Digital Investigation 9 (2012) S118–S130

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

An analysis of Ext4 for digital forensics

Kevin D. Fairbanks

Johns Hopkins University Applied Physics Laboratory, Laurel, MD 20723, USA

Keywords:
Ext4
File system forensics
Digital forensics
Extents
Flex block groups

A B S T R A C T

This paper presents a low-level study and analysis of Ext4 file system data structures. It includes descriptions of extents, extent trees, directory indexing HTrees, and flex block groups. Currently, data about the file system is scattered with most sources focusing on one particular feature. This paper provides a more comprehensive analysis with the forensics community in mind and states some brief implications of the file system behavior with respect to data recovery.

1. Introduction

Ext4 is currently the default file system on new installations of several popular Linux distributions. Despite the fact that in Tso (2008) it is characterized as a stop-gap solution until Btrfs was ready, its current status and adoption by Android (Tso, 2010) greatly increases the probability that it will be the file system used on a digital device that must be analyzed. To that end, tools must be developed and tested that can properly interpret the data structures of this file system. While there are several papers that document the design of Ext4, or highlight certain features from the point of view of the developers of the file system, there is a lack of information presented from the perspective of digital forensics such as Beebe et al. (2009) does for ZFS. This article addresses this problem by providing a more comprehensive low-level study of the Ext4 file system with special attention paid to the on-disk data structures than has been hitherto presented.

Specifically, this article will discuss changes in file system topology and important data structures such as the file system superblock, block group layout, and inode data mapping. It also provides some benefits of the changes. For example, changes in block group layout allow data to be contiguously allocated across block group boundaries while changes in the inode structure allow nanosecond time resolution. Furthermore, details are given on how extents are implemented and used in Ext4 including an in-depth look into extent trees. As an Ext4 system can contain many times the data of an Ext3 file system, information is provided on how this is accomplished while staying somewhat compatible with the previous version of the file system. It is also shown how deleted data can potentially be recovered from the file system either by examining HTrees or inode structures and extent index nodes when extent trees are created.

In Section 2, an overview of the previous versions of the file system are provided. Section 3 describes the generation of the test images, and Section 4 details features of the Ext4 file system while pointing out major differences between it and its predecessors. In Section 5, experimental results are displayed and discussed in the context of forensic data recovery. A brief overview of related work is provided in Section 6 before conclusions and future work are presented in Section 7.

2. ExtX

This section contains an overview of the Ext2 and Ext3 file systems to provide reference points and help display the differences detailed in the Section 4 discussion of Ext4. While it may be considered as primer, this section contains a discussion of HTrees, used for directory indexing, which are were optional in Ext3 but are default in Ext4.

E-mail address: Kevin.Fairbanks@jhuapl.edu.

1742-2876/$ – see front matter © 2012 The John Hopkins University, Applied Physics Laboratory. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2012.05.010

2.1. Topology

In Carrier (2005), Carrier discusses Ext2 and Ext3 together using the term ExtX because Ext3 can be thought of as an extension of Ext2. Both file systems share the same basic data structures and a key factor in the development of Ext3 is its backward compatibility with the already popular Ext2. For these reasons, this discussion serves as an introduction to both file systems with any key differences between them further explained.

The basic unit of work for most hard disks is the *sector* which is usually 512 bytes in size. File systems, on the other hand, tend to operate on units that are comprised of multiple sectors. In ExtX, these are called *blocks*. ExtX supports 1024, 2048, and 4096 byte block sizes. As displayed in Fig. 1, these blocks are organized into units appropriately called block groups. Each block group, with the possible exception of the last, has an equal number of data blocks and inodes. An *inode* stores basic metadata information about a file such as timestamps, user and group permissions, as well as pointers to data blocks. As every file must have a unique inode, the number of files that a system can support is limited not only by the amount of free data blocks but by the total number of inodes as well. In ExtX, not all blocks are equal. Along with data blocks, the file system uses group descriptors, inode and data block bitmaps, inode tables, and a file system superblock.

The ExtX superblock is a metadata repository for the entire file system. It contains information such as the total number of blocks and inodes in the file system, the number of blocks per block group, the number of available blocks in the file system, as well as the first inode available. A group descriptor contains meta information about a particular block group such as the range of inodes in the group, the blocks included in the group, and the offset of key blocks into the group. Originally, the superblock and group descriptors were replicated in every block group with those located in block group 0 designated as the primary copies. This is no longer common practice due to the Sparse SuperBlock Option (Carrier, 2002). The Sparse SuperBlock Option only replicates the file system superblock and group descriptors in a fraction of the block groups. This can allow for a significant amount of space savings on larger partitions that can have a multitude of block groups and therefore a larger set of group descriptors.

Each group has an inode bitmap and a data block bitmap that is limited to the size of one block. These bitmap blocks limit the number of inodes and data blocks of a particular group. As the maximum block size currently supported in ExtX is 4096 bytes, the maximum number of inodes or blocks in a group is 32k (2^{15}). In the bitmaps, a '1' is used to identify an unavailable object and a '0' denotes that the object is unused. If the number of objects that a group can theoretically contain is greater than the number that it actually contains, the corresponding bitmap spaces are set to '1'.

In ExtX, an inode is 128 bytes in size by default. Inodes are stored sequentially in a structure called the inode table that is present in each group. The size of the inode table is related to the block size the file system uses. For example, using a 4096 byte block, the maximum number of inodes per a group is 32k inodes. As each block can hold 32 inodes, the maximum size of the inode table in this example would be 1024 blocks.

In Fig. 2, the almost ubiquitous diagram portraying the inode data block pointer system can be seen. In an effort to keep the inode structure from becoming too large while giving it the ability to support large file sizes, different types of pointers were implemented. The most straightforward method uses single indirect pointers which are stored inside of the inode structure. These point directly to file data. The double indirect pointer, on the other hand, points to a file system block which acts as a repository for single indirect pointers. The triple indirect pointer adds another level of indirection onto this scheme. More advanced file systems, such as Ext4 in Section 4, may use structures known as extents to denote sections of contiguous data blocks, but Ext3 uses the pointer system described above for compatibility reasons (Eckstein, 2004).

The primary intention of this detailed explanation of the pointer system is to bring attention to how Ext2 and Ext3 handle inode structures differently. Upon file deletion, Ext2 marks the inode associated with the deleted file as available by editing its inode and data block bitmaps. This practice makes it possible to recover a deleted file if its contents have not been overwritten, as the inode structure will contain pointers to the data blocks. Ext3, in contrast, performs the additional step of overwriting block pointers in the inode structure upon deletion. This increases the difficulty of file recovery and requires the use of techniques known as data carving, not discussed in this document, to

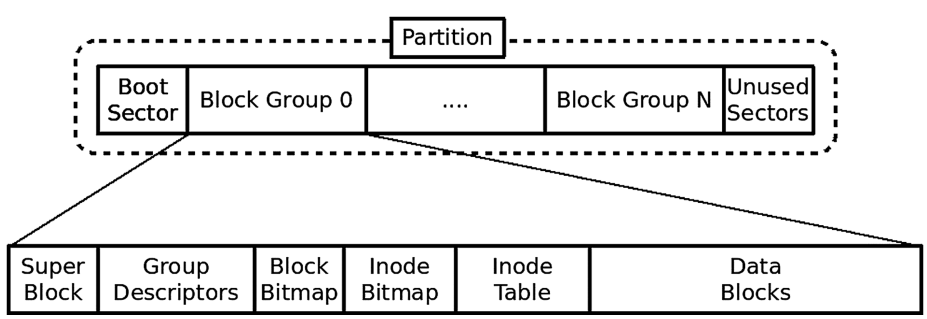


Fig. 1. File system layout. Adapted from Bovet and Cesati (2005).

