

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Cheng-Yu Lu 184679IVCM

**ANALYSE JOURNAL OF XFS FILESYSTEM
FOR ASSISTING IN EVENT
RECONSTRUCTION**

Master's thesis

Supervisor: Pavel Laptev

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Cheng-Yu Lu 184679IVCM

**PÄEVIKUGA XFS FAILISÜSTEEMI
ANALÜÜS AITAMAKS SÜNDMUSTE
REKONSTRUEERIMISEL**

Magistritöö

Juhendaja: Pavel Laptev

Tallinn 2020

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Cheng-Yu Lu

18.05.2020

Abstract

When forensics specialists investigate incidents at a crime scene, event reconstruction helps understand what happened on the system by methods such as examining deleted files and previous versions of files in the period of time. To identify deleted files and previous versions of files on the system, analysis of journal can be applied. Journaling is one of filesystem features, which is originally used to avoid data inconsistency because of system crash or power failure by writing data into journal to record changes made on the system instead of directly writing data into filesystem. This paper discusses journaling on XFS filesystem, which is a high-performance filesystem and is a default filesystem of CentOS and RedHat Linux, and it is also found in high-end storage devices such as network-attached storage (NAS) and storage area network (SAN) systems. It also proposes the method for analysing XFS filesystem journals to assist in event reconstruction, and use a self-written Python script to verify the accuracy of logic by conducting the experiment.

This thesis is written in English and is 45 pages long, including 7 chapters, 50 figures and 12 tables.

Annotatsioon

PÄEVIKUGA XFS FAILISÜSTEEMI ANALÜÜS

AITAMAKS SÜNDMUSTE REKONSTRUEERIMISEL

Kohtueksperdid kasutavad erinevaid sündmuste rekonstrueerimise võtteid, et uurida juhtumeid. Võtteid nagu: kustutatud failide uurimine ja failide eelmiste versioonide uurimine teatud aja hetketel. Et uurida kustutatud ja eelmiseid faili versioone saab kasutada failisüsteemi päevikupidamise funktsionaalsust. Failisüsteemi päeviku pidamist kasutatakse, et vältida andmete kadu süsteemi kokku jooksmise või elektrikatkestuse ajal. Seda tehakse kirjutades failidele tehtavad muudatused päevikusse mitte otse faili süsteemi. Antud lõputöö uurib päevikuga XFS failisüsteeme, mis on suure jõudlusega failisüsteem, mis on kasutusel CentOS ja RedHat Linuxites. Sammuti kasutatakse seda failisüsteemi kallihinnalistes mäluseadmetes nagu võrgumälud (NAS) ja salvesti võrgud (SAN). Lõputöös pakutakse välja ka meetod, mis abistab kohtueksperditel kasutada XFS failisüsteemide päevikupidamise funktsionaalsust, et rekonstrueerida sündmuseid. Välja töötatud meetodite põhjal loodi Phytoni skript millega viidi läbi erinevad katsed, et kinnitada meetodi täpsus.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 45 leheküljel, 7 peatükki, 50 joonist, 12 tabelit.

List of abbreviations and terms

EXT	Extended file system
AG	Allocation group
INODE	Index node
OS	Operating system
FS	Filesystem
DF	Digital forensics
ER	Event reconstruction
NAS	Network-attached storage
SAN	Storage area network

Table of contents

List of figures.....	9
List of tables	11
1 Introduction	12
1.1 Research objective.....	12
1.2 Scope & Novelty	13
2 Related work.....	14
3 Overview of XFS filesystem	18
3.1 Allocation groups.....	18
3.2 Superblock	19
3.3 Inode core	20
3.4 Data fork.....	22
3.5 Directories.....	23
3.6 Extent.....	33
4 Journaling of XFS	35
4.1 Log records	35
4.2 Log operations.....	36
4.3 Log items	37
5 Journal analysis for event reconstruction	41
5.1 Analysis procedure.....	41
5.2 Steps of procedure.....	41
6 Experiment procedure, limitation, and comparison	47
6.1 Environment setting	47
6.2 Steps of experiment procedure.....	47
6.3 Test case	48
6.4 Test objectives.....	48
6.5 Test result.....	49
6.6 Limitation	50
6.7 Comparison with other forensics tools	52

7 Summary	55
References	56
Appendix 1 – Source Code.....	60
Appendix 2 – Shell script used for creating test case	68

List of figures

Figure 1. Layout of allocation group [6].....	18
Figure 2. Structure of superblock	19
Figure 3. Value of Read-write incompatible feature flags [28].....	20
Figure 4. Structure of inode core	21
Figure 5. Definition of type of file [30]	23
Figure 6. Definition of data fork format [28]	23
Figure 7. A directory file with data stored within an inode	23
Figure 8 Header of short form directory	24
Figure 9. Layout of directory entry.....	25
Figure 10. Value of file type [32]	26
Figure 11. Block directory with extent	26
Figure 12. Overview layout of block directory	27
Figure 13. Structure of block directory	27
Figure 14. Leaf directory with 2 data extents and leaf extent	29
Figure 15. Overview layout of leaf directory	29
Figure 16. Structure of leaf directory's 1 st data extent.....	30
Figure 17. Structure of leaf directory's 2 nd data extent.....	30
Figure 18. Node directory with 17 extents.....	31
Figure 19. Overview layout of node directory	31
Figure 20. Structure of node directory's data extent	31
Figure 21. Layout of B+ tree's root node.....	32
Figure 22. One level B+ tree with 3 leaves	32
Figure 23. Structure of B+ tree's leaf	33
Figure 24. The structure of extend record	34
Figure 25. The layout of log record header	36
Figure 26. The structure of log operation header	36
Figure 27. Value of log operation's originator [36].....	37
Figure 28. Value of log operation's flag [36].....	37
Figure 29. Magic number of log items [36]	38

Figure 30. The structure of transaction header	38
Figure 31. Inode update and Inode core.....	39
Figure 32. Values to specify which parts of the inode are being updated [36]	40
Figure 33. Steps of procedure.....	41
Figure 34. Determine size of superblock by checking sector size.....	42
Figure 35. First block of journal	43
Figure 36. Byte offset of journal's first block	43
Figure 37. Log record header	43
Figure 38. Log operations	44
Figure 39. Inode update & Change to Inode	45
Figure 40. Proposed analysis technique	46
Figure 41. Value of default XFS settings.....	47
Figure 42. Layout of test case.....	48
Figure 43. Result of test case.....	50
Figure 44. Remove of transaction types [40]	51
Figure 45. Storage properties of test case shown in UFS explorer.....	52
Figure 46. Unknown data type	53
Figure 47. Failure to recover deleted file	53
Figure 48. XFS test case shown in PhotoRec.....	54
Figure 49. Files recovered from PhotoRec.....	54
Figure 50. Unrecognizable file names and properties	54

List of tables

Table 1. Structure of superblock [27]	20
Table 2. Structure of inode core [27]	22
Table 3. Header of short form directory [27]	25
Table 4. Layout of directory entry [27].....	25
Table 5 Structure of block directory's header [27].....	27
Table 6. Structure of directory entry [27]	28
Table 7. Structure of header of B+ tree's leaf [27]	33
Table 8. The layout of log record header [27].....	36
Table 9. The structure of log operation header [27]	37
Table 10. The structure of transaction header [27].....	38
Table 11. Structure of inode update [27]	40
Table 12. Environment setting.....	47

1 Introduction

As the rapid growth of new technology, the cost to acquire huge volume of storage is getting cheaper and cheaper. Traditional filesystems are limited to support large FSs because of the initial design, so new FSs were created. XFS is one of FSs in support of large FSs, which includes mechanisms for managing large files, large numbers of files, large directories, and high performance I/O [1]. Digital forensics process models have been evolved year after year, and there are different existing models. Even though the term is used differently in these models, one of the common steps is to examine the artefact collected from the storage media [2]. Large amount of data brings obstacle to DF investigation. With limited amount of time and human resource, it is hard for DF specialists to investigate all the data. Understanding what happened on the system is crucial to prioritize order of events for improving performance of event reconstruction. ER examines events related to incident in order to explain the reason why objects have properties, such as file creation time and file location [3]. Journaling is one of FS features, which is originally used to avoid data inconsistency because of system crash or power failure by writing data into journal to keep track of changes made on system instead of directly writing data to FS. Journals can also be used to give information such as deleted files and previous versions of files for assisting in ER, but few DF tools take value of journaling into consideration [4]. XFS is a default FS of operating system such as CentOS and Red Hat Linux, and it is also found in high-end storage devices like network-attached storage (NAS) and storage area network (SAN) systems [5]. Related studies about XFS FS are insufficient, and official documentation is incomplete, which makes investigation difficult to conduct [6]. Even though there are few DF tools support XFS FS, but most of them are proprietary [7], which makes investigation on XFS system like using a “black box study” without considering how the conclusions were reached.

1.1 Research objective

Few forensics tools on the market supports XFS FS, and little of them are open source tools, which makes it difficult for investigators to interpret evidence due to lack of

knowledge of its internal structure, and also limit investigators to rely on current proprietary solutions. Besides, related studies about XFS FS forensics are limited, and the official documentation is incomplete even though XFS can be found on popular OS, such as CentOS and Red Hat Linux, and high-end storage devices like NAS and SAN, which also brings obstacle to investigations. Journal was proven to be forensically valuable to realized events occurred on system, but few of tools had applied it. Therefore, the structure of XFS FS, especially journal feature, were analysed in the paper to understand what useful information could be retrieved from journal, and the method was proposed to present an overview of all events happened on system during a period of time for reaching target of evidence examination, which reviews and identifies all properties of objects, and is also a first step in event reconstruction process.

1.2 Scope & Novelty

As mentioned above, the objective of the paper is mainly focused on evidence examination in event reconstruction process, which is an important first step to list and identify characteristics of entire objects occurred on system in a period of time. To achieve the objective, the journal of XFS FS was chosen as the primary scope to verify the assumption that that it is possible for DF investigators to realize events occurred on systems through journal.

The internal structure of journal on XFS FS has not yet been researched in detail, which is short of academic papers to introduce operation principles behind it, and the official documentation had already stopped updating. The novelty of the paper is to analyse the journal of XFS FS, and propose a method that how journal can be applied in XFS system forensics. Besides, result of this paper was used to compare with current proprietary and open source tools to comprehend advantages and disadvantages by using retrieved information from journal.

The rest of this paper is organized as follows. Related work is presented in Section 2 to review and learn from previous research work. The overview of XFS FS is introduced in Section 3, and journaling feature of XFS FS is described in Section 4. In Section 5, analysing of journal is performed to realize how journaling can help in ER, and experiment procedure is in Section 6 to verify the logic from Section 5 is feasible. In the end, summary and future work is presented in Section 7.

2 Related work

Event reconstruction

To get better understanding of digital investigation, important definitions, such as digital data, and digital object, were introduced to help comprehend fundamental terminology of investigation. Digital object is composed of digital data, and digital data can be presented in physical or numerical form. Characteristics of digital object have states to identify their values. When there is change of state in digital object, digital event is occurred, and digital incident is recognized if law violation is involved in digital event. An event-based DF framework, which was improved from process model using in physical crime scenes, was proposed by Carrier and Spafford. The model is divided into five phases, and event reconstruction is critical in both phases of physical and digital crime scene investigation to reconstruct physical and digital events for answering questions about incident [8]. Carrier and Spafford mentioned that the objective of digital investigation was to find a person who should be responsible of incident. For reaching the objective, event reconstruction plays an important role. Event reconstruction is the process to realize cause and effect of objects by examining characteristics of object, such as creation time and location of objects, and reconstructing timeline of events that lead to incident to test the hypothesis. Moreover, evidence examination is the first step in event reconstruction to record properties of evidence by inspecting all associated information of objects and distinguishing characteristics they have [3]. Large volume of data in investigation due to increase of storage capacity, resulted in one of challenges in event reconstruction. With overwhelming amount of data, investigators had trouble conducting analysis with limited amount of time and resource as stated by Chabot, Aurélie, Christophe, and Tahar. About event reconstruction tools, data source can be used as one of ways to categorize event reconstruction methods. Single source event reconstruction tool, like timestamp in FS, cannot present whole picture of events happened on system. Therefore, multi-source approach is recommended to provide complete information by collecting data from sources, such as logs files, FS, and OS information, but it is also complicated to analyse data from distinct sources [9]. Jeyaraman and Atallah classified event reconstruction tools as tools using ex post evidence and ex ante logging according to usage time of tool. Hard

disk image is the main source of ex post evidence, and tools like The Sleuth Kit examines retrieved data after occurrence of incident. In comparison, host-based logging, such as Windows Event Viewer, which starts recording log events before happening of incident, is an example of ex ante logging tool, and it provides more information as reference [10].

Desired information

Questions related to who, what, when, how, where and why should be taken into consideration by investigators, but only few of them could be responded by information collected on today's systems because OSs and FSs were designed without thinking DF in mind, or because storage space was limited as noted by Buchholz, and Spafford. Preferable information, just as file creation time and user id of the process performing file creation, access, or modification, were also discussed, but how to retrieve and collect them acts differently depending on characteristic of each system [11]. Lillis, Becker, Sullivan, and Scanlon mentioned the significance of timeline reconstruction for assisting in investigation, but it could encounter problems, such as retrieving temporary information from unstructured text, and collecting time data with inconsistent format from different sources [12]. Fourteen types of timestamp changing rule were categorized by Jang, Hwang, and Kim to understand actions leading to change of timestamp on system. For instance, creation, copy, and copy from different file system, can alter the value of timestamp, which is useful information to help identify factors influencing change of timestamp under different conditions for assisting in investigation [13].

Filesystem journal

Journaling filesystem was originally designed to prevent from time-consuming consistency checks on filesystem through reviewing journal to see recent disk write operations for decreasing required time to remount the system. FSs, such as XFS, only recorded limited amount of operations which influence metadata, but it was enough to get the system back to consistent state by log records. Nevertheless, file content was still possible to get corrupted because operations on blocks of file data were not logged as said by Bovet and Cesati [14]. Carrier remarked that file system journals had not been applied to most forensics tools yet even though it was valuable to investigations. FS events happened lately can be observed by journal, which can help with event reconstruction of incident [15]. Precious evidence in association with network intrusions, financial fraud,

software piracy, and child pornography could be retrieved from file system journals. Also, the next generation of computer forensics tools should take file system journal into consideration as said by Choo [16]. Swenson, Philhps, and ShenoI also discussed that deleted files and previous version of files could be observed through analysing journal feature of journaling FSs, and only few digital forensics tools considered journaling during recovery process. Reiser and ext3 journaling analysis were conducted to verify the argument, and research on other FSs, such as NTFS of Windows and HFSJ of Mac OS X, could be executed as future work [4].

