ERNW
providing security.

ERNW WHITEPAPER 61

WHAT IS NEW IN EXT4 FROM AN INCIDENT ANALYSIS PERSPECTIVE

Version:	1.0
Date:	11/16/2017
Classification:	Public
Author(s):	Dr.-Ing. Andreas Dewald, Sabine Seufert

Matthias Luft

Digitally signed by Matthias Luft
Date: 2017.11.16 13:19:38 +01'00'

TABLE OF CONTENT

ABSTRACT

In incident analysis, especially in the field of postmortem file system analysis, the reconstruction of lost or deleted files plays an important role. The techniques that can be applied to this end strongly depend on the specifics of the file system in question. Various file systems are already well-investigated, such as FAT16/32, NTFS for Microsoft Windows systems, and Ext2/3 as the most common file system for Linux systems. There also exist tools, such as the famous Sleuthkit (Carrier, The sleuth kit (TSK), 2010), that provide file recovery features for those file systems by interpreting the file system internal data structures. In case of an Ext file system, the interpretation of the so-called superblock is essential to interpret the data. The Ext4 file system can mainly be analyzed with the tools and techniques that have been developed for its predecessor Ext3, because most principles and internal structures remained unchanged. However, some innovations have been implemented that have to be considered in incident analysis when it comes to manual analysis, or choosing appropriate tools and the interpretation of results. In this article, we report on changes in Ext4 compared to Ext3 that are relevant for incident analysis and illustrate why in cases where the superblock has been corrupted or overwritten, e.g. because of a re-formatting of the volume, traditional approaches and tools might fail and for those cases refer to an open source tool that we published earlier this year.

1 Introduction

Data reconstruction plays an important role in the field of incident analysis and it is specific to the used file system (Carrier, File system forensic analysis, 2005). The Ext file system family is encounter as the standard file system on Linux and Android systems (Fairbanks, Lee, & Owen III, 2010), which we thus have to deal with in various incident analysis cases. In particular, Ext4 is widely used on Linux-based server environments, or Android-based company mobile devices. This article provides insight about what changed with the development from Ext3 to its successor Ext4 and illustrates an approach that enables reconstructing data without information from the superblock or the group descriptor table of the Ext4 file system.

1.1 Motivation

The Ext4 file system is a widely-used file system, which is nowadays not only standard among Linux distributions, but is also used on mobile devices (Fairbanks, Lee, & Owen III, 2010). Ext4 and its predecessors save the metadata in the so-called superblock or the group descriptor table. Without these metadata structures it is difficult to interpret the file system correctly and to reconstruct the data. Of course, there remains the option of file carving, which however will not be able to recover file system metadata and on the other hand (besides specific techniques for some specific file types) is not able to cope with file fragmentation, too. For previous Ext versions, there are approaches which use the available contents of the metadata structures (Pomeranz, EXT3 File Recovery via Indirect Blocks) for file recovery. For example, on the Ext3 file system, indirect block pointers in inodes - data structures where metadata is to be found on individual files - are saved on content data blocks. By dereferencing these block pointers, it is possible to reconstruct file contents. However, referencing the data contents is handled differently on the Ext4 file system, hence other techniques become necessary for manual analysis as well as for tools. This is the reason for which we want to provide an understanding of the basic principles and inner structures of the Ext file system and answer the question what changed with Ext4, and how to deal with those novelties.

1.2 Related Work

Brian Carrier in his book (Carrier, File system forensic analysis, 2005) describes different partition and file systems. Carrier introduces different methods and tools to support the forensic analysis of the different file systems. The Sleuthkit (Carrier, The sleuth kit (TSK), 2010) is one of his developments, which provides various command line tools for digital forensics.

In his paper, Craiger (Craiger, 2005) describes digital forensic procedures for recovering data from Linux systems. He emphasizes the recovering of deleted and hidden files, data from volatile memory and files with modified extensions.