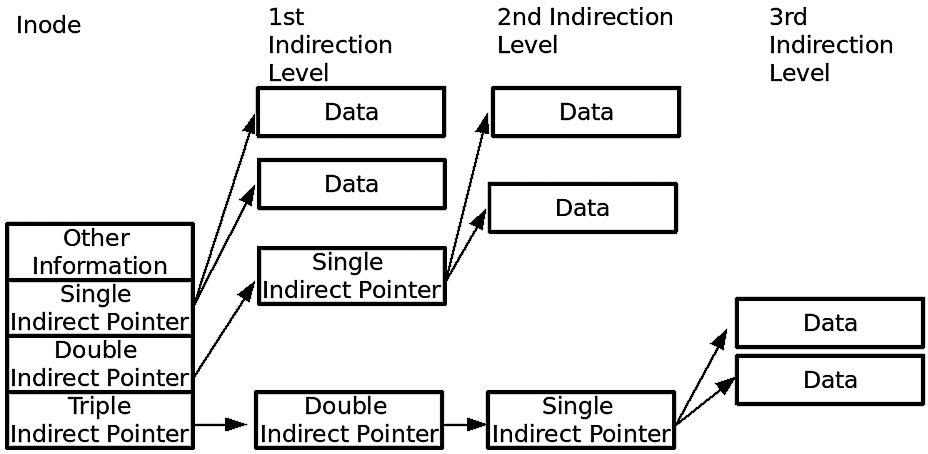


Fig. 2. Inode data block pointers. Adapted from Carrier (2005).

reconstruct the file. A more complete description of ExtX can be found in Carrier (2005) and Bovet and Cesati (2005).

2.2. Journaling

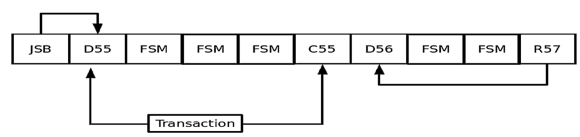
Journaling is a technique employed by file systems in crash recovery situations. It offers reduced recovery time at the cost of additional file system overhead in terms of time and/or space. The journal, in Ext3, takes on the form of a fixed size log that regularly overwrites itself. Actions are recorded to the journal and either replayed in their entirety or not at all during file system recovery. This feature, called atomicity, allows Ext3 to come to a consistent state more quickly than through the use of a file system check program such as `e2fsck`.

With Ext3, the actual journaling action is handled separately from the file system by the Journal Block Device. The Ext3 file system is used to identify transactions which are then passed to the Journal Block Device to be recorded. A transaction is a set of updates sent to the device which are uniquely identified by sequence numbers. The transaction sequence number is used in descriptor and commit blocks to denote the start and end of a transaction respectively. The data that is being journaled is nested between the descriptor and commit blocks. The descriptor block also contains a list of the file system blocks that are updated during a transaction. It is important to note that the commit block is only created once the file system data has been recorded to the disk to ensure atomicity.

There also exists a revoke block. This, like the descriptor block, has a sequence number and a list of file system blocks, but its purpose is to prevent changes during the recovery process. Any file system block listed in the revoke block that is also in a non-committed transaction with a sequence number less than that of the revoke block, is not restored.

The next structure that the journal uses is probably one of the most important: the superblock. This should not be confused with the file system superblock, as this is specific to the journal. This structure has the task of identifying the first descriptor block in the journal. The usage of the journal as a circular log makes this task necessary, as it cannot be assumed that the beginning of the journal is located at the beginning of its physical storage space (Carrier, 2005). The relationship between the different types of journal blocks is displayed in Fig. 3. In this figure, transaction 55 is complete as the set of metadata blocks associated with it are enclosed with a descriptor block and commit block. However, transaction 56 was not completed so no commit block was created. When the file system is mounted again and the journal checked, a revoke block will be created. Since transaction 56 has no commit block and it has a sequence number less than 57, the changes associated with this transaction will not be replayed.

The three documented modes of journaling are Journal, Ordered, and Writeback (Bovet and Cesati, 2005). Journal mode, while certainly being the safest method, is the most costly of the three in terms of performance. It writes both file data and metadata to the journal and then copies the information to the actual file system area on the disk. To avoid writing everything twice, Ordered mode only writes metadata to the journal. In this mode, care is taken to write to the file system disk area before writing to the journal area. In most Linux distributions, it is the default mode in which the journal operates. In Writeback mode, only the



Key

Abbreviation	Meaning
JSB	Journal Super Block
D[X]	Descriptor Block X
FSM	File System Metadata
C[X]	Commit Block X
R[X]	Revoke Block X

Fig. 3. Ext3 journal blocks. Adapted from Carrier (2005).

metadata is journaled, but no precaution is followed to ensure data is written to the file system area first.

2.2.1. Directory indexing

Although the directory indexing originally implemented by Phillips (2002) has been an optional feature since 2002, it is a default option for Ext4. ExtX systems default to using a linked list structure for storing directory entries. While performance benefits of switching to a tree structure are very compelling when a large amount of entries are stored in a directory, this section is primarily focused with providing an overview of HTrees. For the sake of brevity, structure definitions have been excluded from this discussion, but can be found in Carrier (2005) and the Linux kernel source file fs/ext4/namei.c.

The fake directory structure (fake_dirent) plays an important part in the layout of directory files. Two of these structures are present in the directory index root structure (dx_root) which allow it to make entries for the current and parent directories in an indexed directory. Every indexed directory begins with a dx_root structure followed by one or more directory index entry (dx_entry) structures. Each dx_entry consists of a hash key and a pointer to a block of the directory file to which that key maps. Example dx_entry structures are highlighted in Fig. 4. While the hash value is 32 bits in length, the lowest bit is used as a flag to denote hash collisions (Phillips, 2002).

The leaf blocks in the HTree implementation are stored as ExtX style linked lists. This allows the indexed directories to be backward compatible as the directory root in the first block sets the record length of the second fake_dirent to point to a block boundary. An example of this can be seen in Fig. 4 as the record length field, encircled in the figure, of the second directory entry is set to `0x03F4` little endian. If the index is corrupted, it can be treated as a collection of deleted entries, skipped over, and the collection of leaf blocks will be read using the linked methods from the previous versions of the file system. An example leaf block that corresponds to Fig. 4 is shown in Fig. 6 with the beginning of the first directory entry highlighted.

As of this writing, the maximum depth the HTree implementation supports is two levels. When the number of dx_entries exceeds what can be held in one block, which depends upon the file system block size, index nodes (dx_nodes) must be made. When the indirect_levels field of the dx_root structure is greater than zero, then each dx_entry is interpreted as a hash and a pointer to a dx_node. The dx_node begins with a fake_dentry structure which is highlighted in Fig. 5. Note that the fake_dentry does not point to a valid inode number and the length of the record is set to `0x400`, which is the size of the entire file system block used in this case. When this entry is read disregarding directory indexing, the entire block will be considered full of deleted directory entries. Following the filename associated with the fake_dentry, is the final dx_entry which points to a leaf node.

3. Testing and validation

This section details the steps taken to create test media for the examination of Ext4. The results of each experimental process were verified by using the `debugfs` tool from the `e2fsprogs`. By opening the image in this environment, we were able to verify the location of inode structures, confirm file deletion, and extract the contents of directory files so that they may be scrutinized using hex editors. The testing environment consisted of an Ubuntu 10.04 LTS 64 bit machine with a 2.6.32-38 kernel.