XFS filesystem

“A single XFS file system can be 18,000 petabytes and a single file can be 9,000 petabytes”. XFS also provides marvellous input and output execution, which makes it a suitable candidate as a large FS. Besides, many of techniques used in JFS, such as extent-based addressing structures and dynamic inode allocation, could also be discovered in XFS as mentioned by Best [17]. Lu and Arpaci-Dusseau discussed the complexity of FSs, and it kept getting more and more complicated. XFS had around 64K line of codes, which increased difficulty for understanding its internal structure [18]. XFS data loss bug was mentioned by Yang, Twohey, Engler, and Musuvathi that crash during the creation of “lost+found” could lead to corruption in root directory even on a clean file system, which could destroy the whole file system [19]. Park, Chang, and Shon gave an overview about analysing the XFS FS based on understanding its internal structure, and they demonstrated a method to recover the deleted files. Journaling was applied as a part of elements to retrieve metadata of deleted files, but it was conducted through reverse engineering without describing the underlying operations and layout of journal because of incomplete official documentation, which limited the possible functionality of journaling in XFS forensics [6]. About journaling mode, three kinds of journaling modes were introduced by Prabhakaran and Arpaci-Dusseau, and they were writeback mode, ordered mode, and data journaling mode. Only FS metadata is written into journal and there is no ordering between journal and data writes in writeback mode. In contrast, data blocks written to fixed location are ordered prior to filesystem metadata in journal in ordered mode, which provides guarantee for both data and metadata to restore from inconsistent state after recovery. In data journaling mode, both data and metadata are written into journal before they are written to their locations, which gives the same consistency as ordered mode but with distinct performance [20]. Tamma and

Venugopalan verified that XFS adopted ordered journaling mode by conducting experiment to postpone the data blocks in the SBA driver, and examined the corresponding metadata writes were postponed by the same time [21]. XFS applied write barrier to assure the recovery, but write ordering could influence performance, which was a trade-off between performance and reliability as stated by Konishi, Amagai, Sato, Hifumi, Kihara, and Moriai [22]. Kieseberg, Schrittwieser, Mulazzani, Huber, and Weippl analysed the internal structure of B+ tree, which was one of data structures used in XFS system to store data. Owing to the characteristics of B+ tree, it can disturb malicious personnel to modify the records, which can prevent data from manipulation [23]. Signatures of B+ trees, proposed by Kieseberg, Schrittwieser, Morgan, Mulazzani, Huber, and Weippl, could be used in index reorganizations to help reconstruct old version of files [24]. Majore, Lee, and Shon mentioned the large filesystem and efficient parallelism support of XFS FS, which makes XFS FS be used commercially and scientifically. Two files explore and recovery tools, UFS explorer and PhotoRec, were compared briefly to know their recovery capability toward XFS FS. The disadvantages of dependence on few supporting tools were indicated to strengthen the reason to develop new tool for XFS FS [25].

3 Overview of XFS filesystem

XFS is a journaling FS which was originally created by Silicon Graphics for IRIX OS, and it was ported to Linux kernel and supported on most Linux distributions [26]. This section describes the overview of XFS FS, especially features which are used together with the journaling feature in Section 4 for understand how journaling can be used as a support in ER in Section 5.

3.1 Allocation groups

XFS FS is partitioned into allocation groups. Each AG has the same size, and can be thought of as an individual FS with its own superblock, free space management, and inode allocation and tracking [6]. Because of characteristic that every AG manages its own space, processing multiple operations simultaneously in XFS is feasible. Multiple operations are executed in AGs in FS, but only one operation can be written in each AG at the same time [25]. Other FSs just as EXT use block group, which is similar to the concept of AG. File objects in XFS FS can be allocated across AGs, and those can be reached by using AG relative pointer. The layout of AG is shown in Figure 1.

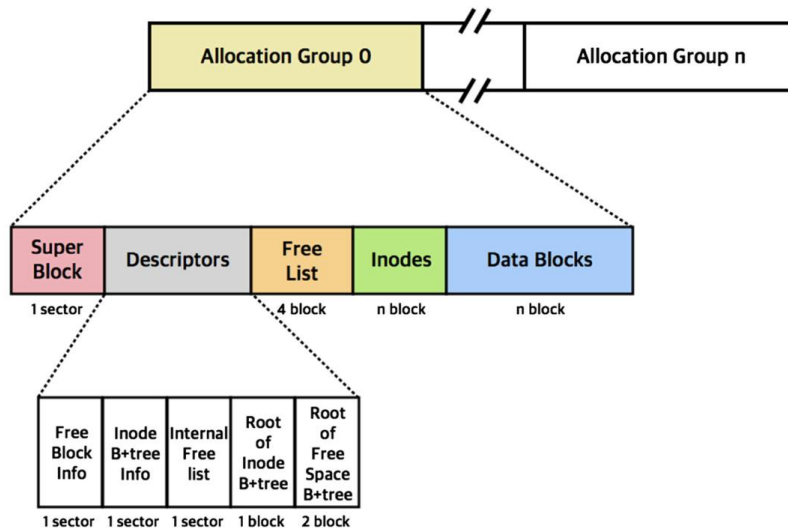


Figure 1. Layout of allocation group [6]

3.2 Superblock

Superblock with one sector size in length contains fields to describe general information of whole FS, and it is stored in big-endian order, which is used for most of XFS fields except for log items which are in host byte order. Overall system information such as block size, inode size, number of free inodes is documented in superblock. Primary superblock located in AG 0 is read to mount FS successfully, and secondary superblocks located at the opening of each AG are used as backups when primary superblock is corrupted [27]. The structure of superblock is shown as in Figure 2, and description of important fields is summarized in Table 1. The correspondence between highlighted area in figures and fields in tables can be classified into two types, and they can be matched orderly. If data is a fixed sized structure, the fields are interpreted by offsets from the beginning. Otherwise, the fields are interpreted by length as shown later in Table 4 if the data is variable size.

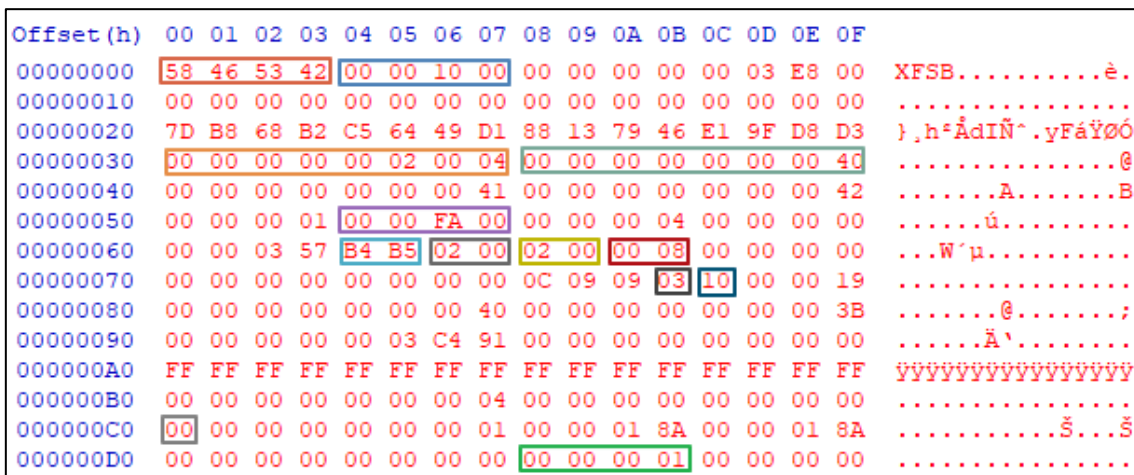


Figure 2. Structure of superblock

Offset (in hex)	Variable name	Description
0x00-03 (4 bytes)	sb_magicnum	Magic number “XFSB”.
0x04-07 (4)	sb_blocksize	Block size (in bytes), which is basic unit of space allocation.
0x30-37 (8)	sb_logstart	First block number for the journaling log.

0x38-3F (8)	sb_rootino	Inode number of root directory.
0x54-57 (4)	sb_agblocks	AG size (in blocks).
0x64-65 (2)	sb_versionnum	Lower nibble is used to identify FS version.
0x66-67 (2)	sb_sectsize	Sector size (in bytes).
0x68-69 (2)	sb_inodesize	Inode size (in bytes).
0x6A-6B (2)	sb_inopblock	Number of inodes per block.
0x7B (1)	sb_inopblog	Log 2 value of sb_inopblock.
0x7C (1)	sb_agblklog	Log 2 value of sb_agblocks.
0xC0 (1)	sb_dirblklog	Log 2 multiplier that determines the granularity of directory block allocations in fsblocks.
0xD8-DB (4)	sb_features_incompat	Read-write incompatible feature flags. The kernel cannot read or write this FS if it doesn't understand the flag. The defined value is shown in Figure 3.

Table 1. Structure of superblock [27]

```

465 #define XFS_SB_FEAT_INCOMPAT_FTYPE (1 << 0) /* filetype in dirent */
466 #define XFS_SB_FEAT_INCOMPAT_SPINODES (1 << 1) /* sparse inode chunks */
467 #define XFS_SB_FEAT_INCOMPAT_META_UUID (1 << 2) /* metadata UUID */

```

Figure 3. Value of Read-write incompatible feature flags [28]

3.3 Inode core

Index node (Inode) includes metadata of FS objects such as regular files and directories. Each FS object is bind to one inode number. Files can have the same inode number when hard links are created. Hard link is like an alias to the original file which points to the same block location as the original file, which means deletion of original file will not

affect access of data. In comparison, soft link has its own inode number, and it is like a shortcut pointing to the original file without containing the content of data, so deletion of the original file will make soft link inaccessible. Inode in XFS FS is composed of three parts: inode core, data fork, and attribute fork [29]. Inode core contains overall information of inode, and it occupies 176 bytes on a v5 FS. The structure of inode core is shown as in Figure 4, and description of important fields is summarized in Table 2.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	49	4E	41	ED	03	01	00	00	00	00	00	00	00	00	00	00	INAI.....
00000010	00	00	00	03	00	00	00	00	00	00	00	00	00	00	00	00
00000020	5E	6E	1E	BC	07	27	4B	1A	5E	6E	1E	A1	3A	3C	17	5A	^n.4.'K.^n.j;<.Z
00000030	5E	6E	1E	A1	3A	3C	17	5A	00	00	00	00	00	00	00	1E	^n.j;<.Z.....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000060	FF	FF	FF	FF	F0	D6	05	B6	00	00	00	00	00	00	00	05	ÿÿÿÿøÖ.¶.....
00000070	00	00	00	01	00	00	00	00	0A	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	5E	6E	1E	4A	33	B3	36	D8	00	00	00	00	00	00	40		^n.J3^6ø.....@
000000A0	7D	B8	68	B2	C5	64	49	D1	88	13	79	46	E1	9F	D8	D3	},h^AdiN^yFáÿøó

Figure 4. Structure of inode core

Offset (in hex)	Variable name	Description
0x00-01 (2 bytes)	di_magic	Magic number “IN”.
0x02-03 (2)	di_mode	The first four bits are used for type of file, and the rest 12 bits are mode access bits.
0x04 (1)	di_version	Version of inode.
0x05 (1)	di_format	Format of data fork.
0x08-0B (4)	di_uid	Owner’s UID.
0x10-13 (4)	di_nlink	Number of links to the inode from directories.
0x30-33 (4)	di_ctime	Last changed time.
0x38-3F (8)	di_size	Regular file: file size (in bytes); Directory: space taken by directory entries; Link: length of symlink

0x40-47 (8)	di_nblocks	Number of blocks used to store the inode's data fork.
0x4C-4F (4)	di_nextents	Number of data extents associated with this inode.
0x50-51 (2)	di_anextents	Number of extended attribute extents associated with this inode.
0x52 (1)	di_forkoff	Offset to inode's extended attribute fork.
0x53 (1)	di_aformat	Format of the attribute fork.
0x90-93 (4)	di_crtime	Creation time.
0x98-9F (8)	di_ino	Inode number.

Table 2. Structure of inode core [27]

3.4 Data fork

Data fork is determined by both format of data fork and type of file listed in inode core, and it starts after inode core at offset 176 (0xb0) in a v3 inode. In Figure 5, stat.h header file, which can be found on POSIX and Unix-like systems, contains definition of constants describing types of file. Type of file includes categories just as regular files, directories, symbolic links, and other file types. For example, S_IFREG represents regular file, and S_IFDIR represents directory. The calculation of filetype is determined by file bitwise AND with S_IFMT. Format of data fork specifies how the data is stored in FS and is defined in kernel file as shown in Figure 6. XFS_DINODE_FMT_LOCAL means all data is stored within inode. XFS_DINODE_FMT_EXTENTS expects that additional extent list is used to point to the location where the data is stored. XFS_DINODE_FMT_BTREE anticipates that root node of B+ tree is stored in data fork, and it points to other nodes or leaves of B+ tree [27]. In this paper, directories will be focused because FS objects are listed as directory entries on the FS. All three type of data fork format mentioned above can be seen in directory. For example, as shown in Figure 7, value of byte offset 0x02's left nibble equals to 4 in decimal, and value of byte offset 0x05 equals to 1 in decimal. By interpreting the values, this inode is a directory and its data is stored locally within inode.

```

9  #define S_IFMT 00170000 mask for file type (octal)
10 #define S_IFSOCK 0140000 socket
11 #define S_IFLNK 0120000 symbolic link
12 #define S_IFREG 0100000 regular
13 #define S_IFBLK 0060000 block special
14 #define S_IFDIR 0040000 directory
15 #define S_IFCHR 0020000 character special
16 #define S_IFIFO 0010000 FIFO (named pipe)
17 #define S_ISUID 0004000
18 #define S_ISGID 0002000
19 #define S_ISVTX 0001000
20
21 #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK) is it a symbolic link?
22 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG) is it a regular file?
23 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR) is it a directory?
24 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR) is it a character device?
25 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK) is it a block device?
26 #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) is it a FIFO (named pipe)?
27 #define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK) is it a socket?

```

Figure 5. Definition of type of file [30]

```

917 /*
918  * Values for di_format
919  *
920  * This enum is used in string mapping in xfs_trace.h; please keep the
921  * TRACE_DEFINE_ENUMs for it up to date.
922  */
923 enum xfs_dinode_fmt {
924     XFS_DINODE_FMT_DEV,          /* xfs_dev_t */
925     XFS_DINODE_FMT_LOCAL,       /* bulk data */
926     XFS_DINODE_FMT_EXTENTS,    /* struct xfs_bmbt_rec */
927     XFS_DINODE_FMT_BTREE,      /* struct xfs_bmdr_block */
928     XFS_DINODE_FMT_UUID        /* added long ago, but never used */
929 };

```

Figure 6. Definition of data fork format [28]

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	49	4E	41	ED	03	D1	00	00	00	00	00	00	00	00	00	00	INAI.....
00000010	00	00	00	03	00	00	00	00	00	00	00	00	00	00	00	00
00000020	5E	6E	1E	BC	07	27	4B	1A	5E	6E	1E	A1	3A	3C	17	5A	^n.4.'K.^n.j:<.Z
00000030	5E	6E	1E	A1	3A	3C	17	5A	00	00	00	00	00	00	00	1E	^n.j:<.Z.....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000060	FF	FF	FF	FF	F0	D6	05	B6	00	00	00	00	00	00	00	05	ÿÿÿÿ80.1.....
00000070	00	00	00	01	00	00	00	0A	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	5E	6E	1E	4A	33	B3	36	D8	00	00	00	00	00	00	00	40	^n.J3*60.....@
000000A0	7D	B8	68	B2	C5	64	49	D1	88	13	79	46	E1	9F	D8	D3	},h*AdIN~.yFáÿ0Ó

Figure 7. A directory file with data stored within an inode

3.5 Directories

Directory is composed of directory entries, and can be differentiated by the value of file mode specified in inode core. Each directory entry contains name of file and inode number, which can be used to match with information documented in inode core to

understand the status change of file for journal analysis in later section. The size of a directory block is calculated by $sb_blocksize \times 2^{sb_dirblklog}$, which is different from FS block size. Directory is categorized into the following types: short form directory, block directory, leaf directory, node directory, and B+ tree directory [27].

Short form directory

Short form directory is used when amount of directory entries is able to store within inode. Short form directory begins with a header to record number of directory entries and parent inode, and it is followed by an array of variable-length directory entries. Each directory entry contains fields like file name, inode number, parent and so on [31]. To improve performance without checking file type from inode every time, file type is cached in directory entry but only if XFS_SB_FEAT_INCOMPAT_FTYPE is set in superblock. Amount of directory entries saved within inode is uncertain, and it depends on factors just as length of file name, inode size, and extended attribute fork [27]. For instance, as shown in Figure 8 and Table 3, there are two directory entries under this short form directory, so each directory entry can be found immediately after the header. Taking first directory entry created for testing as an example, it is called “lqaz”, which is a regular file with inode number 67 in decimal as shown in Figure 9 and Table 4.

