FAU Erlangen-Nürnberg

Hochschule Albstadt-Sigmaringen

Masterthesis

zur Erlangung des akademischen Grades

Master of Science

Forensic Examination of Ceph

Florian Bausch

Matrikel-Nummer: 84444

03.09.2018

1. Prüfer: Prof. Dr.-Ing. Felix Freiling, FAU, Erlangen

2. Prüfer: Dr.-Ing. Andreas Dewald, ERNW, Heidelberg

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

CephFS Ceph File System

CFReDS Computer Forensic Reference Data Sets

CRUSH Controlled Replication Under Scalable Hashing

KV Key-Value

LVM Logical Volume Manager

MDS Metadata Server

OSD Object Store Device

PG Placement Group

RADOS Reliable Autonomic Distributed Object Store

RBD RADOS Block Device

RGW RADOS Gateway

SDS Software Defined Storage

SES SUSE Enterprise Storage

VM Virtual Machine

XFS X File System

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Verwendung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch in keiner anderen Prüfungsbehörde vorgelegen. Alle eingereichten Versionen der Arbeit sind identisch.

03.09.2018, Heidelberg

Florian Bausch

Abstract

The concept of *Software Defined Storage* (SDS) has become very popular over the last few years. It is used in public, private, and hybrid clouds to store enterprise, private, and other kinds of data. Ceph is an open-source software that implements an SDS stack.

This thesis analyzes the data found on storage devices (OSDs) used to store Ceph BlueStore data from a data forensics point of view. The OSD data is categorized using the model proposed by Carrier into the five categories: file system, content, metadata, file name, and application category. It then describes how the different data can be connected to present useful information about the content of an OSD and presents the implementation of a forensic software tool for OSD analysis.

Abstract (deutsch)

In Laufe der letzten Jahre wurde das Konzept von *Software Defined Storage* (SDS) immer populärer. Es wird sowohl im Rahmen von Public, Private als auch Hybrid Clouds genutzt, um Unternehmensdaten, private Daten und andere Arten von Daten zu speichern. Ceph ist eine Open Source Software, die einen SDS-Stack implementiert.

Diese Thesis stellt eine forensische Analyse von Daten auf Speichermedien (OSDs) dar, die Ceph-BlueStore-Daten beinhalten. Die Daten werden nach dem Carrier-Modell in fünf Kategorien eingeteilt: Dateisystem-, Inhalts-, Metadaten-, Dateinamen- und Anwendungskategorie. Die Thesis beschreibt, wie die unterschiedlichen Daten verknüpft werden können, um nützliche Informationen über den Inhalt einer OSD zu erhalten, und zeigt die Implementierung einer forensischen Software für die OSD-Analyse.

Acknowledgement

Special thanks goes to the *SAP on Lenovo Servers* team of Lenovo Global Technology Germany GmbH in Walldorf and Stuttgart; especially to Martin Bachmaier for deep conversations about Ceph internals and Andreas Müller for providing the test environment in the Walldorf Lab.

1. Introduction

Ceph is a software implementation for distributed network storage and originates from the Ph.D. thesis of Sage Weil in 2007 [2].

The term *Software Defined Storage* (SDS) is used to describe the concept of using commodity hardware and a software stack to create a single storage. Entity availability and reliability of the SDS is achieved by using clusters of commodity servers and a software that replicates and balances data between the commodity servers. SDS solutions provide different interfaces to access the storage [4].

Weil's intention was to develop "a distributed file system that provides excellent performance, reliability, and scalability" [2, p. 2]. As underlying technology Weil introduced *Reliable Autonomic Distributed Object Store* (RADOS), a large logical storage that spans a cluster of servers. Data is stored redundantly in this cluster, while a placement algorithm called *Controlled Replication Under Scalable Hashing* (CRUSH) determines where the data is stored in the cluster.

In 2012, Weil founded the enterprise Inktank to provide professional services and support around Ceph. This enterprise was bought by Red Hat in 2014 [5].

In the meantime, not only Red Hat contributes actively to the development, but also other companies, such as Intel, and SUSE [6]. Ceph is used in private [7] and business environments [8] to create storage backends of private and public clouds.

The standard release cycle for new Ceph releases is 9 months [9]. New releases are then also integrated in enterprise offerings such as SES [10].

The underlying technology of Ceph is the *Reliable Autonomic Distributed Object Store* (RADOS). This object store is distributed over several *storage nodes* which donate their storage capacity to the object store. Data is stored redundantly on more than one storage node either by using replicas or erasure coding [11, 12, 13]. Replicas are exact copies of a dataset which are stored on other storage devices. The number of copies is indicated by the replication factor. A replication factor of `3` for example means that there are three exact copies of a dataset stored on different devices. Erasure coding provides redundancy by computing and storing parity information [12].

Data placement is determined by the CRUSH algorithm [14]. The algorithm uses information about the topology of a Ceph cluster and configurable rules to place the replicas onto different *Object Store Devices* (OSDs) of different storage nodes. An OSD is a storage device – HDD, SSD, or another type – which is managed by a Ceph daemon running on the storage node. Therefore, a single failing storage device or a single failing server is not affecting the availability of the whole object store.

There are different approaches to store data on OSDs. The older – but still usable – approach is to format the device with XFS and organize the object store data in files. The newer, now recommended approach is called BlueStore: The daemon writes the data directly to disk without using an intermediate file system. It was officially announced on September 1st, 2017 [1]. BlueStore improves performance by removing the intermediate file system layer and managing the available disk space on its own. By doing so, the data distribution and fragmentation on disk is managed completely by the Ceph software.

Every OSD uses a part of the storage device as an area for metadata – based on a *Key-Value* (KV) store – while the rest of the OSD is used to store the actual data, so called *objects*.

There are several applications that run on top of RADOS, for example RBD [15] and CephFS [16]. They provide different interfaces to clients, store data in form of objects in RADOS and add additional metadata to them.

During a forensic analysis of an OSD both types of data – metadata and objects – have to be brought together to gain a broader view on the content that is stored on an OSD.

A forensic analysis is executed in regards of the rules of *computer forensics*. The field of computer forensics focuses on evaluating and analyzing computer systems using scientific methods to explain certain states of computer systems or changes between states of computer systems. The intention is to provide a scientifically correct foundation for computer analyses, used for instance for reports in court [17, p. 1 ff].

A forensic analysis may have the goal to restore data from a drive, create a timeline of events by using knowledge of how a system is changed by certain actions, or find evidence that a certain action happened or did not happen in a computer system.

1.1. Motivation

Ceph is used by enterprises to store business critical data. This implies that high expectations are to be fulfilled in terms of data security and data safety. For example, Lenovo offers a storage solution as a storage backend for SAP HANA in-memory databases, called DSS-C [18].

From a forensics point of view, many efforts have been invested in examining storage devices (e.g. HDDs) [17, p. 181 ff] and file systems (e.g. ext2, ext3, ext4, XFS, APFS) [19, 20, 21, 22, 23, 24] which have been in use for several years. Wanted as well as unwanted behaviors are often known and can be used for data protection and data recovery [23, 25].

Ceph is in comparison to HDDs and file systems such as XFS a younger project – Ceph was first introduced in 2007 while XFS and ext development started in the early 1990s [26, 27] – but with growing importance [28]. BlueStore is an even newer concept in organizing the Ceph data on disk, officially announced in September 2017 [1].

Forensic evidence of OSDs using BlueStore is therefore not yet easily analyzable since data structures and behavior are not documented and there are no forensic tools for the investigation of OSDs. Yet, this would have a benefit for storage device forensics in data centers. Scenarios which could require BlueStore OSDs analysis are for example:

- An employee is caught at smuggling a hard drive from a Ceph cluster out of a data center. An internal investigation may find that the hard drive holds business critical data that – if in hands of competitors – could harm the company.
- A company running Ceph is in focus of a law enforcement agency. During the search for evidence someone finds hard drives that happened to be used in a Ceph cluster. The hard drives may hold evidence relevant for this investigation.

In both cases relevant evidence may be files or fragments of files with accounting information, intellectual property, or correspondence of high importance. The investigation can also focus on solving the question when certain data was stored in a Ceph cluster or finding deleted data.

Before the introduction of BlueStore, Ceph relied on an underlying XFS file system to store data on disk [1]. In this case, established and well-known forensic tools with XFS support can be used to analyze an OSD. But additional knowledge about Ceph is necessary to identify and interpret the data found within XFS. This is described in more detail in Chapter 2.2.1.

In case of BlueStore, there is no underlying technology that is known to forensic tools. (See Chapter 2.2.2.)

1.2. Contribution

This thesis analyzes the behavior of storage devices that are or were used as OSDs in Ceph setups with BlueStore. It documents the structure of an OSD, data structures and how to find them. It also shows how to reconstruct data and metadata – this also includes

data that was deleted from the object store but still physically resides on the OSD. For example, it is possible to restore topology information of a Ceph cluster, even after it is deleted. This information includes server IP addresses and events like disappearing and reappearing OSDs. This work shows how to combine data and metadata to get a more comprehensive view of the data.

It shows how to find the KV store, how to locate the object content on disk and the corresponding metadata in the KV store. Additionally, it presents how to make use of metadata stored by RBD and CephFS to give a context about objects. This includes, for example, file names and permission of files stored in CephFS. However, the fact that RADOS is a distributed data store implies that not every object and object metadata is stored on each of the OSDs. The thesis shows that data cannot always be completely restored basing on a single OSD. An aspect is how the data of two or more – or even all – OSDs may be used to restore more information.

In certain scenarios, Ceph uses a *Metadata Server* (MDS) [29]. It stores additional metadata for CephFS – a file system on top of Ceph – in RADOS. Therefore, this work examines how this additional data can be used and if there is useful information that can be extracted.

Ceph also provides a caching mechanism – called Cache Tiering – to improve performance of Ceph clusters [30]. An OSD that was used as a caching device may have to be analyzed differently and provides other information than a conventional OSD. This is also an aspect of this work.

Furthermore, this work presents the implementation of *Vampyr* – a software tool to automize extraction and restoration of data based on the findings. This software's purpose is to extract as much detailed and correct information as possible and to present it in a way that is helpful to people analyzing an OSD.

The thesis only covers the analysis of Ceph installations using BlueStore in combination with RBD and CephFS. Other configurations are not analyzed.

1.3. Outline

Chapter 1.4 gives an overview about related work. Chapter 2 provides a deeper insight into Ceph (Chapters 2.1, 2.2) and file system forensics (Chapters 2.3, 2.4).

The forensic analysis of Ceph BlueStore is described in Chapter 3 with detailed descriptions of BlueFS (Chapter 3.3.2), RBD (Chapter 3.3.3), and CephFS (Chapter 3.3.4).

Afterwards, the thesis shows in Chapter 4 how these findings are used to implement a forensic software – called *Vampyr* – that analyzes Ceph OSDs. It shows in Chapter 5 how correctness and completeness of the software were considered in this implementation.

It closes with a summary (Chapter 6) including an analysis of aspects that need further investigation.

1.4. Related Work

This chapter gives an overview about work related to Ceph and file system forensics. It also introduces the Carrier model which is used in this thesis to categorize data.

1.4.1. Ceph

The initial idea and concept of Ceph was introduced by Sage Weil in his Ph.D. thesis [2]. It outlines the architecture of a Ceph cluster and defines the key concepts: An OSD is the combination of CPU, network interface, cache and the actual storage device. Clients write or access data in form of objects with variable size to or from OSDs, while each OSD handles the data distribution on the low-level block device. Clients execute metadata operations by communication with a metadata server (MDS), but directly send data to or retrieve data from the OSDs. The goal is to improve scalability by reducing bottlenecks in form of centralized services.

Ceph explicitly does not use concepts like file allocation tables. Instead, the name of an object is computed in a manner that every party can compute the name and does

not need to look it up in an allocation table [2, p. 19]. The terms file and object are not interchangable. If a file is sent by a client to the object store, it may be split into several chunks; each chunk is saved as an object.

The theoretical background of the data distribution algorithm CRUSH is described by Weil et al. (2006) [14] and Weil (2007) [2]. CRUSH bases on a *cluster map* which consists of devices and buckets. A bucket may consist of devices and buckets of any number. Buckets and devices have a weight and a unique identifier.

By using buckets, devices can be grouped arbitrarily. For example, all devices of a server may be in a bucket, while all servers of a data center are in a bucket. Another possibility is that all SSDs and HDDs of a server are in distinct buckets to adapt the workload to the underlying storage devices.

One design feature of CRUSH was that "CRUSH's data distribution should appear random – uncorrelated to object identifiers x or storage targets – and result in a balanced distribution across devices with equal weight" [14, p. 7]. This means that objects in RADOS are distributed more or less evenly across all OSDs and that there is no specific pattern that would allow determining the location of objects with similar properties, for example similar names. Additionally, CRUSH offers an overload protection, moving workload from poor performing devices (partial failures) to other devices.

The technical background and motivation to implement a new storage backend (BlueStore) for Ceph was outlined by Sage Weil [31, 1] and Tim Serong [32, 33]. FileStore has several performance bottlenecks caused by the XFS file system. In order to ensure consistency of the object data on an OSD, Ceph has to write the data to a journal and then write it to the file that represents the object data in XFS. This means Ceph executes two write operations to XFS per write operation to RADOS, decreasing the I/O (input/output) throughput. Furthermore, organizing the files that store object data turned out to be complex. This complexity comes with a certain overhead when files have to be moved. Additionally, XFS itself implements a cache, journal, and a data dis-

tribution algorithm for on-disk data. Tests and experience from Ceph cluster operations showed that the XFS design does not match the workload patterns of Ceph, resulting in non-optimal performance.

These aspects led to the decision to implement an alternative to FileStore. This alternative was called BlueStore. The most important difference to FileStore is that Ceph manages the disk space completely on its own. Therefore, there are significant performance improvements – up to three times faster writes for certain workloads – because the allocation strategies for data units and caching and flushing strategies for data held in memory could be optimized for Ceph-specific workloads.

Disc write behavior was analyzed in respect to I/O performance and optimization in 2016 [34] and 2017 [35].

Oh et al. [34] analyzed the FileStore I/O process and identified potential bottlenecks that could decrease I/O throughput. By optimizing the locking mechanism, reducing blocking operations, and using light-weight transactions, it was possible to improve the FileStore performance. The optimizations were submitted to the Ceph community.

Lee et al. [35] analyzed the performance impact of deciding for BlueStore or FileStore, mainly the write amplification factor (WAF). The WAF indicates how much I/O overhead is created by a write operation to a storage – in this case RADOS. A higher WAF indicates a bigger overhead. This work found that BlueStore provides in general a better I/O performance than FileStore. For both, HDDs and SSDs, BlueStore increases throughput by removing the intermediate file system and reducing the WAF.

1.4.2. File System Forensics

Without knowing any information about a file system and how it is structured, a storage device may be analyzed forensically by using *data carving* or *file carving*. It is a technique that scans storage devices for known patterns of certain file formats [36]. There is a wide

range of works about file carving. A literature overview was compiled by Poisel and Tjoa in 2013 that selected 70 key papers. [36]

In 2007 Garfinkel presented an analysis of more than 300 file systems on drives, bought from the secondary market, in regards of file carving. He outlines the problems of file fragmentation for file carving and shows that it is possible to reconstruct fragmented files. This is achieved by scanning a drive for certain file signatures, and finding header structures and footer structures in different fragments on disk. [37]

Pal and Memon outlined in 2009 different file carvers (file structure-based and graph theoretic carvers) and carving algorithms (for example parallel unique path, shortest path first, and bifragment gap carving). [38]

There are forensic tools that implement different file carving techniques with different kinds of optimizations. Some of them focus on finding certain file formats. Some of them support a bigger number of file formats. There are for example:

- Scalpel, as part of the forensic suit The Sleuth Kit, is able to carve for different file types based on file signatures. The default list of signatures – including JPEG, GIF, and ZIP signatures – can be extended by a user to support other file types [39].

- Foremost is a tool similar to Scalpel and carves for many different file formats. Foremost was used as basis for Scalpel development [40].

- bulk_extractor carves for certain data, for example EXIF data (from JPEG images), IP addresses, e-mail addresses, and ZIP-compressed data. It can also create histograms to find data that appears more frequent in a hard drive [41].

- EVTXtract carves only for Windows event log files in the EVTX file format [42].

Since file systems hold more data than recoverable by file carving, each file system type has to be analyzed specifically.

A fundamental work on file system and storage device forensics was done by Garfinkel and Shelat [43] when analyzing data of 158 used hard drives purchased from the secondary market between 2000 and 2002. It shows that storage devices can hold recoverable sensitive data and that even Level 3 data – data of a usage before a drive was reformatted – can be restored. Garfinkel and Shelat were able to find medical, corporate, and credit card information.

Storage devices usually contain some metadata about how the device is divided into smaller portions (partitions), so-called partition tables. Carrier describes DOS partitions and the corresponding metadata concepts. DOS partitions are described by an MBR (Master Boot Record) that is located at the start of a device in the first 512-byte sector. It contains boot code, as well as starting and ending address, length in sectors, type of partition, and flags for each partition. Carrier also explains the concept of extended partitions that are used to overcome some limitations of the MBR [23, p. 81 ff].

Carrier also describes metadata of software used for disk spanning. "Disk spanning makes multiple disks appear to be one large disk" [23, p. 156]. He outlines the concept of Linux LVM which combines physical devices to logical volumes. The LVM metadata is located at the start of a drive and can be analyzed by commands like `vgscan` [23, p. 160 ff].

The GPT partition table format was analyzed by Nikkel in 2009. It provides several benefits compared to the MBR; it supports larger partitions, a larger amount of partitions per drive, a backup copy of the partition table, and integrity checks via CRC32 checksums. Backup copy and checksums make it possible to detect data corruption and manipulations of a partition table. GPT tables also contain information that opens new possibilities for the reconstruction of deleted partitions [44].

Carrier compiled a wide-ranging overview about file system forensics [23] including approaches to volume, partition, and file system analysis. He shows the analysis of the Windows file systems FAT and NTFS. The file system FAT – developed by Microsoft –

make use of a File Allocation Table (FAT). It is used to look up data units that belong to a file and contains an entry for every data unit of a partition. Carrier lists analysis techniques for FAT file systems and highlights characteristics, for example the accuracy of different timestamps, allocation strategies and effects of deletion of files [23, p. 211 ff].

The file system NTFS – also developed by Microsoft – uses other concepts. It treats everything as files, even metadata. The central file is the Master File Table (MFT) containing metadata of every file in the file system. The content of a file is looked up using the MFT. Other files like `$LogFile` (for the file system journal) or `$Bitmap` (for the data unit allocation status) store other relevant metadata. Carrier presents how the data and metadata is organized and stored in the different kind of files. He also shows strategies to analyze NTFS file systems [23, p. 273 ff].

Shullich documented in 2009 data structures and characteristics of the exFAT (extended FAT) file system and the difference and similarities to FAT and NTFS. The paper also explains what certain fields and flags in data structures stand for [45].

The characteristics and data structures of ext4 – a common file system for Linux installations – were examined by Fairbanks (2012). Ext4 makes use of *extents* to address phyiscal disk areas. An extent comprises a start address and a length. Metadata of files, for example timestamps and the list of allocated extents, is stored in *inodes*. Each file has a unqiue inode. Additionally, ext4 uses a journal to record changes, for example file or metadata updates, that are not yet applied to the internal file system structures. The journal improves the file system behavior in crash scenarios [46].

An analysis of the XFS file system was executed by Bausch (2016). It shows that XFS uses similar concepts as ext4. File content is stored in extents and metadata is organized in inodes. The paper also explains data structures and where to find them [21].

CFReDS [47] is a collection of different file system images that was compiled by NIST to serve as a kind of training and reference data. The file systems used in this collection are therefore analyzed frequently by different people and software.