3.1. Extent tree image

To make an Ext4 file system that contained a fairly complex extent tree, a 100 MB zero-filled file was created using dd. Next, each sector of the file was filled with its sector number. Then the `mke2fs` command was run with the following options: "-q -F -t ext4 -b 1024 -G2 -i 1024 -E lazy_itable_init=1". This created an Ext4 file system with 2 block groups per flex group, a 1024 byte file system block size, and 1024 bytes per-inode. To ensure the image would fill with data and file creation would not cease due to exhaustion of all inodes, 1 inode for every file system data block needed to be created. The next step was to mount the image and fill the entire drive with 1024 byte files.

The files were filled with text strings in the following format: [filename]-[file sector]-[fs block number]-[cntr] as displayed in Fig. 7. The cntr field is used to determine file position in the case of partial data recovery. This allows file contents to be uniquely identified when testing for deletion and recovery. Every 1000 files were grouped into a subdirectory. Next, every other file in the file system was deleted. Finally, one large file was created and written to until a disk full error was received. By following these steps, the large file was highly fragmented and forced to create an extent tree that mapped to blocks across different block groups and different flex groups. Figs. 4 and 6 in Section 2.2.1 are derived from this image. This entire process has been automated through the use of python and shell scripting.

Fig. 8, shows part of the extent map of the large extent tree. Clearly, there are more than 4 extents which make up

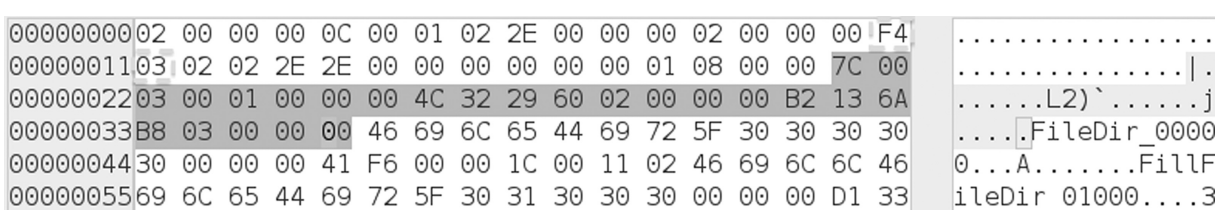


Fig. 4. Partial dx_root. File system block size = 1024 bytes in examples.

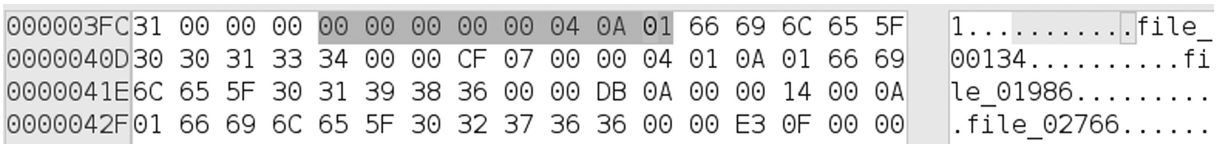


Fig. 5. Partial dx_node. File system block size = 1024 bytes in examples.

this file; moreover, the pattern of each individual extent being 1 block in size is as expected.

3.2. Directory hashing

Although the previously described image contains the desired effect of a highly fragmented file, it does not contain enough entries in any one directory to demonstrate a two level constant depth HTree. To obtain this data structure, the same process described above was repeated; however, all of the small files were kept in the file system root directory. Therefore, before deletion of half of the small files, the root directory contained on the order of 80,500 files. It is from this image that Fig. 5 in Section 2.2.1 is derived.

4. Ext4

This section begins with detailed descriptions of Ext4 data structures followed by discussions of how issues such as scalability, compatibility, and reliability are addressed in the file system. It is explained how an Ext4 file system can contain many times the data of an Ext3 file system and how the maximum file size has been increased from 2 TB to 16 TB. Also covered, is the implementation of persistent preallocation with extents and the usage of checksums in the file system journal and group descriptors to improve reliability.

4.1. Topology

4.1.1. Superblock

The Ext4 superblock is much like the ExtX superblock with the addition of new fields to support 64 bit block numbers. Thus far, the timestamp fields in the superblock are still 32 bits and do not have a high bits field associated with them; however, the s_log_groups_per_flex field plays an important part in block group organization. In the most recent versions of the kernel, the superblock also contains fields for error tracking and snapshots.

4.1.2. Block groups

Although Ext4 has the same basic data structures as its predecessors, there are differences. The first major difference is the organization of the block groups. In Mathur et al. (2007), it is stated that the 128 MB block group size limit has the effect of limiting the entire file system size. This is due to the fact that only a limited number of group descriptors can fit into the span of one block group. In the literature, there have been two solutions mentioned for this problem: the meta-block group feature and the flex block group feature.

The meta-block group feature, mentioned in Tso and Tweedie (2002), allows the creation of meta-block groups that each consist of a series of block groups that can be described by a single descriptor block. More detail is given in Mathur et al. (2007) as backups of the meta-block group descriptors are said to be located in the second and last group of each meta-block group.

Fig. 9 graphically demonstrates how the flex block group feature extends the previous notion of creating large contiguous swaths of block groups by moving the block and inode bitmaps as well as the inode tables to the first block group in a flex block group along with the group descriptors (Kumar et al., 2008). With the sparse superblock feature being enabled by default, some block groups may contain backup copies of the superblock, group descriptors and group descriptor growth blocks. By lifting the restriction of metadata blocks having to reside in the block group to which they refer, a larger virtual block group can be created which allows free contiguous spans of data blocks to be allocated across group boundaries. On the development mail list archives, there can be observed discussions about whether the two options should have different feature flags. It was agreed that this was necessary so that a clear definition of META_BG and FLEX_BG is maintained. The FLEX_BG feature has been detailed as it is selected by default in the `mke2fs` configuration file.

The number of block groups in a flex group must be a power of 2. Also, the flex group block and inode bitmaps are a concatenation of the bitmaps from the constituent

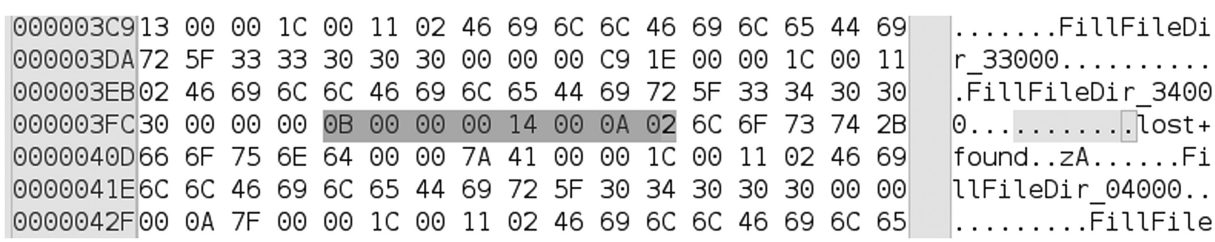


Fig. 6. Partial directory index leaf node. File system block size = 1024 bytes in examples.

```
00263AE3 30 2D 31 30 7C 66 69 6C 65 5F 38 31 30 34 30 2D 31   0-10|file_81040-1
00263AF4 2D 38 31 30 34 30 2D 31 31 7C 66 69 6C 65 5F 38 31   -81040-11|file_81
00263B05 30 34 30 2D 31 2D 38 31 30 34 30 2D 31 32 7C 66 69   040-1-81040-12|fi
00263B16 6C 65 5F 38 31 30 34 30 2D 31 2D 38 31 30 34 30 2D   le_81040-1-81040-
```

Fig. 7. Example file content.

block groups. The size of each of these structures will be equal to the file system block size multiplied by the number of groups per flex group. Similarly, the flex group inode table is a concatenation of block group inode tables. By first calculating the size of the inode table per group and multiplying the result by the number of groups in a flex group, the size of the flex group inode table can be obtained. As these metadata structures now lay outside of the block group they are used to administer, the maximum span of free contiguous data blocks has increased, albeit inside of this larger virtual block group, to be greater than the 128 MB maximum imposed by the block group scheme used by Ext3. The flex group feature does not currently affect the location backup superblocks, group descriptors, or group descriptor growth blocks. While these structures may interrupt spans of free contiguous data blocks, their occurrence becomes less frequent as the number of block groups increase.