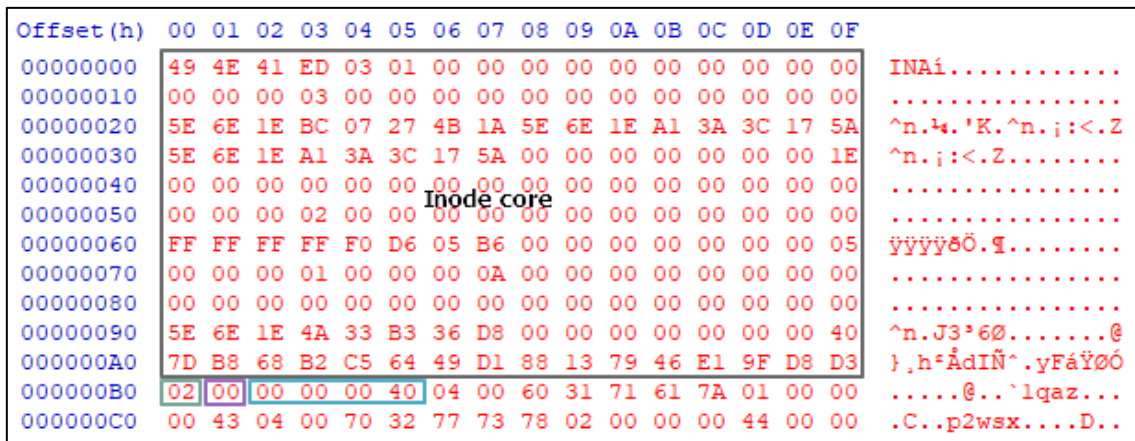


Figure 8 Header of short form directory

Offset (in hex)	Variable name	Description
0x00 (1 byte)	count	Number of directory entries.

0x01 (1)	i8count	Number of directory entries requiring 64-bit entries.
0x02-05 (4)	parent	Inode number of parent directory.

Table 3. Header of short form directory [27]

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	49	4E	41	ED	03	01	00	00	00	00	00	00	00	00	00	00	INaI.....
00000010	00	00	00	03	00	00	00	00	00	00	00	00	00	00	00	00
00000020	5E	6E	1E	BC	07	27	4B	1A	5E	6E	1E	A1	3A	3C	17	5A	^n.4.'K.^n.j;<.Z.....
00000030	5E	6E	1E	A1	3A	3C	17	5A	00	00	00	00	00	00	00	1E	^n.j;<.Z.....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000060	FF	FF	FF	FF	F0	D6	05	B6	00	00	00	00	00	00	05	yyy80.4.....	
00000070	00	00	00	01	00	00	00	0A	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	5E	6E	1E	4A	33	B3	36	D8	00	00	00	00	00	00	40	^n.J3*60.....@	
000000A0	7D	B8	68	B2	C5	64	49	D1	88	13	79	46	E1	9F	D8	D3	},h*AdIN".yFaY00
000000B0	02	00	00	00	00	40	04	00	60	31	71	61	7A	01	00	00@..'lqaz...
000000C0	00	43	04	00	70	32	77	73	78	02	00	00	00	44	00	00	..C..p2wsx....D..

2nd entry

1st entry

Figure 9. Layout of directory entry

Length (in bytes)	Variable name	Description
1	namelen	Length of name (in bytes)
2	offset	Offset tag used to assist with directory iteration.
varies, depending on namelen	name	Name of directory entry
1 byte	ftype	File type of inode, the value of file type is defined in source code as shown in Figure 10.
4 or 8 bits, depending on count and i8count in header	inumber	Inode number

Table 4. Layout of directory entry [27]

```

145  /*
146   * Dirents in version 3 directories have a file type field. Additions to this
147   * list are an on-disk format change, requiring feature bits. Valid values
148   * are as follows:
149   */
150  #define XFS_DIR3_FT_UNKNOWN      0
151  #define XFS_DIR3_FT_REG_FILE    1
152  #define XFS_DIR3_FT_DIR         2
153  #define XFS_DIR3_FT_CHRDEV     3
154  #define XFS_DIR3_FT_BLKDEV     4
155  #define XFS_DIR3_FT_FIFO       5
156  #define XFS_DIR3_FT_SOCKET     6
157  #define XFS_DIR3_FT_SYMLINK    7
158  #define XFS_DIR3_FT_WHT        8
159
160  #define XFS_DIR3_FT_MAX        9

```

Figure 10. Value of file type [32]

Block directory

Block directory is used when amount of directory entries exceeds free space in inode. Extent map is stored in data fork area within inode core, and block of directory entries is pointed to by the offset specified in extent map [6]. For example, as shown in Figure 11, directory with extent inside inode can be recognized by checking left nibble of byte offset 0x02 (4: directory) and 0x05 (2: extent list) in inode core, and there is one extent record following inode core by checking byte offset 0x4C-4F. The fields of extent record are not byte aligned but presented in bits, so the conversion between hexadecimal and binary is necessary. The introduction of extent and address conversion are presented in later sections. Taking the extent record as an example, the location of extent record can be reached by converting absolute block number into block address, and it is offset 65536 bytes in this case.

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 49 4E 41 ED 03 02 00 00 00 00 00 00 00 00 00 00  INAI.....
00000010 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000020 5E 7F BD 95 30 C1 D4 80 5E 7F BD 94 2E BB 08 29  ^..0ÁÔÈ^..»..)
00000030 5E 7F BD 94 2E BB 08 29 00 00 00 00 00 00 00 10  ^..»..)
00000040 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01  .....
00000050 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000060 FF FF FF FF 08 1A 69 A4 00 00 00 00 00 00 00 36  yÿÿÿ..i#.....6
00000070 00 00 00 01 00 00 02 00 00 00 00 00 00 00 00 00  .....
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000090 5E 7F BD 74 36 E3 1A 08 00 00 00 00 00 00 00 40  ^..st6Ä.....@
000000A0 A9 4F 5B 76 44 20 4D EA B8 B0 F7 A2 D3 AF 9D 88  ©O[vD Mè,°÷cÓ~.^
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 01  .....
Hex: 0000000000000000000000000000000000000001, Bin: 00 ... 00100000000000000000000000000001
Flag: 0 (1 bit)
Logical block offset: 0 .. 0 (54 bits) = 0 in decimal
Absolute block number: 0 .. 010000 (52 bits) = 16
Number of blocks: 0 .. 1 (21 bits) = 1

```

Figure 11. Block directory with extent

As seen from the layout of block directory in Figure 12, block directory begins with a header to describe general information, and it is followed by free space array to track unused space. Next, directory entries are listed to provide information such as inode number and name, which can be used for analysis later. Unused entries are free space documented in free space array, which can be used when new directory entries added. Leaf entry stores hash value of directory entry's name for fast directory entry lookup, and tail record documents amount of leaf and free leaf entries [1].

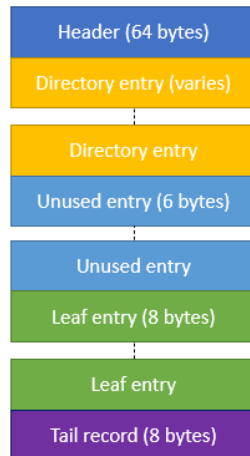


Figure 12. Overview layout of block directory

Continuing with the sample case, block directory is reached by moving 65536 bytes from the beginning of FS, and it can be assured by magic number at the first four bytes. Different from short form directory seen in last section, “.” and “..” are always the first two directory entries, which are used to present current directory and parent directory [27]. The third directory entry is a regular file called “test01” with inode number 67, which was created for testing. The structure of header and directory entry are shown in Figure 13 and Table 5 and 6.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	58	44	42	33	61	CC	75	4A	00	00	00	00	00	00	00	80	XDB3aïuJ.....€
00000010	00	00	00	01	00	00	00	02	A9	4F	5B	76	44	20	4D	EA@O[vD Mè
00000020	B8	B0	F7	A2	D3	AF	9D	88	00	00	00	00	00	00	00	40	,°±eÓ¯.^.....@
00000030	05	10	09	48	00	00	00	00	00	00	00	00	00	00	00	00	...H.....
00000040	00	00	00	00	00	00	00	40	01	2E	02	00	00	00	00	40@.....@
00000050	00	00	00	00	00	00	00	40	02	2E	2E	02	00	00	00	50@.....P
00000060	00	00	00	00	00	00	00	43	06	74	65	73	74	30	31	01C.test01.
00000070	00	00	00	00	00	00	00	60	00	00	00	00	00	00	00	44`.....D
00000080	06	74	65	73	74	30	32	01	00	00	00	00	00	00	00	78	.test02.....x

Figure 13. Structure of block directory

Offset (in hex)	Variable name	Description
0x00-03 (4 bytes)	magic	Magic number “XDB3”.
0x28-2F (8)	owner	Inode number that this block belongs to.

Table 5 Structure of block directory's header [27]

Length (in bytes)	Variable name	Description
8	inumber	Inode number that this entry points to.
1	namelen	Length of name (bytes).
namelen	name	Name of this entry.
1	ftype	File type of the inode. The defined value is the same as short form directory shown in Figure 10.
varies	padding	Padding for 64 bits alignment
2	tag	Offset from the start of block (in bytes).

Table 6. Structure of directory entry [27]

Leaf directory

Leaf directory is used when directory entries occupy more than one extent. Extent list is adopted to point to block of directory entries, and leaf has its own separate extent instead of storing all together within the same extent as the block directory does. Leaf extent is the last extent in extent list, and logical block offset of leaf extent is calculated by $XFS_DIR2_LEAF_OFFSET (32\text{ GB}) / sb_blocksize$ [27]. As displayed in Figure 14, leaf directory can also be identified by using the same method as described in block directory section, and it has two data extents and leaf extent in this sample case. From the overview layout of leaf directory shown in Figure 15, leaf entries and tail record are moved to its own leaf extent with additional header and free space array to track unused entries in each extent, which makes it have more space to save directory entries.

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 49 4E 41 ED 03 02 00 00 00 00 00 00 00 00 00 00 INAI.....
00000010 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000020 5E 7E 4E 32 2D B7 02 2C 5E 7E 4E 28 1D 80 9F BE ^~N2-.,^~N(.eY%
00000030 5E 7E 4E 28 1D 80 9F BE 00 00 00 00 00 00 30 00 ^~N(.eY%.....0.
00000040 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 .....
00000050 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000060 FF FF FF FF 24 38 77 4C 00 00 00 00 00 00 01 F9 yyyyswL.....u
00000070 00 00 00 01 00 00 01 77 00 00 00 00 00 00 00 00 .....w.....
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000090 5E 7E 4B 4B 20 D9 9B 60 00 00 00 00 00 00 00 40 ^~KK Ū>`.....@
000000A0 72 63 6C 46 25 0F 4C 2D 8D A5 1C D5 E4 03 94 43 rclF$.L-.$.ôa."C
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 01 1st.data.extent....
000000C0 00 00 00 00 00 02 00 00 00 00 00 00 02 40 00 02 2nd.data.extent.@..
000000D0 00 00 00 01 00 00 00 00 00 00 00 00 02 20 00 01 Leaf-extent.....

1st data extent
Hex: 0 .. 02000001, Bin: 0 .. 01000000000000000000000000000001
Flag: 0 (1 bit)
Logical block offset: 0 .. 0 (54 bits) = 0 in decimal
Absolute block number: 0 .. 010000 (52 bits) = 16
Number of blocks: 0 .. 1 (21 bits) = 1

2nd data extent
Flag: 0 (1 bit)
Logical block offset: 1
Absolute block number: 18
Number of blocks: 2

Leaf extent
Flag: 0 (1 bit)
Logical block offset: 8388608
Absolute block number: 17
Number of blocks: 1

```

Figure 14. Leaf directory with 2 data extents and leaf extent

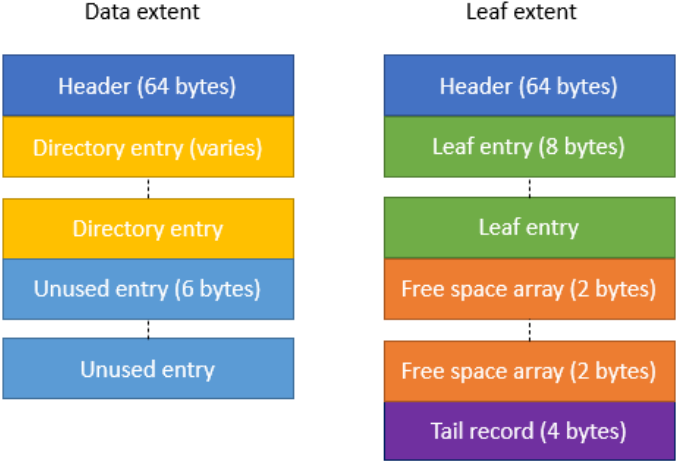


Figure 15. Overview layout of leaf directory

In the sample case as shown in Figure 16 and 17, both data extents can be reached by using the same address conversion as mentioned in later section. Leaf directory can be identified by its magic number at first four bytes, and it also begins with two directory entries to indicate current and parent directory in its first data extent. The structure of leaf directory’s header and directory entry are the same as block directory except that magic number “XDD3” is used in leaf directory.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	58	44	44	33	57	31	27	7B	00	00	00	00	00	00	00	80	XDD3Wl' {.....€
00000010	00	00	00	01	00	00	00	58	72	63	6C	46	25	0F	4C	2DXrclF%.L-
00000020	8D	A5	1C	D5	E4	03	94	43	00	00	00	00	00	00	00	40	.¥.Öä."C.....@
00000030	0F	F0	00	10	00	00	00	00	00	00	00	00	00	00	00	00	.ð.....
00000040	00	00	00	00	00	00	00	40	01	2E	02	00	00	00	00	40@.....@
00000050	00	00	00	00	00	00	00	40	02	2E	2E	02	00	00	00	50@.....P
00000060	00	00	00	00	00	00	00	43	06	74	65	73	74	30	31	01C.test01.
00000070	00	00	00	00	00	00	00	60	00	00	00	00	00	00	00	44`.....D
00000080	06	74	65	73	74	30	32	01	00	00	00	00	00	00	00	78	.test02.....x

Figure 16. Structure of leaf directory's 1st data extent

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	58	44	44	33	DF	DA	0E	FA	00	00	00	00	00	00	00	90	XDD3&U.ú.....
00000010	00	00	00	01	00	00	00	58	72	63	6C	46	25	0F	4C	2DXrclF%.L-
00000020	8D	A5	1C	D5	E4	03	94	43	00	00	00	00	00	00	00	40	.¥.Öä."C.....@
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	01	09	07	74	65	73	74	31	36	37test167
00000050	01	00	00	00	00	00	00	40	00	00	00	00	00	00	01	0A@.....
00000060	07	74	65	73	74	31	36	38	01	00	00	00	00	00	00	58	.test168.....X

Figure 17. Structure of leaf directory's 2nd data extent

Node directory

Node directory is similar to leaf directory, but additional node extent is added to track location of leaf entries, and free space array is moved to its own extent. Node directory is used when there are multiple leaf extents, which means there are more directory entries to be recorded than leaf directory [27]. As displayed in Figure 18, node directory can also be identified by using the same method as shown in block directory section, and it has ten data extents, node extent, five leaf extents, and freeindex extent in this sample case. The overview layout of node directory is displayed in Figure 19.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	49	4E	41	ED	03	02	00	00	00	00	00	00	00	00	00	00	INAi.....
00000010	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000020	5E	81	E9	1C	11	57	49	FF	5E	81	E9	1A	34	63	75	36	^.é..WIÿ^..é.4cu6
00000030	5E	81	E9	1A	34	63	75	36	00	00	00	00	00	00	D0	00	^.é.4cu6.....D.
00000040	00	00	00	00	00	00	00	15	00	00	00	00	00	00	00	11
00000050	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000060	FF	FF	FF	FF	04	55	D0	50	00	00	00	00	00	00	08	37	ÿÿÿÿ.UDP.....7
00000070	00	00	00	01	00	00	03	42	00	00	00	00	00	00	00	00B.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	5E	81	E8	9F	15	36	19	78	00	00	00	00	00	00	00	40	^.èÿ.6.x.....@
000000A0	12	C9	0B	AB	DD	4C	4B	A5	8A	B3	17	55	07	60	C7	9A	.É.«ÿLKÿŠ³.U.ÇŠ
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00	01
000000C0	00	00	00	00	00	00	02	00	00	00	00	00	02	40	00	02@.....
000000D0	00	00	00	00	00	00	06	00	00	00	00	00	09	A0	00	01
000000E0	00	00	00	00	00	00	08	00	00	00	00	00	0D	00	00	01
000000F0	00	00	00	00	00	00	0A	00	00	00	00	00	0D	40	00	02	..Data extent@.....
00000100	00	00	00	00	00	00	0E	00	00	00	00	00	15	80	00	01é.....
00000110	00	00	00	00	00	00	10	00	00	00	00	00	15	C0	00	01À.....
00000120	00	00	00	00	00	00	12	00	00	00	00	00	1B	00	00	02
00000130	00	00	00	00	00	00	16	00	00	00	00	00	1B	60	00	01`.....
00000140	00	00	00	00	00	00	18	00	00	00	00	00	23	80	00	01#é.....
00000150	00	00	00	01	00	00	00	00	00	00	00	00	02	20	00	01	..Node extent.....
00000160	00	00	00	01	00	00	02	00	00	00	00	00	09	C0	00	02À.....
00000170	00	00	00	01	00	00	06	00	00	00	00	00	0D	20	00	01
00000180	00	00	00	01	00	00	08	00	00	00	00	00	15	A0	00	01	..Leaf extent.....
00000190	00	00	00	01	00	00	0A	00	00	00	00	00	15	E0	00	01à.....
000001A0	00	00	00	01	00	00	0C	00	00	00	00	00	1B	40	00	01@.....
000001B0	00	00	00	02	00	00	00	00	00	00	00	00	09	80	00	01	.Freeindex extent.....