The Sleuth Kit – together with Autopsy – is a forensic analysis tool for various common file systems. The website lists NTFS, FAT, exFAT, UFS 1, UFS 2, ext2, ex3, ext4, HFS, ISO 9660, and YAFFS2 as supported file system types. It can analyse content to find certain information, for example geo locations and thumbnails from JPEG files, or e-mails [48].

X-Ways Forensics is another forensic analysis tool that supports for example ext4, XFS, and also disk spanning technology like LVM [49].

All these tools and papers do not deal with Ceph or BlueStore, but show a general approach to forensic analyses of storage devices. There has not been published any forensic examination of Ceph OSDs.

There is a small software tool called *ceph-recovery* [50], but it does not analyse data structures from a forensic researcher's point of view. It is a simple tool that only works with FileStore OSDs. It restores RBD contents by searching for files containing RBD objects. The tool *rbd_restore.sh* [51] uses the same approach and restores RBD content. It also works only with FileStore OSDs. Both tools do not execute further metadata analysis or support other data than RBD data.

1.4.3. Carrier Model

A method to categorize data that is found in file systems was introduced by Carrier [23, pages 171 ff]. The basic assumption is that all data of a file system can be assigned to one of the following data categories: file system category, file name category, metadata category, content category, and application category. Carrier does not only categorize the data, but also shows basic analysis techniques for each category.

The following sections summarize the Carrier model as described by Carrier (2013) [23] in order to give an overview since it is used later to categorize the data found on Ceph OSDs.

File System Category

Each file system holds data that describes the general layout of the file system. It contains a name, ID, version, and creation date of the file system and addresses of important data structures of the file system.

In general, data structures of the file system category are the entry point of a file system analysis because they point to data structures of other data categories.

Content Category

Data of the content category contains the actual content of files or directories and the data structures that keep track of the allocation status of allocation units. Allocation units are blocks or extents of the device. The allocation status can be *allocated* or *unallocated*.

There are different approaches to track the allocation status and there are different algorithms to determine which data unit should be allocated next. Carrier found, for example, that FAT file systems use a *next available* algorithm; this means the search for a free allocation unit starts at the last allocated unit [23, p. 224]. He also found that NTFS uses the *best-fit* algorithm; as the name indicates, the data is written to an area of the disc where the free disk space is used most efficiently [23, p. 313].

Metadata Category

Data of the metadata category holds descriptive data of the content category. These are for instance timestamps (creation data, last access), addresses of allocation units that contain the file content, and the size of the file.

File Name Category

Every file or directory has a human readable name which needs to be translated to the file system internal name or ID. The data structures that contain this mapping count to the file name category.

The data structures may include additional information about a file, for example about the file type.

Important is that the name of the root directory of a file system is known. By using this name as an entry point to the file system, it is possible to traverse through the directory tree and find files and directories.

Application Category

Some file systems define special areas of a storage device and special data structures that are not mandatory for the file system to work, but provide benefits.

An example are file system journals. Here the file system writes transactions to a special area of the device and later applies the changes to the file system data structures.

Essential and Non-Essential Data

Furthermore, Carrier divides data of all categories in essential and non-essential data. Essential data are data structures and information necessary to retrieve a file with correct content from a drive or store a file with correct content on a drive.

Non-essential data is all kind of information that is available, but not mandatory for the file system to work properly. This information is stored for “convenience”, such as timestamps, and permissions.

The key difference for Carrier is that essential data must be correct, while non-essential data may be incorrect.

2. Background

This chapter gives a short overview about the architecture of Ceph clusters and a deeper view into the Carrier model for categorizing file system data. At last, it highlights the difference between event-based and state-based analysis of file system data.

2.1. Architecture of a Ceph Cluster

In order to understand the data on a block device used by Ceph, it is necessary to have an understanding of concepts and terminology used by Ceph.

A Ceph cluster consists of at least three servers that share storage resources. On each of these servers the Ceph software is installed.

Storage devices (HDDs, SSDs, NVMes) that are contributed by a server are called *Object Store Device* (OSD). Each OSD is managed by an OSD daemon process in Linux.

OSDs hold the objects, i.e. the actual content, and a KV store for metadata. There are different supported disk layouts and KV backends, but this work explicitly focuses on Bluestore with RocksDB KV backend. RocksDB provides a KV store, written in C++ and optimized for low latencies [52].

The Ceph software logically joins all available OSDs to a single storage, an object store (RADOS). Within the object store data is stored depending on the configured rules using the CRUSH algorithm. The cluster configuration specifies replication levels and a hierarchical cluster map. This cluster map groups available resources hierarchically,

for example into server racks, servers, and storage arrays. The CRUSH algorithm then places data replica for highest availability: Two replica should not reside on the same server.

The object store is divided into subsets called pools. Pools are used to logically separate data of different workloads on application level [13]. Each pool consists of a number of placement groups (PGs) and each object in a pool is assigned a PG. The replication and placement rules of the CRUSH algorithm are applied per PG, not per object. This means that all objects within the same PG are replicated to the same OSDs. Each OSD holds objects of different PGs [53, 54].

Ceph redistributes data if one or more OSDs is missing for a longer time to restore the configured replication level. If the downtime of an OSD is below a configurable threshold the data is not redistributed, but the data on the reappearing OSD is updated after being reincluded into the cluster.

There are different applications that make use of Ceph and its RADOS (see also Figure 2.1). Two applications are RBD and CephFS. RBD provides an interface to clients to use a part of the object store as a block device [15]. CephFS provides an interface for clients to use a part of the object store like a normal file system [16]. Ceph provides APIs – RGW and librados – for general object access that allow to implement other applications that store data in form of objects in a Ceph object store [55].

2.2. Storage Backends – BlueStore and FileStore

Ceph OSDs can be configured using either FileStore or BlueStore. The older approach is FileStore. BlueStore is the newer implementation that was implemented especially for the needs of Ceph [1].

Figure 2.2 schematically shows the difference between both implementations.

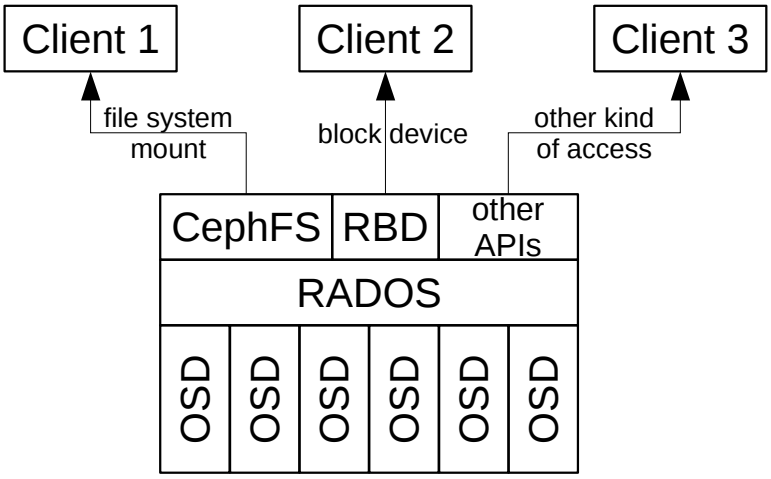

Figure 2.1.: Applications on top of RADOS provide different APIs to store data in the Ceph storage.

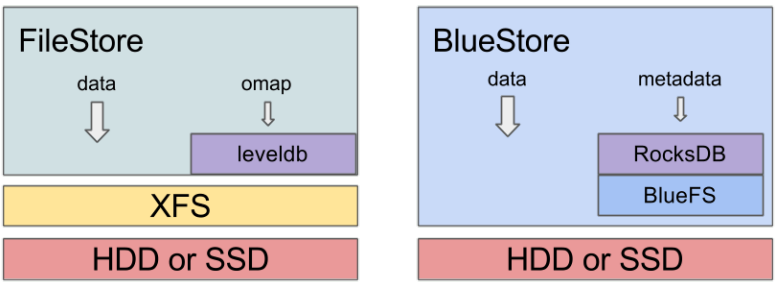

Figure 2.2.: FileStore vs. BlueStore OSDs [1]

2.2.1. FileStore

The storage device is formatted with an XFS file system. This file system contains a directory `current`. The RADOS objects that are stored on this OSD are located in this directory. It also holds the KV store in the subdirectory `omap` and additional metadata in the subdirectory `meta`. The objects are located in subdirectories that are named after their PG. The KV store is implemented as a LevelDB. Furthermore, the root directory of the XFS file system contains a symlink `journal` to a journal device [33, 32]. If no dedicated journal device is used, the journal is located on the OSD.

This means that the raw content of the objects stored on an OSD can be found as files using forensic tools that support XFS. The LevelDB files can also be restored using such a tool. However, the content of the database – the actual metadata – cannot be analyzed automated because there is no tool that implements an automated analysis.

2.2.2. BlueStore

The storage device is managed by LVM. The OSD content (objects, metadata, etc.) is located in the only large partition spanning the whole device [56, 1].

This partition is completely managed by Ceph: Ceph can load the KV store that is located on this partition in a minimal file system-like environment called BlueFS. The KV store is implemented using RocksDB. The rest of the partition is used to store the objects directly on the block device.

In order to improve the I/O performance of this primary device, Ceph can configure two additional types of devices: DB device and WAL device. If the WAL device is configured Ceph will write the write-ahead log (WAL) of RocksDB to this device. If the DB device is configured, Ceph writes the RocksDB to this device, but may overflow to the primary device if needed [31, 33, 32, 56].

2.3. Carrier Model

The different data categories of the Carrier model were introduced in Chapter 1.4.3. However, Carrier did not only categorize the data, but also suggested analysis techniques and strategies for the different categories [23, p. 171 ff]. Some of them are described in the following chapters because they are used later.

2.3.1. File System Category

Data in this category is described by Carrier as "typically single and independent values" [23, p. 178]. These values should be simply presented to the investigator. Additionally, there should be a consistency check of values if possible. For example, if the data shows a certain size of the file system, the image of the file system should have at least this size. If the image is larger than the file system, there is a *volume slack* that might be of interest to the investigator. Depending on how the data is structured there might be additional places where data could be hidden in unused areas.

2.3.2. Content Category

To analyze data of the content category Carrier describes different techniques [23, p. 181 ff].

Data Unit Viewing An investigator may have knowledge that a certain data unit contains certain data or that a certain data unit should be empty. The address of the data unit must then be computed correctly on the basis of the data unit size of the file system. Afterwards, the content can be checked using a hex editor or other tools that can interpret the data of this data unit.

Logical File System-Level Searching An investigator knows that certain data should be in the image but does not know where. The strategy is to scan the image for the

data. This is for example achieved by file carving tools. One challenge is that the data may be fragmented.

Data Unit Allocation Status File systems hold metadata that describes which data units are allocated and which data units are unallocated. Using this knowledge it is possible to extract all data from a file system that is located in unallocated data units or to scan those data units for specific content. By this, an investigator may find data that was deleted by users or applications.

Data Unit Allocation Order Data units are allocated using different strategies. These strategies depend on the operating system. With knowledge about the strategies an investigator may trace back certain events and their order depending on the allocation order of data units.

Consistency Checks In most file systems there should be only one metadata set per allocation unit that marks it as allocated. Furthermore, for every allocated data unit there should be one metadata entry. An allocated data unit without metadata entry is called *orphan*.

2.3.3. Metadata Category

Carrier describes the following techniques and strategies to analyze data of the metadata category [23, p. 192 ff].

Metadata Lookup An investigator finds data that points to a metadata structure of a file or directory. The address of the metadata must be interpreted correctly. After reading the metadata structure at this address, it can be decoded. Afterwards, metadata like allocated data units, size, timestamps, or permissions are known to the investigator.

Logical File Viewing After reading the metadata structure, the data units that belong to a specific file can be read in the correct order. The file content can be extracted or displayed. Here it is important to keep the slack space in mind where data could be hidden.

Logical File Searching It might be necessary to find a file with a specific content. After reading the metadata structures it is possible to scan the file system per file, while the Logical File System-Level Searching scans the file system per data unit. The advantage of Logical File Searching is that an investigator can identify content that is spread across non-consecutive data units. A disadvantage is that it only finds content in allocated data units.

Unallocated Metadata Analysis A metadata structure can still exist after all references to this structure were removed. By looking for such unallocated metadata, it might be possible to find additional information about file system access or deleted files.

Metadata Attribute Searching and Sorting During an analysis of a file system it is often not useful to just display every metadata information that is available. It can be helpful to sort and search the metadata based on different criteria. These can be, for instance, the timestamps of files or a certain set of permissions or owner information.

An investigator may also search for a metadata entry that references a certain data unit. One reason for this is, for example, that a logical file system search showed a data unit containing a certain data structure and the investigator wants to find the matching metadata.

Data Structure Allocation Order Like data units for file content metadata space is allocated using certain strategies. Knowledge about these strategies – which depend on

file system and operating system – can help reconstructing an order of events that led to the metadata constellation found in a file system.

Consistency Checks Consistency checks of metadata should focus on essential data, for instance data unit addresses, size, and allocation status of metadata entries. Data unit addresses should not lie outside of the file system address space and should not be used by more than one metadata entry.

It is also possible that a metadata entry is not referenced by any other data structure. Further analysis could identify hidden or lost data.

2.3.4. File Name Category

The following stategies can be applied to use data of the file name category [23, p. 202 ff].

File Name Listing Since the data of the file name category assign file names to file metadata structures it is obvious that an investigator can use the data to create a list of file names and the corresponding metadata. The file names should not only contain the name of the file, but also the full path starting from the root directory of the file system.

Usually, it is necessary to locate the root directory metadata at first. Afterwards, the files and directories that are contained in the root directory are analyzed, then the sub-directories, and so on. Most forensic tools show data of the file name and metadata category next to each other.

By listing file names, linking them to the matching metadata, and retrieving the data from the data units listed in the metadata, it is possible to get the file content of a specific file.

File Name Searching If only a part of a file name or a naming scheme is known, an investigator can scan the data of the file name category to create a list of relevant files.

For example, configuration files of certain applications can have a certain name. This has the risk that an investigator may miss relevant files when looking for the wrong names or when somebody deliberately chose a file name not following a certain pattern.

Data Structure Allocation Order As in the categories before, it is possible to draw certain conclusions about the order of events by looking at the order of data structures on disk.

Consistency Checks Every metadata category entry should be assigned one file name category entry. In many file systems more than one file name category entry can point to the same metadata entry.

2.3.5. Application Category

Carrier does not specify analysis techniques for data of the application category. Because the use case of this kind of data is often very specific, every use case needs its own, specific procedure.

2.4. Analysis Methodologies for File Systems

There are two main approaches to determine the behavior of a file system under different conditions (for example, writing or deleting data). Both are described by Freiling et al. [57, p. 101 ff].

Event-Based Every access (*event*) to a file system is logged at runtime of the system. If a file content, file name, or metadata is changed, the event logger saves information about this event in a log. Afterwards, it is possible to determine the order and impact of events. This methodology is very similar to the live response phase of the common model as outlined by Dewald et al. (2015) [17, p. 170 f].

The problem of this approach is that the logger must be configured in a way that it does not miss any event. Furthermore, this approach does usually not provide information about data structures on disk. However, it might be useful for example to explain data structure allocation orders after certain events.

State-Based An image of a file system is created – persisting a *state* of the file system. Afterwards, a defined set of events is applied to the file system. Once finished, a second image of the file system is created. Both images can now be compared to determine the impact of the events on the file system.

A disadvantage of this approach is that it is not always easily possible to determine which of the events caused a certain change in the file system. Another disadvantage is that the effect of background activities of the operating system – which can cause changes to the file system – may be interpreted as an effect of the defined set of events.

The state-based approach allows to determine how data structures on disk changed and which data remained on disk even though it had been deleted.

3. Forensic Examination of Ceph

The analysis of BlueStore consisted of three actions which were executed in parallel:

- Reading documentation
- Reading source code
- Setting up a test environment and analyzing OSDs from this environment

In a further step, data from other Ceph environments was used to check whether the understanding and findings hold when applied to real-life setups.

The OSDs were analyzed using the state-based methodology – explained in Chapter 2.4 – in order to find differences in data structures between defined states. The different states are outlined later.

This chapter gives an overview over the structure of an OSD device (Section 3.1) and describes common data structures and data types written to disk by Ceph (Section 3.2). It then describes the OSD data using the Carrier model (Section 3.3) and looks at Ceph Cache Tiering (Section 3.4).

3.1. Overview

A storage device used for BlueStore starts with an LVM header. This means that the actual OSD content is encapsulated by LVM. An example of LVM metadata of an OSD is attached in Appendix D.1. The LVM header analysis is not part of the thesis.

The start of the actual OSD content is marked by the BlueStore label containing the string “bluestore block device”. (See Chapter 3.3.1.) The BlueStore label is followed by the BlueFS superblock and the OSD superblock.

The actual BlueFS content is placed at a variable offset from the start of the OSD. It is not located at the start or end of the OSD. In the test setup it is located at about 6% of the OSD length. (See Chapter 3.3.2.)

This general structure of a BlueStore OSD is illustrated by Figure 3.1.

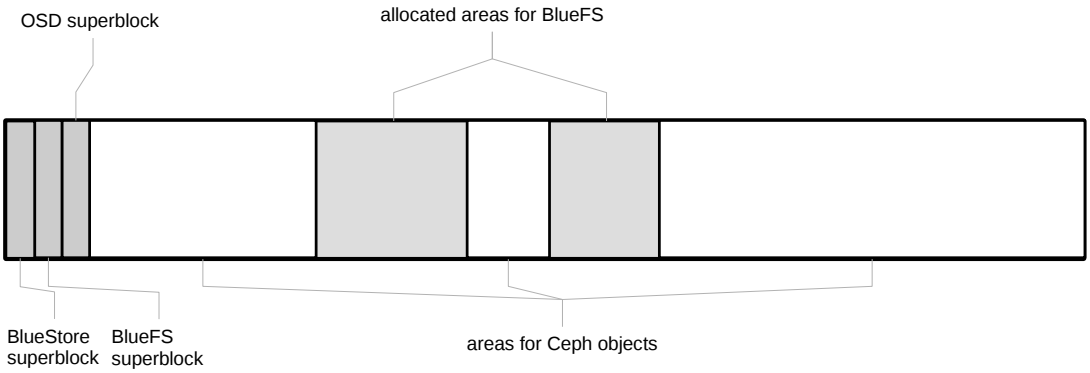

Figure 3.1.: Overview about a BlueStore OSD.

The KV store is the central place for metadata in a BlueStore OSD. It is a simple RocksDB database – located in BlueFS – that stores key-value pairs. There are no database tables to distinguish different types of metadata. Different metadata types are distinguished by the prefix (see Table 3.1) of the key of a key-value pair. The prefix is followed by a null character and the actual key. A key-value pair looks like shown in Listing 3.1. The value can be empty.

```
<prefix><NUL><key> --> <value>
```

List of Listings 3.1: Structure of a key-value pair stored in RocksDB.

Prefix	Description
S	OSD metadata (S = super)
T	OSD statistics (statfs data structure)
C	Collection information
O	Object metadata
M	Object metadata
P	PG metadata
L	Deferred transactions
B	Bitmap metadata
b	Bitmaps
X	Shared blob information

Table 3.1.: Prefixes used in the KV store [3, lines 63 ff].

In the following *<prefix>-row* will refer to a key-value pair with a certain prefix, *<prefix>-key* will only refer to the key of such a row, and *<prefix>-value* will refer to the value of such a row, for instance `M`-row, `B`-key, `O`-value.

3.2. General Data Structures and Encoding

Ceph uses different reappearing data structures and ways of data encoding which are described in this section. Later on, they are referenced, for example, in the description of superblock data structures.

All complex data structures consist of the following basic data types.