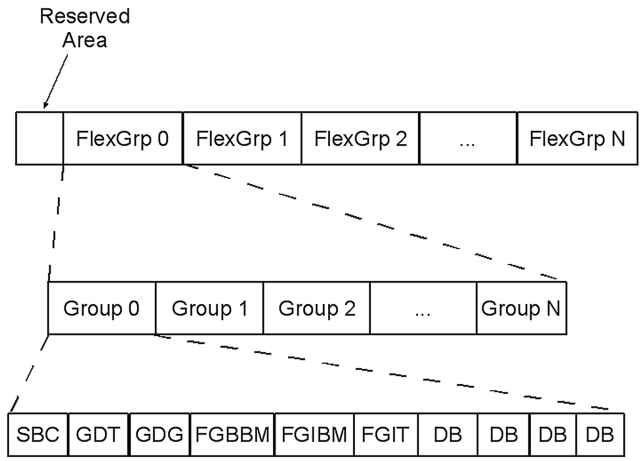
4.1.3. Extents

Another major feature in Ext4 is the use of extents rather than the previously described block mapping method shown in Fig. 2. Extents are more efficient at mapping data blocks of large contiguous files as their structure generally consists of the address of the first physical data block followed by a length. The Ext4 extent structure is shown in Fig. 10. In Ext4, an unbroken span of up to 2^{15} blocks can be summarized with one extent entry. If the file system block size is 4 KB, this maps up to 128 MB of data with a single entry. The highest bit in the length field is used for persistent preallocation purposes. Through its usage, space can be allocated for files without having to actually initialize the reserved blocks. When these special extents are read from, the virtual file system (VFS) returns zeros to the requesting application (Mathur et al., 2007). The 48 bit physical block number indicates the file system block on which the extent begins, while the 32 bit logical block number identifies the offset into the file on which the block run begins. Fig. 10 also displays the structure of the extent header and extent index. While the extent header is always present, the extent index plays a role in extent tree structures.

In Mathur et al. (2007), it is stated that an inode can hold four extents. By comparing the inode data structures of Ext3 and Ext4, one is able to determine that the placement of the Ext4 extents is precisely located where the block pointers in Ext3 were. At this location, there is an array 60 bytes long. The first 12 bytes of this array contain an extent header. The eh_magic, eh_entries, eh_max, and eh_depth fields of the extent header structure are each 2 bytes, while the eh_generation field is 4 bytes. The magic number, present in every extent header, is statically defined in the Ext4 source code as 0xf30A. If a file is highly fragmented, very large, or sparse then a tree of extent structures is constructed as shown in Fig. 11. While the magic number can be thought of as a type of version number for extent structures, the other fields in the extent header play a more active role. The eh_entries field stores the number of valid extent entries in a tree node, and the eh_max field denotes how many entries the current node can hold. While the number of valid entries will vary with size and fragmentation of a given file, the maximum number of entries varies with block size. In a file system with the default block size of 4 KB, 340 entries can be held in a single block. The kernel code states that eh_generation holds the "generation of the tree".

```
Inode: 14   Type: regular    Mode:  0644   Flags: 0x80000
Generation: 2319071877    Version: 0x00000001
User:     0   Group:     0   Size: 41216000
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 81474
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x4f295827 -- Wed Feb  1 10:20:07 2012
atime: 0x4f295806 -- Wed Feb  1 10:19:34 2012
mtime: 0x4f295827 -- Wed Feb  1 10:20:07 2012
EXTENTS:
(0): 2447, (1): 2449, (2): 2451, (3): 2453, (4): 2455, (5): 2459, (6): 2461, (7): 2463, (8): 2465, (
9): 2467, (10): 2469, (11): 2471, (12): 2473, (13): 2475, (14): 2477, (15): 2479, (16): 2481, (17):
2483, (18): 2485, (19): 2487, (20): 2489, (21): 2491, (22): 2493, (23): 2495, (24): 2497, (25): 2499
, (26): 2501, (27): 2503, (28): 2505, (29): 2507, (30): 2509, (31): 2511, (32): 2513, (33): 2515, (3
4): 2517, (35): 2519, (36): 2521, (37): 2523, (38): 2525, (39): 2527, (40): 2529, (41): 2531, (42):
2533, (43): 2535, (44): 2537, (45): 2539, (46): 2541, (47): 2543, (48): 2545, (49): 2547, (50): 2549
```

Fig. 8. Large file extent tree.



Key

Abbreviation	Meaning
SBC	Super Block Copy
GDT	Group Descriptor Table
GDG	Group Descriptor Growth Blocks
FGBBM	Flex Group Block Bitmap
FGIBM	Flex Group Inode Bitmap
FGIT	Flex Group Inode Table
DB	Datablock

Fig. 9. Flex block group layout.

The extent index data structure should be thought of as an intermediary structure as it is used to point to a block containing other structures that will eventually point to data. Although these structures are also 12 bytes wide, only 10 bytes have been given a purpose. The 4 byte ei_block field is used to denote the starting file block the index references, while the leaf fields are used to point to the file system block that contains the normal extent structures. As the file system only supports 48 bits for addressing, only 16 bits are used to define the ei_leaf_hi field.

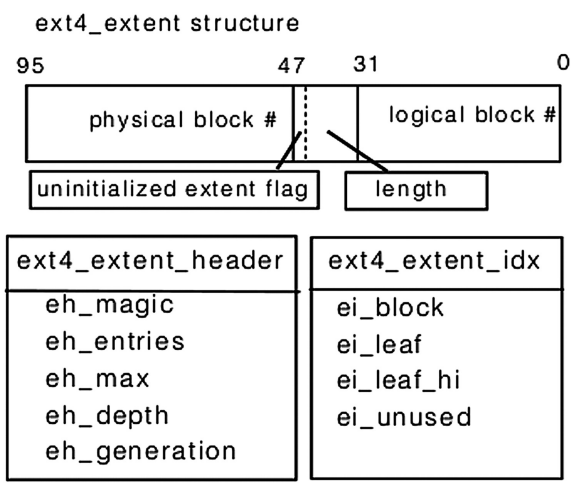


Fig. 10. Ext4 extent structure. Originally appeared in Mathur et al. (2007).

Valid Ext4 extent trees must adhere to certain rules. First, each node in a tree, including the extents in the inode structure, must include the previously described extent header. The in-inode structures are extent indexes known as roots when a tree is built. As show in Fig. 11, these roots point to index nodes which hold one or more extent indexes. Each index then points to a leaf node which contain file extents. At each level in this hierarchy, the depth field of the header is decreased. This results in the current maximum tree depth being stored in the inode structure. Also, the extents in a leaf node must have ee_block fields that are in increasing order and do not overlap when the extent length is taken into consideration. Similarly, the ei_block values in an index node with multiple index structures must increase. According to Mathur et al. (2007), the block coverage of the indexes can be confirmed by comparing the range of blocks addressed in the extent leaf block.