Fairbanks et al. (Fairbanks, Lee, & Owen III, 2010) compare the Ext4 file system with its predecessor from a forensic perspective, whose results we revisit in this paper. In another paper, Fairbanks (Fairbanks K. , 2012) thoroughly describes the Ext4 file system and introduces the upgrade compared to Ext3. Moreover, the paper documents especially low-level features, such as extents, HTrees, and flex groups. Lee et al. (Lee & Shon, 2014) introduce procedures for recovering deleted files through metadata structures on Ext2 and Ext3 file systems and compare these with existing methods. Narváez (Narváez, 2007) describes a procedure to reconstruct files from an Ext3 file system using the journal.

The work of Pomeranz (Pomeranz, EXT3 File Recovery via Indirect Blocks) illustrates an approach to data recovery on Ext2 and Ext3 file systems that enables the recovery of user data by using indirect block pointers. The author exploits the fact that typically, the first 48 KiB of a file content are not highly fragmented. Consequently, the first 12 block pointers are usually sequentially numbered. However, this particular search pattern cannot be applied to Ext4 file systems because normally (as we explain later), extent structures are used instead of indirect block pointers in order to reference file content.

2 Novelties of the Ext4 File System

In this section, we provide the relevant information about Ext4 in by means of manual analysis, as they are given in (Ext4, 2016). The general layout of Ext4 is very similar to Ext3, but has changed in some ways that we want to focus on now.

First of all, Ext4 introduced an optional 64 bit mode, in which block addresses are 64 bit. This mode is deactivated by default but can be enabled when formatting a volume. If this setting is enabled is stored and can be seen during analysis in the respective flag of the Superblock. Table 1 gives the numbers of blocks, inodes, and so on in comparison between the 32 bit mode and the 64 bit mode.

number of ...	32 bit	64 bit
blocks in FS	2^{32}	2^{64}
inodes in FS	2^{32}	2^{32}
blocks in block group	$8 \cdot b$	$8 \cdot b$
inodes in block group	$8 \cdot b$	$8 \cdot b$
blocks per file (extents)	2^{32}	2^{32}

Table 1: Number of blocks and inodes in 32 vs. 64 mode

The maximum file system size is then the maximum number of blocks in the file system multiplied with the block size: $2^{32} \cdot b$ or $2^{64} \cdot b$, depending on the mode. For a block size of 4096 Byte, this means a maximum size of the file system of 16 TiB or 64 ZiB respectively. Thus, the 64 bit mode is able cope with high volume storage servers as used in modern cloud data centers, for example.

The Ext layout, in general, is based on sequential blocks of 1024, 2048 or 4096 bytes that are numbered and grouped together in block groups. Depending on the block size, each block group consists of 8192, 16384, or 32768 blocks, because exactly one block is used as block bitmap with one bit in the block for one block in the block group. Each block group contains metadata that documents its inner structure. The general layout of all block groups is identical and is illustrated in Figure 1.

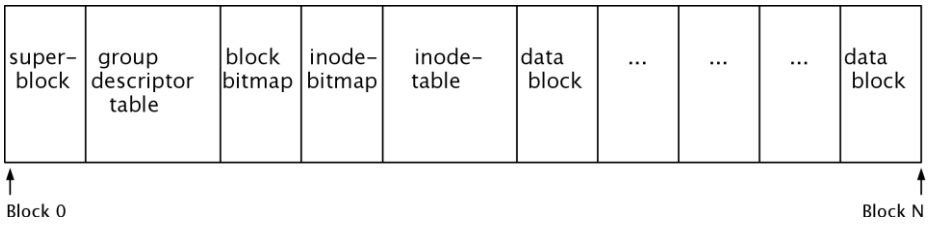

Figure 1: General block group layout.

The superblock contains many essential metadata of the file system, such as the number and size of blocks, number of inodes and reserved blocks, for example. The following group descriptor table contains one group descriptor per block group in the file system and the block bitmap stores the free/used state of each block in the block group in a single bit each. Similarly, the inode bitmap stores the free/used state of each inode (entry) in the inode table. The rest of the block group consists of consecutive data blocks that are used to store data. Table 2 shows the structure of a group descriptor table entry in Ext3.

Offset	Length	Description
0	4	starting block address of block bitmap
4	4	starting block address of inode bitmap
8	4	starting block address of inode table
12	2	number of unallocated blocks in block group
14	2	number of unallocated inodes in block group
16	2	number of folders in block group
18	30	unused

Table 2: Structure of group descriptor table entry.

