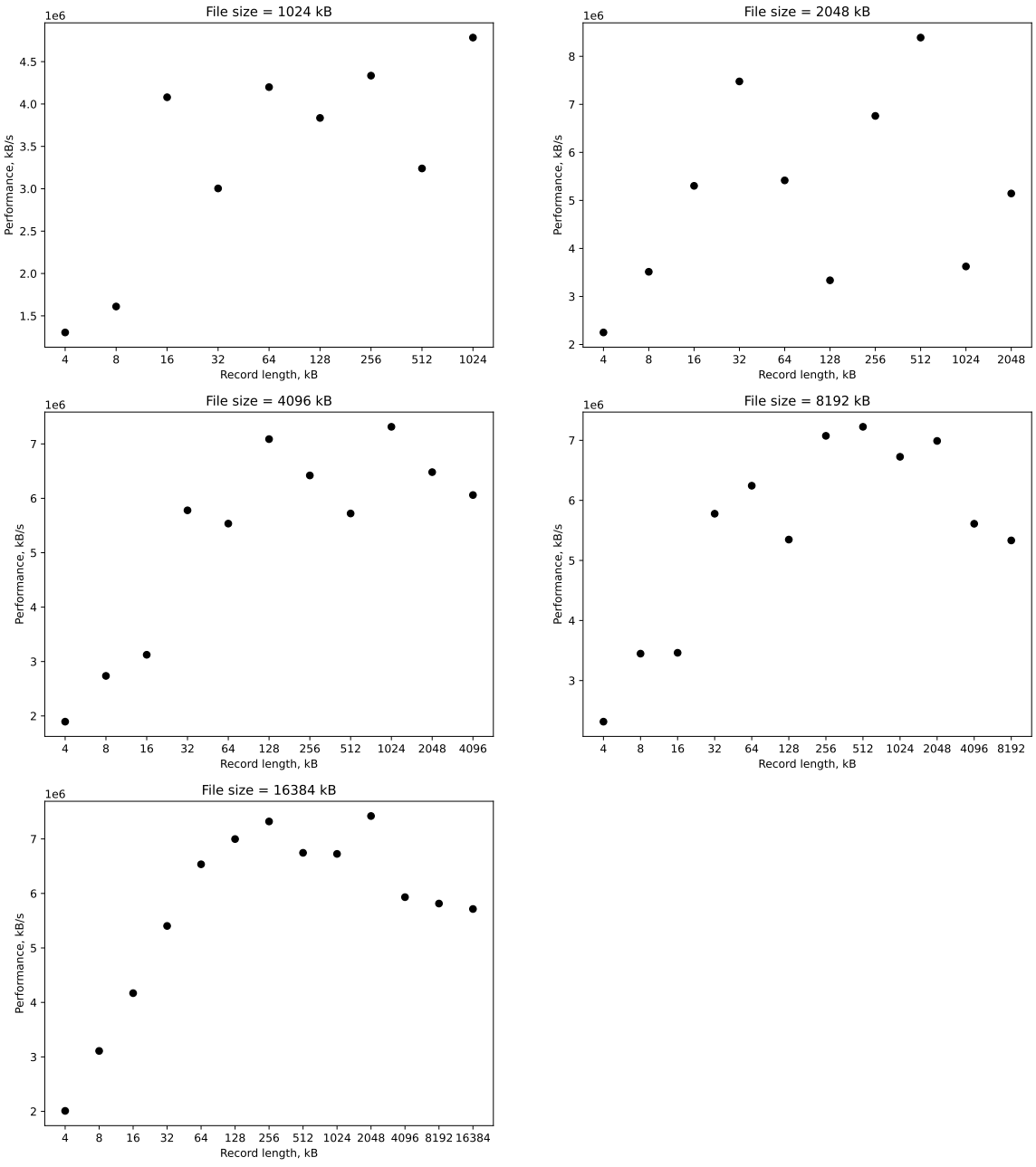

Figure C.11: IOZone output for Random read

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

File size (kB)	Buffer size (kB)													
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	2504637	4029784	6093831	8122014	9677584	6327241	10557785	7577494	15743686				
2048	2519185	3744839	5640860	6717153	10555264	11380325	8029434	7999524	9149853	10889797			
4096	2483786	3503374	5360658	5800471	6272759	6905631	8409937	8865630	6638778	5666546	7135072		
8192	2494390	3771731	5177930	6660405	7172016	7461712	6621897	6267959	6287458	8444674	3995873	5569014	
16384	2780610	1256141	606254	4280039	5968441	6689455	3986324	8159708	6191610	6389653	5640230	5170063	5550036