3.2.1. Integers

Depending on the CPU architecture the values are either stored in little-endian or big-endian representation. Depending on the size (8 bits, 16 bits, 32 bits, 64 bits) of the integer datatype either 1, 2, 4, or 8 bytes are written to disk [58, lines 305 ff].

Keys in the KV store contain big-endian encoded integers.

3.2.2. Varint

Integers may be encoded in a variable length format. While decoding a varint, the decoder reads one byte and continues reading as long as the high bit of the last read byte is set. This means that each byte holds seven bits of the encoded value. The first byte represents the lowest seven bits of the encoded value. If there is a second encoded byte, it represents the 8th to 14th lowest bits; the 3rd byte represents the 15th to 22nd lowest bit, and so on [58, lines 375 ff].

3.2.3. Varintlowz

The varintlowz is another encoding format for integers. It is an extension to the varint format. At first the decoder decodes the value like a varint. Afterwards, the two lowest bits of the varint are interpreted as the number of low zero nibbles of the result. This means there are 0, 4, 8, or 12 zero bits as lowest bits of the result. The remaining bits of the decoded varint value are treated as high bits of the result [58, lines 440 ff].

3.2.4. LBA

The LBA encoding is another integer encoding format. The decoder reads four bytes, then the lowest one to three bits indicate the number of low zeros of the result. The comment from the Ceph source code explains the encoding (Listing 3.2) [58, lines 517 ff].

```
// first 1-3 bits = how many low zero bits
//     *0 = 12 (common 4 K alignment case)
//    *01 = 16
//   *011 = 20
//   *111 = byte
// then 28-30 bits of data
// then last bit = another byte follows
// high bit of each subsequent byte = another byte follows
```

List of Listings 3.2: LBA encoding explained as source code comment.

The name LBA (Logical Block Addressing) indicates that this format is mainly used to encode addresses on disk.

3.2.5. Strings

String data types are stored ASCII-encoded by Ceph. On disk they are preceded by an unsigned 32-bit integer representing the number of characters in the string [58, lines 681 ff].

In some rare cases strings are stored without the preceding 32-bit integer when the length is fixed or can be computed in another way.

3.2.6. Escaped Strings

For object names in the KV store there is a special encoding format: All letters are ASCII-encoded; the ASCII code-point of all non-letter characters is encoded as big-endian 16-bit integer instead of the character, preceded by `#` or `~` – depending on whether the following ASCII code-point comes before `#` or after `~` in the ASCII table; the end of the string is marked by `!` [3, lines 127 ff].

For example, the string `abc!1` would be encoded to the hexadecimal values `61 62 63 23 00 21 31 21` (see also Table 3.2).

Character	Encoded to (hex)
a	61
b	62
c	63
!	23 00 21
1	31
String end	21

Table 3.2.: Escaped string encoding.

3.2.7. List Types

A 32-bit integer represents the number of elements in the list. The elements are then encoded depending on their data type [59, lines 613 ff].

3.2.8. Map Types

A 32-bit integer represents the number of key-value pairs in the map. Then each key-value pair is encoded depending on the data types [59, lines 887 ff].

3.2.9. Pairs

Pair datatypes are encoded by simply encoding the first element followed by the second element [58, lines 817 ff].

3.2.10. Utime

Two unsigned 64-bit integers representing a timestamp. The first represents the Unix timestamp in second-precision, the second the nanoseconds within this second [60].

3.2.11. UUID

16-byte binary representation of an UUID [61].

3.2.12. Bufferlist

Ceph internally uses bufferlists as data structures for encoded data. Encoded data can also be encoded inline within other data structures. Such an encoded bufferlist starts with a 32-bit integer indicating the length, followed by this number of bytes holding the bufferlist [58, lines 775 ff].

3.2.13. Data Structure Headers

Encoded data structures are often preceded by a header containing hints for decoding and interpreting the following data [58, lines 1640 ff], [59, lines 1202 ff].

- Unsigned 8-bit integer: Version
- Unsigned 8-bit integer: Compat
- Unsigned 32-bit integer: Length

The first byte indicates the version of the data encoder that wrote the data to disk. This is not the version of the Ceph software, but every data structure has its own encoder with its own version.

The second byte indicates the minimal version of the data decoder that is able to decode usable data from this encoded data structure.

The last four bytes indicate the length of the encoded data that will follow. Especially the information about the length of the following data structure is useful to check the integrity of on-disk data.

3.3. Applying the Carrier Model

The Carrier model as described in Chapter 1.4.3 can be used to categorize the different data structures found on OSDs. Also BlueFS, RBD, and CephFS data can be categorized using this model.

3.3.1. OSD

The general OSD layout was described before. This section analyzes the data in more detail.

File System Category

The BlueStore superblock is located in the first block of the OSD (0x0 - 0x1000 offset [3, lines 75 ff], excluding the LVM header) and contains the basic information about the layout of the OSD. See Table 3.3 for the general superblock structure and Appendix B.1 for a hexdump of a superblock.

It starts with the string "bluestore block device\n" followed by a string containing the OSD UUID, followed by a line break. This label is always 60 bytes long.

In the metadata information block `whoami` represents an OSD ID. This ID is used in other data structures to identify the OSD in OSD lists. For BlueStore the `kv_backend` is always `rocksdb`. The `ceph_fsid` is the UUID of the Ceph cluster.

There is a CRC32 checksum after the superblock for integrity checks [3, lines 4277 ff].

Type	Description
fixed length string	BlueStore label, 60 characters
Header	Data structure header
uuid	OSD UUID
int64	OSD length in bytes
utime	Creation time of OSD
string	Description; `main` in case of a primary device, `bluefs db` in case of a DB device, `bluefs wal` in case of a WAL device
dict(string, string)	Metadata; contains for example the UUID of the Ceph cluster
int32	CRC32 checksum

Table 3.3.: BlueStore Superblock structure

Furthermore, the KV store stores data of the file system category in `S`-rows and `T`-rows.

Analysis Techniques If the position of the BlueStore superblock is unknown, it can be identified by scanning the image for the string "bluestore block device".

The data encoded in the superblock and the data from the KV store can be displayed.

Furthermore, the slack space between the end of the superblock data structure and the next block at `0x1000` offset can be extracted and analyzed.

```
----------------
Bitmap Metadata:
----------------
blocks --> 0xba3680
blocks_per_key --> 0x80
bytes_per_block --> 0x4000
size --> 0x2e8d930000
```

List of Listings 3.3: Bitmap metadata in B-rows indicates how to interpret the bitmaps stored in the KV store.

The size of the image can be compared to the OSD length advertized in the superblock. If the image is too short this may lead to errors during analysis; if the image is longer it contains a volume slack that can be analyzed.

Content Category

The content of an OSD are the objects of RADOS that were distributed by the CRUSH algorithm to this OSD.

When a client sends data to a Ceph cluster, this data may be split into several objects if the object exceeds a certain size – the so-called object size. These objects may belong to different PGs and will therefore reside on different OSDs.

Each object can consist of several physical extents. The size of the extents is not fixed, but it is allocated depending on the size of an object. There is a "minimal allocation size", i.e. the minimal size of a physical extent. It is defined in the KV store in the `B`-value of the key `bytes_per_block`. The size of physical extents is a multiple of the minimal allocation size. Listing 3.3 show `B`-rows found in a KV store; in this example, the minimal allocation size is `0x4000`, which is 16384 bytes (16 KiB).

Data Unit Allocation Status Whether an area of the disk is allocated is tracked by the *bitmap allocator*. Free extents are tracked by a bitmap [62],[3, lines 4496 ff]. The area

```
physical offset    --> bitmask for 128 blocks
------------------------------------------------------
0x0000000000000000 --> ffffffffffffffffffffffffffffffff
0x0000000000200000 --> fffffffffffffffff030000000000000
0x0000000000400000 --> 00000000000000000000000000000000
```

List of Listings 3.4: Bitmaps are stored in b-rows. Every hexadecimal value represents four blocks.

allocated by BlueFS is marked as used while the extents within the BlueFS are tracked by BlueFS.

The bitmap is stored within the KV store in `b`-rows. Information for interpreting the bitmap is stored in `B`-rows.

The bitmap consists of multiple rows in the KV store of an OSD. Each `b`-value contains a bitmap for a number of consecutive blocks. The number of blocks is defined in the `B`-row `blocks_per_key`. Each block has a size of `bytes_per_block` bytes. This is the minimal allocation size of a block on disk. Each `b`-key contains the physical offset its bitmap refers to. The KV store, however, usually does not store bitmaps for all possible physical offsets. If an area of the disk was not used before, there is no bitmap stored. Listing 3.4 shows three `b`-rows.

Using the bitmap information makes it possible to extract and analyze unallocated areas of the OSD.

Metadata Category

Each object has a cluster-unique OID (object ID). This OID is used to identify the metadata of an object in different locations of the KV store. Object metadata can be found in two locations:

- `O`-values
- `M`-rows

The O-values contain the essential data of the metadata. There are two kinds of O-rows; they can be distinguished by the key's last character. If the last character is an 'o', the value is an onode data structure, optionally followed by the start of the extent map. If the last character is an 'x', the value contains the continuation of the extent map. For an object there is always one key with o-suffix and zero or more keys with x-suffix.

The extent map is a list of logical extents of the object. An empty object does not have an extent map. Non-empty files have one or more logical extents. A logical extent has an offset and length within the object and contains a list of one or more physical extents. A physical extent has an offset and length within the OSD.

O-values with o-suffix contain onode data structures. Table 3.4 shows the general structure of these data structures. The onode data structure is optionally followed by the beginning of the extend map. O-values with x-suffix contain the continuation of the extent map.

Type	Description
Header	Data structure header
varint	OID
varint	Size
dict(string, bufferlist)	Dictionary with extended attribute names and their encoded contents
int8	Flags
list(shard info)	Information about extent map
varint	Expected object size
varint	Expected write size
varint	Flags

Table 3.4.: Onode data structure

Most O-values contain the extended attributes _ and snapset, but other extended attributes can exist, too.

The extended attribute _ contains information such as size, mtime, OID and pool of this object.

The extended attribute `snapset` contains information about snapshots and exists even if there are no snapshots.

`M`-rows contain further information about an object. The key contains the OID followed by the type of information. The value contains encoded data structures.

The way the different metadata sets are related in the KV store is shown in Figure 3.2.

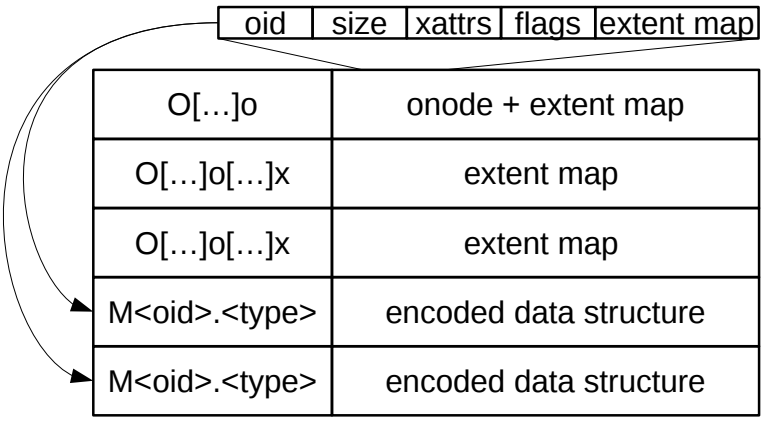

Figure 3.2.: Metadata relationships in the KV store

Slack Space Objects are created with a size that is a multiple of the minimal allocation size. But in a lot of cases an object does not have a size of a multiple of the minimal allocation size. Therefore, it will not fill a full physical extent with data, but it will leave slack.

Tests showed that physical extents are filled in steps of 4096 bytes. This means that if the length of the content is not a multiple of 4096, Ceph will append as many zero-bytes as needed to reach a length that is divisible by 4096.

If the minimal allocation size is M and an object has a size of s bytes ($s \bmod M \neq 0$) and the metadata contains a list of physical extents with a total size of p bytes ($p \bmod M = 0$), then the last $p - s$ bytes are slack space. $(p - s) \bmod 4096 = z$ is

the number of appended zero bytes for the 4096 bytes-alignment. Hence, only the last $p - s - z$ bytes may contain data of a previous object in the last physical extent.

Metadata Lookup Metadata of certain objects can be found by reading all `O`-values and `M`-rows of the KV store.

Logical File Viewing After finding the metadata of an object the allocated physical extents can be identified by interpreting the extent map of this object. The content, slack space, and the matching metadata can be extracted for further analysis.

File Name Category

OSDs do not store files but objects. However, every object has a unique name that has a similar purpose as a file name. The names are stored in `O`-keys.

Depending on the purpose of the object, the name follows a certain naming scheme. If the object is part of an RBD, the name prefix is `rbd_data.`; osdmap names start with `osdmap.` or `inc_osdmap.`, etc. Since data that logically belongs together can be split over more than one object, the name in the KV store often also contains an ID. This ID indicates in which order the objects have to be read from the object store.

Table 3.5 shows examples for the naming schemes of different object types. The objects are explained later. There is also data that is not split into more than one object, for example `osd_superblock` or `rbd_info`. Therefore, the object names do not contain IDs.

Prefix	Example	Description
`osdmap`	`osdmap.5`	osdmap of Ceph epoch 5
`inc_osdmap`	`inc_osdmap.10`	inc_osdmap of epoch 10
`rbd_data`	`rbd_data.2330174b0dc51.0000000000000002`	Third block of the RBD with RBD ID 2330174b0dc51
`<CephFS inode>`	`1000000eb98.00000000`	First object of the CephFS file with inode number 1000000eb98

Table 3.5.: Example of Ceph object naming schemes.

An `O`-key contains the following information:

- Shard
- Pool ID
- Hash
- Namespace
- Key / Name, as escaped string
- Snapshot ID
- Generation

Since values of `O`-keys with `o`-suffix contain onode data structures (Table 3.4), the look-up of the matching OID is simple.

File Name Listing A list of object names and the corresponding metadata can be generated by reading all `O`-rows and `M`-rows.

Application Category

OSDs store a lot of additional application data depending on the purpose of the Ceph cluster and enabled features, for example:

- CephFS
- RBD
- RGW
- osdmap

These applications store additional application-specific metadata in the KV store and additional objects with specific content.

osdmap If the osdmap feature is activated in Ceph, the ceph cluster will write a log of events, and status and topology information to the OSDs. This log contains information about IP addresses of other servers, status changes – for example when an OSD or server disappears or reappears –, snapshots, pools, and much more.

The osdmap consists of three different types of objects:

1. `osd_superblock`

2. `inc_osdmap.<epoch>`

3. `osdmap.<epoch>`

The content of these objects are encoded data structures.

Per OSD there is one `osd_superblock` object that contains basic information about the OSD and the OSD's initial state.

The `osdmap` and `inc_osdmap` objects are updates to the OSD's initial state. They are created alternatingly, starting with `inc_osdmap.1` and `osdmap.1`. The osdmap objects are created when the cluster reaches a new epoch. The `inc_osdmap` objects contain updates between two epochs. When a topology or configuration change is detected, a new `inc_osdmap` object is created [63]. Figure 3.3 schematically shows when objects are created.

Older osdmaps are not deleted immediately. They are referenced by the KV store, but after some time the KV entries may be removed and the space deallocated. Nevertheless the encoded data remains on the OSD as long as it is not overwritten by new data.

The encoded osdmaps contain the epoch. Therefore, it is possible to scan the whole OSD for osdmap and inc_osdmap objects, to bring them into the right chronological order and to create a timeline of events.

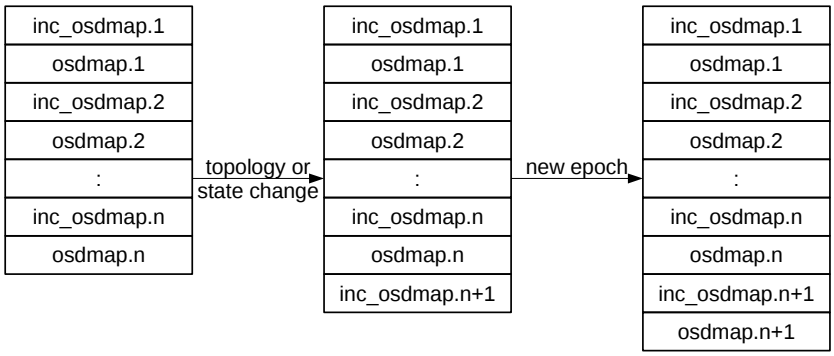

Figure 3.3.: Osdmap and inc_osdmap objects are created alternatingly. On disk they do not appear in this strict order. The location of the objects is determined by the bitmap allocator.

OSD Superblock The OSD superblock starts at offset 0x10000 of the OSD. The position can also be found in the KV store. See also Appendix B.3 for a hexdump of an OSD superblock. The data stored in the superblock is described in Table 3.6.

Type	Description
Header	Data structure header
UUID	Cluster UUID
int32	Whoami
int32	Current epoch
int32	Oldest map
int32	Newest map
float	Weight
int64	Features mask
dict(int64, string)	List of features, mask to name mapping
int32	Clean thru
int32	Mounted
UUID	OSD UUID
int32	Last epoch marked full
int32	Number pool last epoch marked full

Table 3.6.: OSD superblock

osdmap Objects osdmap objects consist of two blocks which have their own data structure headers. At the very end of the object Ceph encodes a CRC32 checksums. The general layout is described in Table 3.7 – the two different blocks are shown in Tables 3.8 and 3.9. These blocks contain information such as the epoch and time of the epoch,

pools in the cluster and their names, states and weights of all OSDs, IP addresses and ports of all OSDs, and the crushmap of this epoch.

Type	Description
Header	Data structure header
Block 1	Table 3.8
Block 2	Table 3.9
int32	CRC

Table 3.7.: General structure of an osdmap object

Type	Description
Header	Data structure header
UUID	Cluster UUID
int32	Epoch
utime	Creation time of OSD. Is the same across all osdmap objects.
utime	Modification time of the osdmap object.
dict(int64, pgpool)	Pools in the cluster.
dict(int64, string)	Names of the pools in the previous dict.
int32	The highest pool number.
int32	Flags.
int32	The highest OSD number.
list(int32)	States of all OSDs.
list(int32)	Weight of all OSDs.
list(entity address)	IP adresses including port of all OSDs.
dict(pg, list(int32))	PG temp
dict(pg, int32)	primary temp
list(int32)	OSD primary affinity
Bufferlist	Encoded crushmap
dict(string, dict(string, string))	erasure code profiles
dict(pg, list(int32))	pg upmap
dict(pg, list(pair(int32, int32)))	pg upmap items
int32	Version of crushmap.
dict(int64, int32)	new removed snaps
dict(int64, int32)	new purged snaps

Table 3.8.: osdmap Block 1

Type	Description
Header	Data structure header
list(entity address)	IP addresses including port of all OSDs (cluster internal communication).
list(osdinfo)	OSD information of all OSDs. The data structure encodes for example the epochs of the last time the OSD came up, went down or was in a clean state.
dict(entity address, utime)	blacklist map
list(entity address)	IP addresses including port of all OSDs.
int32	cluster snapshot epoch
string	cluster snapshot
list(uuid)	UUIDs of all OSDs.
list(osdxinfo)	Extended OSD information of all OSDs. This data structure encodes for instance the timestamp of the last time the OSD went down.
list(entity address)	IP addresses including port of all OSDs (client access).
int32	Nearfull ratio: Indicates the fill level compared to the nearfull threshold.
int32	Full ratio: Indicates the fill level compared to the maximum size.
int32	Backfillfull ratio: Indicates the fill level compared to the backfillfull threshold.
int8	Required minimum version of the client.
int8	Require OSD release.
dict(int64, int32)	Queue of snapshots to remove.