Figure 18. Node directory with 17 extents

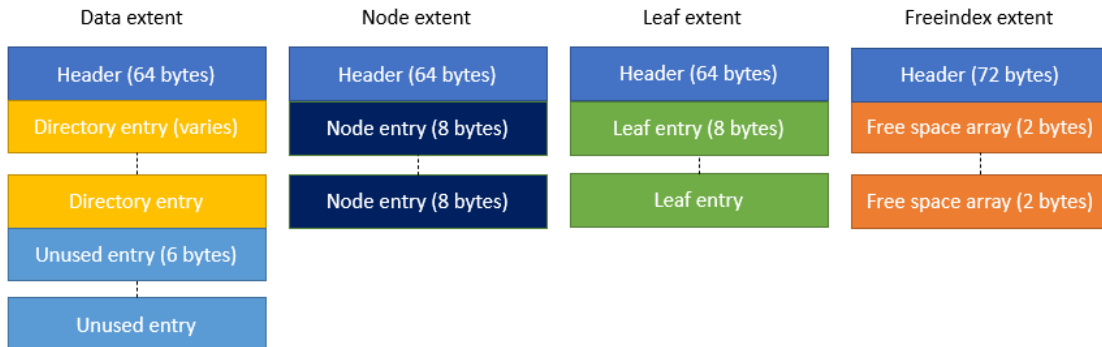


Figure 19. Overview layout of node directory

In the sample case, data extents can be reached by using the same address conversion as mention in later section. The structure of leaf directory's header and directory entry are the same as leaf directory.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	58	44	44	33	A8	9F	41	51	00	00	00	00	00	00	00	80	XDD3"ÿAQ.....é
00000010	00	00	00	01	00	00	00	02	12	C9	0B	AB	DD	4C	4B	A5É.«ÿLKÿ
00000020	8A	B3	17	55	07	60	C7	9A	00	00	00	00	00	00	40		Š³.U.ÇŠ.....@
00000030	0F	FO	00	10	00	00	00	00	00	00	00	00	00	00	00	00	.8.....
00000040	00	00	00	00	00	00	00	40	01	2E	02	00	00	00	00	40@.....@
00000050	00	00	00	00	00	00	00	40	02	2E	2E	02	00	00	00	50@.....P
00000060	00	00	00	00	00	00	43	08	74	65	73	74	30	30	30	C.test000
00000070	81	01	00	00	00	00	00	60	00	00	00	00	00	00	00	44	l.....`.....D
00000080	08	74	65	73	74	30	30	30	32	01	00	00	00	00	00	78	.test0002.....x

Figure 20. Structure of node directory's data extent

B+ tree directory

B+ tree directory is used when extent maps exceed space available in inode. B+ tree extent list is adopted to store block of directory entries. Root node of B+ tree is stored in data fork area of inode with information like offset and block number for locating B+ tree's leaf and node, and its layout is displayed in Figure 21. Directory entries are stored in data extent pointed by leaf of B+ tree, and leaf of B+ tree can be directly reached if B+ tree is only one level, or indirectly reached with node extent involved if B+ tree is multilevel [27]. There are also four types of extents in B+ tree directory as described in node directory. For example, as shown in Figure 22, B+ tree directory can be recognized by checking left nibble of byte offset 0x02 (4: directory) and 0x05 (3: B+ tree root). Root node of B+ tree begins with information to describe this B+ tree, and it is the one level B+ tree with three leaves in the sample case. After general information of information, it continues with an array of offset and block number, which can help reach location of each leaf in B+ tree.

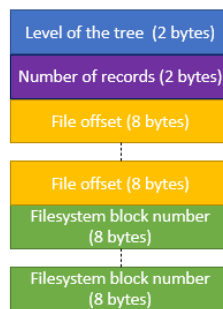


Figure 21. Layout of B+ tree's root node

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	49	4E	41	ED	83	03	00	00	00	00	00	00	00	00	00	00	INAI.....
00000010	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000020	5E	82	1F	A8	34	FC	4A	96	5E	82	1F	97	21	E9	7D	3C	^,"4üJ-^,,-!é)<
00000030	5E	82	1F	97	21	E9	7D	3C	00	00	00	00	00	25	40	00	^,,-!é)<.....%@.
00000040	00	00	00	00	00	00	03	A9	00	00	00	00	00	00	02	E5@.....â
00000050	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
00000060	FF	FF	FF	FF	45	A9	F4	CD	00	00	00	00	00	01	86	A3	yyyyE@öi.....t&
00000070	00	00	00	0B	00	00	17	A2	00	00	00	00	00	00	00	00ö.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	5E	82	1E	93	37	E6	70	A8	00	00	00	00	00	00	00	40	^,"7ap".....@
000000A0	AB	15	89	E4	99	EE	4E	3F	85	57	3B	C0	B1	CF	71	34	«.ba"iN?..W;ÄiIq4
000000B0	00	01	00	03	00	00	00	00	00	00	00	00	00	00	00	00
000000C0	00	00	01	73	00	00	00	00	80	00	63	00	00	00	00	00	...s.....e.c....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	01	4E	00	00	00	00	00K.....
00000160	00	00	11	CA	00	00	00	00	00	00	23	5E	00	00	00	00	...È.....#_....

Figure 22. One level B+ tree with 3 leaves

Taking first leaf as an example, by moving 1355776 bytes from the beginning of FS, the first leaf of B+ tree is reached by checking magic number in its header. The structure of its header is shown in Figure 23 and Table 7. Directory entries can be further discovered by analysing data extents in leaf of B+ tree, and the layout of data extent is the same as the one discussed in previous section.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	42	4D	41	33	00	00	00	FB	FF	FF	FF	FF	FF	FF	FF	FF	BMA3...Ûÿÿÿÿÿÿÿÿÿÿÿ
00000010	00	00	00	00	00	00	11	CA	00	00	00	00	00	00	0A	58È.....X
00000020	00	00	00	08	00	00	06	C8	AB	15	89	E4	99	EE	4E	3FÈ«.:%â™iN?
00000030	85	57	3B	C0	B1	CF	71	34	00	00	00	00	00	00	00	40	..W:À±İq4.....@
00000040	B8	CF	BE	36	00	00	00	00	00	00	00	00	00	00	00	00	1st data extent ...
00000050	00	00	00	00	02	00	00	01	00	00	00	00	00	00	02	00	2nd data extent ...
00000060	00	00	00	00	02	40	00	02	00	00	00	00	00	00	06	00@.....

Figure 23. Structure of B+ tree's leaf

Offset (in hex)	Variable name	Description
0x00-03 (4 bytes)	bb_magic	Magic number "BMA3".
0x04-05 (2)	bb_level	Level of the tree in which this block is found.
0x06-07 (2)	bb_numrecs	Number of records in this block.
0x08-0F (8)	bb_leftsib	FS block number of the left sibling of this B+tree node.
0x10-17 (8)	bb_rightsib	FS block number of the left sibling of this B+tree node.
0x38-3F (8)	bb_owner	AG number that this B+tree block belongs to.

Table 7. Structure of header of B+ tree's leaf [27]

3.6 Extent

Extent is a region of continuous blocks used to store data of file as close as possible to reduce the possibility of fragmentation for improving FS performance, and more than one extent can be allocated to a file. Extent list maps the offset to the corresponding extent,

which makes existence of sparse file feasible [33]. The structure of extent is shown in Figure 24, and value is presented in bits, not byte aligned.

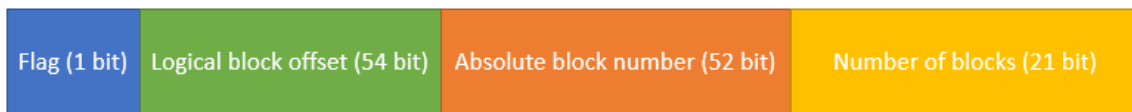


Figure 24. The structure of extent record

Two types of extent lists exist in XFS FS, which are extent list within inode data fork, and B+ tree extent list. Extent list within inode data fork is adopted when there is free space to store whole extent list in data fork of inode, and it can have up to 21 extent maps assuming there is no extended attribute fork. B+ tree extent list is used when there are too many extent maps to fit inside inode. Only root node of B+ tree is stored in data fork of inode, and it stores offset and block number to point directly to leaves or through other nodes in between depending on levels of B+ tree [27].

4 Journaling of XFS

Power failure and system crash between write to FS can make data inconsistent, and traditional FSs take time to check entire FS when mounting FS at the next boot [34]. Journaling is a FS feature used to store changed operations which have not been committed to FS, and XFS is one of the journaling FSs. By using the journal, FS only read and execute uncommitted operations in the journal to recover to consistent status instead of spending time checking whole FS [35]. Owing to the characteristic of journal, analysing journal can also be used to help identify operations occurred on the system, which can assist in understanding previous versions of files and deleted files. Official documentation of XFS data structure stopped updating since 2006, leaving journaling part empty without finishing, which could make it difficult to realize the logic behind journaling. Due to the contribution of XFS development community, many parts of data structure including journaling were introduced to the public, which gives an opportunity to save time focusing on analysis instead of starting to do research from the scratch. XFS journal is composed of log records, and each log record contains part or entire transaction. Transaction is made up of log operations, and it starts with an operation to begin a new transaction and ends with commitment. Each transaction is stored in circular queue and is hold in the cache until oldest item is overwritten, so it is possible to contain copy of the most recent data to aid in ER [1].

4.1 Log records

From the information recorded in superblock, the first block of journal can be reached. Log record begins with a 512 bytes header to document general information of this log record, and it starts with a magic number “0xfeedbabe” to help make sure at the start location of this log record. Log sequence number corresponds to the given location in journaling log, and it is split into two parts. The first four bytes describe cycle number, which increases as circular queue is full, and the rest four bytes describe block number, which is block offset from the beginning of journaling log and it is set when commitment is done [27]. The layout of log record header is shown in Figure 25 and Table 8.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	FE	ED	BA	BE	00	00	00	01	00	00	00	02	00	00	06	00	bi*%.....
00000010	00	00	00	01	00	00	00	02	00	00	00	01	00	00	00	02
00000020	1E	1C	83	7D	00	00	00	00	00	00	00	0F	12	5C	01	54	..f}.....\..T
00000030	74	1E	6E	5E	00	00	00	00	00	00	00	00	00	00	00	00	t.n^.....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 25. The layout of log record header

Offset (in hex)	Variable name	Description
0x00-04 (4 bytes)	h_magicno	Magic number “0xfeedbabe”.
0x08-0B (4)	h_version	Log record version.
0x0C-0F (4)	h_len	Length of log record (in bytes).
0x10-17 (8)	h_lsn	Log sequence number.
0x28-2B (4)	h_num_logops	Number of log operations.

Table 8. The layout of log record header [27]

4.2 Log operations

After the log record header, a series of log operations are used to represent part or entire transaction. XFS applies a mechanism of write ahead transaction log, and it can asynchronously write logs into system, which can bundle many log operations into one log write [1]. Each log operation can be divided into header and data. For example, as shown in Figure 26 and Table 9, the first one in transaction always starts with an operation to start a new transaction, which can be identified by originator XFS_TRANSACTION (0x69) and flag XLOG_START_TRANS (0x01) of this operation [27]. The originator and flag’s value of the log operation are defined in source code as depicted in Figure 27 and 28.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	00	00	00	01	00	00	00	00	69	01	00	00	12	5C	01	54i.....\..T
00000010	00	00	00	10	69	00	00	00	4E	41	52	54	28	00	00	00i...NART(...
00000020	54	01	5C	12	0C	00	00	00	12	5C	01	54	00	00	00	18	T.\.....\..T....

Figure 26. The structure of log operation header

Length (in bytes)	Variable name	Description
4	oh_tid	Transaction ID.
4	oh_len	Number of bytes in log item.
1	oh_clientid	Originator of this operation.
1	oh_flags	Flag of this operation
2	oh_res2	Padding

Table 9. The structure of log operation header [27]

```

73  /* Log Clients */
74  #define XFS_TRANSACTION      0x69
75  #define XFS_VOLUME          0x2
76  #define XFS_LOG              0xaa

```

Figure 27. Value of log operation's originator [36]

```

122  /*
123   * Flags to log operation header
124   *
125   * The first write of a new transaction will be preceded with a start
126   * record, XLOG_START_TRANS. Once a transaction is committed, a commit
127   * record is written, XLOG_COMMIT_TRANS. If a single region can not fit into
128   * the remainder of the current active in-core log, it is split up into
129   * multiple regions. Each partial region will be marked with a
130   * XLOG_CONTINUE_TRANS until the last one, which gets marked with XLOG_END_TRANS.
131   *
132   */
133  #define XLOG_START_TRANS      0x01 /* Start a new transaction */
134  #define XLOG_COMMIT_TRANS     0x02 /* Commit this transaction */
135  #define XLOG_CONTINUE_TRANS   0x04 /* Cont this trans into new region */
136  #define XLOG_WAS_CONT_TRANS   0x08 /* Cont this trans into new region */
137  #define XLOG_END_TRANS        0x10 /* End a continued transaction */
138  #define XLOG_UNMOUNT_TRANS    0x20 /* Unmount a filesystem transaction */

```

Figure 28. Value of log operation's flag [36]

4.3 Log items

Each log operation can be divided into header and data. Data stored in log operation includes categories such as log item or element of AG like superblock. As shown in Figure 29, each log item can be distinguished by its own magic number, and it is stored in host byte order, depending on machine's processor to determine how data is presented, which is different from the big-endian order which is typically used in XFS [27].

```

219 #define XFS_TRANS_HEADER_MAGIC 0x5452414e /* TRAN */
220
221 /*
222  * The only type valid for th_type in CIL-enabled file system logs:
223  */
224 #define XFS_TRANS_CHECKPOINT 40
225
226 /*
227  * Log item types.
228  */
229 #define XFS_LI_EFI 0x1236
230 #define XFS_LI_EFD 0x1237
231 #define XFS_LI_IUNLINK 0x1238
232 #define XFS_LI_INODE 0x123b /* aligned ino chunks, var-size ibufs */
233 #define XFS_LI_BUF 0x123c /* v2 bufs, variable sized inode bufs */
234 #define XFS_LI_DQUOT 0x123d
235 #define XFS_LI_QUOTAOFF 0x123e
236 #define XFS_LI_ICREATE 0x123f
237 #define XFS_LI_RUI 0x1240 /* rmap update intent */
238 #define XFS_LI_RUD 0x1241
239 #define XFS_LI_CUI 0x1242 /* refcount update intent */
240 #define XFS_LI_CUD 0x1243
241 #define XFS_LI_BUI 0x1244 /* bmbt update intent */
242 #define XFS_LI_BUD 0x1245

```

Figure 29. Magic number of log items [36]

Transaction header is always used as a first data payload to start a transaction, and it can be followed by number of other log items or elements of AG. As shown in Figure 30 and Table 10, the data is presented in little-endian order because of testing machine’s processor, so the order of bytes needs to be read from right to left.