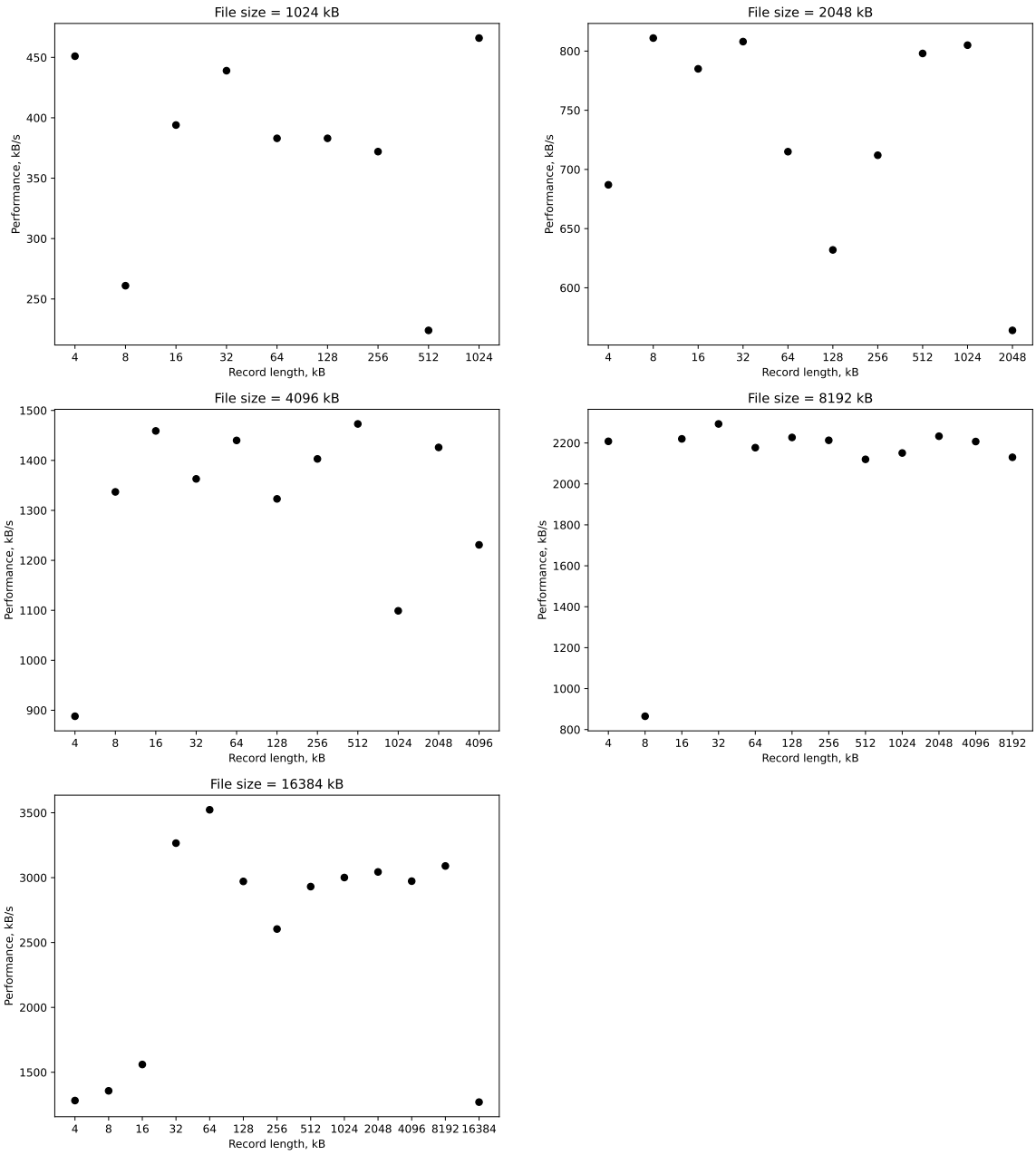

Figure C.12: IOZone output for Random write

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

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	642400	951728	1408275	1909888	2751762	3791442	2179295	3048658	3643490				
2048	608446	945527	1662201	1959469	2066474	3372826	3828288	2598439	2888532	2367804			
4096	677926	1044889	1340731	2741845	1937548	2393804	3516281	2509913	2592481	2261757	2666519		
8192	598353	862858	1344287	1329412	1790987	1617704	1997565	1398624	1787539	2196801	1596582	2065036	
16384	652360	347573	666533	1155516	1502005	1012176	1213357	1719758	1153731	1169438	1312296	1423211	1557847

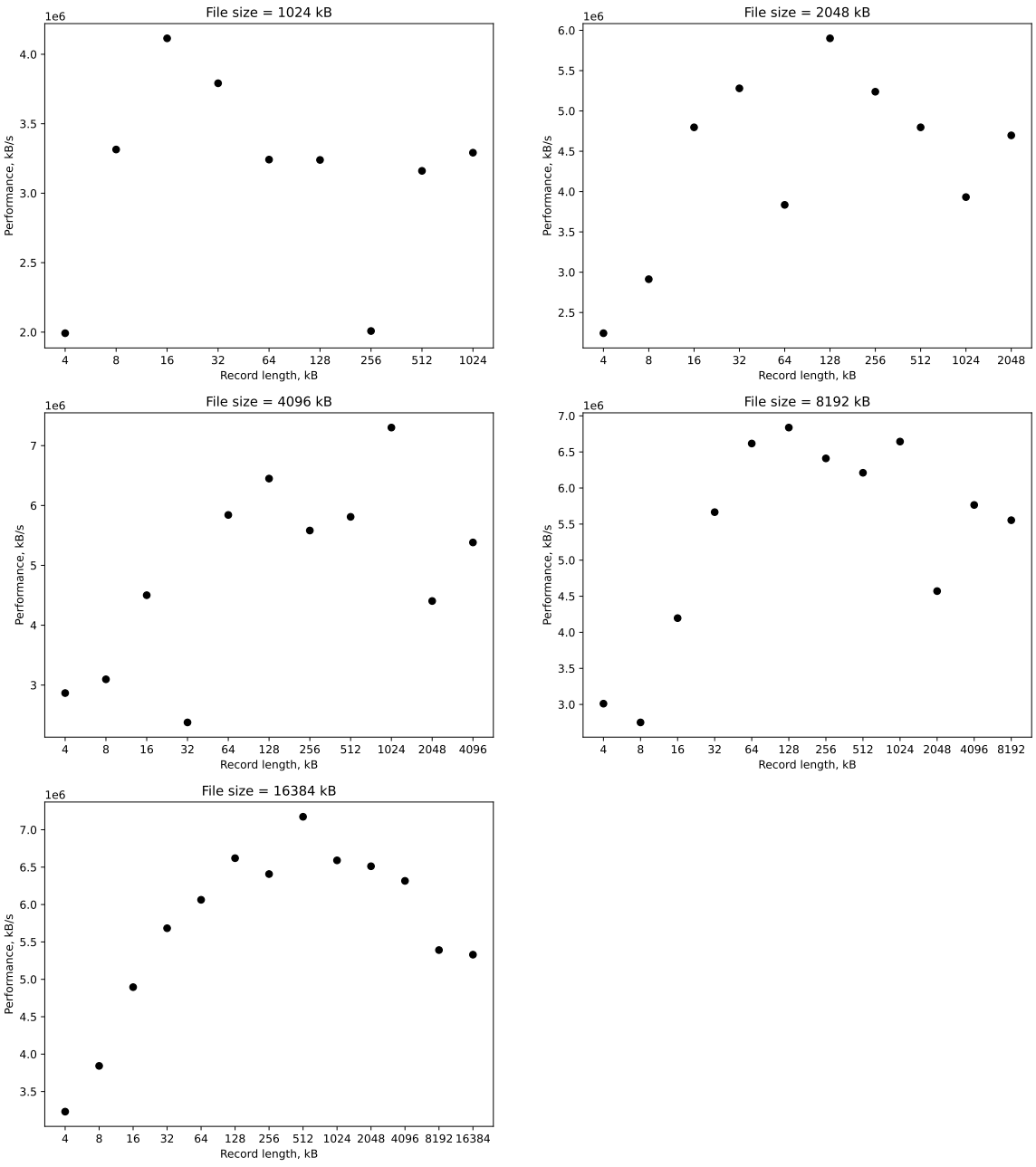

Figure C.13: IOZone output for Fejk Forward Read

Table C.23: IOZone result for APFS Random read

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	1305935	2900425	4783849	6489770	8262639	6355328	7269678	8122014	14044758				
2048	1742802	2771120	5626082	5750369	10179991	10848538	9664580	8643474	8940345	9752360			
4096	1772435	2610602	4591331	5409609	6361016	7515892	6512939	9801354	6370451	5885924	7314299		
8192	1733431	2736125	4855947	5281396	6491781	7243074	5911041	6866721	6748045	8325988	3730372	5561803	
16384	1941677	2037532	1415967	3636950	5324694	6395004	5206097	8135557	6054683	5804105	5681264	5243037	5045925