2.1 Inodes

In all Ext file systems, almost all file/directory metadata, such as timestamps, access rights, references to data blocks for example, are stored in the inode of the file (file names, for example, are not, although they are not always considered as metadata). Inodes are numbered, starting with inode number 1 and stored in the inode table of their respective block group.

The inodes 1 to 10 are reserved for special functions. For example, inode 2 represents the root directory of the volume and inode 8 represents the file system journal, which has been added in Ext3. In Ext4, for the sake of compatibility with prior versions, only few changes to the inode structure have been implemented. To recall the full original structure of Ext3 inodes, refer to Table 3. In Ext4, some of the unused space of Ext3 has been used to introduce new attributes, as shown in Table 4.

Offset	Len.	Description
0	2	mode (file type and access)
2	2	lower 16 bit user-ID
4	4	lower 32 bit file size
8	4	atime
12	4	ctime
16	4	mtime
20	4	dtime
24	2	lower 16 bit group ID
26	2	link count
28	4	sector count
32	4	flags
36	4	unused
40	48	12 direct block pointers
88	4	1 indirect block pointer

92	4	1 double indirect block pointer
96	4	1 triple indirect block pointer
100	4	generation number
104	4	extended attributes (file-ACL)
108	4	upper 32 bits file size/directory-ACL
112	4	fragment block address
116	1	fragmentindex in block
117	1	fragmentsize
118	2	unused
120	2	upper 16 bit user-ID
122	2	upper 16 bit group-ID
124	4	unused

Table 3: Structure of Inodes in Ext2/3, offsets and lengths given in bytes.

Offset	Length	Description
0	2	mode (file type and access)
2	2	lower 16 bit user-ID
4	4	lower 32 bit file size
8	4	atime
12	4	ctime
16	4	mtime
20	4	dtime
24	2	lower 16 bit group ID

26	2	link count
28	4	sector count
32	4	flags
36	4	OS specific
40	60	indirect block pointers or extent data structure
100	4	generation number
104	4	extended attributes (file-ACL)
108	4	upper 32 bits file size/directory-ACL
112	4	fragment block address
116	12	OS specific
128	2	additional size for inode
130	2	upper 16 bit of inode checksum
132	4	additional bits ctime
136	4	additional bits mtime
140	4	additional bits ctime
144	4	cr(eation) time
148	4	additional bits crtime
152	4	upper 32 bit version number

Table 4: Additional inode attributes in Ext4

Amongst others, there has been added space for checksums and additional fields for timestamps, which allow to store timestamps in nanoseconds (10^9 times more fine grained than in Ext3). Further, the problem in Ext3 that timestamps will overflow in year 2038 has been addressed (Pomeranz, SANS Digital Forensics and Incident Response Blog, 2011) by interpreting the stored values as unsigned numbers,

providing one additional bit and shifting the problem to the year 2106. Further, the lowest 2 bits of the newly introduced timestamp fields for the nanoseconds (which only needs 30 bits, but 32 are available) are used in addition, which shifts the problem even further to the ear 2514.

To provide backwards compatibility, the concept of (single/double/triple) (in)direct block pointers to refer to content data blocks is still supported in Ext4. However, this concept is only used, when an old Ext3 file system is converted to Ext4. In all other cases, Ext4 uses an entirely new concept for data block references, which is called *Extents*, which are able to address more storage and allow for bigger files. Further, so called inline files and inline folders can be stored directly in the space for extended attributes. Figure 2 illustrates exemplary how data blocks of a file are referenced by extents:

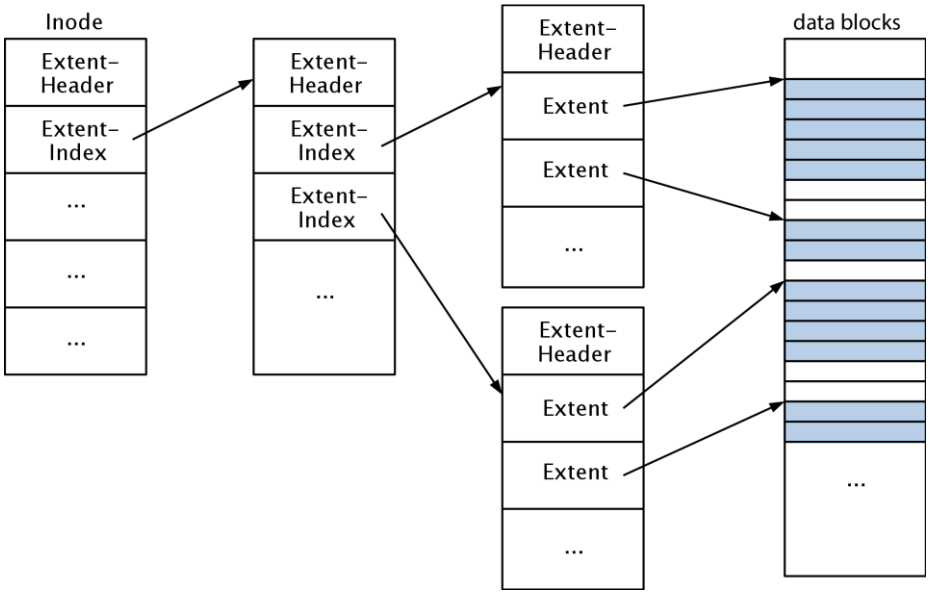

Figure 2: Extent data structure of Ext4

The inode of the example-file on the left side of the figure has 60 bytes to store its extent structure. The size of one extent entry is 12 bytes, thus there can be 5 extent entries stored directly. Each extent structure starts with an extent header with a size of 12 bytes as well. For completeness, the detailed structure of a extent header is shown in Table 5.

Offset	Len.	Description
0	2	Magic Number (`0xf30a`)
2	2	number of valid extent entries after header
4	2	max. number of extent entries
6	2	depth of this node in extent tree
8	4	generation number

Table 5: Ext4 Extent header in extent tree

The extent header is followed by extent nodes, that can be either inner nodes of the extent tree (extent indexes) and point to other extent headers, or leaves (extents) that point to data block runs. For completeness, the structure of those entries is shown in Table 6 and Table 7.

Offset	Len.	Description
0	4	id of first block of own tree-part from beginning of the file
4	4	lower 32 Bit of block address of child entry
8	2	upper 16 Bit of block address of child entry
10	2	unused

Table 6: Ext4 Extent entry for inner node in the extent tree.

Offset	Len.	Description
0	4	id of first block of own tree-part from beginning of the file
4	2	block count covered by this extent
6	2	upper 16 bit of block address of referenced data blocks
8	4	lower 32 bit of block address of referenced data blocks

Table 7: Ext4 Extent entry for a leaf in extent tree.

2.2 Flex-Groups

Another newly introduces feature of Ext4 is the concept of so-called *flex groups*. Flex groups combine multiple block groups to one single logical block group. Only the first block group holds the block and inode bitmaps, as well as the inode tables of all the block groups together.

The Flex-Groups feature can be combines with the Sparse-Superblock-Feature, which results in block groups that consist of data block only and do not contain any metadata. This is achieved by arranging Flex-Groups in potencies of 2 and Sparse-Superblocks in potencies of 3, 5, and 7. Thereby, the only block group that contains superblock, group descriptor table, bitmaps and inode tables is block group 0. Because of the Sparse-Superblock-Feature, block group 1 then contains a superblock and a group descriptor table. Depending on the size of the flex groups, in block group 2 might then begin a new flex group, and would then contain bitmaps and an inode table. The rest of the block groups in the flex group then only consist of data blocks, with the exception of those block groups that again contain copies of the superblock and the group descriptor table because of the Sparse-Superblock-Feature.

The entries of the group descriptor table, whose structure has already been shown in Table 2, has likewise been extended in Ext4. The first 18 Bytes have been left unchanged, while the unused space after is now used as shown in Table 8.

Offset	Length	Description
18	2	flags for block group
20	4	(lower) 32 bit of address for snapshot exclusion bitmap
24	2	(lower) 16 bit of block bitmap checksum
26	2	(lower) 16 bit of inode bitmap checksum
28	2	(lower) 16 bit of number of unallocated inodes
30	2	group descriptor checksum

Table 8: Structure of entry in group descriptor table in Ext4.