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 00 00 00 01 00 00 00 00 69 01 00 00 12 5C 01 54 .....i.....T
00000010 00 00 00 10 69 00 00 00 4E 41 52 54 28 00 00 00 ....i...NART(...)
00000020 54 01 5C 12 0C 00 00 00 12 5C 01 54 00 00 00 18 T.\.....\T....

```

Figure 30. The structure of transaction header

Length (in bytes)	Variable name	Description
4	th_magic	Magic number “TRAN”.
4	th_tid	Transaction ID.
4	th_num_items	Number of operations appearing after this operation, not including the commit operation.

Table 10. The structure of transaction header [27]

To understand how metadata of FS object is changed to achieve the goal of aiding in ER, inode update log item and inode afterward are taken into consideration. Inode update log item documents metadata change to inode, and updated inode is stored immediately right

after it in the next log operation [27]. By analysing the updated inode in journal, it is possible to have a general overview of what occurred on the FS related to FS objects during the period time based on current data saved in circular journaling log. For example, as shown in Figure 31 and Table 11, inode update begins with a magic number, and it is presented in host byte order, which is little endian order on testing machine. From the data of inode update, there are two log operations following this inode update, and inode core and data fork's local data are updated. The inode being updated is the one with inode number 64. Therefore, inode core and data fork after inode update can be used to understand information of inode at the moment. For instance, inode number 64 is a short form directory with one regular file called “lqaz” under it.

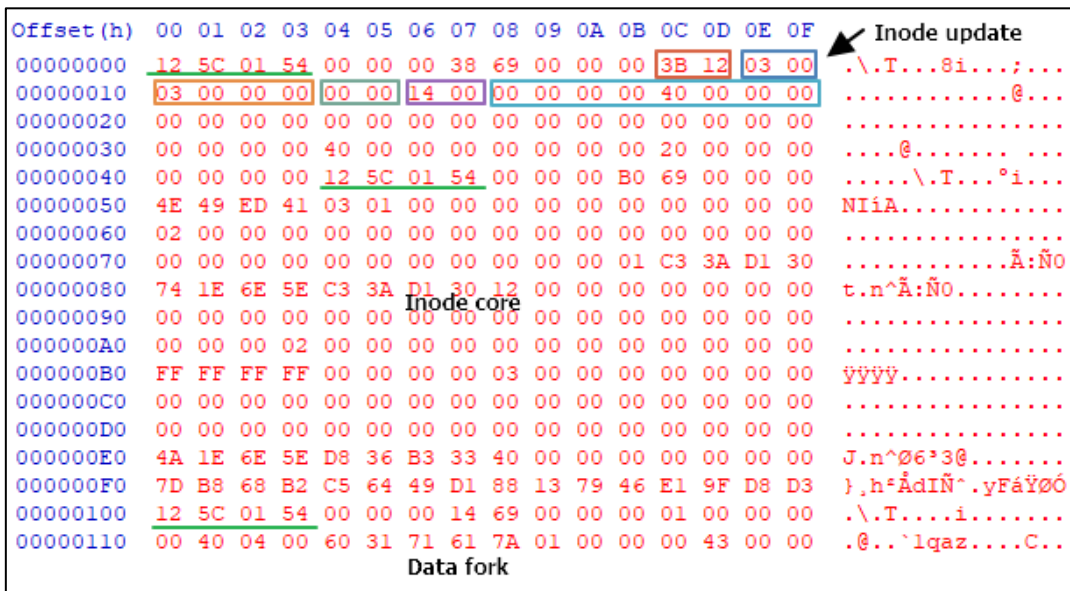


Figure 31. Inode update and Inode core

Length (in bytes)	Variable name	Description
2	ilf_type	Magic number “0x123b”.
2	ilf_size	Number of operations involved in this update, including this format operation.
4	ilf_fields	Specifies which parts of the inode are being updated. This can be certain combinations of the values shown in Figure 32.

2	ilf_asize	Size of the attribute fork (in bytes).
2	ilf_dsize	Size of the data fork (in bytes).
8	ilf_ino	Absolute node number.

Table 11. Structure of inode update [27]

```

305
306
307  /*
308   * Flags for xfs_trans_log_inode flags field.
309   */
310 #define XFS_ILOG_CORE    0x001  /* log standard inode fields */
311 #define XFS_ILOG_DDATA  0x002  /* log i_df.if_data */
312 #define XFS_ILOG_DEXT   0x004  /* log i_df.if_extents */
313 #define XFS_ILOG_DBROOT 0x008  /* log i_df.i_broot */
314 #define XFS_ILOG_DEV    0x010  /* log the dev field */
315 #define XFS_ILOG_UUID   0x020  /* added long ago, but never used */
316 #define XFS_ILOG_ADATA  0x040  /* log i_af.if_data */
317 #define XFS_ILOG_AEXT   0x080  /* log i_af.if_extents */
318 #define XFS_ILOG_ABROOT 0x100  /* log i_af.i_broot */
319 #define XFS_ILOG_DOWNER 0x200  /* change the data fork owner on replay */
320 #define XFS_ILOG_AOWNER 0x400  /* change the attr fork owner on replay */
321
322
323  /*
324   * The timestamps are dirty, but not necessarily anything else in the inode
325   * core. Unlike the other fields above this one must never make it to disk
326   * in the ilf_fields of the inode_log_format, but is purely store in-memory in
327   * ili_fields in the inode_log_item.
328   */
329 #define XFS_ILOG_TIMESTAMP    0x4000
330
331 #define XFS_ILOG_NONCORE      (XFS_ILOG_DDATA | XFS_ILOG_DEXT | \
332                               XFS_ILOG_DBROOT | XFS_ILOG_DEV | \
333                               XFS_ILOG_ADATA | XFS_ILOG_AEXT | \
334                               XFS_ILOG_ABROOT | XFS_ILOG_DOWNER | \
335                               XFS_ILOG_AOWNER)
336
337 #define XFS_ILOG_DFORK        (XFS_ILOG_DDATA | XFS_ILOG_DEXT | \
338                               XFS_ILOG_DBROOT)
339
340 #define XFS_ILOG_AFORK        (XFS_ILOG_ADATA | XFS_ILOG_AEXT | \
341                               XFS_ILOG_ABROOT)
342
343 #define XFS_ILOG_ALL          (XFS_ILOG_CORE | XFS_ILOG_DDATA | \
344                               XFS_ILOG_DEXT | XFS_ILOG_DBROOT | \
345                               XFS_ILOG_DEV | XFS_ILOG_ADATA | \
346                               XFS_ILOG_AEXT | XFS_ILOG_ABROOT | \
347                               XFS_ILOG_TIMESTAMP | XFS_ILOG_DOWNER | \
348                               XFS_ILOG_AOWNER)

```

Figure 32. Values to specify which parts of the inode are being updated [36]

5 Journal analysis for event reconstruction

5.1 Analysis procedure

Based on the features of XFS introduced above, the analysis procedure is designed to understand what happened on the system [6]. To begin with, superblock analysis is conducted to get the overall information of FS and first block of journal for reaching the start of journaling log. Second, by moving to the beginning of journaling log, journal analysis is executed to traverse transactions to find inode update information and inode followed by for realizing change of inode in different period of time. Filename is documented under directory entry, and FS is designed in hierarchical structure, starting from root directory [37]. By using the inode number of root directory getting from superblock, directory entries analysis is conducted by traversing all files under directory files. Finally, by comparing the information stored in directory entries with information stored in inode core of journal, timeline of FS change with metadata can be summarized to help understand status of files in the period of time for assisting in ER. The flow of analysis procedure can be seen graphically as displayed in Figure 33. In the following steps of procedure, the sample case of analysis is also conducted to help interpret the procedure.



Figure 33. Steps of procedure

5.2 Steps of procedure

Superblock analysis

Every AG begins with superblock, and the primary superblock is stored at AG 0 with one sector in length. First of all, value of sector size is read from primary superblock. With value of sector size, entire length of superblock can be determined and read to get the

entire superblock. As show in Figure 34, The sector size of sample case image is 0x0200, which is 512 bytes in decimal.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	58	46	53	42	00	00	10	00	00	00	00	00	00	03	E8	00	XFSB.....è.
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	7D	B8	68	B2	C5	64	49	D1	88	13	79	46	E1	9F	D8	D3	},h°ÀdIÑ°.yFáYØÓ
00000030	00	00	00	00	00	02	00	04	00	00	00	00	00	00	00	40@

Figure 34. Determine size of superblock by checking sector size

Second, to move to the beginning location of journal, first block of journal in superblock is read. There are two types of addressing schemes in XFS FS: absolute and relative address. Absolute address includes both AG number and offset from the start of that AG, and it is 64-bit address which is used in superblocks and directory entries. In contrast, relative address contains only offset from the start of AG [38]. Owing to the characteristics of addressing scheme in XFS, address conversion is necessary to reach to the exact location of journal. Log 2 value of AG, which is number of bits for the relative block offset, can be used to help convert block number into absolute block address. AG number is number of bits above relative block offset. The byte address of journal’s first block is calculated by using the following equation, and can be reached by moving result value of bytes from the beginning of FS. Value of AG size and block size can also be retrieved from superblock.

$$(AG\ no.\ * \ AG\ size\ +\ relative\ block\ offset) \ * \ block\ size$$

In the sample case as shown in Figure 35, the first block of journal is 0x00000000000020004, which is 00100000000000000100 in binary. Log 2 value of AG is 0x10, which is 16 in decimal, meaning the last 16 bits of journal’s block number represent relative block offset, and the remaining bits above is used to represent AG number. For this reason, AG number is 2 and the relative offset is 4, so the absolute block address of journal’s first block is $(2 * 64000 + 4) * 4096 = 524304384$ bytes. After moving the value of byte offset from the start of FS, journaling log is reached by spotting magic number “0xfeedbabe” at first four bytes as shown in Figure 36.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	58	46	53	42	00	00	10	00	00	00	00	00	03	E8	00		XFSB.....è.
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000020	7D	B8	68	B2	C5	64	49	D1	88	13	79	46	E1	9F	D8	D3	},h`ÁdIÑ`.yFáÿÓ
00000030	00	00	00	00	00	02	00	04	00	00	00	00	00	00	00	40@
00000040	00	00	00	00	00	00	00	41	00	00	00	00	00	00	00	42A.....B
00000050	00	00	00	01	00	00	FA	00	00	00	04	00	00	00	00	ú.....
00000060	00	00	03	57	B4	B5	02	00	02	00	00	08	00	00	00		...W`µ.....
00000070	00	00	00	00	00	00	00	00	0C	09	09	03	10	00	00	19

Figure 35. First block of journal

1f404000	FE	ED	BA	BE	00	00	00	01-00	00	00	02	00	00	02	00		bi%.....
1f404010	00	00	00	01	00	00	00	00-00	00	00	01	00	00	00	00	
1f404020	00	00	00	00	FF	FF	FF	FF-00	00	00	01	B0	C0	D0	D0		...yyy`*`ÁDD
1f404030	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	

Sel start = 524304384, len = 4; log sec = 1024032

Figure 36. Byte offset of journal's first block

Journal analysis

Journal is consisted of sequence of log records. Each log record begins with a 512 bytes header, and the first four bytes of header are presented as 0xfeedbabe in hexadecimal format, which is a magic number of log record. Magic number, just like file signature, is used to identify format of file, and it is largely used in file carving in DF. When traversing through log records, the first four bytes are compared with magical number to assure it is located at the header of log record. Next, from the start of header, value of bytes occupied by log operations and data region is read, and this value can be used to check there are still any following log records or not if the length of log record is greater than zero while going over log records. In the sample case, the magic number “0xfeedbabe” is stored in the first four bytes, and the following log operations and data region occupy 1536 bytes as displayed in Figure 37.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	FE	ED	BA	BE	00	00	00	01	00	00	00	02	00	00	06	00	bi%.....
00000010	00	00	00	01	00	00	00	02	00	00	00	01	00	00	00	02

Figure 37. Log record header

After getting the value of bytes occupied by log operations and data, the pointer can be moved 512 bytes behind from the start of log record to check log operations in the log record. Each log operation starts with an operation header, and the first four bytes of data are used for storing transaction ID. The first log operation is used to start a transaction, and the last one is used to commit a transaction. Existence of transaction ID is reviewed

together with first and last log operation to know the scope of log operations in the log record. From each log operation header, number of bytes in data region, originator of the operation, and log operation flag can be retrieved. Based on the values gotten above, operation type, such as start of transaction, commitment of transaction, and any log items or data can be identified and handled separately. As shown in Figure 38, the first log operation with transaction ID set to 0x00000001, originator set to XFS_TRANSACTION (0x69) and flag set to XLOG_START_TRANS (0x01) indicates a beginning of new transaction, and it is always followed by the transaction header as payload to start a new transaction, which can be detected by its magic number (0x5452414E). After a series of log operations, the transaction is finished by commitment with log operation's flag set to XLOG_COMMIT_TRANS (0x02), and padding the remaining space with zero.

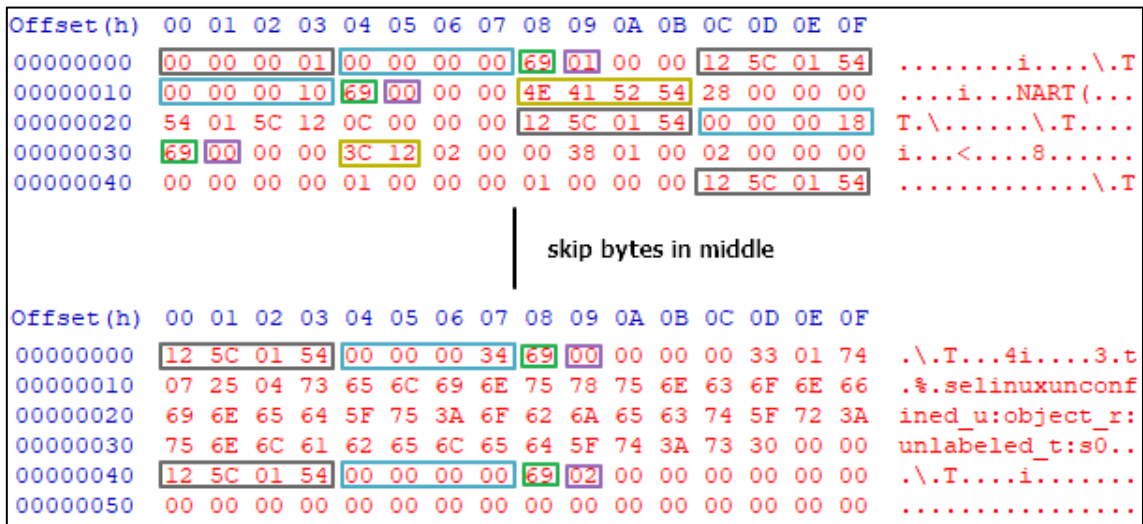


Figure 38. Log operations

Log items followed by log operation header come with different types. As mentioned in previous section, inode update and the inode afterward are focused in this analysis. Inode updates are used to record change to different parts of inode. Update to inode can be done granularly in prevention of resource competency caused by simultaneous updates [1]. Inode update has magic number of 0x123b, and it is also presented in host-endian order. Inode updates records information related to inode being changed, such as parts of the inode being updated and absolute inode number. The inode is recorded right after inode update, which can be used for directory entries analysis later. In the sample case as displayed in Figure 39, data related to inode change is saved in log operations after inode update. Inode core can be recognized by magic number (0x494E), and this inode is a

directory whose data fork is stored within inode. From the next log operation, the data fork of inode is stored to tell us how many directory entries under this directory, and there is one regular file called “lqaz” under this directory.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	12	5C	01	54	00	00	00	38	69	00	00	00	3B	12	03	00	.\.T...8i...;...
00000010	03	00	00	00	00	00	14	00	00	00	00	00	40	00	00	00@...
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	40	00	00	00	00	00	00	00	20	00	00	00@.....
00000040	00	00	00	00	12	5C	01	54	00	00	00	B0	69	00	00	00\.T...°i...
00000050	4E	49	ED	41	03	01	00	00	00	00	00	00	00	00	00	00	NIiA.....
00000060	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	01	C3	3A	D1	30Ã:Ñ0
00000080	74	1E	6E	5E	C3	3A	D1	30	12	00	00	00	00	00	00	00	t.n^Ã:Ñ0.....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00
000000B0	FF	FF	FF	FF	00	00	00	00	03	00	00	00	00	00	00	00	ÿÿÿÿ.....
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	4A	1E	6E	5E	D8	36	B3	33	40	00	00	00	00	00	00	00	J.n^06°3@.....
000000F0	7D	B8	68	B2	C5	64	49	D1	88	13	79	46	E1	9F	D8	D3	},h^ÃdIÑ°.yFáÿ0Ó
00000100	12	5C	01	54	00	00	00	14	69	00	00	00	01	00	00	00	.\.T...i.....
00000110	00	40	04	00	60	31	71	61	7A	01	00	00	00	43	00	00	.\.T...lqaz....C..

Figure 39. Inode update & Change to Inode

Directory entries analysis

For understanding change of FS objects on system to help prioritize the order of DF investigation, a technique is proposed. The visual display of the proposed technique is shown in Figure 40, and the process is introduced as follows. First, journaling log is traversed to find all inode update and change to inode. All inode core items in the journaling log are stored in a separate list with some information such as inode number and file type. At the same time, if inode is a directory file, the data fork of inode is stored in another list. Next, from the list of directory inode’s data fork, directories are iterated to compare with list of inode core items. Only inode core items whose file creation time is smaller or equal to directory last changed time are filtered out to compare in each round. Directory entries under the directory are traversed to match with inode core item by comparing condition such as inode number, file type, and time. After comparison, action executed on FS object can be determined just like creation, modification, or deletion. By combing matched information from inode core item and directory entry, table of FS objects can be created to achieve of goal of assisting in ER. Additional operations are also necessary for handling different situations such as renaming of FS object, which changes

the name of directory entry but also has the same inode number, and it is similar to behaviour of hard link.

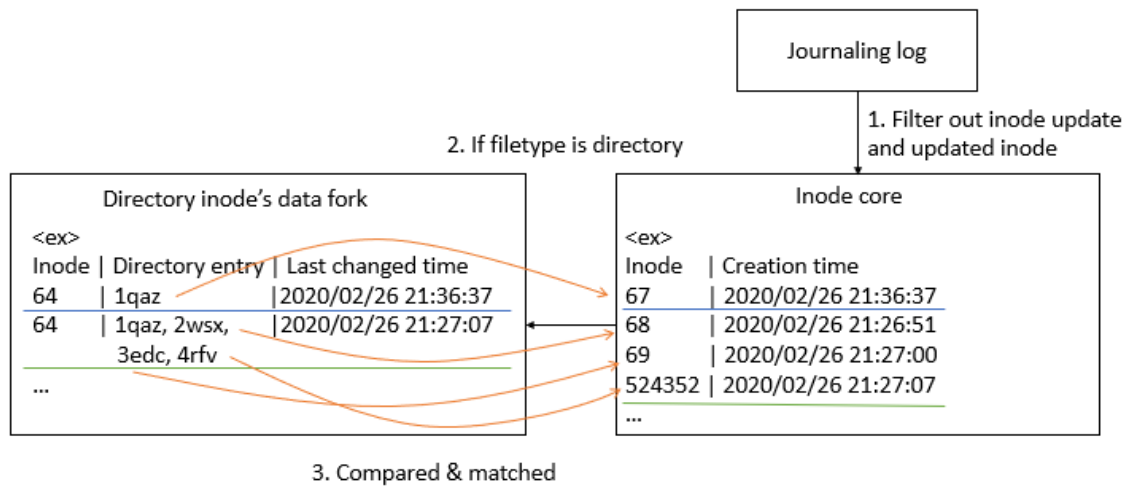


Figure 40. Proposed analysis technique

6 Experiment procedure, limitation, and comparison

6.1 Environment setting

Physical / Virtual machine	Virtual machine
Operation system	CentOS 7 64-bits (Host machine: Windows 10)
Kernel version	3.10.0-1062

Table 12. Environment setting

6.2 Steps of experiment procedure

- (1) Creating a 1 GB file with null characters inserted
- (2) Formatting the newly created file as XFS FS with default options. The value of default setting is show in Figure 41.