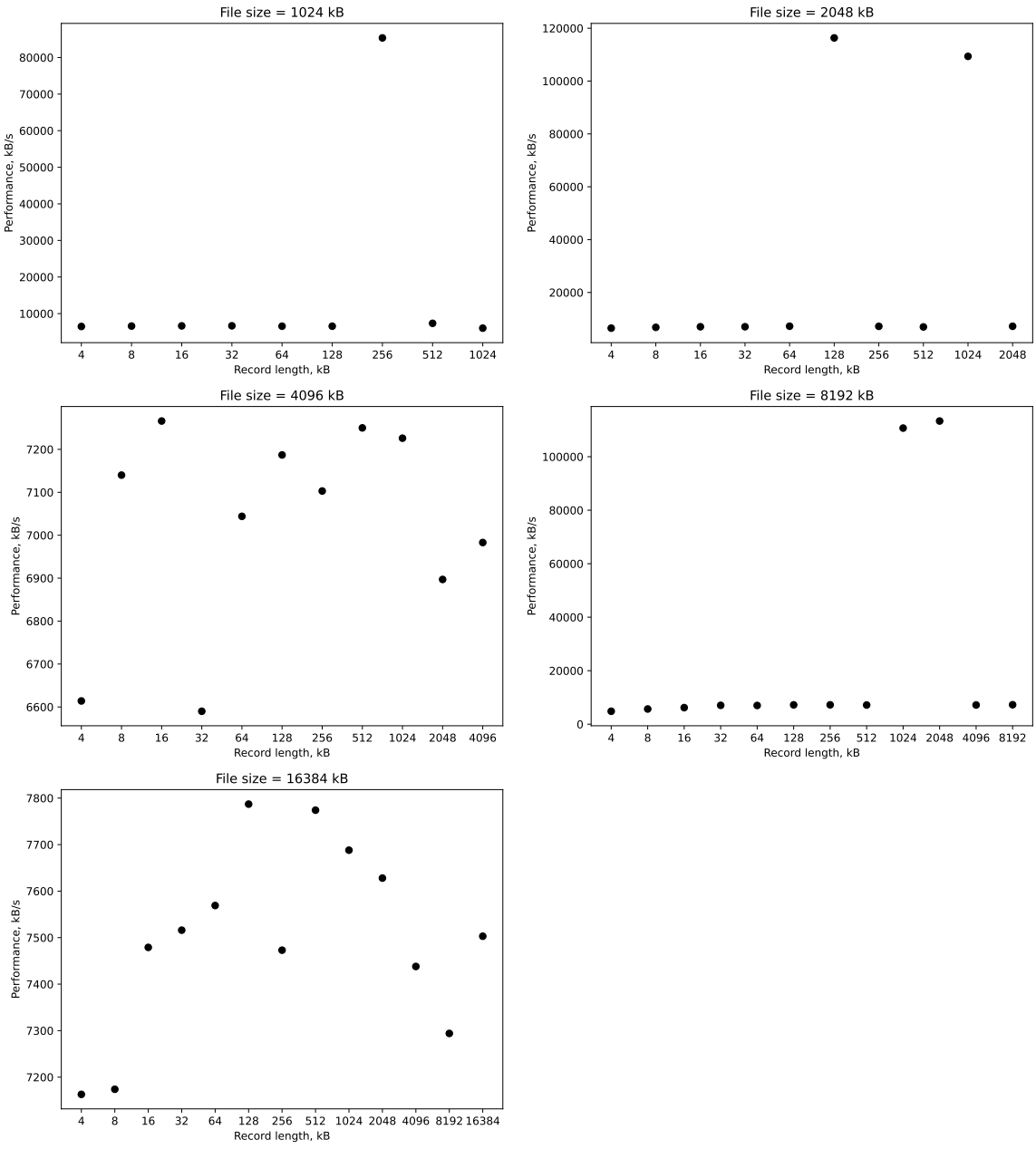

Figure C.14: IOZone output for Fejk Forward Write

Table C.24: IOZone result for APFS Random write

	Buffer size (kB)												
File size (kB)	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
1024	495165	889815	1211964	2348509	3335105	1935711	1968537	3281592	5120336				
2048	556530	1037456	2048243	1863404	3029098	4730646	4257067	3606479	4088876	4441984			
4096	585548	894700	1375841	2080188	2254337	2908979	2860543	3373360	3669496	2907010	2901120		
8192	589696	925644	1667397	1933928	2231469	2482495	2348612	2431548	2029780	2487707	1891555	2121651	
16384	577247	401085	669005	1328919	1809178	1825082	1656297	1702755	1716836	1669131	1971592	1699428	1775476