In Ext3, there has been developed a feature to make file searching in directories more efficient: The Hash-Index-Feature. Ext4 now contains this as an official feature. To avoid a linear search for a given file name, an H-tree (which is a modified B-tree) over hash sums of the directory names is built. A flag in the inode of a directory stores if this directory is using the hash index entries. Table 9 shows the beginning of such an H-tree, which is contained in the directory after the two directory entries for '.' and '..' that are stored in the beginning of the directory for the sake of downwards compatibility.

Offset	Length	Description
0	4	inode number
4	2	length of the directory entry
6	1	length of the file name
7	1	file type

8	4	file name in ASCII
12	4	inode number
16	2	length of the directory entry
18	1	length of the file name
19	1	file type
20	4	file name in ASCII
24	4	reserved
28	1	hash version
29	1	tree information length
30	1	tree depth
31	1	unused flags
32	2	max. number of entries
34	2	number of following entries
36	4	block id within directory
40	4	min. hash value of second child element
44	4	block id within directory
		...

Table 9: Root node of an H-tree

After the two predefined entries in the directory, the metadata that describes the H-tree is stored. First, there is general information about the tree, and then there is information about child nodes. The first reference to a child node is a bit different, as can be seen from the table and is explained in the following, but all further references to child nodes are structured as follows: First, the lower boundary of hash values (of file names) that are stored in this part of the tree is given. The upper boundary is implicitly given

by the lower boundary of the next element. As the entire tree spans the entire range of an unsigned 4 byte integer, the lower boundary of the entire tree is known (zero), and the upper boundary as well (2^{32}). This is the reason for which the reference to the first child node does not need an explicit lower boundary for its hash values. Those 4 Bytes are used to store the maximum and the actual number of following child node references instead.

To find a file, the hash sum of the searched file name is calculated and compared to the entries of the H-tree. Because of the sorted storage of the entries according to their hash values and the hierarchical structure of the tree, the block that contains the searched file can be found in logarithmic run time. This block can then be searched linear for the given file name (a block stores multiple file names). Table 10 shows the structure of an inner node of the H-tree, which contains a traditional directory entry in the beginning that signals an empty block, when interpreted by the traditional linear file search method.

Offset	Length	Description
0	4	inode number (0)
4	2	length of directory entry (block size)
6	1	name length (0)
7	1	file type (0)
8	2	max. number of entries
10	2	number of following entries
12	4	id of block within directory

Table 10: Inner node of a directory H-tree.

Listing 3 shows an example of a directory with Hash-Index structure:

```
0x82309000  70 14 02 00 0c 00 01 02  2e 00 00 00 04 fd 01 00  |p...............|
0x82309010  f4 0f 02 02 2e 2e 00 00  00 00 00 00 01 08 00 00  |................|
0x82309020  fc 01 0c 00 01 00 00 00  fe b0 c3 1c 08 00 00 00  |................|
0x82309030  34 24 79 3d 04 00 00 00  08 0f 1a 5c 07 00 00 00  |4$y=.......\....|
0x82309040  7c 61 de 77 02 00 00 00  2e e6 d6 85 0c 00 00 00  ||a.w............|
0x82309050  44 47 b9 93 05 00 00 00  58 02 38 a4 0b 00 00 00  |DG......X.8.....|
0x82309060  2e a8 76 b7 03 00 00 00  0e b7 68 cb 09 00 00 00  |..v.......h.....|
0x82309070  f4 81 a0 df 06 00 00 00  c2 00 1c f0 0a 00 00 00  |................|
0x82309080  7f 14 02 00 10 00 08 01  61 63 6f 6e 6e 65 63 74  |........aconnect|
0x82309090  80 14 02 00 14 00 0c 01  61 63 70 69 5f 66 61 6b  |........acpi_fak|
...
```

Listing 3: Hexdump of a directory with Hash-Index (/usr/bin/)

The directory starts with the first two entries for '.' and '..' in the first 24 Bytes. Then there follows the general tree information, marked in red. The first 4 Byte are zero by default and are followed by the used hash version, here MD4. After this, the length of the tree information is stored (here 8) and the depth of the tree (here 0). In purple, the first child reference with the maximum and actual number of child references (instead of a lower hash boundary as discussed) is given. In our example, this is `0x1fc` and `0x0c`, meaning 12 of 508 possible entries. Next, highlighted in light blue, there follow 11 entries of 8 Byte each. Thereby, the first 4 Byte denote the hash value of the lower boundary of the child and the last 4 Byte denote the block number of the block that stores the file names of the referred files.