```
meta-data=test.img      isize=512    agcount=4, agsize=64000 blks
                        =                    sectsz=512   attr=2, projid32bit=1
                        =                    crc=1      finobt=0, sparse=0
data                    =                    bsize=4096  blocks=256000, imaxpct=25
                        =                    sunit=0     swidth=0 blks
naming                 =version 2          bsize=4096  ascii-ci=0 ftype=1
log                    =internal log      bsize=4096  blocks=855, version=2
                        =                    sectsz=512   sunit=0 blks, lazy-count=1
realtime               =none              extsz=4096  blocks=0, rtextents=0
```

Figure 41. Value of default XFS settings

- (3) Creating a loop device with the newly created file
- (4) Making a new directory to be used as a mount point of XFS FS
- (5) Mounting the XFS FS on the newly created directory
- (6) Creating, Modifying, Deleting FS objects on XFS FS by shell script
- (7) Creating a bit-by-bit copy image of XFS FS
- (8) Copying the image file to host machine
- (9) Analysing with the self-written Python script (Source code in Appendix 1)

6.3 Test case

After mounting the XFS FS, the self-written shell script was executed. The content of shell script is shown in Appendix 2. In this shell script, FS objects test01 to test07 and dir01 to dir03 were created sequentially as the layout shown in Figure 42. Second, directory test02 was renamed to test02n, and character special device test03 under renamed directory test02n was deleted. Next, directory dir02 was renamed to dir02n, and FIFO test05 under this directory was deleted.

```
Root dir
- test06 Socket
- test07 Symlink (Link to test01)

- dir01 Directory
- test01 Regular file
- test02 Directory (Renamed to test02n)
- test03 Character special device (Deleted)

- dir02 Directory (Renamed to dir02n)
- test04 Block special device
- test05 FIFO (Deleted)
- dir03 Directory
```

Figure 42. Layout of test case

6.4 Test objectives

As mentioned in previous section, evidence examination is the first step in event reconstruction process by inspecting all associated information of objects and distinguishing characteristics they have. For achieving the goal to assist in event reconstruction, the following test objectives were designed:

- (1) Creation of different FS objects, ranging from regular file to symbolic link.
- (2) Testing FS objects under different locations, such as root directory and multi-layer directories.
- (3) Update of parent location's name after renaming directory.
- (4) Deletion of different FS objects.

6.5 Test result

(1) Creation of different FS objects, ranging from regular file to symbolic link

As shown in Figure 43, all types of FS objects created from test case were successfully identified by checking the column Filetype from the output result. For example, dir01 is a directory, and test01 is a regular file, which were as expected as presented in the test case. By recognizing type of FS objects, classification of files can be conducted to investigate by groups.

(2) Testing FS objects under different locations, such as root directory and multi-layer directories.

FS system objects were intentionally created under different directories. For example, socket file test06 was created under root directory, and regular file test01 was created under directory dir01, which is one layer under root directory. Under directory test02, which is two layers under root directory, character special device file test03 was created. From FileLocation field of the output result in Figure 43, all FS objects located under different directories were successfully identified, which assured that files under different locations could be searched.

(3) Update of parent location's name after renaming directory.

Renaming of file is frequently used as a method to evade investigation. In the test case, dir02 and test02 were renamed for checking update status of parent location. As seen in Figure 43, by checking fields of Filename and Action, dir02 was modified to dir02n, and test02 was modified to test02n.

(4) Deletion of different FS objects.

Understanding deleted files is important topic in data recovery. In the test case, test03 and test05 were deleted. From the Action field of output result, the deletion of file was successfully identified. Also, because these two files were deleted after directory renaming, the FileLocation of both files were also updated to new directory names. However, there is some observation deserved for further investigation. First, after renaming directory, there were no log items in journaling log to update all FS objects under this directory, but new directory was given only until actions happened such as data

deletion in this case. Besides, the journaling log was not shown or inode number was modified to zero when deletion of FS objects like socket and symbolic link happened, which made it difficult to differentiate the happening of events.

Inode	Filename	Filetype	FileLocation	Action	Account (UID)	Creation Time	Last Changed Time
67	dir01	Directory	Root Directory	Modified	0	4/4/2020 23:44:38	4/4/2020 23:44:58
67	dir01	Directory	Root Directory	Modified	0	4/4/2020 23:44:38	4/4/2020 23:46:08
524352	dir02	Directory	Root Directory	Created	0	4/4/2020 23:44:38	4/4/2020 23:44:38
524352	dir02	Directory	Root Directory	Modified	0	4/4/2020 23:44:38	4/4/2020 23:45:28
524352	dir02	Directory	Root Directory	Modified	0	4/4/2020 23:44:38	4/4/2020 23:45:38
524352	dir02n	Directory	Root Directory	Modified	0	4/4/2020 23:44:38	4/4/2020 23:46:28
524352	dir02n	Directory	Root Directory	Modified	0	4/4/2020 23:44:38	4/4/2020 23:46:38
68	test01	Regular file	dir01	Created	0	4/4/2020 23:44:48	4/4/2020 23:44:48
1055488	test02	Directory	dir01	Created	0	4/4/2020 23:44:58	4/4/2020 23:44:58
1055488	test02	Directory	dir01	Modified	0	4/4/2020 23:44:58	4/4/2020 23:45:08
1055488	test02n	Directory	dir01	Modified	0	4/4/2020 23:44:58	4/4/2020 23:46:18
1055489	test03	Character special device	test02	Created	0	4/4/2020 23:45:08	4/4/2020 23:45:08
1055489	test03	Deleted File	test02n	Deleted	0	4/4/2020 23:45:08	4/4/2020 23:46:18
524353	test04	Block special device	dir02	Created	0	4/4/2020 23:45:18	4/4/2020 23:45:18
524354	test05	FIFO	dir02	Created	0	4/4/2020 23:45:28	4/4/2020 23:45:28
524354	test05	Deleted File	dir02n	Deleted	0	4/4/2020 23:45:28	4/4/2020 23:46:38
1572928	dir03	Directory	dir02	Created	0	4/4/2020 23:45:38	4/4/2020 23:45:38
69	test06	Socket	Root Directory	Created	0	4/4/2020 23:45:48	4/4/2020 23:45:48
70	test07	Symlink	Root Directory	Created	0	4/4/2020 23:45:58	4/4/2020 23:45:58

Figure 43. Result of test case

6.6 Limitation

XFS source code is evolving year after year because of active XFS development community. Even though the general data structure of XFS hasn't changed, there are still some minor change comparing to information in documentation such as some defined values were removed. Depending on kernel version used, it is possible to have difference as what was documented in this paper. For example, transaction type of transaction header in log items was originally defined with different type of values such XFS_TRANS_CREATE for identifying usage of transaction, but most values were removed with only XFS_TRANS_CHECKPOINT left as shown in Figure 44. Journaling logs are stored in circular queue, so it can only show you change of FS objects during a period of time based on journal size. Owing to characteristic of circular log, it is possible to have some residual data of previous records left, and it can be spotted by checking scope of transactions according to transaction ID. Journaling log is also possible to store in separate physical storage, which can be identified by checking the value in superblock. Python is a popular language used in DF because of simplicity of syntax and comprehensive inbuilt modules, and that is also one of the reasons why the test script written in Python, but the possibility of incorporation the script with open source tools is

relatively low now. For instance, Autopsy, one of the most famous open source DF tools developed by Dr. Brian Carrier, provides chance to write custom modules using Python. As mentioned in Open Source Digital Forensics Conference, there are three challenges developing forensics application including input types, user interaction, and analytics, and Autopsy takes care of the first two of them, allowing analytics module to be developed [39]. However, supporting of XFS FS was not available in Autopsy when the time paper was written, which made it difficult to contribute XFS related code to the open source community. Besides, PhotoRec, which would be used as tool comparison in the next section, supports XFS filesystem, but the source codes were written in C language, and is mainly focused on data recovery by using data carving instead of journaling technique proposed in this paper. The self-written Python script doesn't take concision and efficiency into consideration and it is developed for static analysis, which means appropriate hardware or software write blocker should be used to avoid corruption of data preservation. The source codes were only used to propose a method for getting an overview of events on system for DF investigation assistance, which can provide people interested in XFS forensics as an inspiration to further explore and develop related functions or tools in XFS system forensics. Also, hard link is the FS object which has the same inode number with the original file but with different name, and it is not defined in FS as any of file types, so how to differentiate between hard link and renamed file from journal can be analysed further.