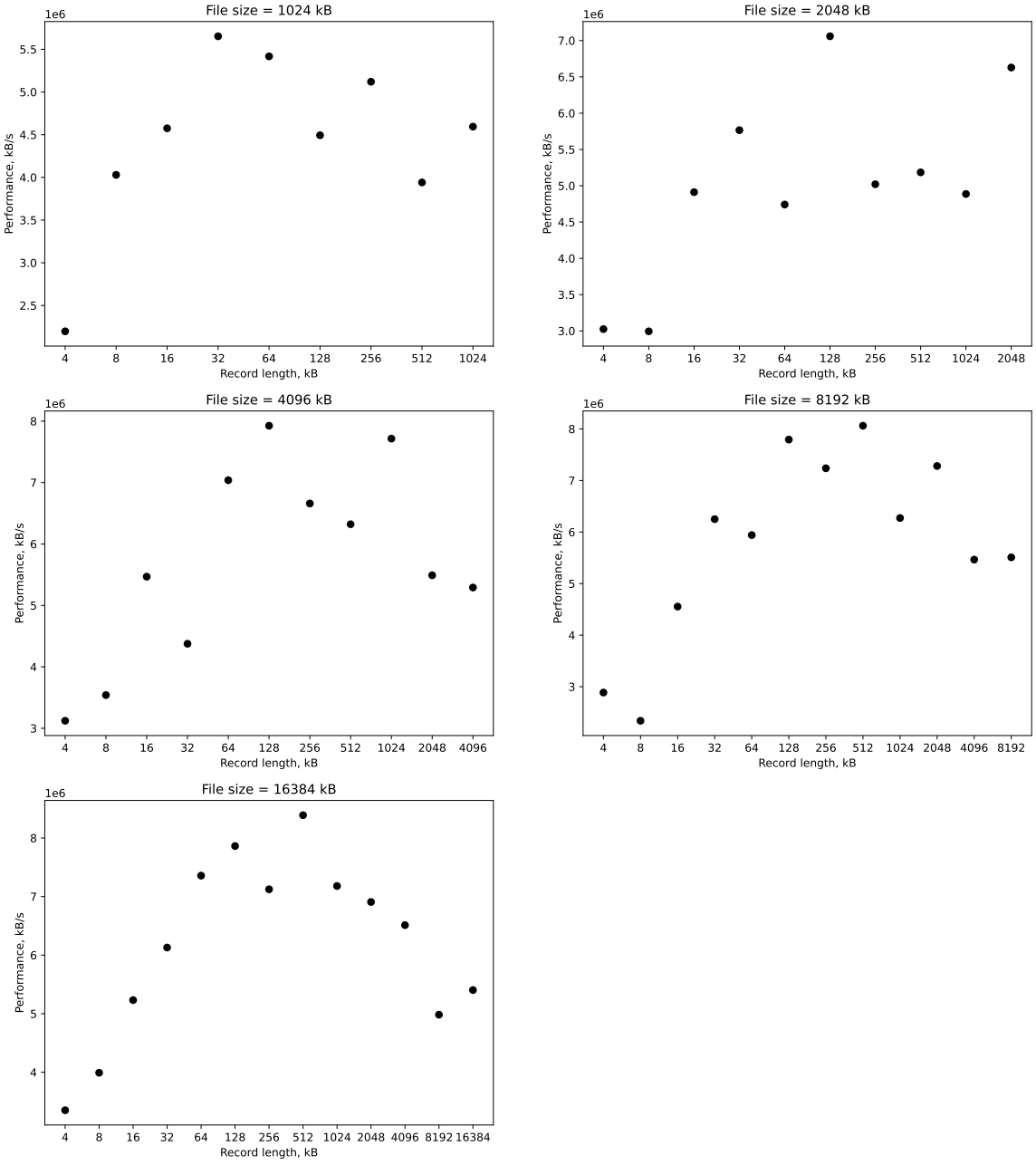

Figure C.15: IOZone output for Fejk Re-Read

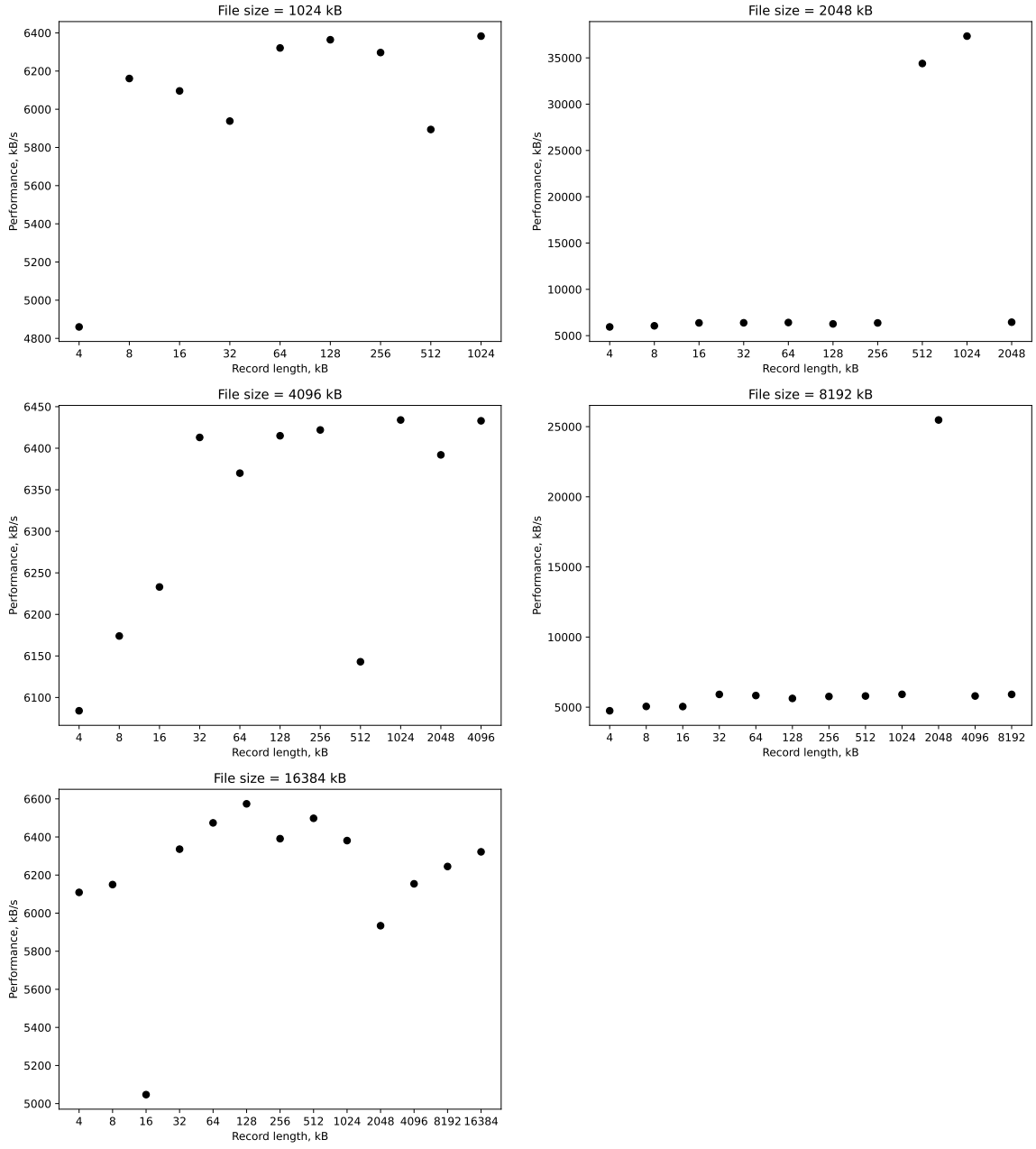

Figure C.16: IOZone output for Fejk Re-Write