4.1.4. Inodes and time

As noted in Mathur et al. (2007) and Xia et al. (2008), Ext3 is limited to second resolution at the file system level. This is addressed in Ext4 by modifying and extending the inode structure. Although Ext3 supports different inode sizes that are powers of two greater than 128 bytes up to the size of a file system block, 128 bytes is the default size. The newer structures in Ext4 will be 256 bytes by default

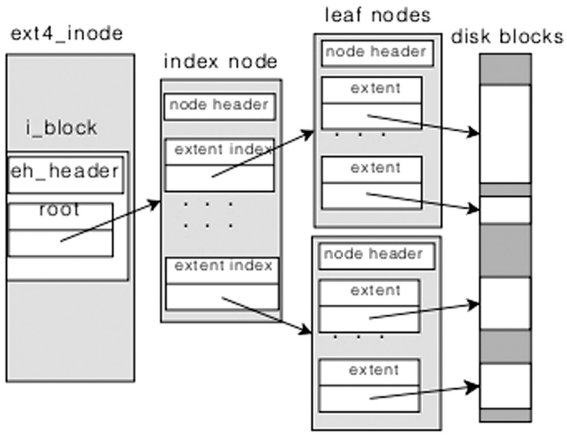


Fig. 11. Ext4 extent tree structure. Originally appeared in Mathur et al. (2007).

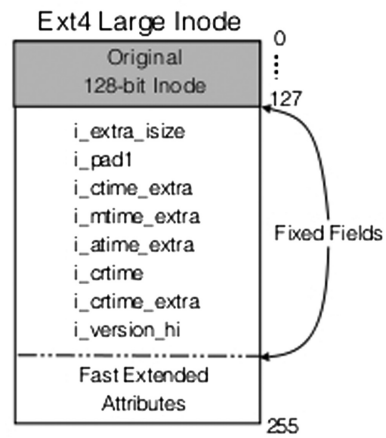


Fig. 12. Ext4 inode. Original appeared in Mathur et al. (2007).

(Mathur et al., 2007). This space allows Ext4 to support nanosecond timestamps and 64 bit inode version numbers. As shown in Fig. 12, the first 128 bytes remains largely the same as the new fields are added to the end of the structure. Also, a new timestamp field has been added to document the file creation time. Each timestamp field in the inode, with the exception of the deletion timestamp, has a corresponding high 32 bit field. In this space, only the high 30 bits are used for nanosecond representation. The 2 remaining bits are used as most significant bits of the second portion of timestamps, thereby delaying the 2038 problem for 272 years.[1] This increase in time resolution from the file system perspective may help future computer forensic and security research. The remaining space is consumed by fast extended attributes. In the first 128 bytes, the 1 byte fragment number, 1 byte fragment size, and a 2 byte padding field have been replaced by the i_blocks high field and file_acl_high field each 16 bits.

Although the default size of Ext4 inodes is 256 bytes, Ext4 file systems can be created with an inode size of 128 bytes. When this is the case the i_ctime_extra, i_mtime- _extra, i_atime_extra, i_crtime, i_crtime_extra, and i_version_hi fields will not be present. For the timestamp related fields, this results in the previously mentioned benefits of subsecond support, a 2 bit timestamp extension, and the creation timestamp being lost.

All of the data structure changes have affected journaling as the JBD (Section 2.2) has been branched into JBD2 which can support journaling of 32 bit and 64 bit file systems.

4.2. Scalability

Setting aside the physical topology of the file system, one of the most noticeable features that Ext4 introduces to users over Ext3 is its ability to support very large file systems. Ext3 is limited to a maximum of file system size of 16 TB, because there are only 32 bits available to address data blocks which have a 4 KB default size. This limit is exceeded by increasing the block number space to 48 bits. When combined with default block size, this yields a maximum file system size of 1 exabyte.[2] This change in the block number field has caused new fields to be introduced in the file system superblock and block group descriptor structures so that they can accommodate values up to 64 bits while maintaining backward compatibility with the pre-existing 32 bit structures.

Along with a drastic increase in maximum file system size, the maximum individual file size has been increased. While in Ext3 the maximum size of a file is 2 TB, an Ext4 file reaches its limit at 16 TB. This increase is achieved by adding a flag (HUGE_FILE) to the file system and in some of the inodes to denote that the files they represent are very large. When these flags are set, the 48 bit i_blocks field that is normally interpreted as the number of 512 byte sectors, is instead interpreted as the number of file system blocks. The original Ext4 inode structure only had a 32 bit i_blocks field which limited the file size to 2 TB ($2^{32} * 2^{9}$). With the HUGE_FILE flag set, the limit using the 32 bit field is raised to 16 TB ($2^{32} * 2^{12}$). However, the larger Ext4 inode i_blocks counter includes a 16 bit high field. Now, the limiting factor in file size is currently the format of the extent structure and not the i_blocks field (Mathur et al., 2007).

In the previous discussions of Figs. 1 and 9, one classification of blocks was not discussed. These are the blocks reserved for the growth of the group descriptor table. These blocks help to facilitate file system expansion as described in Cao et al. (2005). When the file system is being expanded, and a new block group must be added, a new group descriptor must be added accordingly. If there is no extra room at the end of the last group descriptor block, a new block can be allocated from the reserved area. This allows a file system to grow over time rather than keeping it at a fixed size as would be the case in an Ext2 or Ext3 formatted partition.

[1] January 19th, 2038 at 03:14:07 UTC is the latest time that can be represented by a signed 32 bit number.

[2] 1 exabyte = 1000 petabytes = 106 terabytes.

Another scaling problem with Ext3 is the 32,000 limit on the number of subdirectories which can be created. Ext4 has no subdirectory limit. When the link counter of a directory overflows its 16 bit space, the count is replaced with a 1 to denote that the directory is not empty. The reason huge directories can be supported is due to the new default way entries are stored. In ExtX, a linked list was used by directories, with the Linux Virtual File System (VFS) caching mechanisms helping ease the associated timing costs. Ext4 breaks with that tradition by using constant depth hash trees (HTrees). These HTrees decrease lookup times, thereby helping to improve performance in large directories.

In 2004, Eckstein stated that advanced file systems make use of BTrees, but the complex nature of BTrees and the size of implementations when compared to the rest of the ExtX base caused developers to seek out a more simplified solution that would also maintain compatibility with existing Ext2 partitions (Phillips, 2002). The HTree implementation takes in a given filename and a seed, usually from the file system superblock, and computes a 32 bit hash. As discussed in Section 2.2.1, the hashes are then used to point either directly to a block containing directory entries or to an index block. Although the current limit to depth of the hash tree is two levels and the number of directory entries that can be held in a leaf block vary based on filename lengths, the root index can accommodate 508 entries to directory indexes which each can hold up to 511 pointers to leaf blocks. Estimates for the amount of directory entries that can be stored in a single index vary, but all are in the tens of millions. More details about this design choice can be found in Cao et al. (2005), Mathur et al. (2007), and Phillips (2002).

4.3. Compatibility

One thing to note is that unlike the compatibility between Ext2 and Ext3, Ext4 is only partially compatible with the older file systems. Ext3 partitions can be mounted using Ext4. In this type of hybrid environment, older files will continue to use indirect block mapping while newly created files will use the larger inode structure and extents. It is possible to obtain some of the benefits of Ext4 by mounting an older partition without extent support. However, due to the fact that the on-disk structures used by the older file systems and the new one are different, Ext4 cannot be mounted by Ext3.