```

5 fs/xfs/libxfs/xfs_log_format.h
xfs @@ -211,6 +211,11 @@ typedef struct xfs_trans_header {
211 211
212 212 #define XFS_TRANS_HEADER_MAGIC 0x5452414e /* TRAN */
213 213
214 214 + /*
215 215 + * The only type valid for th_type in CIL-enabled file system logs:
216 216 + */
217 217 + #define XFS_TRANS_CHECKPOINT 40
218 218 +
219 219 + /*
220 220 + * Log item types.
221 221 + */
222 222
97 fs/xfs/libxfs/xfs_shared.h
xfs @@ -55,103 +55,6 @@ extern const struct xfs_buf_ops xfs_sb_quiet_buf_ops;
55 55 extern const struct xfs_buf_ops xfs_symlink_buf_ops;
56 56 extern const struct xfs_buf_ops xfs_rtbuf_ops;
57 57
58 58 - /*
59 59 - * Transaction types. Used to distinguish types of buffers. These never reach
60 60 - * the log.
61 61 - */
62 62 - #define XFS_TRANS_SETATTR_NOT_SIZE 1
63 63 - #define XFS_TRANS_SETATTR_SIZE 2
64 64 - #define XFS_TRANS_INACTIVE 3
65 65 - #define XFS_TRANS_CREATE 4
66 66 - #define XFS_TRANS_CREATE_TRUNC 5
67 67 - #define XFS_TRANS_TRUNCATE_FILE 6
68 68 - #define XFS_TRANS_REMOVE 7

```

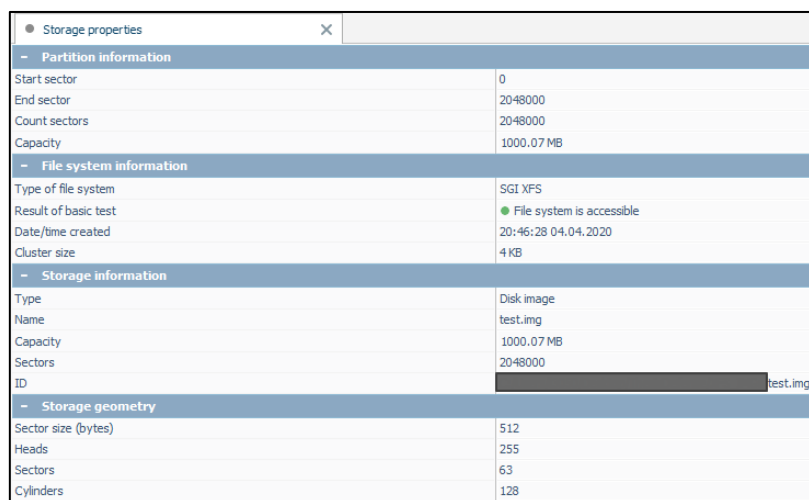
Figure 44. Remove of transaction types [40]

6.7 Comparison with other forensics tools

As discussed in previous section, there are few tools supporting XFS FS, but most of tools mentioned in studies are proprietary with only one open source tool Photorec [7][9]. In this section, one proprietary tool, UFS explorer, and one open source tool, PhotoRec, were used in comparison with the test result achieved in this paper.

(1) UFS explorer

UFS explorer was developed by SysDev Laboratories LLC in 2004 as a data access solution for non-Windows FSs. From the company's website, there are different kind of tools provided for data recovery, such as UFS Explorer Standard Recovery and UFS Explorer Professional Recovery. UFS Explorer Professional Recovery were used to test the test case created above. After downloading and installing the software from the website, the test case was loaded into the software for analysis. First, from the Storage properties as seen in Figure 45, the software successfully identified the image belongs to XFS FS, and information was the same as default settings set in creation of FS. Next, when exploring the FS as shown in Figure 46, the structure was basically the same as the test case, but filetype of test06 was shown unknown from the output, which was not able to recognize its type as a socket file. The same situation could also be applied to test04 under directory dir02n. Filetype of block special device was unable to identify in the case. In Figure 47, by executing option to scan for lost data, deleted files, just as test03 under directory test02n and test05 under dir02, were not able to recover from the system. Also, the existence of those deleted files could not be realized.



The screenshot shows the 'Storage properties' window in UFS Explorer. It displays detailed information about the 'test.img' file system, including partition and storage geometry details.

Partition information	
Start sector	0
End sector	2048000
Count sectors	2048000
Capacity	1000.07 MB

File system information	
Type of file system	SGI XFS
Result of basic test	● File system is accessible
Date/time created	20:46:28 04.04.2020
Cluster size	4 KB

Storage information	
Type	Disk image
Name	test.img
Capacity	1000.07 MB
Sectors	2048000
ID	[REDACTED] test.img

Storage geometry	
Sector size (bytes)	512
Heads	255
Sectors	63
Cylinders	128

Figure 45. Storage properties of test case shown in UFS explorer

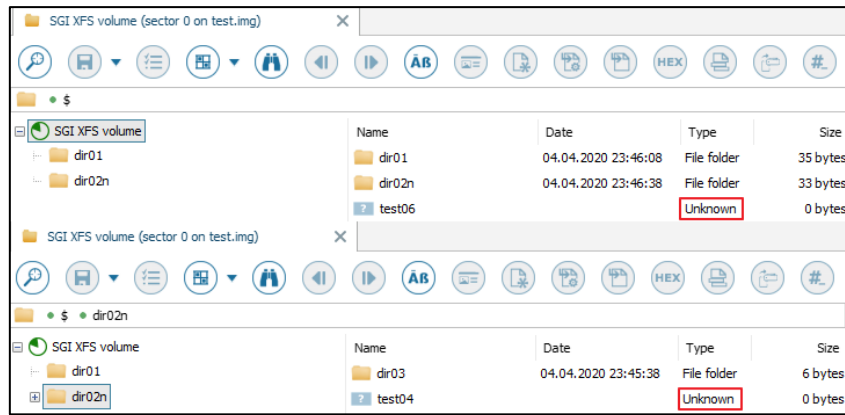


Figure 46. Unknown data type

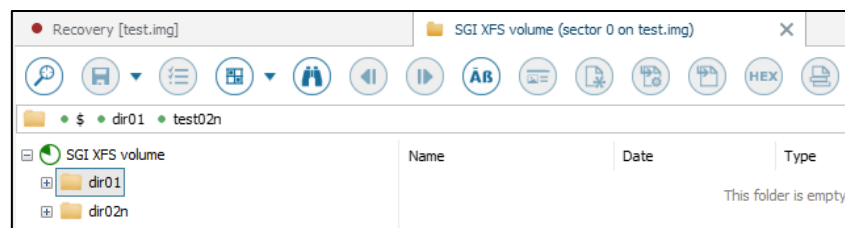


Figure 47. Failure to recover deleted file

(2) PhotoRec

PhotoRec is a free and open-source software developed by Christophe GRENIER for data recovery using data carving techniques. As shown in Figure 48, by adding the test image into the software, XFS filesystem was also successfully identified as the UFS explorer did but without system properties. Owing to the characteristic that the software was mainly designed for data recovery, many file formats can be chosen to recover from the system. The result was saved to the chosen folder after specifying destination folder and clicking Search button, and there were four files recovered from the image as shown in Figure 49. By checking the result files in the directory, the names of files were all randomized without knowing the original filenames and other file properties as shown in Figure 50.

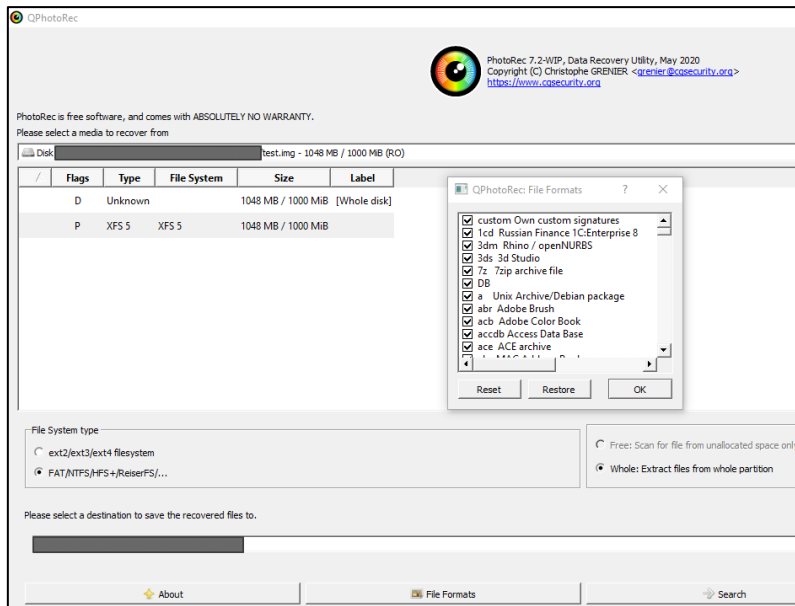


Figure 48. XFS test case shown in PhotoRec

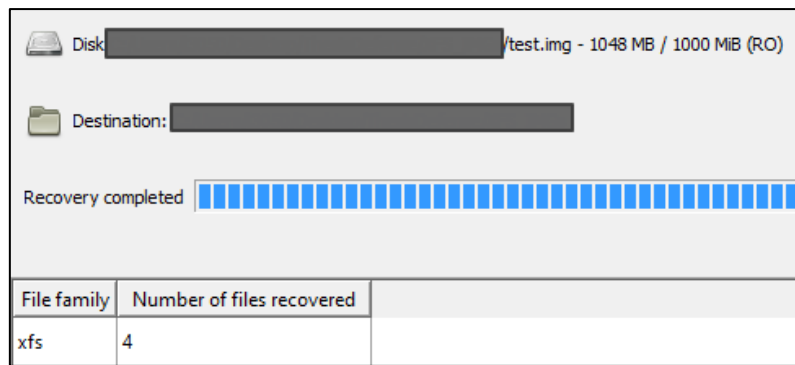


Figure 49. Files recovered from PhotoRec

Name	Date modified	Type	Size
f0000000.xfs	5/10/2020 10:06 PM	XFS File	4 KB
f0512000.xfs	5/10/2020 10:06 PM	XFS File	4 KB
f1024000.xfs	5/10/2020 10:06 PM	XFS File	4 KB
f1536000.xfs	5/10/2020 10:06 PM	XFS File	4 KB

Figure 50. Unrecognizable file names and properties

7 Summary

With limited amount of time and human resource, how to arrange time appropriately in DF investigation becomes important. Inexpensive cost of storage acquirement makes setting up a large system in a short time easy no matter through lease from cloud service provider or personal owned server, which brings not only convenience to enterprises but also more and more data to be analysed by DF specialists. Identifying events from FS can help prioritize order of investigation for ER to understand cause and effect of events during a period of time. Journaling is originally developed to prevent data inconsistency caused by system crash, but it can also be applied to realize status change of FS objects on system due to the characteristic that any change is first stored in journal before makes a commitment to the system. CentOS and Red Hat Linux are used by many enterprises as OS of server because of its high performance, which is owing to the design of XFS FS. There are not enough studies about XFS, and official documentation is incomplete, which makes knowledge to underlying structure of XFS insufficient. In this paper, an overview layout of XFS was introduced to first understand how data is stored on system, and a method to analyse XFS journal was proposed to help understand change of FS objects. To prove the accuracy of the logic, experiment was also conducted. Even though experiment was done with few directory entries involved, it can be expanded to research on more directory entries by referencing the data structure and journal analysis above. The concept of journaling analysis can be applied to other journaling FSs with similar data structures to get an overview of events happened on system in future work. Besides, further actions such as file carving and file recovering can be combined together to resume file into previous versions of files based on the result gotten from journal analysis for assisting in DF investigation.

References

- [1] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, “Scalability in the XFS File System”, USENIX Annual Technical Conference, 1996.
- [2] X. Du, N. Le-Khac, and M. Scanlon, “Evaluation of Digital Forensic Process Models with Respect to Digital Forensics as a Service”, arXiv.org, Cornell University, 2017.
- [3] B. Carrier, and E. Spafford, “Defining Event Reconstruction of Digital Crime Scenes”, J Forensic Sci, Vol. 49, No. 6, 2004.
- [4] C. Swenson, R. Philhps, and S. Sheno, “File System Journal Forensics”, Advances in Digital Forensics III. DigitalForensics 2007. IFIP — The International Federation for Information Processing, vol 242. Springer, New York, NY, 2007.
- [5] R. Carbone, “Forensic analysis of SGI IRIX disk volume”, Defence Research and Development Canada, 2016
- [6] Y. Park, H. Chang, and T. Shon, “Data investigation based on XFS file system metadata”, Multimed Tools Appl 75, 2015.
- [7] K. Ghazinour, D. Vakharia, K. Kannaji, and R. Satyakumar, “A Study on Digital Forensic Tools”, IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI), 2017.
- [8] B. Carrier, EH. Spafford, “An event-based digital forensic investigation framework”, Digital Forensic Research Conference, 2004.
- [9] Y. Chabot, B. Aurélie, N. Christophe, K. Tahar, “Event Reconstruction: A state of the art”, Handbook of Research on Digital Crime, Cyberspace Security, and Information Assurance, 2015
- [10] S. Jeyaraman, M. Atallah, “An empirical study of automatic event reconstruction systems”, Digital Investigation, Volume 3, 2006.

- [11] F. Buchholz, E. Spafford, “On the role of file system metadata in digital forensics”, *Digital Investigation*, Volume 1, Issue 4, 2004
- [12] D. Lillis, B. Becker, T. O'Sullivan, M. Scanlon, “Current challenges and future research areas for digital forensic investigation”, arXiv.org, Cornell University, 2016.
- [13] D. Jang, G. Hwang, and K. Kim, “Understanding anti-forensic techniques with timestamp manipulation”, *IEEE 17th International Conference on Information Reuse and Integration*, 2016.
- [14] D. Bovet, and M. Cesati, “Understanding the Linux kernel”, O'Reilly Media, Inc., 2006.
- [15] B. Carrier, “File system forensic analysis”, Addison Wesley Professional, 2005.
- [16] K. Choo, “Organised crime groups in cyberspace: a typology”, *Trends Organ Crim* 11, 2008.
- [17] S. Best, “Journaling File Systems”, *Linux Magazine*, 2002.
- [18] L. Lu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and S. Lu, “A Study of Linux File System Evolution”, *11th USENIX Conference on File and Storage Technologies (FAST '13)*, 2013.
- [19] J Yang, P Twohey, D Engler, and M Musuvathi, “Using model checking to find serious file system errors”, *ACM Transactions on Computer Systems*, 2006.
- [20] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Analysis and Evolution of Journaling File Systems”, *USENIX Annual*, 2005.
- [21] K. Tamma, and S. Venugopalan, “Failure Analysis of SGI XFS File System”, *Computer Sciences Department, University of Wisconsin*, 2014.
- [22] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, “The Linux Implementation of a Log-structured File System”, *ACM SIGOPS Operating Systems Review*, 2006.

- [23] P. Kieseberg, S. Schrittwieser, M. Mulazzani, M. Huber, and E. Weippl, "Trees Cannot Lie: Using Data Structures for Forensics Purposes," European Intelligence and Security Informatics Conference, Athens, 2011.
- [24] P. Kieseberg, S. Schrittwieser, L. Morgan, M. Mulazzani, M. Huber, and E. Weippl, "Using the structure of B+-trees for enhancing logging mechanisms of databases", International Journal of Web Information Systems, 2013.
- [25] S. Majore, C. Lee, and Taeshik Shon, "XFS File System and File Recovery Tools", International Journal of Smart Home Vol. 7, No. 1, 2013.
- [26] J. Mostek, B. Earl, S. Levine, S. Lord, R. Cattelan, K. McDonell, T. Kline, B. Gaffey, and R. Ananthanarayanan, "Porting the SGI XFS File System to Linux", USENIX Annual Technical Conference, 2000.
- [27] XFS.org, "XFS Filesystem Disk Structures 3rd Edition" [Online], Available: https://mirrors.edge.kernel.org/pub/linux/utils/fs/xfs/docs/xfs_filesystem_structure.pdf [Accessed: 23.03.2020].
- [28] GitHub, Source code of Linux kernel [Online], Available: https://github.com/torvalds/linux/blob/master/fs/xfs/libxfs/xfs_format.h [Accessed: 24.03.2020].
- [29] C. Hellwig, "XFS the big storage file system for Linux", ;login:: the magazine of USENIX & SAGE, Vol. 34, N°. 5, 2009.
- [30] GitHub, Source code of Linux kernel [Online], Available: <https://github.com/torvalds/linux/blob/master/include/uapi/linux/stat.h> [Accessed: 24.03.2020].
- [31] Z. Wang, "Research of Data Storage Mode and Recovery Method Based on XFS File System", 7th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2016.
- [32] GitHub, Source code of Linux kernel [Online], Available: https://github.com/torvalds/linux/blob/master/fs/xfs/libxfs/xfs_da_format.h [Accessed: 25.03.2020].

- [33] J. Florido, “Journal File Systems”, Linux Gazette, 2000.
- [34] K. Eckstein, “Forensics for advanced UNIX file systems”, Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.
- [35] K. Fairbanks, “A technique for measuring data persistence using the Ext4 file system journal”, IEEE 39th Annual Computer Software and Applications Conference, Taichung, 2015.
- [36] GitHub, Source code of Linux kernel [Online], Available: https://github.com/torvalds/linux/blob/master/fs/xfs/libxfs/xfs_log_format.h [Accessed: 25.03.2020].
- [37] K. Fairbanks, “An analysis of Ext4 for digital forensics”, IEEE 39th Annual Computer Software and Applications Conference, 2012.
- [38] Hal Pomeranz, “XFS (Part 1) – The Superblock” [Online], Available: <https://righteousit.wordpress.com/2018/05/21/xfs-part-1-superblock/> [Accessed: 26.03.2020].
- [39] Eugene Livis, “Writing Autopsy Python Modules” [Online], <http://www.osdfcon.org/presentations/2018/Eugene-Livis-Writing-Autopsy-Python-Modules.pdf> [Accessed: 10.05.2020].
- [40] GitHub, xfs: remove transaction types [Online], <https://github.com/torvalds/linux/commit/710b1e2c2948c1e5d0499def5273ecbc6472342d> [Accessed: 26.03.2020].

Appendix 1 – Source Code

```
import sys
import re
import platform
import datetime
from bitstring import BitArray
import pandas as pd