Table 3.9.: osdmap Block 2

inc_osdmap Objects inc_osdmap objects also consists of two blocks. After the second block there are two CRC32 checksums. The general layout is described in Table 3.10 – the two different blocks are shown in Tables 3.11 and 3.12. These blocks contain information such as the epoch, crushmap, new pools and their names, and states and weights of OSDs.

Type	Description
Header	Data structure header
Block 1	Table 3.11
Block 2	Table 3.12
int32	CRC
int32	CRC

Table 3.10.: General structure of an inc_osdmap object

Type	Description
Header	Data structure header
UUID	Cluster UUID
int32	Epoch
utime	Modification time
int64	new pool max
int32	new flags
Bufferlist	Inline encoded osdmap object
Bufferlist	Encoded crushmap
int32	new max osd
dict(int64, pgpool)	new pools
dict(int64, string)	new pool names
list(int64)	old pools
dict(int32, entity address)	new up clients
dict(int32, int32)	new state
dict(int32, int32)	new weight
dict(pg, list(int32, int32))	number new pg temp
dict(pg, int32)	number new primary temp
dict(int32, int32)	new primary affinity
dict(string, dict(string, string))	new erasure code profiles
list(string)	old erasure code profiles
dict(pg, list(int32))	new pg upmap
list(pg)	old pg upmap
dict(pg, list(pair(int32, int32)))	new pg upmap items
list(pg)	old pg upmap items
dict(int64, string)	new removed snaps
dict(int64, string)	new purged snaps

Table 3.11.: inc_osdmap Block 1

Type	Description
Header	Data structure header
dict(int32, entity address)	new hb back up
dict(int32, int32)	new up thru
dict(int32, pair(int32, int32))	new last clean interval
dict(int32, int32)	new lost
list(entity address)	new blacklist
list(entity address)	osd blacklist
dict(int32, entity address)	new up cluster
string	cluster snapshot
dict(int32, uuid)	new UUID
dict(int32, osdxinfo)	new extended info
dict(int32, entity address)	new hb front up
int64	encode features
float	new nearfull ratio
float	new full ratio
float	new backfill ratio
int8	new require min comat client
int8	new require osd release

Table 3.12.: inc_osdmap Block 2

3.3.2. BlueFS

BlueFS is a simple file system used only to store the KV store (RocksDB database).

It consists of the superblock at offset 0x1000, a transaction log and the actual data extents. BlueFS does not store the current state of the file system, but simply replays the transaction log to restore the file system state in-memory when Ceph is restarted. When any activity on the file system happens, the new transactions are appended to the on-disk transaction log.

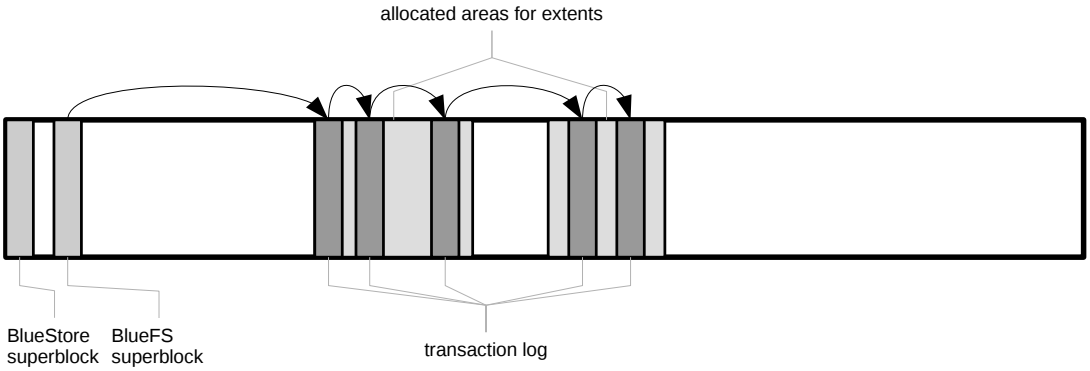

Figure 3.4.: Overview about BlueFS data structures on disk. Areas not used for BlueFS are used to store RADOS objects.

BlueFS can only create first-level directories. There is no possiblity to create subdirectories.

The following section describes a central data structure of BlueFS – the transaction log. It is described before the analysis using the category scheme by Carrier because it will be referenced in the different categories.

Transaction Log

The transaction log is a central data structure of BlueFS. It contains a list of transactions that were applied to the file system. When the Ceph OSD daemon is started, the transaction log has to be replayed to get the latest status of the file system.

Transactions are sets of one or more operations that happened on the file system. Table 3.13 shows all possible operations in a transaction [64, lines 144 ff].

The superblock points to the physical extent storing the transaction log. This physical extent is divided into smaller blocks (transactions). Each holds a list of operations.

The transactions can be read by sequentially jumping from one block to another. If a valid header is found, the list of transactions can be replayed. If there is no valid data, the end of the transaction log is reached. The structure of a transaction is outlined in Table 3.14, while Figure 3.5 illustrates the structure of the transaction log.

Code	Operation	Description
0	NONE	No action
1	INIT	Initial action. First entry of the log
2	ALLOC_ADD	Add an area of the disk to BlueFS
3	ALLOC_RM	Remove an area of the disk from BlueFS
4	DIR_LINK	Add a file to a directory
5	DIR_UNLINK	Remove a file from a directory
6	DIR_CREATE	Create a directory
7	DIR_REMOVE	Remove a directory
8	FILE_UPDATE	Create or update a file, contains an fnode data structure
9	FILE_REMOVE	Remove a file
10	JUMP	While replaying the log, skip a certain amount of transactions
11	JUMP_SEQ	While replaying the log, skip the next transaction

Table 3.13.: BlueFS Operations in the Transaction Log

File System Category

The BlueFS superblock that contains general information about the file system is located at offset 0x1000 to 0x2000 of the OSD (see Table 3.15) [65, lines 325 ff]. By reading the fnode data structure that is located within the superblock, the transaction log can be identified on disk (Table 3.16). See also Appendix B.2 for a hexdump of the superblock.

As an integrity check a CRC32 checksum follows the superblock [66, lines 517 f].

Type	Description
Header	Data structure header
UUID	UUID of the BlueFS
int64	Sequence number of this transaction; increased by one for every new transaction.
int32	Length of the following list of operations (in bytes).
Operation(s)	One or more operations. The decoder tries to decode further operations as long as the length of the operations list (previous field) is not reached.
int32	CRC32 checksum.

Table 3.14.: BlueFS Transaction in the Transaction Log

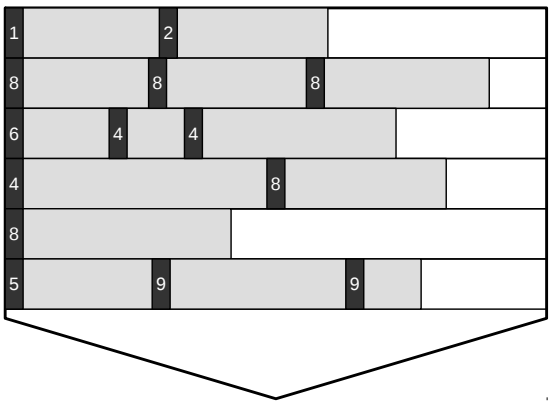

Figure 3.5.: A transaction does not have a fixed length, but it is variable in size. Every operation starts with the operation code and is followed by the payload of the operation. In this example the transaction would be read from top to bottom and the operations from left to right.

There is no backup copy of this superblock. If the superblock is lost, it may be possible to find file system metadata and content by scanning the disk for known data structures.

The test data from the test environment shows that the fnode data structure pointing to the transaction log does not seem to always show all information. For example, the modification time and list of physical extents are not always updated in the fnode data structure. The inode number of this fnode is always `1` in the data from the test environments. This data structure shows that the transaction log is treated as a file on file system level.

The transaction log holds data of the file system category in `ALLOC_ADD` and `ALLOC_RM` operations: They hold information about areas of the OSD that were allocated for or de-allocated from BlueFS. Furthermore, `FILE_UPDATE` operations to inode `1` (the transaction log) hold information about changes to the transaction log, for example when additional extents were allocated for the transaction log.

Type	**Description**
Header	Data structure header
UUID	Filesystem UUID
UUID	OSD UUID
int64	Version
int32	Block size of BlueFS extents
fnode	BlueFS transaction log metadata
int32	CRC32 checksum

Table 3.15.: BlueFS superblock

Type	**Description**
Header	Data structure header
int64	Inode
int64	size
utime	mtime
int8	prefer bdev
list(extent)	List of physical extents on block device

Table 3.16.: BlueFS fnode data structure

Analysis Techniques The file system data from the superblock and – after reading the transaction log – other file system category data like allocated areas can be decoded and checked for consistency.

The slack space of the superblock can be extracted for further analysis.

Since BlueFS is located within the OSD, there is no real volume slack. The area of the OSD that is not part of BlueFS is analyzed using OSD analysis techniques.

Content Category

The content of files in the file system is organized in extents. The extent addresses are absolute addresses within the OSD. Extents of files are located in areas that were allocated via `ALLOC_ADD` operations.

Analysis of the test data shows that the allocation strategy for new extents seems to be: Allocate data after the last-used extent (next-available algorithm). Figure 3.4 illustrates how allocated areas and extents are ordered on an OSD.

There is no on-disk data structure like a bitmap that stores allocation information about used or unused blocks or extents. The information about the allocation status is only available after reading the transaction log. The necessary information is stored in `FILE_UPDATE` and `FILE_REMOVE` operations. `FILE_UPDATE` operations in the transaction log consist mainly of a fnode data structure (see Table 3.16).

Data Unit Allocation Unit The allocation state of data units – extents that are in use or were used before – can be printed after reading the transaction log.

Consistency Checks The consistency of the allocation state can be verified by comparing the allocated extents with the allocated areas.

Metadata Category

The transaction log holds information of the metadata category in the `DIR_`, and the `FILE_` operations.

`DIR_CREATE` creates a directory. There is no disk space or inode allocated for the directory, it is simply addressed by its name. A `DIR_REMOVE` operation marks a directory as deleted.

`FILE_UPDATE` operations contain the inode, size, modification time and extents of a file. The transaction log may store more than one `FILE_UPDATE` operation per inode. Therefore, it is possible to see how a file has changed over time, but without knowing the content of the file at every point in time.

A `FILE_REMOVE` operation marks an inode as removed.

BlueFS does not compress or encrypt data. Therefore, there is no need to decrypt or decompress data.

Slack Space The slack space of files stored in BlueFS can be extracted for further analysis. The slack can, for example, hold data from deleted RocksDB files.

Metadata Lookup After reading the transaction log, all used inodes are known. Per inode the latest `FILE_UPDATE` operation contains the file's metadata in an fnode data structure.

Logical File Viewing The latest `FILE_UPDATE` data structure contains the list of extents that have to be read to get a file's content.

Unallocated Metadata Analysis Since the transaction log stores older transactions it is possible to view metadata entries that refer to deleted files or to files that were updated in the meantime.

Consistency Checks There are three ways to execute a consistency check of the metadata in the transaction log:

1. The transaction log entries contain a CRC checksum which are computed per entry.
2. Every transaction log entry contains the UUID of the BlueFS which has to match the superblock's UUID.
3. The sequence number of every transaction must be greater than the previously read transaction.

File Name Category

The information of the file name category is stored in the transaction log in `DIR_LINK` operations. Such an operation consists of three values: directory, file name, and inode. Afterwards, the file with this inode can be accessed under this file name in this directory.

A `DIR_UNLINK` unlinks the inode from the directory. The file remains on disk, but it is not accessible via this name anymore.

File Name Listing After reading the transaction log, the names of files can be listed. Not only current file names can be listed, but file names of deleted files and their metadata can also be found in the transaction log.

Application Category

There is no additional data that can be referred to as application category. Even though the transaction log could be interpreted as a journal, it does not seem justified to count it to the application category. In Carrier's definition of the application category the application data must be "not essential to the file system" [23, p. 205]. In BlueFS, however, the transaction log is crucial for reading the file system.

3.3.3. Rados Block Device (RBD)

RBD is an application on top of a Ceph cluster (see also Figure 2.1). Storage consumers can map an RBD using a Linux kernel module to get access to a block device. This RBD is created with a certain size and consists of a set of objects that reside in the Ceph cluster. The client sees a standard Linux block device – like a hard drive – and can create, for example, a file system on this block device.

The Ceph cluster creates several objects for an RBD:

- `rbd_data.<id>.<block>` objects
- `rbd_header.<id>` object
- `rbd_object_map.<id>` object
- `rbd_id.<name>` object
- `rbd_info` object
- `rbd_directory` object

The objects `rbd_info` and `rbd_directory` are created once per RADOS, while the other objects are created for each RBD.

File System Category

The metadata that belongs to the file system category is not stored in a superblock, but as metadata in the KV store.

The `rbd_header.<id>` object is an object without content. Its name contains the unique ID of the RBD and the metadata is stored in `M`-keys. (See also Chapter 3.3.1.)

Listing 3.5 shows decoded data of the `rbd_header` object of an RBD.

While creating an RBD the administrator assigns a name to the RBD. This name is stored in an object with name `rbd_id.<name>`. The content of the object is just the ID

```
Key: shard: 0x-1, key: rbd_header.2330174b0dc51, name: rbd_header.2330174b0dc51, ↩
    ↪poolid: 0x1, snap: 0xfffffffffffffffe, gen: 0xffffffffffffffff
Value:
oid: 5191, object_size: 0, shards:
   snapset: snapid: 0x0, snaps: Number of elements: 0 (0x0), clones: Number of ↩
    ↪elements: 0 (0x0)
_lock.rbd_lock: desc: , type: 1, tag: internal
         _: size: 0x0, mtime: 2018-03-30 14:42:22.971968445, soid: key: , oid: ↩
    ↪rbd_header.2330174b0dc51, nspace: , pool: 0x1

Additional Metadata from KV Store (M prefix)
create_timestamp: 2018-03-30 11:17:25.190355644
features: 1 (0x1)
flags: 0 (0x0)
object_prefix: rbd_data.2330174b0dc51
order: 22 (0x16)
size: 52428800 (0x3200000)
snap_seq: 0 (0x0)
```

List of Listings 3.5: Decoded data of the file system category of an RBD.

of the RBD encoded as a string. There is also the `rbd_directory` object. It is an object without content, but there is metadata in `M`-rows to map RBD IDs to RBD names and vice versa.

Since this metadata is stored in form of objects which are distributed using the CRUSH algorithm, it cannot be found on every OSD. Therefore, it may happen during an analysis of a single OSD that the existence of an RBD is known, but its name, size, or creation date remains unknown.

Content Category

The data that the client writes to the block device is located in the `rbd_data` objects. The name has the format `rbd_data.<rbdid>.<block>`. The RBD space is lazily allocated: The corresponding object is only created when a client writes data to a block of the RBD for the first time. The total size of the `rbd_data` objects cannot exceed the size of the RBD.

As RBDs can have a size of several GiB or TiB, there are a lot of `rbd_data` objects. Not all of them will be located on the same OSD, leading to the situation that only portions of a block device can be reconstructed. A useful analysis of the contents of an RBD might not be possible.

On the other hand, by using the metadata information of the `rbd_data` objects it is possible to get the modification time of a certain area of the RBD. This is in contrast to a physical block device like a hard drive, where it is not possible to tell when a certain block was last accessed.

The data that tracks RBD block allocation status seems to be stored in the `rbd_object_map` object [67], but in the data from the test environment this object does not contain data.

File Name Category and Metadata Category

There is no data on a raw block device that matches the definition of the file name or metadata category.

Application Category

The `rbd_header` object holds information about the lock status of an RBD. If a client mounts an RBD the lock is written to the `rbd_header` object as an extended attribute `_lock.rbd_lock`.

3.3.4. CephFS

CephFS is a POSIX-compatible file system that can be configured on top of a Ceph cluster (see also Figure 2.1). It is not to be confused with BlueFS.

CephFS is an application that extends RADOS and stores files and directories as objects within the object store.

CephFS stores files and directories as objects in two pools in the Ceph cluster. One pool is used for data, i.e. files, and the other pool is used for metadata, i.e. directories.

Furthermore, CephFS needs at least one server that is configured as MDS (metadata server). MDS store additional metadata, organize the CephFS data and serve clients with information which objects to access. Clients need access to at least one MDS, but then retrieve file contents directly from the server on which the objects are stored.

File System Category

There is no superblock or a similar data structure on disk. While BlueFS and the OSD store this data in a block at a known address, CephFS stores the information about its presence in RADOS. This means that the information is stored in replicas on different OSDs' KV store. This implies that – when only analyzing a single OSD – there might be no evidence of a certain category although one sees evidence of other categories.

Content Category

The content of files is stored in objects. Therefore, the allocation status of blocks of the underlying block device can be extracted from the bitmap. But without further analysis it is not possible to determine if an allocation unit is used for CephFS or for another use case, such as RBD.

```
Key: shard: 0x-1, key: 1000000eb98.00000000, name: 1000000eb98.00000000, poolid: 0x2, ↩
    ↪snap: 0xfffffffffffffffe, gen: 0xffffffffffffffff
Value:
oid: 5625, object_size: 0, shards:
   _parent: inode: 0x1000000eb98 -> ancestors: ino: 0x1000000eb6b, dname: factsheet, ↩
    ↪ver: 0xfe->ino: 0x1000000eb47, dname: components, ver: 0x1fc->ino: 0↩
    ↪x1000000c0d7, dname: ushell, ver: 0xc1c->ino: 0x1000000c09e, dname: sap, ver: 0↩
    ↪x155->ino: 0x1000000c09d, dname: resources, ver: 0x13f->ino: 0x1000000bf9d, ↩
    ↪dname: ui5, ver: 0x9d->ino: 0x1000000bf9b, dname: sap, ver: 0x67->ino: 0↩
    ↪x1000000bf99, dname: resources, ver: 0xcc->ino: 0x1000000a8dc, dname: ↩
    ↪hdbcockpit, ver: 0x484->ino: 0x1000000a8db, dname: hdb, ver: 0x10b->ino: 0↩
    ↪x1000000a8d9, dname: global, ver: 0x1cc->ino: 0x1, dname: SES, ver: 0x921c3, ↩
    ↪pool: 0x2
   _layout: objectsize: 0x0, poolid: 0xffffffffffffffff, pool_ns:
         _: size: 0x0, mtime: 2018-03-21 10:31:58.654878684, soid: key: , oid: 1000000↩
    ↪eb98.00000000, nspace: , pool: 0x2
   snapset: snapid: 0x0, snaps: Number of elements: 0 (0x0), clones: Number of ↩
    ↪elements: 0 (0x0)
```

List of Listings 3.6: O-row of a directory object

Metadata Category

File system metadata – inode and directory information – is stored in a distributed fashion in the KV store. That means, not every OSD holds the complete metadata information but just a subset. The metadata stored on an OSD does not necessarily refer to files stored as objects on the same OSD.

Each CephFS directory and CephFS file is an object in the object store (see Figure 3.6); and therefore, it has metadata stored in `O`-rows. The object name in the `O`-key is the inode number of this directory or file. The `O`-value holds additional metadata in form of extended attributes, e.g. information about the full path in the CephFS including the file or directory name, parent directory names, and parent directory inode numbers.

The decoded metadata of a file may look like in Listing 3.6. One can see that the full path, including inode number of every parent directory, is included in the `_parent` attribute.

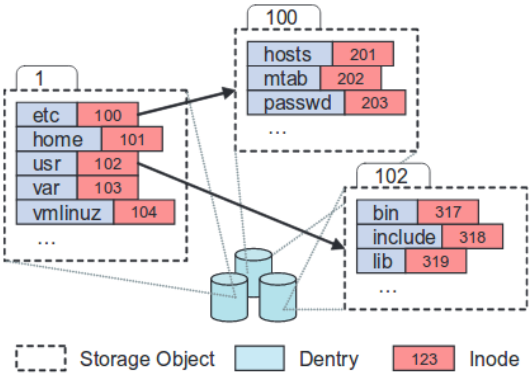

Figure 3.6.: Every directory containing file names is stored as object [2, p. 38]

If an object is a CephFS directory it has additional metadata in the `M`-rows. The matching `M`-rows can be looked up via the OID that is stored in `O`-values (see also Figure 3.2). There are two kinds of metadata stored for a CephFS directory: directory information about itself and child information about files and directories that are located within this directory.