4.4. Block allocation

Along with changes to improve scalability, the way that blocks are allocated by the file system has been altered. Preallocation, one of the more interesting additions, is the process of reserving blocks for a file before they are actually needed. More importantly, when these blocks are allocated, they are not initialized with data or zeroed out. This is different from the block reservation system proposed in Mathur et al. (2007) for Ext3. In that scheme, blocks were allocated based on reservations that were stored in memory structures. This means that across reboots the preallocated range could be lost. In Ext4, this is solved through the use of extents. The most significant bit of the extent length field is used to denote a span of blocks that contains uninitialized data. When these spans are read from, they are interpreted as zero-filled blocks. This allows an expansive collection of contiguous blocks to be saved for a particular file while also saving time as large sparse files do not have to initialize every block they reserve. By examining the kernel source code, it can be found that there is a special case when the extent length field has a value of `0x8000`. Rather than interpreting this as a pre-allocated extent with zero length, it is interpreted as an extent with a length of 2^{15}. Therefore, the maximum length of an uninitialized extent is $2^{15} - 1$.

Another change that has sparked some debate is delayed allocation. The basic idea behind this feature is that instead of the file system blocks being used one at a time during a write, allocation requests are kept in memory. Blocks are then allocated when a page is flushed to disk. This saves time as what previously took multiple requests can be summarized with one. Other benefits include not having to allocate blocks for short lived files and a reduction in file fragmentation. In Kumar et al. (2008), it is stated that delayed allocation is handled at the VFS layer.

The Ext4 multiple block allocator is detailed in Kumar et al. (2008). It is designed to handle large and small files by using per-inode preallocation and per-CPU locality respectively. The strategy that is used for a particular file depends on a tunable parameter that specifies the number of blocks requested for allocation. The general idea is that smaller files using the same CPU will be placed physically close together and larger files will achieve less fragmentation through preallocation.

4.5. Reliability

The journaling used with Ext4 is a little different than what was previously described with Ext3. Mathur et al. (2007) describes a CRC32 checksum being added to the commit block of a transaction. The CRC is computed over all of the transaction blocks. During recovery, if the checksum does not match, then that transaction and all subsequent transactions are treated as invalid. The cost of computing a checksum for a transaction is offset by the fact that commit blocks can now be written to the journal with the rest of the transaction as opposed to the two-stage commit process previously used. This is possible because the checksum can detect blocks that are not written in the journal. In Mathur et al. (2007), it is stated that the end result is the file system speed is actually increased by up to 20%.

Group descriptors are made more robust with the use of a CRC16 checksum. This is used to verify that the unused inode count field in the descriptor is valid when a file system check, done by `e2fsck`, occurs. This unused field along with new flags that denote whether the bitmaps and inode table are uninitialized let `e2fsck` skip areas that have not been used thereby reducing the time it takes for the file system to be examined. Furthermore, there exists a `lazy_bg` option for the `mke2fs` command which saves time when creating large file systems by not writing the inode tables at the creation of the file system.

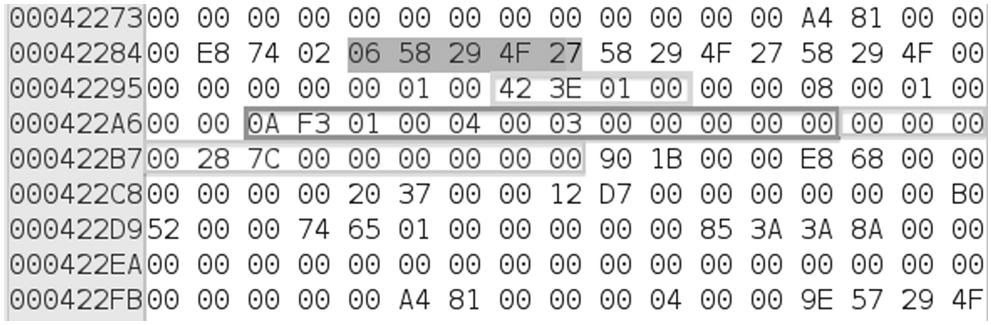


Fig. 13. Inode with extent index before deletion.

Fragmentation is one of the banes to performance in many file systems. As the file system ages and free space is depleted, the risk of fragmentation increases. This is especially true for very large files. In Sato (2007), several techniques are discussed which combat this trend while the file system is still mounted. The overall goal is to reduce single file, relevant file, and free space fragmentation. Generally, this is done by either moving fragmented data to contiguous free space or making uninterrupted space for larger files by moving the data belonging to smaller files to another area. Online defragmentation patches are available, but support has not been merged into the main line kernel yet (Bugs, 2009).

As Ext4 is fairly recent, some features may continue to be modified and added, however, the on-disk data structures have become stable and should remain the same. This file system is more greatly detailed in Kumar et al. (2008), Mathur et al. (2007), Sato (2007), and Tso and Tweedie (2002).

5. Forensic implications

In 2004, Eckstein classified file systems either as traditional or advanced based on the features that the given file system supported. At the time of the writing, Ext3 was classified as a traditional file system because the only feature it shared with more advanced file systems such as JFS was the journaling capability. If that is ignored, Ext3 and Ext2 could be analyzed in the same manner. Ext4 differs in that it implements extents, uses tree structures, and has the ability for limited expansion. While it may not be in the same league as file systems which make use of binary trees and/or use special files to store file system and file metadata, it is clearly more advanced than Ext3. This section discusses how the Ext4 data structure changes affect digital forensics when compared to Ext3.

5.1. Deleted files

In Fairbanks et al. (2010), it is shown that the inode resident extents of a file are zeroed out upon deletion of that file. However, the experiment did not show how the extent tree was affected. Fig. 13, displays the inode structure of a file that makes use of an extent tree. The inode structure starts at `0x00042280`. Highlighted are several fields that can be used to verify this structure matches the `debugfs` output displayed in Fig. 8. The fields highlighted are: the atime, i_blocks_lo, extent header, and extent index respectively. By examining the extent header, `0x000422A8`, it can be determined that there is only one valid entry resident in the inode and that the tree has a depth of 3. The extent index structure, `0x000422B4`, reveals that the next node is located at `0x007C28`. Fig. 14 displays this same inode after the file has been deleted. It can be seen at `0x00042284` that the size of the file has been set to zero. Also, the change, modification, and deletion time fields have all been set to the same value confirming deletion. A major point of interest in the comparison between Figs. 13 and 14 is that although the extent header in the latter has been reset to show that there are no valid extents present, the following string of bytes clearly contains the past index extent.

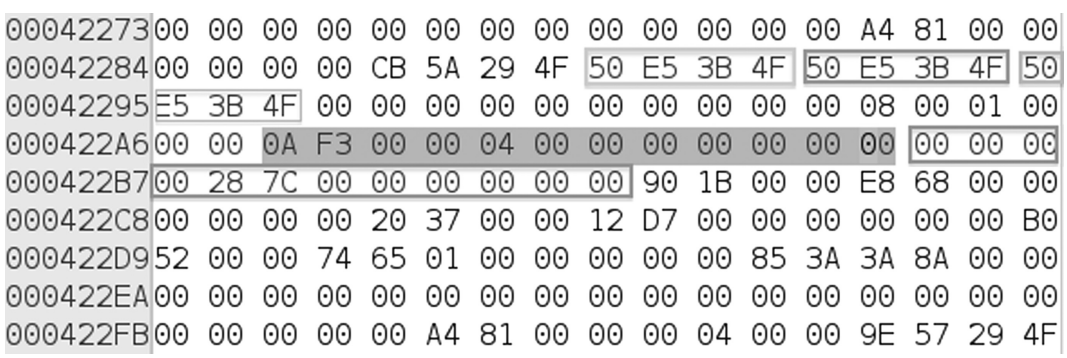


Fig. 14. Inode with extent index after deletion.