# Current architectures: https://en.wikipedia.org/wiki/Endianness
# Values in host byte order: transaction header(0x5452414E), buffer write
log(0x123C), inode update(0x123B), inode core(0x494E), inode
creation(0x123F), efi(0x1236)
print(platform.processor(), '\n')

sb = None # superblock
sb_blocksize = None # block size
sb_sectsize = None # sector size
sb_logstart = None # first block of journal
sb_rootino = None # Root inode number for the filesystem
sb_agblocks = None # AG size (in blocks)
sb_inodesize = None # Inode size (in bytes)
sb_inopblock = None # Inodes/block
sb_inopblog = None # log2(inode/block)
sb_agblklog = None # log2(AG size) rounded up

is_shortformdir = False

# Open in binary mode (read using byte data)
with open(r"test.img", "rb") as f:
    f.seek(102)
    sb_sectsize = int.from_bytes(f.read(2), byteorder='big')

print("Sector size:", sb_sectsize, "bytes\n")

with open(r"test.img", "rb") as f:
    sb = f.read(sb_sectsize)

# Block address conversion
sb_blocksize = int.from_bytes(sb[4:8], byteorder='big')
sb_agblklog = int.from_bytes(sb[124:125], byteorder='big')
# For an external log device, this will be zero. Conditional judgement here !
bin_len = len(BitArray(bytes=sb[48:56]).bin)
agno = int(BitArray(bytes=sb[48:56]).bin[:bin_len-sb_agblklog], 2)
rel_offset = int(BitArray(bytes=sb[48:56]).bin[-sb_agblklog:], 2)
sb_agblocks = int.from_bytes(sb[84:88], byteorder='big')
sb_logstart = (agno * sb_agblocks + rel_offset) * sb_blocksize

# Inode address conversion
sb_inopblog = int.from_bytes(sb[123:124], byteorder='big')
```

```

bin_len_rootino = len(BitArray(bytes=sb[56:64]).bin)
agno_rootino = int(BitArray(bytes=sb[56:64]).bin[:bin_len_rootino-
(sb_inopblog + sb_agblklog)], 2)
rel_offset_rootino = int(BitArray(bytes=sb[56:64]).bin[-(sb_inopblog +
sb_agblklog):], 2)
sb_inopblock = int.from_bytes(sb[106:108], byteorder='big')
rel_block = int(rel_offset_rootino / sb_inopblock)
rel_inode = rel_offset_rootino % sb_inopblock
sb_inodesize = int.from_bytes(sb[104:106], byteorder='big')
sb_rootino = int.from_bytes(sb[56:64], byteorder='big')
sb_rootino_offset = ((agno_rootino * sb_agblocks + rel_block) * sb_blocksize)
+ (rel_inode * sb_inodesize)

```

```

h_magicno = b'\xfe\xed\xba\xbe' # The magic number of log records
pattern = re.compile(h_magicno)
h_len = None # Length of the log record, in bytes
h_num_logops = None # The number of log operations in this record
xlog_op = None # Log operation
oh_tid = None # Transaction ID of this operation
oh_len = None # Number of bytes in the data region
oh_clientid = None # The originator of this operation
oh_flags = None # Specifies flags associated with this operation
xlog_item = None # Log item

```

```

ctime = None
location = None
direntries = []
num_direntries = 0
inode_list = []

```

```

def inode_core(arg):
    print('Inode core')
    byte_ord = 'little'
    di_mode_filetype = int.from_bytes(arg[3:4], byteorder=byte_ord) >> 4
    di_format = int.from_bytes(arg[5:6], byteorder=byte_ord)
    di_uid = int.from_bytes(arg[8:12], byteorder=byte_ord)
    di_nlink = int.from_bytes(arg[16:20], byteorder=byte_ord)
    di_atime = datetime.datetime.fromtimestamp(int.from_bytes(arg[32:36],
byteorder=byte_ord)).strftime('%Y-%m-%d %H:%M:%S')
    di_mtime = datetime.datetime.fromtimestamp(int.from_bytes(arg[40:44],
byteorder=byte_ord)).strftime('%Y-%m-%d %H:%M:%S')
    di_ctime = datetime.datetime.fromtimestamp(int.from_bytes(arg[48:52],
byteorder=byte_ord)).strftime('%Y-%m-%d %H:%M:%S')
    di_size = int.from_bytes(arg[56:64], byteorder=byte_ord)
    di_blocks = int.from_bytes(arg[64:72], byteorder=byte_ord)
    di_nextents = int.from_bytes(arg[76:80], byteorder=byte_ord)
    di_anextents = int.from_bytes(arg[80:82], byteorder=byte_ord)
    di_forkoff = int.from_bytes(arg[82:83], byteorder=byte_ord)
    di_aformat = int.from_bytes(arg[83:84], byteorder=byte_ord)
    di_next_unlinked = int.from_bytes(arg[96:100], byteorder=byte_ord)
    di_crtime = datetime.datetime.fromtimestamp(int.from_bytes(arg[144:148],
byteorder=byte_ord)).strftime('%Y-%m-%d %H:%M:%S')

```

```

di_ino = int.from_bytes(arg[152:160], byteorder=byte_ord)

global inode_list
inode_list.append([di_ino, di_mode_filetype, di_uid, di_crtime,
di_ctime])

global is_shortformdir
global ctime
global location
# How to identify it has short form directory ?
# 82 Inode offset to xattr (8 byte multiples) 0x23 = 35 * 8 = 280
# 83 Extended attribute type flag (see below) 1
if (di_ino == sb_rootino or (di_mode_filetype == 4 and di_format == 1 and
di_forkoff != 0)):
    is_shortformdir = True
    ctime = di_ctime
    location = di_ino

def buffer_log(arg):
    print('Buffer log')

def inode_update(arg):
    print('Inode update')

def inode_creation(arg):
    print('Inode creation')

def efi(arg):
    print('EFI')

def efd(arg):
    print('EFD')

def others(arg):
    global is_shortformdir

    if (arg[0:4] == b'\x4E\x41\x52\x54'):
        print('Transaction header')
    elif (arg[0:4] == b'\x58\x41\x47\x49'):
        print('AGI')
    elif (arg[0:4] == b'\x49\x41\x42\x33'):
        print('Inode b+ tree')
    elif (arg[0:4] == b'\x58\x46\x53\x42'):
        print('Superblock')
    elif (arg[0:4] == b'\x58\x41\x47\x46'):
        print('AG free')
    elif (arg[0:4] == b'\x41\x42\x33\x43'):
        print('Free b+tree count')
    elif (arg[0:4] == b'\x41\x42\x33\x42'):
        print('Free b+tree offset')
    elif is_shortformdir:

```

```

print(arg)
count = int.from_bytes(arg[0:1], byteorder='big')
i8count = int.from_bytes(arg[1:2], byteorder='big')
len_inumber = 4 if (i8count == 0) else 8
parent = int.from_bytes(arg[2:6], byteorder='big')
dir_entries = arg[6:]

global ctime
global location
global direntries
global num_direntries
direntries.append([location, ctime])

if (count + i8count != 0):
    for i in range(count + i8count):
        namelen = int.from_bytes(dir_entries[0:1], byteorder='big')
        if (namelen == 0):
            break
        offset = int.from_bytes(dir_entries[1:3], byteorder='big')
        name = dir_entries[3:3+namelen]
        ftype = dir_entries[3+namelen:4+namelen]
        inumber =
int.from_bytes(dir_entries[4+namelen:4+namelen+len_inumber], byteorder='big')
        dir_entries = dir_entries[4 + namelen + len_inumber:]

        direntries[num_direntries].append([name, ftype, inumber])

        print('Name:', name, 'Length:', namelen, 'Inode:', inumber)

is_shortformdir = False
num_direntries += 1

def identify_logitem(arg):
    switcher = {
        b'\x4E\x49': inode_core,
        b'\x3C\x12': buffer_log,
        b'\x3B\x12': inode_update,
        b'\x3F\x12': inode_creation,
        b'\x36\x12': efi,
        b'\x37\x12': efd,
    }
    func = switcher.get(arg[0:2], others)
    func(arg)

with open(r"test.img", "rb") as f:
    # Move to the start of journal
    f.seek(sb_logstart)
    while True:
        # Check if the first 4 byte match the journal magic no.
        if(pattern.match(f.read(4))):
            h_len = int.from_bytes(f.read(16-4)[-4:], byteorder='big')

```

```

        # Length of log record is 0, means no more log record
        if h_len == 0:
            break
        h_num_logops = int.from_bytes(f.read(44-16)[-3:],
byteorder='big')
        # Read the remaining of log operation
        f.read(512-44)
        xlog_op = f.read(h_len)
        i = 0
        remaining = h_len
        # Check if trans. ID exists
        while (int.from_bytes(xlog_op[i:i+5], byteorder='big') != 0):
            if (xlog_op[i+4:i+8] == b'\x00\x00\x00\x01'): # Why 128???
                oh_len = 128
            else:
                oh_len = int.from_bytes(xlog_op[i+4:i+8],
byteorder='big')
            oh_clientid = xlog_op[i+8:i+9]
            oh_flags = xlog_op[i+9:i+10]
            # Make sure remaining data are still enough
            remaining = remaining - 12
            if (oh_len > remaining):
                break
            # XFS_TRANSACTION: Operation came from a transaction
            #if (oh_clientid == b'\x69'):
            if (oh_flags == b'\x01'):
                print('Start a new transaction')
            elif (oh_flags == b'\x02'):
                print('Commit this transaction\n')
                is_shortformdir = False
            else:
                xlog_item = xlog_op[i+12:i+12+oh_len]
                identify_logitem(xlog_item)
            # Move to next log item
            i = i + 12 + oh_len

for i in range(len(direntries)):
    for j in range(2, len(direntries[i])):
        if (direntries[i][j][1] == b'\x01'):
            direntries[i][j][1] = 'Regular file'
        elif (direntries[i][j][1] == b'\x02'):
            direntries[i][j][1] = 'Directory'
        elif (direntries[i][j][1] == b'\x03'):
            direntries[i][j][1] = 'Character special device'
        elif (direntries[i][j][1] == b'\x04'):
            direntries[i][j][1] = 'Block special device'
        elif (direntries[i][j][1] == b'\x05'):
            direntries[i][j][1] = 'FIFO'
        elif (direntries[i][j][1] == b'\x06'):
            direntries[i][j][1] = 'Socket'
        elif (direntries[i][j][1] == b'\x07'):

```



```

        direntries[i][j][1] = 'Symlink'

for i in range(len(inode_list)):
    if (inode_list[i][1] == 8):
        inode_list[i][1] = 'Regular file'
    elif (inode_list[i][1] == 4):
        inode_list[i][1] = 'Directory'
    elif (inode_list[i][1] == 2):
        inode_list[i][1] = 'Character special device'
    elif (inode_list[i][1] == 6):
        inode_list[i][1] = 'Block special device'
    elif (inode_list[i][1] == 1):
        inode_list[i][1] = 'FIFO'
    elif (inode_list[i][1] == 12):
        inode_list[i][1] = 'Socket'
    elif (inode_list[i][1] == 10):
        inode_list[i][1] = 'Symlink'
    elif (inode_list[i][1] == 0):
        inode_list[i][1] = 'Deleted File'

print(direntries)
print(inode_list)
results = []
inode_temp = []
file_temp = [[64, 'Root Directory']]
is_hardlink = False
for i in range(len(direntries)):
    # No directory entries
    if len(direntries[i]) == 2:
        continue
    # Update parent directory name, especially root directory
    for n in range(len(file_temp)):
        if (direntries[i][0] == file_temp[n][0]):
            direntries[i][0] = file_temp[n][1]
    # Filter out inode creation time <= directory last change time
    for j in range(len(inode_list)):
        if (inode_list[j][3] <= direntries[i][1]):
            inode_temp.append(inode_list[j])
    # Traverse directory entries
    for k in range(2, len(direntries[i])):
        for l in range(len(inode_temp)):
            # Created file: inode num is the same, inode creation time =
            # changed time
            if (direntries[i][k][2] == inode_temp[l][0] and inode_temp[l][3]
            == inode_temp[l][4]):
                results.append([inode_temp[l][0],
                direntries[i][k][0].decode('utf-8'), inode_temp[l][1], direntries[i][0],
                'Created', inode_temp[l][2], inode_temp[l][3], inode_temp[l][4]])
            # Update current file inode num & name, used for update
            # parent directory
            for o in range(len(file_temp)):

```

```

        if [inode_temp[1][0], direntries[i][k][0].decode('utf-8')] not in file_temp and file_temp[o][0] != inode_temp[1][0]:
            file_temp.append([inode_temp[1][0],
direntries[i][k][0].decode('utf-8')])
            # Deleted file: file type is deleted file
            elif (direntries[i][k][2] == inode_temp[1][0] and
inode_temp[1][1] == 'Deleted File'):
                results.append([inode_temp[1][0],
direntries[i][k][0].decode('utf-8'), inode_temp[1][1], direntries[i][0],
'Deleted', inode_temp[1][2], inode_temp[1][3], inode_temp[1][4]])
                for o in range(len(file_temp)):
                    if [inode_temp[1][0], direntries[i][k][0].decode('utf-8')] not in file_temp and file_temp[o][0] != inode_temp[1][0]:
                        file_temp.append([inode_temp[1][0],
direntries[i][k][0].decode('utf-8')])
                        # Modified file: inode num is the same, inode creation time !=
changed time
                        elif (direntries[i][k][2] == inode_temp[1][0] and
inode_temp[1][3] != inode_temp[1][4]):
                            #print(direntries[i][k])
                            results.append([inode_temp[1][0],
direntries[i][k][0].decode('utf-8'), inode_temp[1][1], direntries[i][0],
'Modified', inode_temp[1][2], inode_temp[1][3], inode_temp[1][4]])
                            for o in range(len(file_temp)):
                                if [inode_temp[1][0], direntries[i][k][0].decode('utf-8')] not in file_temp and file_temp[o][0] != inode_temp[1][0]:
                                    file_temp.append([inode_temp[1][0],
direntries[i][k][0].decode('utf-8')])
                                    # Rename: inode num is the same, file name not the same, directory
last change time <= inode last changed time
                                    temp_name = None
                                    temp_name1 = None
                                    temp_ctime = None
                                    for p in range(len(results)):
                                        if (direntries[i][k][2] == results[p][0] and
direntries[i][k][0].decode('utf-8') != results[p][1] and direntries[i][1] <=
results[p][7]):
                                            temp_name = results[p][1]
                                            temp_name1 = direntries[i][k][0].decode('utf-8')
                                            temp_ctime = results[p][7]
                                            results[p][1] = direntries[i][k][0].decode('utf-8')
                                            results[p][2] = direntries[i][k][1]
                                            results[p][3] = direntries[i][0]
                                            # Update current file inode num & name, used for update
parent directory
                                            for o in range(len(file_temp)):
                                                if (direntries[i][k][2] == file_temp[o][0] and
direntries[i][k][0].decode('utf-8') != file_temp[o][1]) :
                                                    file_temp[o][1] = direntries[i][k][0].decode('utf-8')
                                                    # Rename: update file location
                                                    for p in range(len(results)):
                                                        if (results[p][3] == temp_name and results[p][7] >= temp_ctime):
                                                            results[p][3] = temp_name1

```

```
# Remove inodes already compared
for m in range(len(inode_temp)):
    if inode_temp[m] in inode_list:
        inode_list.remove(inode_temp[m])
inode_temp = []

test = [['Inode', 'Filename', 'Filetype', 'FileLocation', 'Action', 'Account
(UID)', 'Creation Time', 'Last Changed Time']]
# Remove duplicate entries
for a in results:
    if a not in test:
        test.append(a)

df = pd.DataFrame(test[1:], columns=test[0])
df.to_csv(r'test.csv', index=False)
```

Appendix 2 – Shell script used for creating test case

```
#!/bin/sh
sudo mkdir dir{01..02}
cd dir01
sleep 10
sudo touch test01
sleep 10
sudo mkdir test02
cd test02
sleep 10
sudo mknod test03 c 89 1
cd ../../dir02
sleep 10
sudo mknod test04 b 89 1
sleep 10
sudo mkfifo test05
sleep 10
sudo mkdir dir03
cd ..
sleep 10
sudo python -c "import socket as s; sock = s.socket(s.AF_UNIX);
sock.bind('test06')"
sleep 10
sudo ln -s test01 test07
sleep 10
sudo mv dir01/test02 dir01/test02n
sleep 10
sudo mv dir02 dir02n
sleep 10
sudo rm dir02n/test05
```