Directory information Metadata of directories is stored in `M`-rows where the key ends with "-". The key also contains the directory's OID.

Every directory metadata `M`-value starts with an fnode data structure. It is different to the BlueFS fnode data structure. The CephFS data structure stores for example the number of files and subdirectories of this directory.

Child information Metadata of children is stored in `M`-rows where the key ends with `_head`. The key also contains the `OID` of the directory and the child's file or directory name: `M<oid>.<filename>_head`

The `M`-value contains an inode data structure that stores for example: Inode number (This inode number is the name of the object that contains this file or directory.), size (0 for directories), creation time, mode, UID, GID, links (for hardlink support), and name.

```
-: fnode: fragstat: mtime: 2018-03-21 10:31:04.911033000, nfiles: 3, nsubdirs: 7, ↩
    ↪change_attr: 10, rstat: rbytes: 430060, rfiles: 27, rsubdirs: 7, rctime: ↩
    ↪2018-03-21 10:31:04.919033000
Component-dbg.js_head: inode: 0x1000000eb99, size: 1782, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.887033000, mode: 100440, uid: 111, gid: 479, nlink: 1, dir_layout: 0x0
Component.js_head: inode: 0x1000000eb9a, size: 1026, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.887033000, mode: 100440, uid: 111, gid: 479, nlink: 1, dir_layout: 0x0
annotation_head: inode: 0x1000000eb9b, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.891033000, mode: 40750, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
controls_head: inode: 0x1000000eb9e, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.899033000, mode: 40750, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
css_head: inode: 0x1000000eba9, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.899033000, mode: 40550, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
factory_head: inode: 0x1000000ebab, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.903033000, mode: 40750, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
resources.json_head: inode: 0x1000000ebae, size: 1683, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.903033000, mode: 100440, uid: 111, gid: 479, nlink: 1, dir_layout: 0x0
tools_head: inode: 0x1000000ebaf, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.907033000, mode: 40750, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
views_head: inode: 0x1000000ebb2, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.911033000, mode: 40750, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
vocabularies_head: inode: 0x1000000ebb7, size: 0, rdev: 0, ctime: 2018-03-21 ↩
    ↪10:31:04.919033000, mode: 40750, uid: 111, gid: 479, nlink: 1, dir_layout: 0x2
```

List of Listings 3.7: M-rows of a directory object

The suffix `_head` is used to identify the current version of file metadata. If a snapshot was created the suffix contains the snapshot ID.

The decoded metadata of a directory and its children may look like shown in Listing 3.7.

Metadata Lookup An investigator may find an inode number of a CephFS file or directory and wants to find the matching metadata.

The following steps have to be taken: Lookup the `O`-rows of the object where the object name is the inode number. The `O`-value then contains the first part of the metadata. By using the OID in the `O`-value it is possible to look up the metadata in the `M`-rows.

Logical File Viewing After executing a metadata lookup as described before, the physical extents allocated for an object are known. Therefore, the content of an object can be extracted for further analysis.

File Name Category

The information of the file name category is stored in the `O`-values as extended attributes to the object metadata.

Files can be split in more than one object. The object with ID 1 holds an extended attribute `_parent`. It contains

- inode: its own inode number, which is the same as its object name
- list of ancestors:
  - inode: ancestor's inode number
  - dname: the name of the file or directory within its parent directory

By reading this information it is possible to build the full path of a file within CephFS. It is also possible to gather some information about directories even though their metadata is not stored on the same OSD.

File Name Listing The `M`-rows can be used to get the names, inode numbers, and other metadata of files and subdirectories in a directory. By using the inode number it is possible to get the object metadata, such as allocated extents.

The inode number of the root directory is `1` [68, line 43].

File Name Searching A specific file name can be searched by reading the `O`-rows and `M`-rows of the KV store and searching for this specific name.

Application Category

CephFS uses certain objects to store application category data [68, lines 51 ff].

Depending on the content CephFS uses different prefixes:

- Prefixes `100.` and `600.`: Inode information.
- Prefixes `200.`, `300.`, `400.` and `500.`: Journaling data.

There are additional objects like the `mds<id>_inotable` objects that store information about free inodes per MDS [69, lines 1142 ff].

3.4. Cache Tiering

Cache tiering is a built-in functionality of Ceph to improve the I/O performance of certain workloads while at the same time saving hardware costs. If a cache tier is active, Ceph determines *hot* and *cold* data. Hot data are objects that were accessed or modified within a certain timespan, for instance within the last two minutes. Ceph stores these objects on fast storage devices so that further access operations to them have a fast response time. Cold data are objects that were not accessed for a longer period of time. Ceph transfers these objects to slower storage devices. The intention behind cache tiering is to reduce hardware costs by having a large storage of slow, but inexpensive storage devices, for example HDDs, while making use of a smaller and faster storage of more expensive devices such as SSDs. The data is transferred between slow and fast storage as needed when data becomes hot or cold [30, 70].

To configure cache tiering one needs to modify the crushmap and define different buckets: One bucket contains the fast storage for hot data (cache tier), one bucket contains the slow storage for cold data (base tier). The replication and redundancy settings for both tiers are configured using CRUSH rulesets [70].

Cache tier and base tier OSDs are normal OSDs. They are not formatted in a different way than OSDs in a non-cache tiering setup. This implies that the results of the previous chapters can be applied to these OSDs, too. OSDs from the cache tier hold objects that were modified or accessed more recently, while OSDs from the base tier hold older objects.

Whether cache tiering is or was active in a Ceph cluster can be determined by analyzing the crushmaps extracted from osdmap objects.

4. Implementation of Vampyr

Vampyr (from *Vampyroteuthis infernalis*, vampir squid, a *ceph*alopod) is the implementation of the findings described above. The development was part of the thesis. It is a Python-based command-line software for Linux, available at `https://github.com/fbausch/vampyr` under the Apache License 2.0.

Vampyr analyzes Ceph BlueStore OSDs, and extracts and displays data of the five different data categories.

4.1. General Design

Vampyr consists of two Python programs: `vampyr.py` and `vampyr-rebuild.py`.

`vampyr.py` is the tool for analyzing and extracting data from an OSD. `vampyr-rebuild.py` can be used to put chunks of RBDs, CephFS files and objects together.

Vampyr was implemented in Python for rapid development. It does not use Ceph libraries or Ceph code in order to rule out side effects that could potentially occur by using them. Overall, Vampyr consists of about 4500 lines of Python code.

In order to analyze the RocksDB-based KV store Vampyr needs the `ldb` tool which is part of RocksDB. `ldb` extracts all database rows stored in the RocksDB files.

4.1.1. Structure

Vampyr consists of different Python modules:

- `vampyr.py`: Contains the logic to parse the command-line arguments. Depending on the command-line arguments it makes use of different modules to extract or present data.

- `vampyr/datatypes.py`: Contains the decoders for basic data types, such as integers, strings, lists or dictionaries.

- `vampyr/osd.py`: Contains the logic to analyze OSDs, for instance the decoder for the BlueStore superblock. Loads BlueFS and the KV store using the respective modules.

- `vampyr/bluefs.py`: Contains the logic to analyze BlueFS, read the transaction log and extract the BlueFS content.

- `vampyr/kv.py`: Contains the parsers for the KV datasets. It also contains the logic to extract object content and metadata.

- `vampyr/decoder.py`: Contains decoders for encoded object content, such as osdmap and inc_osdmap.

- `vampyr/exceptions.py`: Defines Vampyr-related exceptions.

- `vampyr-rebuild.py`: Rebuilds files or RBDs using extracted objects. It does not have dependencies to other modules.

The central Vampyr class is `OSD` (Figure 4.1). It gives access to the BlueStore label, the KV store and BlueFS. The class for accessing the KV store is called `RDBKV` (for RocksDB KV store) and has an attribute for each of the KV prefixes. For example, the attribute `pO` handles all `O`-rows. The class `BlueFS` allows accessing the BlueFS superblock, the transaction log, and the directories and files.

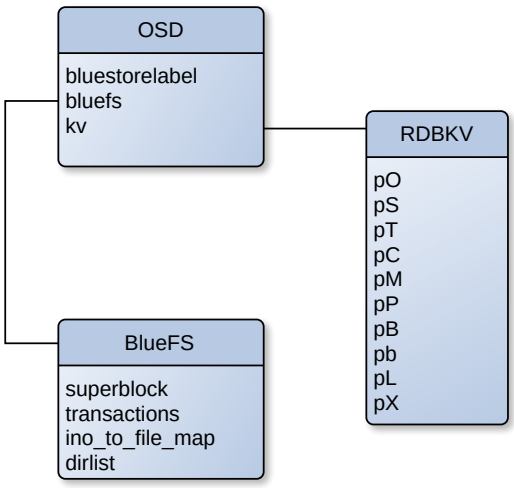

Figure 4.1.: General overview

The class `BlueFSSuperblock` has an attribute `log_fnode` – an instance of `BlueFSFNode` – which points to the transaction log (Figure 4.2). The class `BlueFSFNode` holds the list of allocated physical extents, inode number, size and modification time.

The superblock also holds a list of transactions (`BlueFSTransaction`). Each transaction contains a list of operations (`BlueFSOperation`). Each operation holds the code of the operation and operation-specific attributes, e.g. `file` (an `BlueFSFnode` instance) for `FILE_UPDATE` operations.

The dictionary `ino_to_file_map` contains all allocated inode numbers and the corresponding `BlueFSFiles`. The list `dirlist` contains a list of `BlueFSDir` objects. Each instance holds the directory name and a directory-specific mapping of inode numbers to file names.

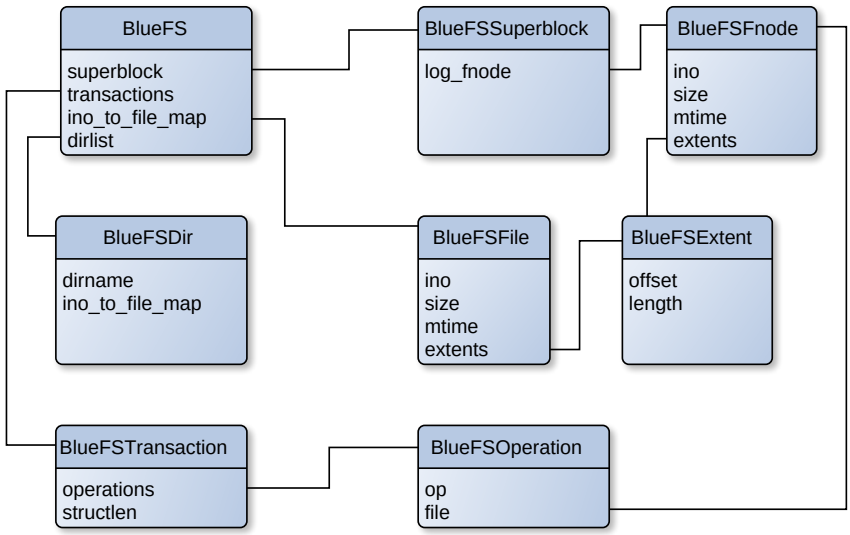

Figure 4.2.: BlueFS overview

Figure 4.3 shows how the `RDBKV` class accesses data. Because of the large amount of implemented classes, this figure only shows a part of the classes. The class `PrefixHandlerO` parses all `O`-rows. `KVOnode` objects hold the data of the `O`-values, including OID, object size, extended attributes, and all logical extents. The dictionary `onode_map` contains the mapping of `O`-keys to values, while the dictionary `oid_map` contains a mapping from OID to `O`-row. The different pool IDs found in the `O`-rows are collected in `poolids`.

The logical extents (`LExtent`) have the attributes `logical_offset` indicating the offset within the object, `length`, and `blob`. The `blob` is a `KVBlob` object which holds a list of `CephPExtent` objects. The `valid` attribute indicates, whether the physical extent holds data or has to be skipped while reading data.

The `PrefixHandlerMP` class handles both, `M`-rows and `P`-rows. However, the attributes `pM` and `pP` of `RDBKV` are different instances of this class. The `meta_map` attribute is a dictionary mapping OIDs to the metadata belonging to the object of this OID.

The `header_map` attribute maps OIDs to `KVFnode` objects, if there is CephFS directory metadata. The `dentries` attribute is a dictionary mapping file names to the `KVInode`

objects of the directory content. The `inode_map` attribute of the `PrefixHandlerMP` class holds a dictionary that maps all known inode numbers to the corresponding `KVInode` objects. The `KVInode` objects hold inode number, creation time, mode, GID, UID, and size of the file or directory.

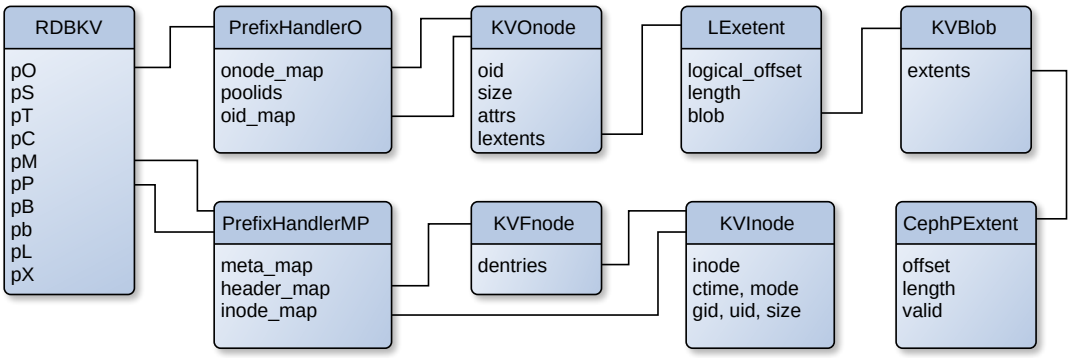

Figure 4.3.: KV overview

4.1.2. Parsing and Reading Data

The workflow to parse and read data implemented in Vampyr is as follows:

1. Decode and validate the BlueStore label.
2. Load BlueFS data:
   a) Decode and validate the BlueFS superblock.
   b) Initialize the `ino_to_file_map` with the data of the transaction log.
   c) Read and decode the transaction log.
3. Extract the content of BlueFS to a temporary directory.
4. Parse and load the data using `ldb`:
   a) Sort the rows alphabetically and by prefix.

b) When data from rows of a certain prefix is needed, parse all datasets with this prefix:

i. Call the prefix handler's `parse_dataset` method for every dataset.

ii. Build the prefix handler's data structures with the parsed data.

4.1.3. Extracting Data

The workflow to extract object data after reading and parsing the KV store is shown here:

1. Loop over all datasets (Ceph objects) stored in the `onode_map` of `PrefixHandlerO`:

a) Get the OID and the `KVOnode` object of this dataset.

b) Check, if the object contains decodable data, for example osdmap and inc_osdmap objects, and decode the data, if possible.

c) Write the decoded data to a file.

d) Extract the object content, slack space, and compute the MD5 checksum of the object content.

i. Loop over all physical extents of this object and read them.

ii. The size of the object content is stored in the `size` attribute of the `KVOnode` class. The first `size` bytes read previously are the content of the Ceph object. The remaining bytes are the slack space.

e) Store the content, slack space, and MD5 checksum in files.

f) If the Ceph object has a `mtime` defined in the extended attributes, set the `mtime` of the file storing the content to this value.

g) Write the content of the dataset (data of the metadata and file name category) to a file.

h) Look up additional metadata from the `meta_map` of `PrefixHandlerMP` using the OID.

i) Write this metadata to the metadata file.

2. Loop over all datasets stored in the `meta_map` of `PrefixHandlerMP`.

a) Write all metadata per OID to a file.

b) Use the OID to look up the object information in the `oid_map` of `PrefixHandlerO`.

c) Use the CephFS related metadata to create symlinks to parent and child directories and files.

4.2. Functionality and Options of vampyr.py

`vampyr.py` tries to implement the results of the research outlined in the earlier chapters. It executes basic consistency checks using the data structure headers. After decoding a data structure the decoder checks whether the number of read bytes matches the data structure length.

In many cases fields of data structures contain values that are predictable, for instance UUIDs at certain positions. These values are checked using assertions. If such a field holds a value that is not as predicted, `vampyr.py` detects it.

Vampyr reads the BlueStore label and checks if the size of an OSD is smaller than the containing image. Afterwards, it reads the BlueFS superblock to locate the transaction log. Then it reads the transaction log and restores the BlueFS state. Both the transaction log and the BlueFS state can be displayed. The file contents are extracted to a temporary directory and loaded using the `ldb` tool.

The BlueFS content can also be extracted to a user-defined directory for manual analysis: Superblock slack space, directories and files including mtime timestamp, and file slack spaces are extracted to this directory.

Vampyr tries to decode all datasets of the KV store. Depending on the command-line parameters Vampyr executes different metadata lookups to extract objects and their metadata, print bitmap and allocation state information, and decode osdmap information.

The command-line parameters are:

--image The `--image` parameter is used to provide a file that holds an image of an OSD.

--offset If the actual OSD does not start at the first byte of the image, but is preceeded by, for example, an LVM header, then a user can provide the offset with this parameter.

Vampyr is implemented in a way that all offsets that are part of the output are relative to the offset.

--verbose Turns on verbose output.

--logging The parameter `--logging` takes the values `INFO` or `DEBUG`. Depending on the log level Vampyr prints status and debug information.

--ldb Via this parameter the user can provide the location of the `ldb` executable. If `ldb` is in the `PATH` environment variable, this option can be skipped.

--clear Vampyr extracts data to different directories (see below). If the `--clear` flag is set, these directories are emptied before extracting to them.

--scan If set, Vampyr does not load or analyze the KV store. Instead it simply scans the image for known data structures. In the current implementation these structures are osdmap, inc_osdmap, and osd_super data structures.

The parameter expects a directory. All data structures that could be identified are extracted to this directory.

--bslabel If this parameter is set, Vampyr prints information about the BlueStore superblock. Additionally, it prints metadata from the KV store that also fits into the file system category of the OSD. Listing 4.1 shows that the analyzed OSD for instance has a size of about 99 GiB, its UUID, and the UUID of the BlueFS. Additionally, the KV store data indicate that the minimal allocation size of physical extents is 64 KiB. As suggested by Carrier, the file system category data is presented and the start of the volume slack is computed.

--bfssuper If this parameter is set, Vampyr prints information about the BlueFS superblock. Listing 4.2 shows the superblock data of a BlueFS. It includes the UUID of BlueFS and the OSD, but also the extents of the transaction log.

--bfstx If this parameter is set, Vampyr prints the BlueFS transaction log. This means, every extracted operation in BlueFS is listed. Listing 4.3 shows two transactions of the transaction log. At the start of the line one finds the offset of the transaction on disk, then the sequence number of the transaction. Afterwards, Vampyr lists the operations in the transaction, including the operation code and a summary of the payload.

Vampyr also tries to read transactions in areas that the transaction log skips. If there is any reconstructable transaction, it appears in the "BlueFS Skipped Transaction List".

--xbfs This parameter expects a directory. Vampyr extracts the BlueFS content into this directory. It preserves the timestamps.