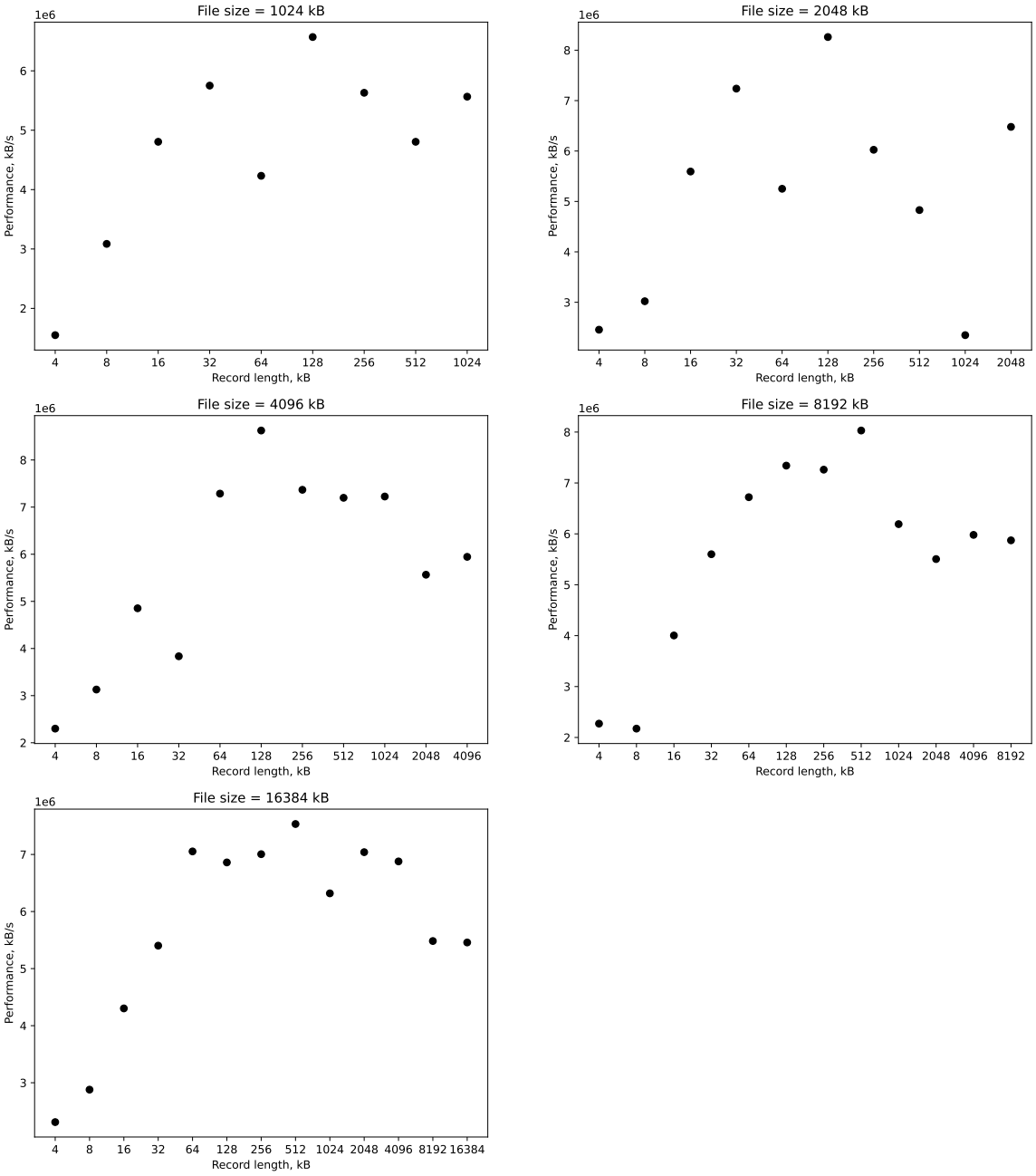

Figure C.17: IOZone output for Fejk Random read

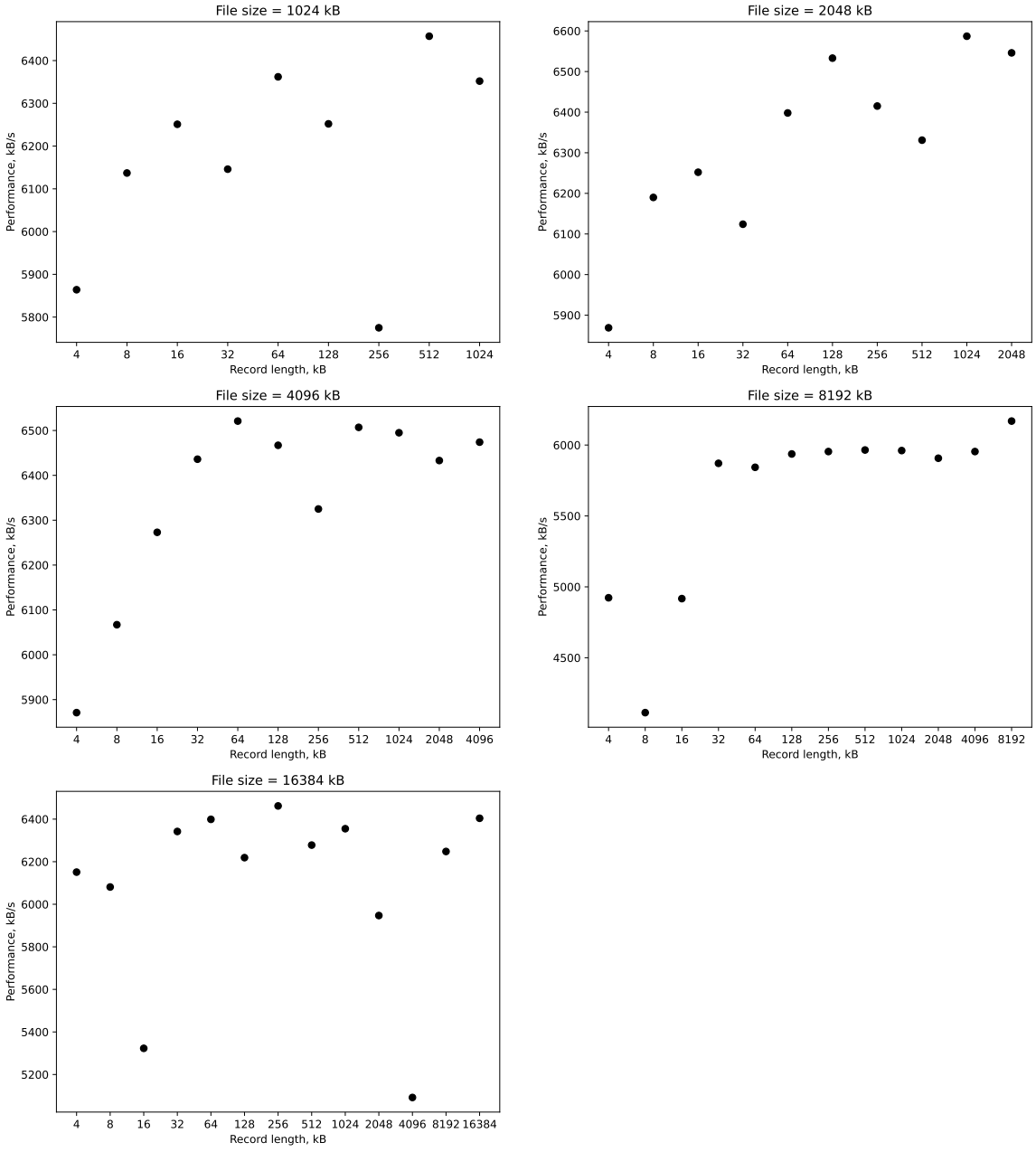

Figure C.18: IOZone output for Fejk Random write