3 Ext4 File Recovery using AFEIC

Using a pattern-based file carving method, a search for metadata structures of inodes can be performed to recover their content data. To this end, we published our tool AFEIC (Dewald & Seufert, 2017), which tries to avoid reading the superblock and the group descriptor table, because our goal is to recover files from corrupted or reformatted Ext4 file systems, in which cases those structured would have been overwritten. Thus, only a minimum of information about the file system is required to compute parameters essential for recovery. To this end, AFEIC uses carving techniques (with more complex patterns, as there are no real magic bytes for an inode) to identify potential inodes on a volume and analyze their metadata structures and handled according to their file type. Furthermore, the interpretation of directory entries allows to recover inode numbers and file names with their complete file paths for regular files.

AFEIC is implemented as Sleuthkit module and provides two distinct recovery modes: The so called *content data mode*, that exclusively recovers the content of regular files, for which only the block size of the Ext4 file system must be provided or detected correctly. And the *metadata mode*, that requires more Ext4 parameters because the necessary inode numbers need to be calculated by the module. Then, file names with their complete file paths can be recovered using directory entries in this mode. The entire recovery process of AFEIC can be divided into the following phases, which are described in the next sections:

1. Initialization
2. Inode carving
3. Directory tree
4. Regular files
5. Files without content

3.1 Initialization

The goal of the initialization is to gather all required Ext4 parameters. These can be specified by the user or estimated in accordance to the file system size. The following parameters are of relevance:

- Offset
- File system size
- Block size
- Inode size
- Inode ratio
- Flex group size
- 64 bit mode
- Sparse superblock
- Number of blocks per block group

- Number of blocks and block groups in the file system
- Number of inodes per block group
- Space for growing group descriptor table

The default values for new Ext4 file systems used by mkfs are shown in Table 11.

mkfs Type	Parameter	Value
floppy	block size	1024 Byte
(to 3 MiB)	inode size	128 Byte
	inode ratio	8192
small	block size	1024 Byte
(to 512 MiB)	inode size	128 Byte
	inode ratio	4096
default	block size	4096 Byte
(to 4 TiB)	inode size	256 Byte
	inode ratio	16384
big	block size	4096 Byte
(to 16 TiB)	inode size	256 Byte
	inode ratio	32768
huge	block size	4096 Byte
(from 16 TiB)	inode size	256 Byte
	inode ratio	65536

Table 11: Default parameters as chosen by mkfs for Ext4.

Furthermore, there are relevant values with mkfs defaults that are independent of the file system size. Those are:

- Number of block groups per flex group (default: 16)
- Block address width (default: 32 bit)
- Usage of sparse superblock (default: activated)

The sparse superblock option causes not every block group to possess a copy of the superblock. All other necessary values can be derived from the already presented parameters. Once all parameters are known, the positions and sizes of the inode tables of all block groups can be computed. Mapping the physical address of an inode to its inode number can then be performed.

With the number of the block groups and the information about whether the 64 bit mode is enabled, the size of the group descriptor table can be calculated. The size of the group descriptor table along with the superblock add to the offset to the inode table within a block group – provided they aren't omitted due to the sparse superblock option. Additional space for a growing group descriptor table must also be taken into account. Finally, if flex groups are enabled, the inode and block bitmaps and the inode tables are grouped at the beginning of a flex group.

3.2 Inode Carving

Due to the inner structure of an inode, not every 128 byte permutation constitutes a valid inode. For an inode to be plausible and correct, certain interrelations between its values must be fulfilled. AFEIC makes use of this fact to formulate search patterns with which potential inodes are carved in a byte-wise manner.

The most significant 4 bits of the first 2 byte structure in the inode indicate its file type. All but two values can be ignored, since only regular files and directories are relevant for recovery. Search patterns can also be defined on timestamps, such as a time interval or their inner consistency. Therefore, the different timestamps of a file can be used, such as modification time (mtime), creation time (ctime) and deletion time (dtime).