--lspes If this flag is set, Vampyr analyzes the physical extents. In order to do this, Vampyr loads the KV, loads all objects and keeps track of all physical extents that are in use by objects. Afterwards, Vampyr knows all allocated and unallocated areas of the OSD and can output them as depicted in Listing 4.4.

```
---------------------------------
BlueStore Superblock Information:
---------------------------------
Start at: 0x0
End at:   0x15c
---------------------------------
bluestore block device
66831e26-3a66-4547-9646-7cd26505a886

OSD UUID: 66831e26-3a66-4547-9646-7cd26505a886
OSD length: 0x18ffc00000 B = ~ 99 GiB
Last used at: 2018-03-04 11:32:04.404952490
Metadata information:
- bluefs: 1
- ceph_fsid: e81f1260-b58f-4f4e-b53a-a21d9b3617c8
- kv_backend: rocksdb
- magic: ceph osd volume v026
- mkfs_done: yes
- osd_key: AQAjy5tatJpMGxAACSpaO6WhtPREOvxEYfmLaQ==
- ready: ready
- whoami: 0
CRC32 checksum: 0xb759e167
---------------------------------
Volume slack starts at offset 0x18ffd00000 of image file
---------------------------------

---------------------------
OSD Metadata from KV Store:
---------------------------
      blobid_max: 174080 (0x2a800)
  bluefs_extents: Number of elements: 1 (0x1)
   freelist_type: bitmap
  min_alloc_size: 65536 (0x10000)
min_compat_ondisk_format: 2 (0x2)
         nid_max: 20507 (0x501b)
   ondisk_format: 2 (0x2)

------------
Statfs Data:
------------
bluestore_statfs: allocated: 0x20000, stored: 0xaf7, compressed_original: 0x0, ↩
    ↪compressed: 0x0, compressed_allocated: 0x0
```

List of Listings 4.1: BlueStore file system category data presented by Vampyr.

```
------------------------------
BlueFS Superblock Information:
------------------------------
Start at: 0x1000
End at:   0x1063
------------------------------
BlueFS UUID: 9a5b96ec-0b8d-4f74-9c43-68fca17311a3
OSD UUID:    66831e26-3a66-4547-9646-7cd26505a886
Version: 3
Block size: 0x1000
Log fnode Information:
- ino: 1
- size: 1048576, 0x100000
- mtime: 1970-01-01 01:00:00.000000000
- prefer block device: 0
- extents:
  - 0xc0ae00000+0x100000 (bdev 1)
    (at ~ 6 GiB offset)
  - 0xc0aa00000+0x400000 (bdev 1)
    (at ~ 6 GiB offset)
CRC32 checksum: 0x5fba1386
------------------------------
```

List of Listings 4.2: BlueFS superblock information presented by Vampyr.

```
------------------------
BlueFS Transaction List:
------------------------
:
0x0000000bffe98000 -> seq: 0x00000099: OP: DIR_UNLINK, dir: db, file: 000025.sst | OP:↩
    ↪ FILE_REMOVE, ino: 39 | OP: DIR_UNLINK, dir: db, file: 000013.sst | OP: ↩
    ↪FILE_REMOVE, ino: 22 | OP: DIR_UNLINK, dir: db, file: 000010.sst | OP: ↩
    ↪FILE_REMOVE, ino: 17 | OP: DIR_UNLINK, dir: db, file: 000004.sst | OP: ↩
    ↪FILE_REMOVE, ino: 8 | OP: FILE_UPDATE, ino: 42, size: 156, mtime: 2018-03-26 ↩
    ↪16:51:53.672210589, extents: 0xc04400000+0x100000
0x0000000bffe99000 -> seq: 0x0000009a: OP: FILE_UPDATE, ino: 42, size: 269, mtime: ↩
    ↪2018-03-26 16:51:53.687759855, extents: 0xc04400000+0x100000
```

List of Listings 4.3: Two transactions extracted by Vampyr from the transaction log.

```
----------------
Allocated areas:
----------------
0x0-0x4000
0x10000-0x750000
0x770000-0xe00000
------------------
Unallocated areas:
------------------
0x4000-0xc000
0x760000-0x10000
0x1570000-0x18fe690000
```

List of Listings 4.4: Vampyr overview over allocated and unallocated areas of the OSD.

--xpes This parameter expects a directory. Vampyr extracts all unallocated areas of the OSD into this directory. Each unallocated area is extracted in its own file. The filenames consist of the offset of the area on the OSD and the length of the area.

Afterwards, the extracted unallocated areas can be analyzed using forensic tools like file carving tools.

--analyzepes Vampyr looks at all unallocated areas of the OSD and analyzes how many 512 KiB blocks are actually empty and how many contain any kind of data. Empty means filled with zeros. It does not analyze whether the data in the blocks is usable in any way.

--lsobjects If this flag is set, Vampyr lists the names of all objects found in the KV store.

The objects are sorted using their prefix, for instance `rbd_data.<rbdid>` and `osdmap`, to make the list shorter and easier to read. Listing 4.5 shows for example that there are inc_osdmap.1 to inc_osdmap.65, but also several other objects, including RBD-related objects.

```
inc_osdmap -> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ↩
    ↪21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,↩
    ↪ 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, ↩
    ↪60, 61, 62, 63, 64, 65
osd_superblock ->
osdmap -> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, ↩
    ↪22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,↩
    ↪ 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, ↩
    ↪61, 62, 63, 64, 65
rbd_data.2330174b0dc51 -> 0000000000000000, 0000000000000001, 0000000000000002, ↩
    ↪0000000000000003
rbd_directory ->
rbd_header -> 2330174b0dc51
rbd_id -> myrbd1
rbd_info ->
rbd_object_map -> 2330174b0dc51
scrub_1 -> 3, c, 11, 13, 14, 16, 1d, 1f, 21, 25, 2a, 2b, 2f, 30, 36, 37, 39, 3b, 3c, 3↩
    ↪d, 40, 44, 47, 49, 4a, 4b, 4d, 50, 57, 5c, 60, 61, 63
snapmapper ->
```

List of Listings 4.5: Vampyr showing the object list sorted by prefix of the object name.

--decobjects If this flag is set, Vampyr tries to decode objects and prints them. This is only usable for objects that are stored as encoded data structures: inc_osdmap, osdmap, osd_superblock, rbd_id. This option should be used with the `objfilter` option.

--objfilter This parameter expects a regular expression. This regex is used as a filter for object names. Objects are only handled if their names match the regex.

--lsbitmap If the flag is set, Vampyr shows the bitmap metadata and the bitmap from the KV store. Listing 4.6, for instance, shows that a bitmap represents 128 blocks and that every block represents 64 KiB. The bitmap list consists of the physical offset to which the bitmap refers and the actual bitmap. `1` means that the block is marked as used.

--xbitmap This parameter expects a directory. Vampyr extracts all blocks that are marked unallocated in the bitmap to this directory. Consecutive unallocated blocks is

```
----------------
Bitmap Metadata:
----------------
blocks --> 0x190000
blocks_per_key --> 0x80
bytes_per_block --> 0x10000
size --> 0x18ffc00000
--------
Bitmaps:
--------
0x0000000000000000 --> 00000000000000000000000000000000000000000000000000 ↩
    ↪000000000000000000000000000000000000000000000000000000000000000000 ↩
    ↪0000100000000000000
0x0000000000800000 --> 00000000000000000000000000000000000000000000000000 ↩
    ↪000000000000000000000000000000001111100011111111111111111111111111 ↩
    ↪1111111111111111111
```

List of Listings 4.6: Bitmap metadata and bitmaps presented by Vampyr.

extracted to the same file. Each file name contains the offset of the unallocated area and the length.

--xall This parameter expects a directory. Vampyr extracts all objects, their metadata and slack space to this directory. If there is CephFS metadata, it is also used to restore information about directories, file names and locations in CephFS.

The slack space of the BlueStore label is extracted into `slack_bslabel` and the slack space of the BlueFS superblock into `slack_bfssuperblock`. It also writes the metadata of `M`-rows and `P`-rows to the files `kvmetadata_M` and `kvmetadata_P`. The data is sorted by OID.

Within the specified directory Vampyr creates a subdirectory for each object prefix, for example `osdmap`, `rbd_data.<rbdid>`, or `100000a` (as an example for a CephFS i-node number). Vampyr extracts content, slack space, and metadata to the matching subdirectory.

The following files may be created by Vampyr depending on whether there is data to extract:

```
ls -l 1000001a802/parent
1000001a802/parent -> ../1000001300b
```

List of Listings 4.7: The parent symlink shows to the directory that holds the parent directory information.

- `object_<ID>`: The content of the object with this ID. For example the file `rbd_data.30572ae8944a/object_0000000000010029` contains the data of the object `rbd_data.30572ae8944a.0000000000010029`.
- `md5_object_<ID>`: Contains the md5 checksum of the object content.
- `vampyrmeta_<ID>`: The metadata that could be extracted by Vampyr. This includes the data from `O`-rows and `M`-rows.
- `slack_<ID>`: The slack space of the object. This file is empty if there was no slack.
- `self_<filename>`: If the object is a CephFS file or directory, this file is created without content to indicate the file or directory name.
- `parent`: If the object is a directory or file in CephFS, this symlink is created and shows to the subdirectory that contains the parent directory's data as shown in Listing 4.7.
- `child_<filename>`: If the object is a CephFS directory, Vampyr creates symlinks to the subdirectories that contain data of all children of this CephFS directory as shown in Listing 4.8.
- `decoded_<id>`: If the object contains a data structure that Vampyr can decode, the decoded data is written to this file.
- `crush_<id>`: If the object contains a crush map, it is extracted to this file. The crushmap is still encoded, but it can be decoded using Ceph's crushtool [71]. Objects that contain crush maps are `osdmap` and `inc_osdmap` objects.

```
ls -l child_*
child_HDB50 -> ../1000000f47a
child_exe -> ../1000000a8f6
child_global -> ../1000000a8db
child_hdbclient -> ../1000001328d
child_hdblcm -> ../1000000f4af
child_profile -> ../1000000a8da
```

List of Listings 4.8: The child_ symlinks point to the directories holding the subdirectory or file information.

--lspgs If this flag is set, Vampyr displays PG metadata from the KV store.

4.3. Functionality and Options of vampyr-rebuild.py

After extracting objects that are part of RBDs or CephFS files using the `--xall` option of `vampyr.py`, the tool `vampyr-rebuild.py` can rebuild an RBD or file using all *known* parts. Obviously it is not possible to rebuild data of *missing* parts.

To increase the number of known parts of an object one can execute Vampyr with the `--xall` option for every known OSD and extract the objects to the *same* directory. If Vampyr finds the same object in more than one OSD, it keeps the newest. Whether an object version is newer is determined by the timestamp.

Afterwards, `vampyr-rebuild.py` can put the objects of the different OSDs together.

After executing the tool the directory with the object data contains a file called `rebuild`. This file contains, for example, a restored CephFS file or a restored RBD. Missing parts are filled using zeros.

--dir This parameter expects the directory with the `object_` files. This could be for instance `./extract/rbd_data.30572ae8944a/` after Vampyr was executed with `--xall ./extract/`.

`vampyr-rebuild.py` looks for all files called `object_<id>` and puts them together.

--blocksize This parameter expects the size in bytes of the different objects. The tool defaults to 4194304 bytes (4 MiB) which should work. If the tool finds an object part that is larger than the blocksize, it aborts.

The test data shows that RBD data and CephFS files are split into objects of 4 MiB size. Smaller objects can exist, too, for example at the end of an file or because RBD objects are allocated lazily.

5. Evaluation

This chapter describes the test environment and test data (Chapter 5.1) and how this test data shows that Vampyr retrieves and displays correct information (Chapter 5.2). Chapter 5.3 contains an analysis regarding the completeness of the results of Vampyr and Chapter 5.4 analyzes its runtime performance.

5.1. Test Environment

In order to generate test data, a clean test environment was set up using three VMs with four virtual CPUs (the underlying hardware was an Intel(R) Xeon(R) CPU E7-8890 v3 @ 2.50GHz) and 32 GB RAM (hostnames vm911, vm912, and vm913). Each VM was running *Debian GNU/Linux 9.3 (stretch)* as operating system and *Ceph version 12.2.4 luminous (stable)*.

The storage layout of each VM consisted of a 16GB virtual hard drive for the operating system and a 100GB virtual hard drive as Ceph OSD.

The cluster was created and managed via an additional VM (hostname vm900).

After creating the Ceph cluster, certain actions were taken and images of the OSDs taken via the command line tool `dd`. Before taking OSD images the OSDs were stopped. The following list contains the list of actions and the name of the step:

1. Empty OSD: `emptyosd`

2. Create a pool called mypool1: `poolcreated`

3. Initialize the pool: `poolinit`
4. Create an RBD: `rbdcreated`
5. Write test file (see below) at offset `0x0` of the RBD: `filewritten`
6. Write test file at offset `0x1e00000` (30 MiB) of the RBD: `filewritten2`
7. Create two pools and a CephFS using these two pools: `cephfscreated`
8. Mount CephFS on vm900, create directories and files within the CephFS in directory `test_files`: `cephfsused`
9. Mount CephFS on vm900, create directories and files within the CephFS in directory `test_files2` with the same files as before. Additionally, create 10,000 empty files in the directory `empty_file` using `emtpy-files.sh <mnt>/empty_files/` (see Appendix C.1 for the script) to fill the file system journal: `cephfsused2`

The image files can be found in the attachment in directories named after their step.

The test data file is called `blockfile` and is attached. It has a size of 13 MiB (MD5 checksum `5d9a8a13529b425c626a6654886c37ef`) and consists of 26 blocks with a size of 512 KiB each. The first block is filled with the ASCII character `a`, the second with `b`, and so on. This layout was chosen to easily see how data is fragmented by Ceph.

The files and directories created in the CephFS are also contained in the attachment (directory `test_files`) and consist of several JPEG files, a ZIP file and two ASCII-encoded text files.

5.1.1. Other Test Data

In order to validate findings, some OSD images of Lenovo test and certification clusters for SAP HANA setups were analyzed.

The servers of these clusters were running *SUSE Linux Enterprise Server for SAP Applications 12 Service Pack 3* with *Ceph 12.2.2-361 luminous (stable)*.

```
-----------------------
Object List:
-----------------------
Prefix        -> Object
-----------------------
rbd_data.2330174b0dc51 -> 0000000000000000, 0000000000000001, 0000000000000002, ↩
    ↪0000000000000003
rbd_directory ->
rbd_header -> 2330174b0dc51
rbd_id -> myrbd1
rbd_info ->
rbd_object_map -> 2330174b0dc51
-----------------------
```

List of Listings 5.1: Vampyr output of objects holding RBD data at state filewritten

5.2. Correctness

In order to ensure that Vampyr extracts valid data from OSDs, the test images were analyzed. The results are presented in the following three sections.

5.2.1. RBD

Analyzing OSD images at states rbdcreated, filewritten, and filewritten2 shows that objects for RBDs are allocated lazily. Only blocks that are used are allocated. After writing the test file to offset `0x0` there are four `rbd_data` objects and several RBD objets holding metadata (Listing 5.1).

The objects with IDs 0 - 2 have a size of 4 MiB, the object with ID 3 has a size of 1 MiB. `vampyr-rebuild.py` creates a file that has the same MD5 checksum as the test file. This shows that Vampyr extracted the correct data from the OSD image and `vampyr-rebuild.py` put them together in the correct order.

After writing the test file to offset `0x1e00000`, the objects with IDs 7 - 10 exist while IDs 4 - 6 are still not allocated (see Listing 5.2). After rebuilding the RBD, the test file can be located at offset `0x0` and `0x1e00000` and the reconstructed file has a size of 43 MiB.

```
-----------------------
Object List:
-----------------------
Prefix        -> Object
-----------------------
rbd_data.2330174b0dc51 -> 0000000000000000, 0000000000000001, 0000000000000002, ↩
    ↪0000000000000003, 0000000000000007, 0000000000000008, 0000000000000009, ↩
    ↪000000000000000a
rbd_directory ->
rbd_header -> 2330174b0dc51
rbd_id -> myrbd1
rbd_info ->
-----------------------
```

List of Listings 5.2: Vampyr output of objects holding RBD data at state filewritten2

```
Additional Metadata from KV Store (M prefix)
create_timestamp: 2018-03-30 11:17:25.190355644
features: 1 (0x1)
flags: 0 (0x0)
object_prefix: rbd_data.2330174b0dc51
order: 22 (0x16)
size: 52428800 (0x3200000)
snap_seq: 0 (0x0)
```

List of Listings 5.3: rbd_header metadata

Another capability of Vampyr is that it finds RBD metadata; the `rbd_header` object metadata for example shows correctly that the RBD was named *myrbd1* with a size of 50 MiB (`0x3200000`) and created at March 30th, 2018 (see Listing 5.3).

Since all RBD-related objects can be found on all three OSD images of each state (rbdcreated, filewritten, filewritten2) it is possible to extract all information from each of the OSDs. In other setups with more OSDs this will be different. In a real-life scenario a Ceph cluster has more than three OSDs and not every OSD contains a full set of objects. An OSD then contains only a subset of objects and it is necessary to analyze more than one OSD to be able to restore the full RBD content. Nevertheless, this test case shows that the data extracted by Vampyr is correct.

```
----------------------
Object List:
----------------------
Prefix        -> Object
----------------------
100 -> 00000000, 00000000.inode
200 -> 00000000, 00000001
400 -> 00000000
500 -> 00000000
600 -> 00000000
601 -> 00000000
602 -> 00000000
603 -> 00000000
604 -> 00000000
605 -> 00000000
606 -> 00000000
607 -> 00000000
608 -> 00000000
609 -> 00000000
mds0_inotable ->
mds0_sessionmap ->
mds_snaptable ->
----------------------
```

List of Listings 5.4: CephFS objects with application category data

5.2.2. CephFS

Analyzing the CephFS content on OSD images (states cephfsused and cephfsused2) shows that the existence of a CephFS can be verified using Vampyr.

Pools containing the string “cephfs” in their name can be found in the decoded osdmap and inc_osdmap objects. A Ceph user may choose names not containing “cephfs”, but it is likely that users choose speaking names for their pools. In case pools are named in a way that their usage is not obvious, the existence of objects described below indicate that a CephFS is present.

The object with inode number `1`, i.e. the root directory, is found. Objects with 3-digit-prefix and prefix `mds` – as used for objects of the application data category – are found (see Listing 5.4). They hold for example journal data of CephFS.

```
----------------------
Object List:
----------------------
Prefix        -> Object
----------------------
10000000003 -> 00000000
10000000004 -> 00000000
10000000005 -> 00000000
10000000006 -> 00000000
10000000007 -> 00000000
10000000008 -> 00000000
10000000009 -> 00000000
1000000000a -> 00000000
1000000000b -> 00000000
1000000000c -> 00000000
1000000000d -> 00000000
1000000000e -> 00000000
1000000000f -> 00000000
10000000010 -> 00000000
10000000011 -> 00000000
10000000012 -> 00000000
10000000013 -> 00000000
10000000015 -> 00000000, 00000001, 00000002, 00000003, 00000004
10000000016 -> 00000000
10000000017 -> 00000000
----------------------
```

List of Listings 5.5: Objects for CephFS files at state cephfsused

As expected (see Chapter 3.3.4 and Figure 3.6), the actual file contents are placed into objects with the object name containing the inode number (see Listing 5.5).

`vampyr-rebuild.py` can recreate the ZIP file that is contained in the test data from the data of inode `10000000015`. The other objects contain the content of the other test files. The extracted files were checked using their MD5 checksums against the original files. The MD5 checksums of the objects and the rebuilt ZIP file match the checksums of the original test data. Table 5.1 contains a mapping of seven original files to the CephFS inode using the MD5 checksum. This shows that Vampyr is able to correctly restore data from an OSD.