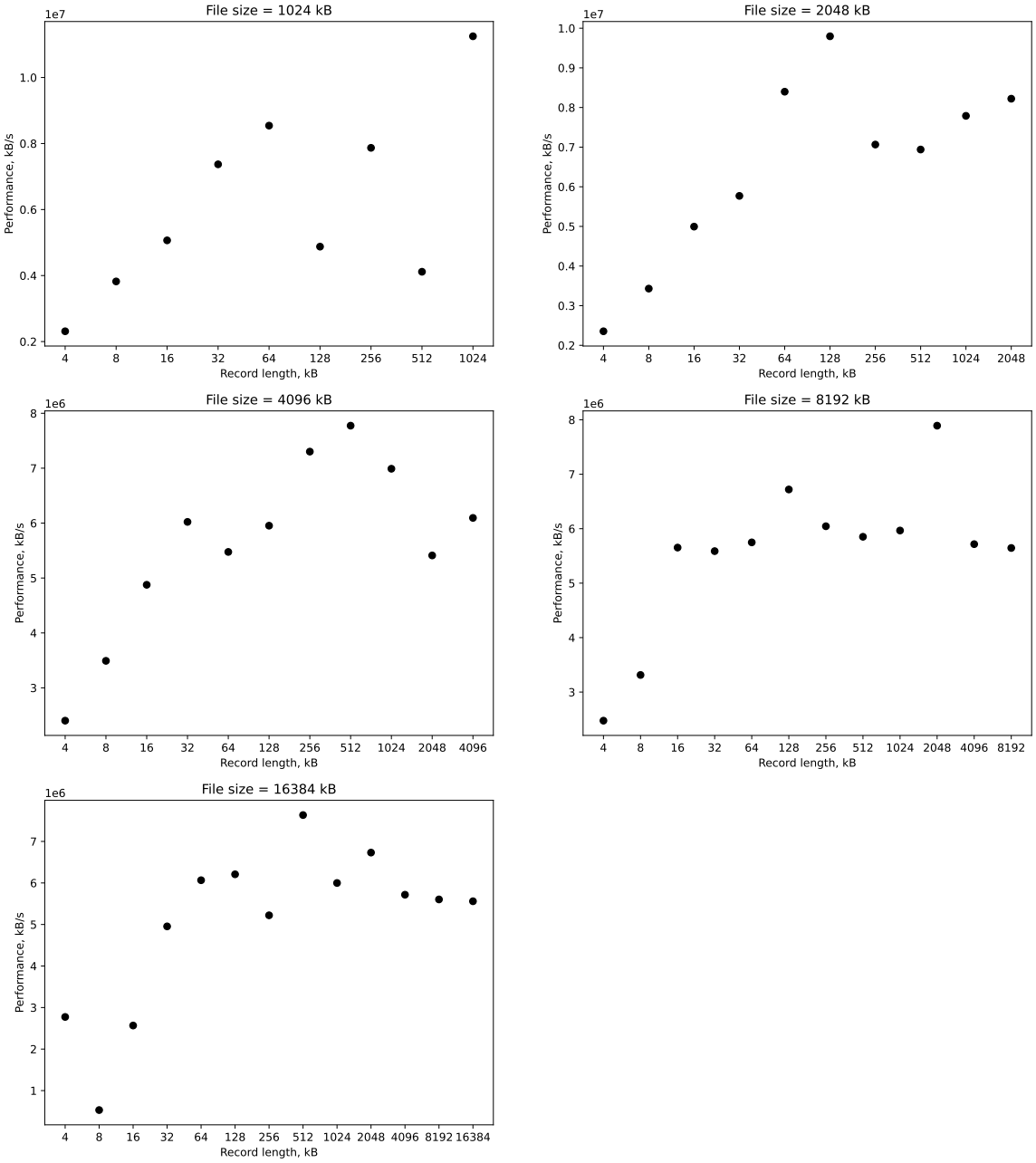

Figure C.19: IOZone output for Forward Read

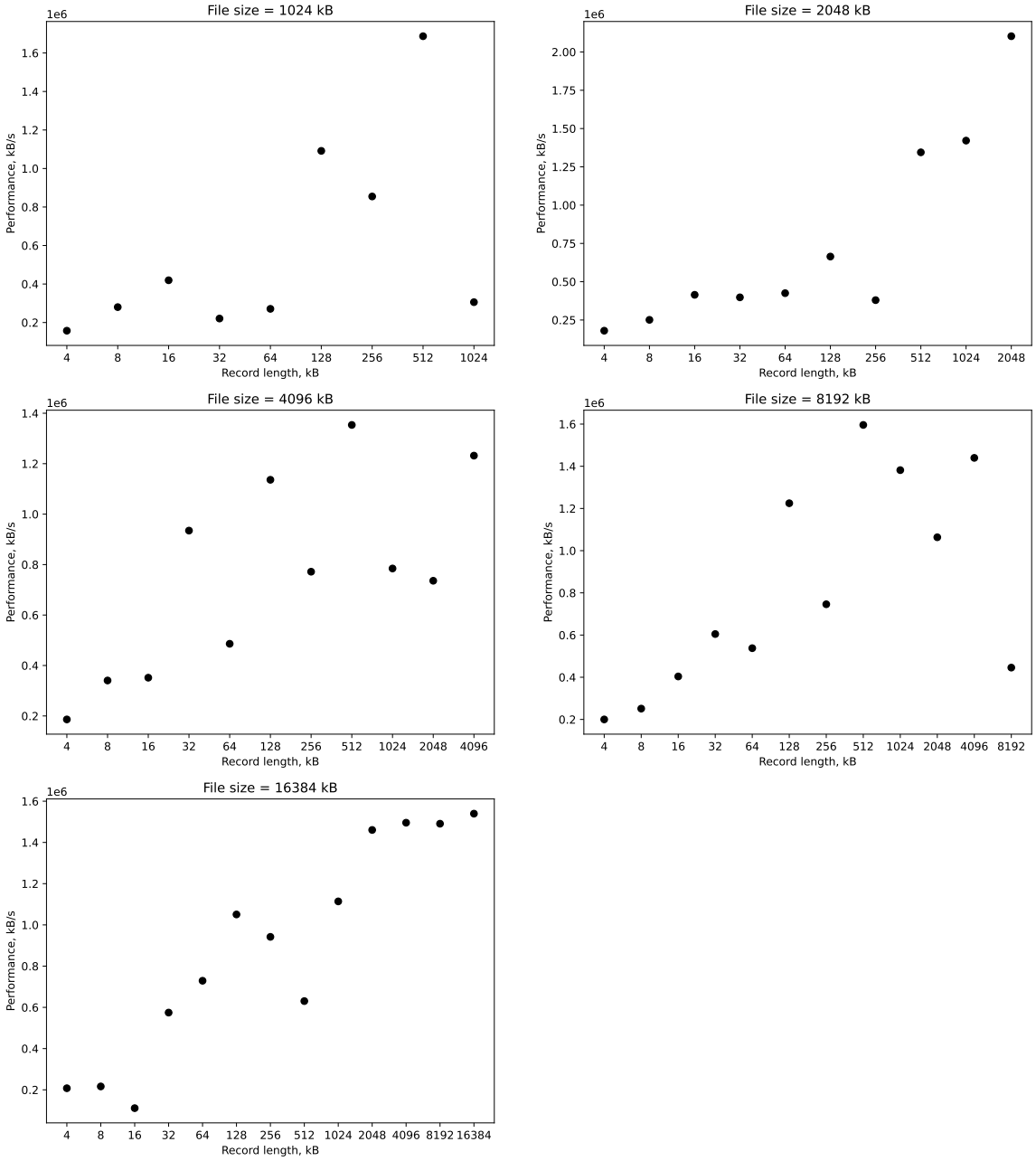

Figure C.20: IOZone output for Forward Write