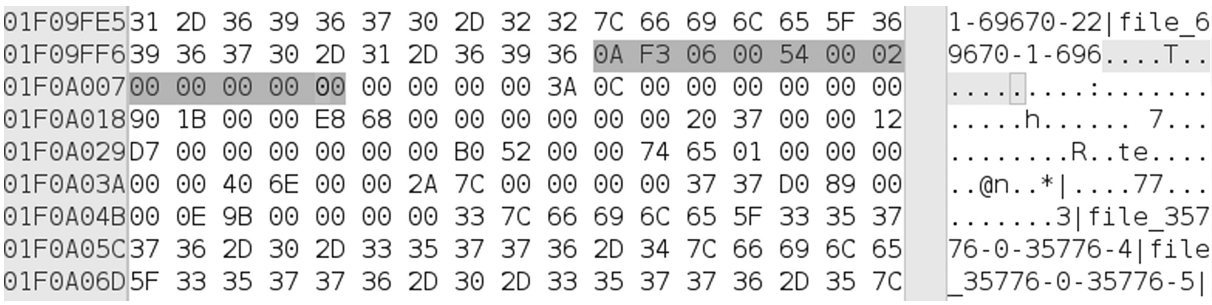


Fig. 15. Extent index node before deletion.

Fig. 15 shows a partial capture of the index node to which the index extent points. The extent header is highlighted to show that the depth has decreased by one and that the node is fully populated with 84 (`0x54`) entries. The index node is not zeroed upon file deletion. If the data blocks that have been designated as index blocks are not reused, it is possible to follow the extent tree all of the way to the actual file data. This is not shown for the sake of brevity.

The result of this first test falls in line with Eckstein (2004) and Pomeranz (2010), as the extent entries were not cleared when trees were constructed. However, this does not refute the results of Fairbanks et al. (2010). When examining one of the inode tables of the test image, it was noted that inodes which were once used by the block sized files had their extents set to zero. It would therefore seem that in Ext4 the determining factor of whether the extents in an inode will be cleared will be the size and/or fragmentation of the file in question. This was confirmed by creating files that incrementally increased in size from one to five file system blocks. The multiple block files were fragmented as they used the blocks freed by the large fragmented file. Examining the inodes of the first four files before and after deletion yielded results similar to Fairbanks et al. (2010); however, it appears that the logical starting block number of the extent structure remains after deletion. When the fifth file that used a small extent tree was examined, no zeroing was detected. If the inode gets recycled, the difference between these situations is moot as the extent index nodes will yield the best opportunity for data recovery. The results of this experiment are shown in Figs. 16 and 17.

5.2. Metadata

Unlike JFS metadata, described in Eckstein (2004), Ext4 has fairly static file system metadata structures. Inodes are stored in tables similar to ExtX, not in a special file. The placement of the structures like inode tables, bitmaps, and group descriptors are defined when a flex group is established. However, the file system is capable of limited growth because of the use of group descriptor growth blocks, as described in Section 4.2.

Although the inode remains the primary repository for file metadata, metadata can be mixed in with normal data blocks, just as in ExtX, when extent trees are constructed. However, the usage of extent headers can work to the benefit of data recovery efforts if metadata carving is attempted as suggested in Eckstein (2004). When attempting to classify extent index nodes, not only can the extent header be used to identify index blocks, it can identify where in the extent tree hierarchy the block should be placed.

5.3. Data

File and directory data continue to be stored in data blocks as in ExtX; however, there are some issues that should be considered when searching for data. As stated in Fairbanks et al. (2010), preallocation can cause data to be captured in large files of the applications which make use of it. It is therefore necessary to check uninitialized extents for data remnants. Also, by examining the figures from the HTree discussion in Section 2.2.1 one can see data remnants in the HTree nodes. Although the remnants in this paper

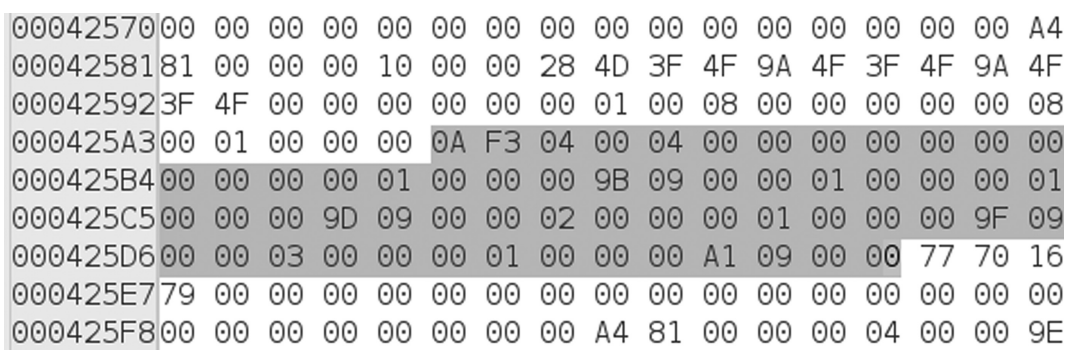


Fig. 16. Embedded inode extents before deletion.

```
00042570 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A4
00042581 81 00 00 00 00 00 00 28 4D 3F 4F D9 57 3F 4F D9 57
00042592 3F 4F D9 57 3F 4F 00 00 00 00 00 00 00 00 00 00 08
000425A3 00 01 00 00 00 0A F3 00 00 04 00 00 00 00 00 00 00
000425B4 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00
000425C5 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00
000425D6 00 00 03 00 00 00 00 00 00 00 00 00 00 00 77 70 16
000425E7 79 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000425F8 00 00 00 00 00 00 00 00 A4 81 00 00 00 04 00 00 9E
```

Fig. 17. Embedded inode extents after deletion.

```
00000FE4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000FF6 00 00 00 00 00 00 00 00 0A F3 25 00 54 00 00 00 36 30
00001008 2D 30 D8 9C 00 00 01 00 00 00 3A 5F 01 00 D9 9C 00 00
0000101A 01 00 00 00 3E 5F 01 00 DA 9C 00 00 01 00 00 00 40 5F
0000102C 01 00 DB 9C 00 00 01 00 00 00 42 5F 01 00 DC 9C 00 00
0000103E 01 00 00 00 44 5F 01 00 DD 9C 00 00 01 00 00 00 46 5F
```

Fig. 18. Journaled extent index block.

contain directory data, there is no reason to believe that other data cannot be obtained from an HTree node. Moreover, if HTree nodes are sparsely populated with entries, the extra space may be theoretically used to hide data without interrupting the operation of the file system. Other potential data hiding places include group descriptor growth blocks and data structures in uninitialized block groups. Each of these potential hiding places will present a different level of volatility which will need to be studied further.

5.4. Journal

In Carrier (2005), Eckstein (2004), Fairbanks et al. (2007), and Swenson et al. (2007), the file system journal is examined as a source of previous data or metadata. Since ordered is the default mode of journaling, metadata such as inodes, bitmaps, and directory entries are expected to be seen in the journal. However, as shown in Fig. 18, extent index nodes can also be found in journal data. This could be a valuable, although limited, resource as the original extent index block and inode may have already been reused.