Data of the file name and metadata category is not yet completely available in the KV store. The reason for this is that the data is still in the CephFS journal and not

MD5 checksum	original file	inode
e8283c2d47ee935cee695eb8354d5216	`blyde-river-canyon_19613751584_o.jpg`	1000000000f
8e280cd327f251205f77c56d09e3ca1b	`blyde-river-canyon_20210140976_o.jpg`	10000000009
616f809681d8509caefc2da34a2f4475	`blyde-river-canyon_20242204791_o.jpg`	1000000000a
1ac540f97ad4bdd5bd4cd5c92217516a	`blyde-river-canyon_20242208371_o.jpg`	10000000004
6b1c5d796553e9f2f258ab1fcb5f141a	`cape-point_19613681554_o.jpg`	1000000000c
5cd85d30b34a867ca9df4ecc6561b49e	`cape-point_20048295750_o.jpg`	10000000011
f9a7a5faa3eaede8ba3e9e570704a296	`cape-point_20236333875_o.jpg`	10000000016

Table 5.1.: MD5 checksums of seven test data files and objects in state cephfsused on vm912.

yet applied to the KV store. Thus, the directory structures and file names cannot be recreated. The data can be found in object `200.00000001`.

The state `cephfsused2` was created to ensure that the journal is applied and some metadata written to the KV store. From this state Vampyr can recreate the file system structure and file names of the different test files in directories `test_files` and `test_files2`. Vampyr is not able to reconstruct the file system structure and file names of all of the 10,000 empty files. This data is available in the KV store only up to file `empty_1819`. The rest of the information is still in the journal. The journal objects now have the IDs `0`, and `6` – `11`.

Comparing Foremost and Vampyr Results

Foremost is – as mentioned previously – a file carving tool. In order to ensure that Vampyr extracts the same amount or more files – but not less – than a file carving tool, Foremost was used to find the CephFS files on `cephfsused2` images. Foremost version 1.5.7 was executed to carve JPEG and ZIP files.

Foremost is able to find the JPEG files, but it is not able to find complete ZIP files in CephFS. It finds both copies of each JPEG file. It also detects the start of the ZIP files, but it is not able to find all fragments.

Extracting CephFS Data from a SAP HANA Test Environment

Vampyr was also tested with an OSD image from a SAP HANA test environment. The OSD is one of 30 OSDs in the cluster. It does not hold information of the application data category – the objects containing this data reside on other OSDs. Only a subset of the files that are stored in the file system can be found on the OSD; and for larger files only a subset of objects that contain the file's data are stored on a single OSD. Therefore, it is only possible to restore smaller files completely. This is an important constraint of OSD analysis that is inherent to the way Ceph distributes across a cluster.

The KV store contains usable data of the file name and metadata category. By using this data it is possible to restore directory structures, file names, file permissions and timestamps.

5.2.3. osd_superblock, osdmap, inc_osdmap

The OSD images were checked whether valid osdmap data can be extracted from them or not.

An interesting finding is that the images of state `filewritten2` contain KV entries for osdmap objects that are not yet written to disk – the physical extents allocated for the osdmaps are still empty. This means that the KV store does not always exactly reflect the state of the on-disk data.

Tests using the `--scan` option show that it can find the same `osd_superblock`, `osdmap`, and `inc_osdmap` objects as found when using the `--xall` option. This might not be true for setups with larger osdmap objects in case they are stored in non-contiguous physical extents.

Vampyr can also decode the different data structures correctly. For example timestamps of events, OSD state changes, the creation of new pools, and IP addresses of servers are correctly decoded and listed. The extracted crushmaps can then be decoded

```
0x00010000:               header --> 8-5-0x1f9: Offset of end: 66047 (0x101ff)
0x00010006:         cluster_fsid --> ab118908-9e1a-3f65-893c-2ea0146fef53
0x00010016:               whoami --> 71 (0x47)
0x0001001a:        current_epoch --> 13950 (0x367e)
0x0001001e:           oldest_map --> 10991 (0x2aef)
0x00010022:           newest_map --> 13950 (0x367e)
:
```

List of Listings 5.6: Decoded osd_superblock object data

using the Ceph crushtool. This proves that the Vampyr decoders for osdmap objects are correct and the recovery is successful.

Scanning an OSD for osdmap Data Structures

During analysis of a single, randomly chosen OSD from a test environment, it was possible to find 13538 valid osdmap data structures, 1537 valid inc_osdmaps and the osd_superblock object. The osd_superblock object (Listing 5.6) shows that the oldest osdmap known to the KV store is osdmap.10991 and the newest osdmap.13950.

By scanning the OSD for osdmap data structures it was possible to decode osdmaps from epoch 416 to 13950, and inc_osdmaps from epoch 417 to 13950.

Object osdmap.416 is from 2018-06-06 13:38:34, osdmap.10991 from 2018-06-13 11:28:29, and osdmap.13950 from 2018-06-19 13:29:18. Object inc_osdmap.417 is from 2018-06-06 13:38:35, inc_osdmap.10991 from 2018-06-13 11:28:29, and inc_osdmap.13950 from 2018-06-13, 13:29:18.

This shows that it is possible to extract significantly more data by fully scanning the OSD itself than by relying on OSD metadata in the KV store. In this test case there are about 4.5 times more valid osdmaps than the metadata showed, but about 3% of the osdmaps (osdmap.1 to osdmap.415) are already lost.

5.3. Completeness

Ceph is a versatile software with several applications and extensions running on top of it. Because of the limited timespan for creating Vampyr it was not possible to implement decoders and handlers for every kind of data structure. However, there are several aspects to consider in regards to completeness of the implementation.

In order to load the KV store, Vampyr needs to parse the complete transaction log. This means that Vampyr has to understand all operations (Table 3.13). If Vampyr did not understand all operations, it could not rebuild the state of BlueFS, locate the file contents, and extract them. Therefore, Vampyr can parse the complete set of BlueFS operations and rebuild every state of a BlueFS. Dozens of tests with OSDs from the test environments show that Vampyr handles the data correctly and restores the files completely.

The most important aspect is that Vampyr must be able to parse all `O`-rows of the KV store. These rows are crucial for finding object data on disk. They also contain data of the metadata and file name category. Even if Vampyr were not able to decode other rows – for instance `M`-rows –, Vampyr would be able to list object names, modification times, and sizes and locate the object content on disk. Vampyr can parse `O`-rows of different setups, but two features are missing: First, Ceph may compress parts of the object data, but Vampyr is not yet able to decompress it. Second, Ceph stores checksums in `O`-values, but Vampyr does not check them.

Another aspect is the high variety of metadata types in `M`-rows. Ceph stores CephFS metadata, but also other metadata with this prefix. The parser for `M`-rows handles metadata types found by code review and was checked against several OSD images. If the parser finds an unknown metadata type, it is not ignored, but presented undecoded as a hexdump.

In summary, it can be stated that Vampyr meets the goal to analyze BlueStore OSDs (see Chapter 1.2). It extracts objects and their metadata. In case of RBD-related and

CephFS-related objects, it can put the metadata in the right context – for example, it restores CephFS file names and directory structures. Furthermore, it is able to decode osdmap data and can even find osdmap data in unallocated areas of an OSD.

This is a first-of-a-kind as no other tool existed so far that can recover Ceph BlueStore data.

5.4. Runtime Performance

Runtime performance was not the key focus during the implementation of Vampyr. Many operations – for example extracting data or reading data from the OSD image – depend on the storage of the computer that runs Vampyr. If the image is located on an SSD, Vampyr may run faster than if the image is stored on an HDD. Other operations like parsing the KV store datasets do not depend on storage performance.

Table 5.2 shows the loading and parsing speed of five different operations. Vampyr was run with four different OSD images: The OSDs of vm911 at the states filewritten2 and cephfsused2, as well as two OSDs of SAP HANA test environments. Both HANA OSDs were used for some time and held more objects than the OSDs of vm911.

The transaction log time is the time that Vampyr needs to read and parse the BlueFS transaction log. Table 5.2 shows that the transaction logs consist at maximum of 4009 transactions and are loaded within up to two seconds. This operation depends on the I/O speed of the storage subsystem. However, Ceph shrinks the transaction log as soon as it exceeds a certain size. Therefore, the size of the transaction log will not grow so much that it will lead to significant loading times for Vamypr.

The ldb time indicates how much time it takes until Vampyr loaded all datasets into memory using `ldb`. The loading time depends on the size of the KV store – and the size of the KV store depends on the number of objects and amount of metadata stored on the OSD.

	filewritten2	cephfsused2	SAP HANA 1	SAP HANA 2
transaction log time	$< 1s$	$2s$	$< 1s$	$1s$
transaction log size	$742tx$	$4009tx$	$130tx$	$1839tx$
parsing speed	$> 742tx/s$	$2004tx/s$	$> 130tx/s$	$1839tx/s$
ldb time	$< 1s$	$< 1s$	$5s$	$1s$
ldb size	$1095d$	$9601d$	$382698d$	$35483d$
loading speed	$> 1095d/s$	$> 9601d/s$	$76540d/s$	$35483d/s$
`O`-rows time	$< 1s$	$2s$	$5s$	$5s$
# `O`-rows	$375d$	$2963d$	$4586d$	$27596d$
parsing speed	$> 375d/s$	$1481d/s$	$917d/s$	$5519d/s$
`M`-rows time	$< 1s$	$3s$	$36s$	$< 1s$
# `M`-rows	$595d$	$6407d$	$364228d$	$4016d$
parsing speed	$> 595d/s$	$2136d/s$	$10117d/s$	$> 4016d/s$
`b`-rows time	$< 1s$	$< 1s$	$< 1s$	$< 1s$
# `b`-rows	$6d$	$22d$	$13838d$	$3616d$
parsing speed	$> 6d/s$	$> 22d/s$	$> 13838d/s$	$> 3616d/s$

Table 5.2.: Parsing and loading times of different datasets. tx stands for transactions of the transaction log. d stands for dataset.

The time to parse certain prefixes is listed for `O`-rows, `M`-rows, and `b`-rows because they hold a significant amount of data. The most interesting data shows the image SAP HANA 1. The parsing speed for `O`-rows is low in comparison to cephfsused2 and SAP HANA 2 ($917d/s$ vs. $1481d/s$ and $5519d/s$). The reason for the lower speed might be that the `O`-rows of SAP HANA 1 hold more or more complex data.

At the same time SAP HANA 1 holds more `M`-rows than SAP HANA 2 ($364228d$ vs. $4016d$). It takes 36 seconds to parse them, but the parsing speed is higher than for cephfsused2 ($10117d/s$ vs. $4016d/s$).

Summarizing the measurements, the overall parsing speed for the KV store depends on the amount, but also on the types of objects and metadata stored on an OSD. Since Vampyr loads datasets with a certain prefix only if they are required, the overall runtime of Vampyr depends on the chosen command line parameters.

6. Summary & Conclusion

This thesis gives an overview about recovery of data located on BlueStore OSDs. Ceph – together with BlueStore – is a sophisticated software that stores data and metadata of different use cases encoded on OSDs of servers in a cluster. RBD and CephFS – as applications on top of RADOS – translate and store their data as objects and metadata in RADOS. RADOS itself stores cluster specific management information in form of osdmaps and a key-value store within BlueFS.

The second part of this thesis takes the knowledge gained in the first part and describes Vampyr, the implementation of a forensic tool that analyzes tha data and metadata of an OSD. An inherent limitation of Vampyr is that Ceph stores data distributed across cluster nodes. If only one OSD is available for analysis, only a part of data and metadata can be analyzed and restored. With a larger number of OSDs a larger percentage of data and metadata can be analyzed by Vampyr.

Since not all aspects and all conditions of BlueStore and Ceph cluster configurations were within scope due to time constraints, there are certain limitations (Chapter 6.1). Further investigations on this topic are thus recommended (Chapter 6.2).

6.1. Current Limitations

At present, the most important limitation of Vampyr to be considered is that it only analyzes BlueStore OSDs – FileStore was not within the scope of this thesis, so it was

not implemented in Vampyr. Since both OSD backend types are completely different (Chapter 2.2), FileStore requires its own analysis and implementation.

This thesis analyzes OSDs, where the KV store is located in the BlueFS of this OSD. However, there are configuration options that allow the KV to be partially located on a faster device, a larger device, or both (Chapter 2.2.2). So far, Vampyr is not able to analyze the databases located outside of the primary device. However, Vampyr is able to recover the full BlueFS data and metadata when located on the primary device.

Furthermore, Vampyr is not yet able to analyze CephFS journaling data from objects with prefix `200.`. These objects hold data from the file name category and the metadata category (Chapter 3.3.4).

This thesis aimed to analyze the applications RBD and CephFS. RGW and `librados` are not part of the analysis. While Vampyr is able to find objects that were written using these interfaces, interface-specific metadata or objects may not be decoded in a useful manner.

Additionally, decompression of compressed object data is not implemented in Vampyr – Ceph compresses object data if it is configured by the administrator and the compressed data is significantly smaller than the uncompressed data. The same applies to checksum validation of CRC32 checksums. So far, no Python-based CRC32 algorithm library was found that can validate the checksums written by Ceph. Such a library is a prerequisite to support checksum validation, for example for the BlueStore superblock.

6.2. Future Work

There are different aspects from a research and implementation point of view that can be part of further analyses and implementation work.

BlueStore and FileStore FileStore was not within the scope of this work. An examination of FileStore can find approaches to analyses of FileStore OSDs. Furthermore,

a comparison of BlueStore and FileStore may reveal differences or similarities between both backends and could lead to a more general approach to Ceph OSD analysis.

RGW and librados Since RGW and RADOS access via `librados` are not part of this research, an analysis of both access interfaces may find and describe RGW-specific or `librados`-specific evidence. These insights would also require additional implementation work to make them accessable for Vampyr.

CephFS This work gives a good understanding of CephFS-specific data on an OSD. However, future work can focus on researching the CephFS behavior when deleting, overwriting, extending, or moving files, or performing other file system operations. Furthermore, the journal objects require further research. They hold information about recent changes in a CephFS that cannot be found elsewhere, but Vampyr is not yet able to decode this information.

Vampyr features Some aspects and features of BlueStore were not implemented in Vampyr. Unhandled features, such as Ceph object compression and checksum validation, need further implementation effort. Furthermore, it is possible to implement additional features that make use of the insights shown in this thesis and present data in a different way. An advanced feature could be to compare and correlate data of multiple OSDs.

BlueStore development Test data for Vampyr was created using Ceph version 12.2 (Luminous). Since Ceph and BlueStore are actively developed and extended with new features, new data structures are expected to appear in future Ceph releases. This leads to the necessity of reviewing and updating the results of this work with data of new Ceph releases.

6.3. Conclusion

There are three main results of this thesis.

The first result is a documentation of Ceph data structures like superblocks, the BlueFS transaction log, or datasets in the key-value store. This was achieved by source code and hexdump analysis. This documentation helps understanding how BlueStore works.

The second result is the comprehensive view of BlueStore and how the different components – mainly BlueFS, RADOS, RBD, and CephFS – work together. It explains – based on the Carrier model – which data structures can be used to find certain information.

The third result is Vampyr. Vampyr is designed and implemented from scratch and allows to extract and present data physically residing on Ceph BlueStore OSDs. This enables forensic analysis of a Ceph cluster. A part of the implementation work is the creation of training data to verify correctness of the analysis. This data includes images of OSDs at different defined states of a Ceph cluster.

Future research could be carried out on analyzing additional applications besides RBD and CephFS. New features of Ceph like compression and deduplication also manifest in different physical data structures that each need their own research and implementation work in Vampyr.

Overall, forensic analysists can benefit from this thesis and from Vampyr when analyzing public, hybrid, or private clouds that use Ceph as their storage backend. Vampyr performs better than a file carving tool because it can handle fragmented data, it presents metadata, and it decodes Ceph-specific content. It can also combine data of different OSDs when restoring files. Additionally, Vampyr provides functionality to extract unallocated areas and slack spaces in order to make them available to other forensic tools such as file carving tools.

Appendices

A. Attachments

The DVD attached to this master thesis contains:

- this thesis as PDF file
- directory `test_files` with data that was copied into CephFS
- directory `osd_images`
  - directories `emptyosd`, `poolcreated`, `poolinit`, `rbdcreated`, `filewritten`, `filewritten2`, `cephfscreated`, `cephfsused`, `cephfsused2`; each containing three OSD images (one per test server)
  - directory `rbd_states` with the extracted RBD data from states `filewritten` and `filewritten2`
- Vampyr source code (`vampyr-1.0.0.tar.gz`)

B. Hexdumps

B.1. BlueStore Superblock

```
00000000: 626c 7565 7374 6f72 6520 626c 6f63 6b20  bluestore block
00000010: 6465 7669 6365 0a36 3638 3331 6532 362d  device.66831e26-
00000020: 3361 3636 2d34 3534 372d 3936 3436 2d37  3a66-4547-9646-7
00000030: 6364 3236 3530 3561 3838 360a 0201 1601  cd26505a886.....
00000040: 0000 6683 1e26 3a66 4547 9646 7cd2 6505  ..f..&:fEG.F|.e.
00000050: a886 0000 c0ff 1800 0000 24cb 9b5a aa15  ..........$..Z..
00000060: 2318 0400 0000 6d61 696e 0800 0000 0600  #.....main......
00000070: 0000 626c 7565 6673 0100 0000 3109 0000  ..bluefs....1...
00000080: 0063 6570 685f 6673 6964 2400 0000 6538  .ceph_fsid$...e8
00000090: 3166 3132 3630 2d62 3538 662d 3466 3465  1f1260-b58f-4f4e
000000a0: 2d62 3533 612d 6132 3164 3962 3336 3137  -b53a-a21d9b3617
000000b0: 6338 0a00 0000 6b76 5f62 6163 6b65 6e64  c8....kv_backend
000000c0: 0700 0000 726f 636b 7364 6205 0000 006d  ....rocksdb....m
000000d0: 6167 6963 1400 0000 6365 7068 206f 7364  agic....ceph osd
000000e0: 2076 6f6c 756d 6520 7630 3236 0900 0000   volume v026....
000000f0: 6d6b 6673 5f64 6f6e 6503 0000 0079 6573  mkfs_done....yes
00000100: 0700 0000 6f73 645f 6b65 7928 0000 0041  ....osd_key(...A
00000110: 5141 6a79 3574 6174 4a70 4d47 7841 4143  QAjy5tatJpMGxAAC
00000120: 5370 614f 3657 6874 5052 454f 7678 4559  SpaO6WhtPREOvxEY
00000130: 666d 4c61 513d 3d05 0000 0072 6561 6479  fmLaQ==....ready
00000140: 0500 0000 7265 6164 7906 0000 0077 686f  ....ready....who
00000150: 616d 6901 0000 0030 67e1 59b7 0000 0000  ami....0g.Y.....
```

List of Listings B.1: Hexdump of a BlueStore superblock.

B.2. BlueFS Superblock

```
00001000: 0101 4f00 0000 9a5b 96ec 0b8d 4f74 9c43  ..O....[....Ot.C
00001010: 68fc a173 11a3 6683 1e26 3a66 4547 9646  h..s..f..&:fEG.F
00001020: 7cd2 6505 a886 0100 0000 0000 0000 0010  |.e.............
00001030: 0000 0101 1d00 0000 0180 2000 0000 0000  .......... .....
```

```
00001040: 0000 0000 0100 0000 0101 0700 0000 f3ff  ................
00001050: 0500 8320 0149 b7e4 5d00 0000 0000 0000  ... .I..].......
```

List of Listings B.2: Hexdump of a BlueFS superblock.