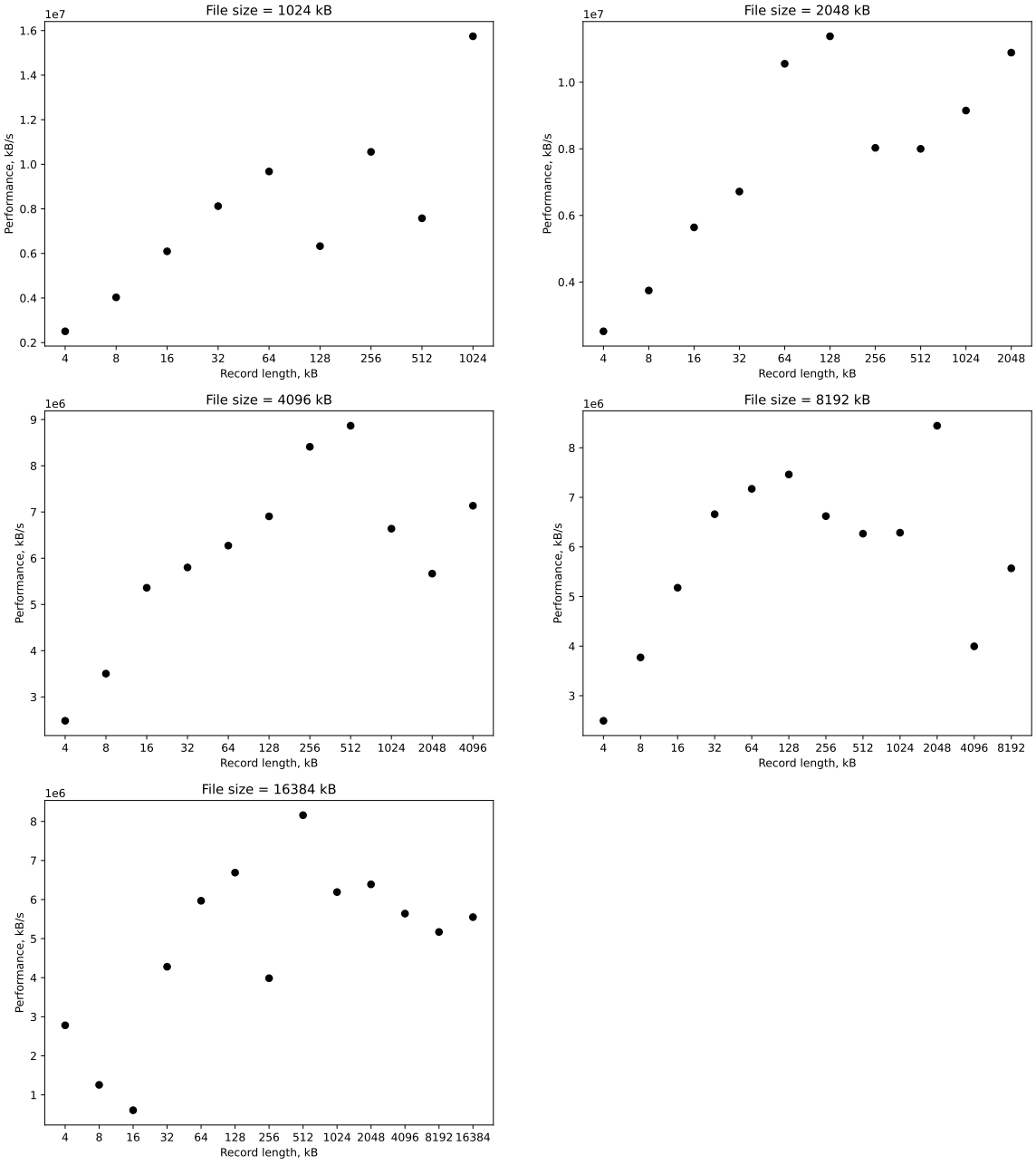

Figure C.21: IOZone output for Re-Read

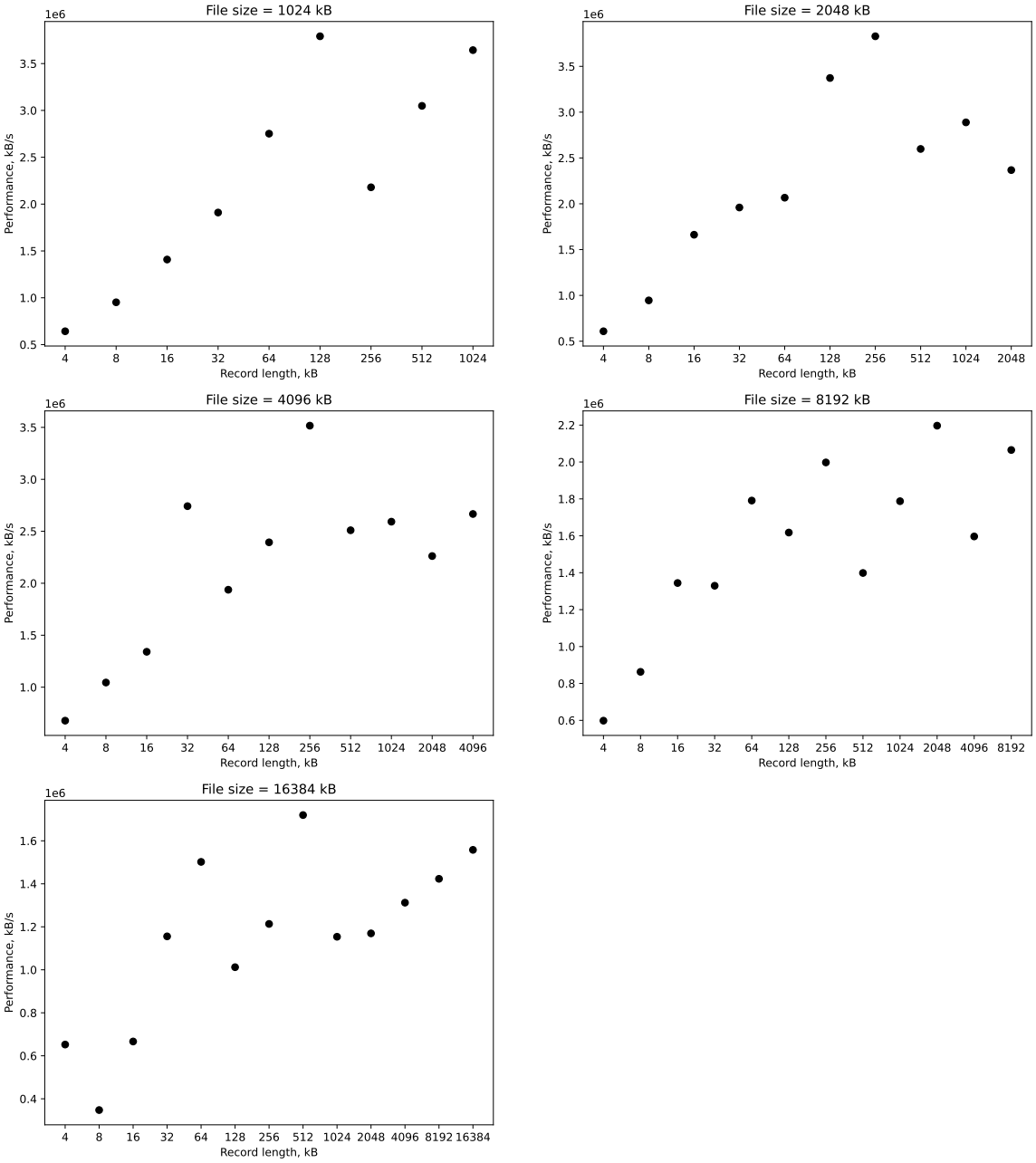

Figure C.22: IOZone output for Re-Write