6. Related work

In 2010, Fairbanks describes the flex block group layout of Ext4 and shows that upon file deletion, the extents resident in the inode are erased. He draws attention to the fact that data can be recovered if a preallocated extent covers previously deleted data, thereby making the argument that files produced by applications which may use preallocation, such as virtual machines or databases, should be examined for data remnants. However, Fairbanks does not explain extent trees or mention what occurs to the tree after the deletion of a file. Also the structure of directory indexes are not discussed in his work.

In a series of blog posts, starting in late December 2010, Pomeranz examines Ext4 data structures (Pomeranz, 2010; Pomeranz, 2011). His experimentation primarily focuses on Ext4 extents and extent trees, which he examines through the use of a hex editor. While he does cover the file deletion and timestamps, he does not mention the changes to the layout of block groups, directory indexing, or the Ext4 journal.

Although he does not provide an Ext4 guide, Ballenthin has made available a set of patches to the Sleuth Kit that support some of the file system features from his website (Ballenthin, 2011). To date, the patches cover the following features: extents, Ext4 superblock, and the updated inode structure. The following features are marked as to do: 64 bit support for multiple data structures, huge files, multiple mount protection, meta-block groups, uninitialized block groups, flex block groups, and journal checksumming. Rather than ignoring these patches and attempting to extend the Sleuth Kit from scratch, we have applied and studied these patches.

The following forensic tools state support of Ext4: Digital Forensics Framework, Encase v7, FTK v4, extcarve.

7. Conclusion & future work

This paper provides a more comprehensive study of the organization and behavior of Ext4 than has hitherto been presented by providing a low-level study of many

important on-disk data structures. Background information on the previous incarnations of the file system was given so that comparisons can be made on the differences present in Ext4. Some of the features of importance to the forensic community include increased timestamp resolution, persistent preallocation, block group layout. These were discussed in detail to show how the topology of the file system has changed with usage of flexible block groups and how data structures were changed to accommodate a larger block address space.

It has been noted and explained how Ext4 can scale to support larger files (16 TB) and more data (1 EB) overall than ExtX file systems (2 TB and 16 TB respectively). Ext4 also does away with the limit on files a directory can contain through the use of constant depth HTrees and overflowing the link counter field as opposed to the use of linked lists in ExtX. Further, it is explained that the Ext4 journaling mechanism, JBD2, differs from Ext3 in that it supports 32 bit and 64 bit file systems and makes use of CRC32 checksums so that transactions can be written in a 1 step process.

The effect deletion has on data structures such as inodes and extent trees had been examined and reveals that inode zeroing is dependent on both file size and/or fragmentation for extent tree creation. While embedded extent zeroing depends upon extent tree creation, it is demonstrated that extent index nodes are not zeroed and can potentially be recovered from the journal of the file system. Armed with this knowledge, the patches made available in Ballenthin (2011) are currently being studied and expanded to extend the Sleuth Kit to natively support Ext4. While this paper has identified potential places for data hiding, this has not been explored in detail and should be in future work. Specifically, the use of CRCs in group descriptors and the journal commit block should be studied to determine the effect they have on the data hiding techniques presented in Eckstein and Jahnke (2005).

Acknowledgments

The author would like to thank the anonymous reviewers for their input and Simson Garfinkel of the Naval Postgraduate School and by extension the Digital Evaluation and Exploitation (DEEP) group for helping to make this publication possible. The author also expresses gratitude to Benjamin Salazar of JHU/APL for helping revise numerous copies of this paper.

References

Ballenthin W. TSK & Ext4, http://www.williballenthin.com/ext4/index.html; 2011 [last accessed 02.07.12].

Beebe NL, Stacy SD, Stuckey D. Digital forensic implications of zfs. Digital Investigation 2009;6(Suppl.):S99–107. The Proceedings of the Ninth Annual DFRWS Conference.

Bovet D, Cesati M. Understanding the Linux Kernel. 3rd ed. Sebastopol, CA: O'Reilly Media, Inc.; 2005.

Bugs Ubuntu. ext4 defrag/defragment tool in jaunty-include, https://bugs.launchpad.net/ubuntu/+source/e2fsprogs/+bug/321528; 2009 [last accessed 11.18.09].

Cao M, Tso TY, Pulavarty B, Bhattacharya S, Dilger A, Tomas A. State of the art: where we are with the ext3 filesystem. In: Proceedings of the Ottawa Linux Symposium (OLS); 2005. p. 69–96.

Carrier B. An investigator's guide to file system internals. In: FIRST conference on computer security, incident handling & response; 2002.

Carrier B. File system forensic analysis. Upper Saddle River, NJ: Pearson Education, Inc.; 2005.

Eckstein K. Forensics for advanced unix file systems. In: IEEE/USMA information assurance workshop; 2004. p. 377–85.

Eckstein K, Jahnke M. Data hiding in journaling file systems. In: Digital forensic research workshop (DFRWS); 2005. p. 1–8.

Fairbanks KD, Lee CP, Xia YH, Owen HL. Timekeeper: a metadata archiving method for honeypot forensics. In: Information assurance and security workshop IAW '07, IEEE SMC; 2007. p. 114–8.

Fairbanks KD, Lee CP, Owen III HL. Forensic implications of ext4. In: Proceedings of the sixth annual workshop on cyber security and information intelligence research, CSIIRW '10. New York, NY, USA: ACM; 2010. p. 22:1–22:4.

Kumar A, Cao M, Santos J, Dilger A. Ext4 block and inode allocator improvements. In: Proceedings of the Linux Symposium, vol. 1; 2008. p. 263–74.

Mathur A, Cao M, Battacharya S, Dilger A, Tomas A, Vivier L. The new ext4 filesystem: current status and future plans. In: Proceedings of the Linux Symposium, vol. 2; 2007. p. 21–33.

Phillips D. A directory index for ext2. In: Proceedings of the Ottawa Linux Symposium; 2002.

Pomeranz H. Understanding Ext4, http://computer-forensics.sans.org/blog/author/halpomeranz; 2010 [last accessed 02.07.12].

Pomeranz H. Ext4: bit by bit (slides). In: The computer enterprise and investigations conference (CEIC), www.deer-run.com/hal/CEIC-EXT4-Bit-By-Bit.pdf; 2011.

Sato T. ext4 online defragmentation. In: Proceedings of the Linux Symposium, vol. 2; 2007. p. 179–86.

Swenson C, Phillips R, Shenoi S. File system journal forensics. In: Advances in digital forensics III. IFIP international federation for information processing, vol. 242. Boston: Springer; 2007. p. 231–44.

Tso T. RE: [RFC] Btrfs mainline plans, https://lkml.org/lkml/2008/10/9/447; 2008 [last accessed 02.14.12].

Tso T. Android will be using ext4 starting with gingerbread, http://thunk.org/tytso/blog/2010/12/12/android-will-be-using-ext4-starting-with-gingerbread/; 2010 [last accessed 02.20.12].

Tso T, Tweedie S. Planned extensions to the linux ext2/3 filesystem. In: USENIX technical conference; 2002.

Xia Y, Fairbanks K, Owen H. A program behavior matching architecture for probabilistic file system forensics. SIGOPS Operating Systems Review 2008;42:4–13.

Kevin D. Fairbanks is currently a Cyber Security Research Engineer in the Information Operations Group of the Asymmetric Operations Department at the Johns Hopkins University Applied Physics Laboratory. His research interests include digital forensics, host-based computer security, and mobile device security. He has also taught courses on the subjects of computer networks, computer security, as well as analog and digital electronics. He earned a Ph.D. and Master's from the Georgia Institute of Technology in 2010 and 2007 respectively. His undergraduate degree in Electrical Engineering with a Concentration in Computers was obtained from Tennessee State University in 2005.