Furthermore, the extent header field must always contain the magic number `0xf30a`. Further, any other inode attribute can be used for the definition of search patterns, depending on the exact recovery use case (e.g. access rights, user and group ID). All found addresses of potential inodes are grouped by their file type (regular files and directories) and used for future recovery steps.

3.3 Directory Tree

In this phase, the potential directory inodes are analyzed. To this end, their extent entries are interpreted in a way analogous to the content data phase, which is described in Section 3.4.

Directory entries are searched linearly whereby directories not starting with entries for '.' and '..' can be discarded. The inode number and file name of a directory entry is saved along with its parent inode number. This reference pattern constitutes a logical tree from which the complete file path can be deduced. However, inode numbers of regular files are not inherently known and the module must map physical inode addresses to inode numbers. Together with the information gathered from the directory tree, file names can be associated to inodes. Hence, the metadata mode is able to reconstruct the whole directory structure of the file system. If a directory is irreparable, its children's file paths cannot be reconstructed. Then, their content is saved in files named after their inode addresses.

The content data mode cannot map physical addresses to inode numbers due to the lack of necessary Ext4 parameters. This fact excludes the directory tree phase from the content data mode. All recovered files are named after their inode address and saved in a flat hierarchy.

3.4 Regular Files

Whether or not the file path has been reconstructed for a given regular file, its content is now recovered. In Ext4 file systems, file content is spread across the volume in a way managed by so-called extents, whereas Ext2/3 file systems use indirect block pointers as described in the previous sections. AFEIC does not support the latter, as those are already well-covered by older tools, for which we recommend to use the standard Sleuthkit, for example.

Since the file system journal can contain copies of existing inodes and because the content data mode does not consider inode table boundaries, duplicate inodes with different physical addresses can be found. Therefore, the file size and the extent structures of the inodes are compared, as they identify same content.

3.5 Files without content

Similarly, to the directory tree phase, this phase is optional and builds upon the results of the directory tree and content data phases and is thus only available in the metadata mode. Files found in the directory tree phase but not in the inode carving phase cannot be recovered with respect to their content but indeed

to their file name. Hence, they can be written as empty files with their original name and complete file path. Examples include symbolic links or device files.

4 Conclusion

In this article, we summarized the changes that have been introduced with Ext4 compared to Ext3 and explained the inner structures as a basis for manual incident analysis. Further, we shortly introduced AFEIC, which has been described in a prior publication and made publicly available. It allows for recovery of files from Ext4 volumes even without the presence of a valid superblock or detailed knowledge about the original configuration of the filesystem.

5 References

Carrier, B. (2005). *File system forensic analysis* (3 ed.). Addison-Wesley Reading.

Carrier, B. (2010). *The sleuth kit (TSK)*. Retrieved 10 14, 2017, from http://www.sleuthkit.org/sleuthkit/

Craiger, P. (2005). *Advances in Digital Forensics - Recovering digital evidence from Linux systems.* Springer.

Dewald, A., & Seufert, S. (2017). AFEIC: Advanced Forensic Ext4 Inode Carving. *Proceedings of DFRWS EU.*

Ext4. (2016). Retrieved 10 14, 2017, from Ext4 Disk Layout: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

Fairbanks, K. (2012). An analysis of Ext4 for digital forensics. (Elsevier, Ed.) *Digital investigation*(9), pp. 118-130.

Fairbanks, K. D., Lee, C. P., & Owen III, H. L. (2010). Forensic implications of ext4. *Proceedings of the sixth annual workshop on cyber security and information intelligence research* (p. 22). ACM.

Lee, S., & Shon, T. (2014). Improved deleted file recovery technique for Ext2/3 filesystem. *The Journal of Supercomputing, 70*(1), pp. 20-30.

Narváez, G. (2007). Taking advantage of ext3 journaling file system in a forensic investigation. *SANS Institute Reading Room*.

Pomeranz, H. (2011, 03 14). Retrieved 10 14, 2017, from SANS Digital Forensics and Incident Response Blog: https://digital-forensics.sans.org/blog/2011/03/14/digital-forensics-understanding-ext4-part-2-timestamps

Pomeranz, H. (n.d.). EXT3 File Recovery via Indirect Blocks.