B.3. OSD Superblock

```
00010000: 0805 f901 0000 e81f 1260 b58f 4f4e b53a  ...........ON.:
00010010: a21d 9b36 17c8 0000 0000 4100 0000 0100  ...6......A.....
00010020: 0000 4100 0000 0000 0000 0000 0000 0000  ..A.............
00010030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010040: 0000 0000 0000 fef7 0100 0000 0000 0f00  ................
00010050: 0000 0100 0000 0000 0000 1a00 0000 696e  ..............in
00010060: 6974 6961 6c20 6665 6174 7572 6520 7365  itial feature se
00010070: 7428 7e76 2e31 3829 0200 0000 0000 0000  t(~v.18)........
00010080: 0d00 0000 7067 696e 666f 206f 626a 6563  ....pginfo objec
00010090: 7403 0000 0000 0000 000e 0000 006f 626a  t............obj
000100a0: 6563 7420 6c6f 6361 746f 7204 0000 0000  ect locator.....
000100b0: 0000 0010 0000 006c 6173 745f 6570 6f63  .......last_epoc
000100c0: 685f 636c 6561 6e05 0000 0000 0000 000a  h_clean.........
000100d0: 0000 0063 6174 6567 6f72 6965 7306 0000  ...categories...
000100e0: 0000 0000 000b 0000 0068 6f62 6a65 6374  .........hobject
000100f0: 706f 6f6c 0700 0000 0000 0000 0700 0000  pool............
00010100: 6269 6769 6e66 6f08 0000 0000 0000 000b  biginfo.........
00010110: 0000 006c 6576 656c 6462 696e 666f 0900  ...leveldbinfo..
00010120: 0000 0000 0000 0a00 0000 6c65 7665 6c64  ..........leveld
00010130: 626c 6f67 0a00 0000 0000 0000 0a00 0000  blog............
00010140: 736e 6170 6d61 7070 6572 0c00 0000 0000  snapmapper......
00010150: 0000 1100 0000 7472 616e 7361 6374 696f  ......transactio
00010160: 6e20 6869 6e74 730d 0000 0000 0000 000e  n hints.........
00010170: 0000 0070 6720 6d65 7461 206f 626a 6563  ...pg meta objec
00010180: 740e 0000 0000 0000 0014 0000 0065 7870  t............exp
00010190: 6c69 6369 7420 6d69 7373 696e 6720 7365  licit missing se
000101a0: 740f 0000 0000 0000 0010 0000 0066 6173  t............fas
000101b0: 7469 6e66 6f20 7067 2061 7474 7210 0000  tinfo pg attr...
000101c0: 0000 0000 0016 0000 0064 656c 6574 6573  .........deletes
000101d0: 2069 6e20 6d69 7373 696e 6720 7365 7441   in missing setA
000101e0: 0000 003b 0000 0066 831e 263a 6645 4796  ...;...f..&:fEG.
000101f0: 467c d265 05a8 8600 0000 0000 0000 0000  F|.e............
```

List of Listings B.3: Hexdump of an OSD superblock object.

C. Helper Scripts

C.1. Script empty-files.sh

```
#!/bin/bash
if [[ -z "$1" ]]
then
    echo "error"
    exit 1
fi
DIR=$1
declare -i i
for i in {0..9999}
do
    x=$((i%10))
    dir=dir_$((i/10))
    if [[ $x == 0 ]]
    then
        mkdir -p $DIR/$dir
    fi
    touch $DIR/$dir/empty_$i
done
sync
```

List of Listings C.1: Script to create 10000 empty files in a certain directory.

D. Other

D.1. LVM Metadata Example

```
# Generated by LVM2 version 2.02.168(2) (2016-11-30): Sat Aug 18 13:32:33 2018

contents = "Text Format Volume Group"
version = 1

description = "vgcfgbackup -f vm911.lvm.backup.txt ceph-e81f1260-b58f-4f4e-b53a-↩
    ↪a21d9b3617c8"

creation_host = "vm911" # Linux vm911 4.9.0-6-amd64 #1 SMP Debian 4.9.88-1+deb9u1 ↩
    ↪(2018-05-07) x86_64
creation_time = 1534591953  # Sat Aug 18 13:32:33 2018

ceph-e81f1260-b58f-4f4e-b53a-a21d9b3617c8 {
    id = "dIWVYa-cckw-VMtv-erJ2-NkcF-03wy-YCReaf"
    seqno = 16
    format = "lvm2"         # informational
    status = ["RESIZEABLE", "READ", "WRITE"]
    flags = []
    extent_size = 8192      # 4 Megabytes
    max_lv = 0
    max_pv = 0
    metadata_copies = 0

    physical_volumes {

        pv0 {
            id = "x6DxI6-cWZl-VPFY-KGs7-9LTv-JQ5b-bY6VNN"
            device = "/dev/sdb" # Hint only

            status = ["ALLOCATABLE"]
            flags = []
            dev_size = 209715200    # 100 Gigabytes
            pe_start = 2048
            pe_count = 25599    # 99.9961 Gigabytes
        }
    }
```

```
    logical_volumes {

        osd-block-66831e26-3a66-4547-9646-7cd26505a886 {
            id = "yL2G7n-9cbV-yQPP-pInm-aLXv-0hA6-tAECnD"
            status = ["READ", "WRITE", "VISIBLE"]
            flags = []
            tags = ["ceph.type=block", "ceph.osd_id=0", "ceph.osd_fsid=66831e26-3a66↩
    ↪-4547-9646-7cd26505a886", "ceph.cluster_name=ceph", "ceph.cluster_fsid=e81f1260↩
    ↪-b58f-4f4e-b53a-a21d9b3617c8", "ceph.encrypted=0", "ceph.cephx_lockbox_secret↩
    ↪=", "ceph.crush_device_class=None", "ceph.block_device=/dev/ceph-e81f1260-b58f↩
    ↪-4f4e-b53a-a21d9b3617c8/osd-block-66831e26-3a66-4547-9646-7cd26505a886", "ceph.↩
    ↪block_uuid=yL2G7n-9cbV-yQPP-pInm-aLXv-0hA6-tAECnD"]
            creation_time = 1520159523  # 2018-03-04 11:32:03 +0100
            creation_host = "vm911"
            segment_count = 1

            segment1 {
                start_extent = 0
                extent_count = 25599    # 99.9961 Gigabytes

                type = "striped"
                stripe_count = 1    # linear

                stripes = [
                    "pv0", 0
                ]
            }
        }
    }

}
```

List of Listings D.1: LVM metadata of a Ceph OSD created using vgcfgbackup.

Bibliography

[1] Sage A. Weil. New in luminous: Bluestore, September 2017. http://ceph.com/community/new-luminous-bluestore/, accessed 19.08.2018.

[2] Sage A. Weil. *Ceph: Reliable, Scalable, And High-Performance Distributed Storage*. PhD thesis, University of California, December 2007. https://ceph.com/wp-content/uploads/2016/08/weil-thesis.pdf, accessed 19.08.2018.

[3] Ceph Source Code on Github. Bluestore.cc, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/os/bluestore/BlueStore.cc, accessed 19.08.2018.

[4] Margaret Rouse. Definition. software-defined storage (sds), August 2017. https://searchstorage.techtarget.com/definition/software-defined-storage, accessed 19.08.2018.

[5] Red Hat. Red hat to acquire inktank, provider of ceph, April 2014. https://www.redhat.com/en/about/press-releases/red-hat-acquire-inktank-provider-ceph, accessed 19.08.2018.

[6] ceph.com. Ceph contributors, December 2017. http://ceph.com/contributors/, accessed 19.08.2018.

[7] Alexandre Oliva. Ceph@home: the domestication of a wild cephalopod, 2014. http://ceph.com/use-cases/cephhome-the-domestication-of-a-wild-cephalopod/, accessed 19.08.2018.

[8] Red Hat. Monash university eresearch centre improves research support with red hat ceph storage, 2017. https://www.redhat.com/en/resources/monash-university-improves-research-ceph-storage-case-study, accessed 19.08.2018.

[9] docs.ceph.com. *Ceph Releases*, 2016. http://docs.ceph.com/docs/master/releases/, accessed 19.08.2018.

[10] SUSE. Suse enterprise storage, 2017. https://www.suse.com/de-de/products/suse-enterprise-storage/, accessed 19.08.2018.

[11] Karan Singh. Erasure coding in ceph, April 2014. https://ceph.com/planet/erasure-coding-in-ceph/, accessed 19.08.2018.

[12] Yuan Zhou. Ceph erasure coding introduction, April 2015. https://software.intel.com/en-us/blogs/2015/04/06/ceph-erasure-coding-introduction, accessed 19.08.2018.

[13] ceph.com. Pools – ceph documentation, 2016. http://docs.ceph.com/docs/luminous/rados/operations/pools/, accessed 19.08.2018.

[14] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *In proceedings of the 2006 ACM/IEEE Conference of Supercomputing (SC '06)*. ACM, 2006. https://www.ssrc.ucsc.edu/Papers/weil-sc06.pdf, accessed 19.08.2018.

[15] ceph.com. Ceph block device – ceph documentation, 2018. http://docs.ceph.com/docs/master/rbd/, accessed 19.08.2018.

[16] ceph.com. Ceph filesystem – ceph documentation, 2018. `http://docs.ceph.com/docs/master/cephfs/`, accessed 19.08.2018.

[17] Andreas Dewald, Felix Freiling, Michael Gruhn, and Christian Riess. *Forensische Informatik*. BoD - Books on Demand, Norderstedt, 2nd edition, October 2015.

[18] Martin Bachmaier and Ilya Krutov. *In-memory Computing with SAP HANA on Lenovo Systems*. Lenovo, 2017. `https://lenovopress.com/sg248086.pdf`, accessed 19.08.2018.

[19] Yongmin Park, Hyunsoo Chang, and Taeshik Shon. Data investigation based on xfs file system metadata. 75, June 2015.

[20] SalvationDATA. [Case Study] Computer Forensics: Fragmented Files Recovery Based on XFS File System, March 2017. `https://blog.salvationdata.com/2017/03/24/fragmented-files-recovery-based-on-xfs-file-system/`, accessed 19.08.2018.

[21] Florian Bausch. Dateisystem-Merkmale. XFS – X Filesystem, December 2016. Unpublished seminar paper.

[22] Andreas Dewald and Jonas Plum. *APFS Internals For Forensic Analysis. ERNW Whitepaper 65*. ERNW GmbH, 1.0 edition, April 2018. `https://static.ernw.de/whitepaper/ERNW_Whitepaper65_APFS-forensics_signed.pdf`, accessed 29.08.2018.

[23] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley, 12. edition, Juni 2013.

[24] Brian Carrier. Why Recovering a Deleted Ext3 File Is Difficult..., August 2005. `http://linux.sys-con.com/node/117909`, accessed 19.08.2018.

[25] Martin Rieger, Patrick Eisoldt, Vasilios Tsantroukis, Felix C. Freiling, Stefan Vömel, Michael Spreizenbarth, Andreas Dewald, and Hans-Georg Eßer. Datenträgerforensik. Modul 111, October 2015.

[26] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings to the First Dutch International Symposium on Linux, Amsterdam, December 8th and 9th, 1994*, 1995.

[27] xfs.org. A brief history of XFS. `http://xfs.org/docs/xfsdocs-xml-dev/XFS_User_Guide/tmp/en-US/html/ch01s02.html`, accessed 19.08.2018.

[28] Eric Burgener, Ritu Jyoti, Amita Potnis, Eric Sheppard, and Natalya Yezhkova. Market Forecast. Worldwide Software-Defined Storage Forecast, 2017-2021: SDS Market Growth Significantly Outpaces Enterprise Storage Growth, Led by HCI, 2017. IDC market forecast.

[29] Sébastien Han. Ceph and mds, December 2012. `https://www.sebastien-han.fr/blog/2012/12/03/ceph-and-mds/`, accessed 19.08.2018.

[30] ceph.com. Cache tiering – ceph documentation, 2016. `http://docs.ceph.com/docs/jewel/rados/operations/cache-tiering/`, accessed 19.08.2018.

[31] Sage A. Weil. Bluestore: A new storage backend for ceph – one year in, March 2017. `https://events.static.linuxfound.org/sites/events/files/slides/20170323%20bluestore.pdf`, accessed 19.08.2018.

[32] Tim Serong. Understanding bluestore. ceph's new storage backend, July 2017. `http://ourobengr.com/wp-uploads/2017/07/Understanding-BlueStore-Cephs-New-Storage-Backend.pdf`, accessed 19.08.2018.

[33] Tim Serong. Understanding bluestore, ceph's new storage backend, September 2017. `https://ceph.com/planet/understanding-bluestore-cephs-new-storage-backend/`, accessed 19.08.2018.

[34] Myoungwon Oh, Jugwan Eom, Jungyeon Yoon, Jae Yeun Yun, Seungmin Kim, and Heon Y. Yeom. Performance optimization for all flash scale-out storage. In *2016 IEEE International Conference on Cluster Computing*, 2016. `https://ceph.com/wp-content/uploads/2017/03/performance_optimization_for_all_flash_scale-out_storage-SK_Telecom.pdf`, accessed 19.08.2018.

[35] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. Understanding write behaviors of storage backends in ceph object store. In *33rd International Conference on Massive Storage Systems and Technology (MSST 2017)*, 2107. `http://storageconference.us/2017/Papers/CephObjectStore.pdf`, accessed 19.08.2018.

[36] Rainer Poisel and Simon Tjoa. A comprehensive literature review of file carving. In *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013. `https://ieeexplore.ieee.org/document/6657278/`, accessed 19.08.2018.

[37] Simson L. Garfinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2 – 12, 2007.

[38] Anandabrata Pal and Nasir Memon. The Evolution of File Carving. *IEEE SIGNAL PROCESSING MAGAZINE*, pages 59 – 71, 2009.

[39] Scalpel Contributors. sleuthkit/scalpel, 2013. `https://github.com/sleuthkit/scalpel`, accessed 19.08.2018.

[40] Foremost Contributers. Foremost, 2010. `http://foremost.sourceforge.net/`, accessed 19.08.2018.

[41] bulk_extractor Contributors. simsong/bulk_extractor, 2018. `https://github.com/simsong/bulk_extractor`, accessed 19.08.2018.

[42] EVTXtract Contributors. williballenthin/evtxtract, 2017. https://github.com/williballenthin/EVTXtract, accessed 19.08.2018.

[43] Simson L. Garfinkel and Abhi Shelat. Remembrance of data passed: A study of disk sanitization practices. *Data Forensics*, pages 17–27, January 2003. https://simson.net/clips/academic/2003.IEEE.DiskDriveForensics.pdf, accessed 19.08.2018.

[44] Bruce J. Nikkel. Forensic Analysis of GPT Disks and GUID Partition Tables. *Digital Investigation. The International Journal of Digital Forensics and Incident Response*, 6, November 2009.

[45] Robert Shullich. *Reverse Engineering the Microsoft Extended FAT File System (exFAT).* SANS Institute InfoSec Reading Room, December 2009. https://www.sans.org/reading-room/whitepapers/forensics/reverse-engineering-microsoft-exfat-file-system-33274, accessed 19.08.2018.

[46] Kevin Fairbanks. An analysis of Ext4 for digital forensics. *Digital Investigation*, 9:118 – 130, 2012.

[47] NIST. The cfreds project. https://www.cfreds.nist.gov/, accessed 19.08.2018.

[48] Brian Carrier. The Sleuth Kit (TSK) & Autopsy: Open Source Digital Forensics Tools, 2003. http://www.sleuthkit.org/, accessed 19.08.2018.

[49] X-Ways AG. X-Ways Forensics: Integrated Computer Forensics Software. http://www.x-ways.net/forensics/, accessed 19.08.2018.

[50] Lennart Bader. ceph-recovery, 2017. https://gitlab.lbader.de/kryptur/ceph-recovery, accessed 19.08.2018.

[51] smmoore. rbd_restore.sh, 2012. https://github.com/smmoore/ceph, accessed 19.08.2018.

[52] Siying Dong, Ed Baunton, Gunnar Kudrjavets, and Igor Canadi. facebook/rocksdb Wiki, 2018. https://github.com/facebook/rocksdb/wiki/Home/b6dea4ac820cf4eca18a56dd2387b57fce0f0184, accessed 19.08.2018.

[53] ceph.com. Placement groups – ceph documentation, 2016. http://docs.ceph.com/docs/master/rados/operations/placement-groups/, accessed 19.08.2018.

[54] Red Hat. Chapter 3. Placement Groups (PGs) – Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/1.3/html/storage_strategies_guide/placement_groups_pgs, accessed 19.08.2018.

[55] ceph.com. Ceph storage cluster apis – ceph documentation, 2018. http://docs.ceph.com/docs/master/rados/api/, accessed 19.08.2018.

[56] ceph.com. Bluestore config reference – ceph documentation, 2018. http://docs.ceph.com/docs/master/rados/configuration/bluestore-config-ref/, accessed 19.08.2018.

[57] Felix Freiling, Christian Riess, Sven Borchert, Andreas Dewald, Torben Griebe, Daniel Günzel, Marc Junginger, Robert Kandzia, Daniel Keller, Marion Liegl, Tilo Müller, Romy Rahnfeld, Felix Ramisch, Mortin Rojak, and Michael Spreitzenbarth. Browser- und Anwendungsforensik, November 2017.

[58] Ceph Source Code on Github. denc.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/include/denc.h, accessed 19.08.2018.

[59] Ceph Source Code on Github. encoding.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/include/encoding.h, accessed 19.08.2018.

[60] Ceph Source Code on Github. utime.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/include/utime.h, accessed 19.08.2018.

[61] Ceph Source Code on Github. uuid.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/include/uuid.h, accessed 19.08.2018.

[62] Wei Jin. Ceph bluestore freelistmanager, February 2018. http://blog.wjin.org/posts/ceph-bluestore-freelistmanager.html, accessed 19.08.2018.

[63] renren security. ceph osdmap crush, April 2016. http://itfish.net/article/61770.htm, accessed 19.08.2018.

[64] Ceph Source Code on Github. bluefs_types.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/os/bluestore/bluefs_types.h, accessed 19.08.2018.

[65] Ceph Source Code on Github. Bluefs.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/os/bluestore/BlueFS.h, accessed 19.08.2018.

[66] Ceph Source Code on Github. Bluefs.cc, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/os/bluestore/BlueFS.cc, accessed 19.08.2018.

[67] Ceph Source Code on Github. cls_rbd.cc, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/cls/rbd/cls_rbd.cc, accessed 19.08.2018.

[68] Ceph Source Code on Github. mdstypes.h, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/mds/mdstypes.h, accessed 19.08.2018.

[69] Ceph Source Code on Github. Journaltool.cc, 2018. https://github.com/ceph/ceph/blob/25508d538b70fd721543bfe22464ab1d7b899923/src/tools/cephfs/JournalTool.cc, accessed 19.08.2018.

[70] W. Zhiqiang. Ceph cache tiering introduction, 2015. https://software.intel.com/en-us/blogs/2015/03/03/ceph-cache-tiering-introduction, accessed 19.08.2018.

[71] ceph.com. crushtool - crush map manipulation tool – ceph documentation, 2018. http://docs.ceph.com/docs/master/man/8/crushtool/, accessed 19.08.2018.

[72] Nurul Azma Abdullah, Rosziati Ibrahim, and Kamaruddin Malik Mohamad. Carving thumbnail/s and embedded jpeg files using image pattern matching. *Journal of Software Engineering and Applications*, 6:22 – 66, 2013.

[73] Brian Carrier and Eugene Spafford. An event-based digital forensic investigation framework. In *The Digital Forensic Research Conference*. DFRWS, 2004. https://www.dfrws.org/sites/default/files/session-files/paper-an_event-based_digital_forensic_investigation_framework.pdf, accessed 19.08.2018.

[74] ceph.com. ceph-osd – ceph object storage daemon – ceph documentation, 2018. http://docs.ceph.com/docs/master/man/8/ceph-osd/, accessed 19.08.2018.

[75] Jeff Hamm. *Extended FAT File System. exFAT.* Paradigm Solutions, 2009. https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf, accessed 19.08.2018.

[76] microsoft.com. *How NTFS Works: Local File Systems.* Microsoft, March 2003. https://technet.microsoft.com/en-us/library/cc781134%28WS.10%29.aspx, accessed 19.08.2018.