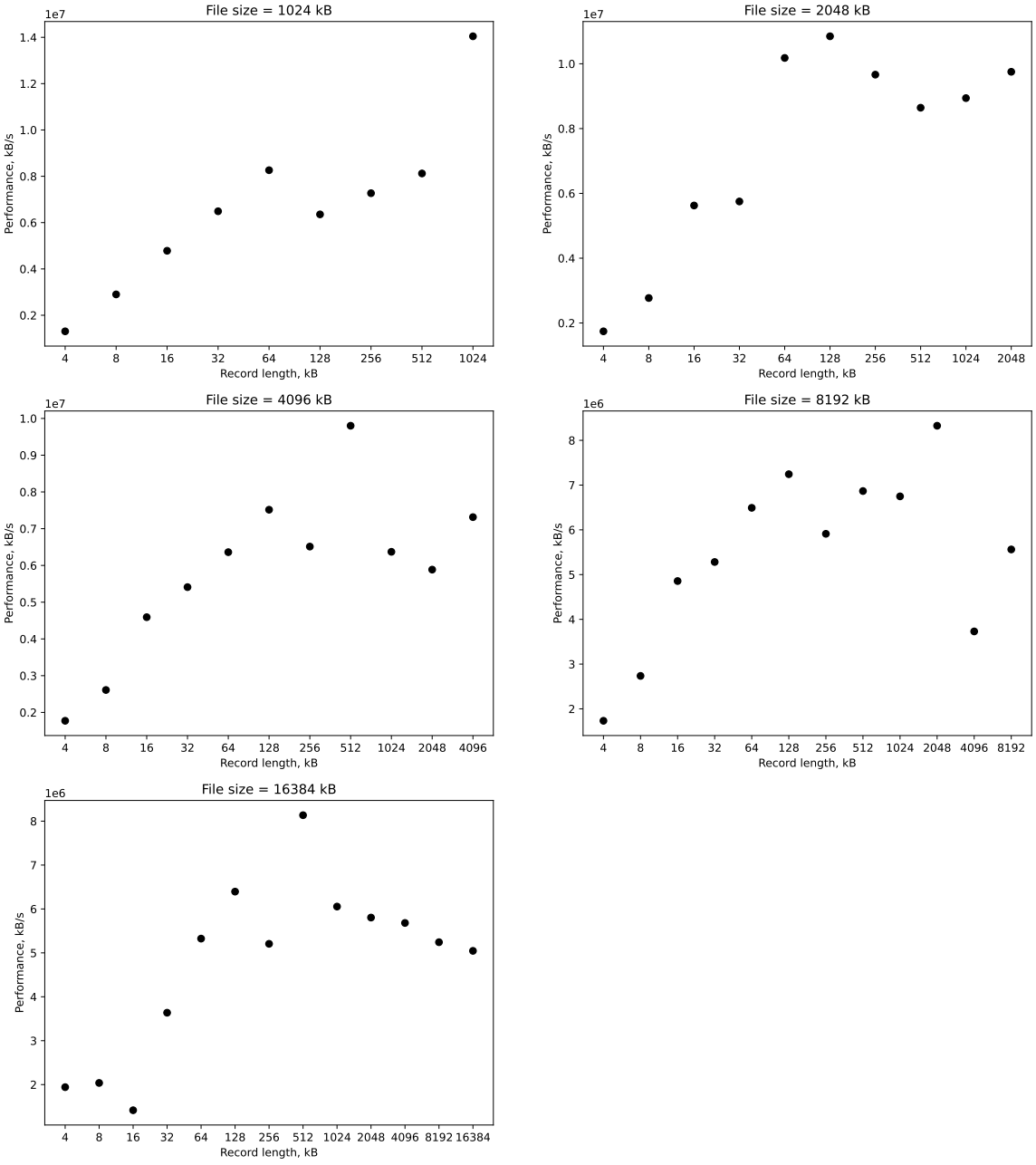

Figure C.23: IOZone output for Random read

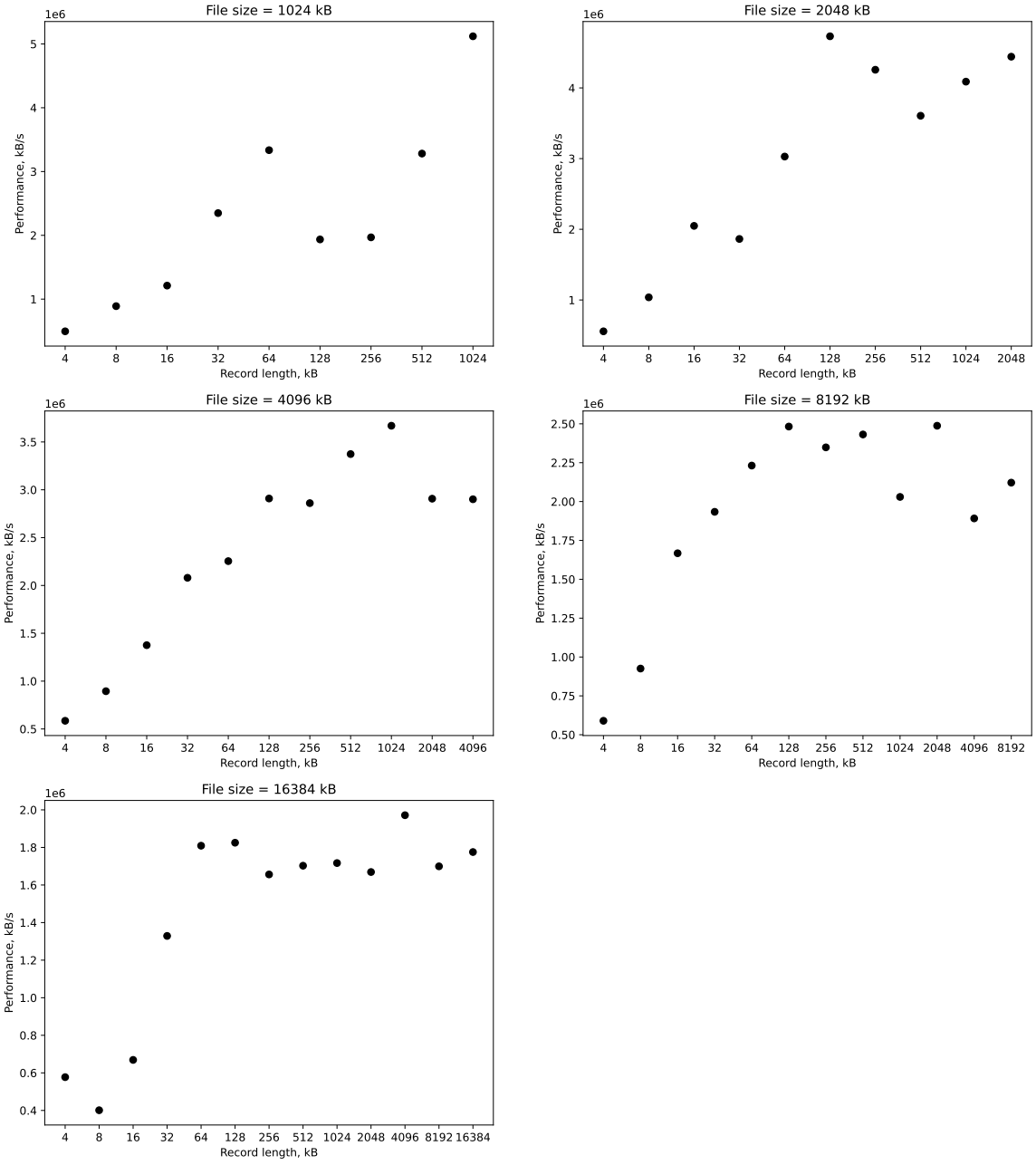

Figure C.24: IOZone output for Random write