Mobile Forensics
Lecture 2

Ext4 file system – Android

Introduction

- In-depth knowledge about Ext4 file system: used by Android cell phones and by Linux distributions
- Understand the structure of this file system to recover data
- Verify tool results
- Detect anti-forensics techniques
- Ext4 replaced Ext2 and Ext3

Requirements

- know how to use a hex editor
- How to interpret multi-byte fields in a structure
- How to read raw data based on the used endianness

Ext4 - File system category

- Ext4 contains multiple block groups

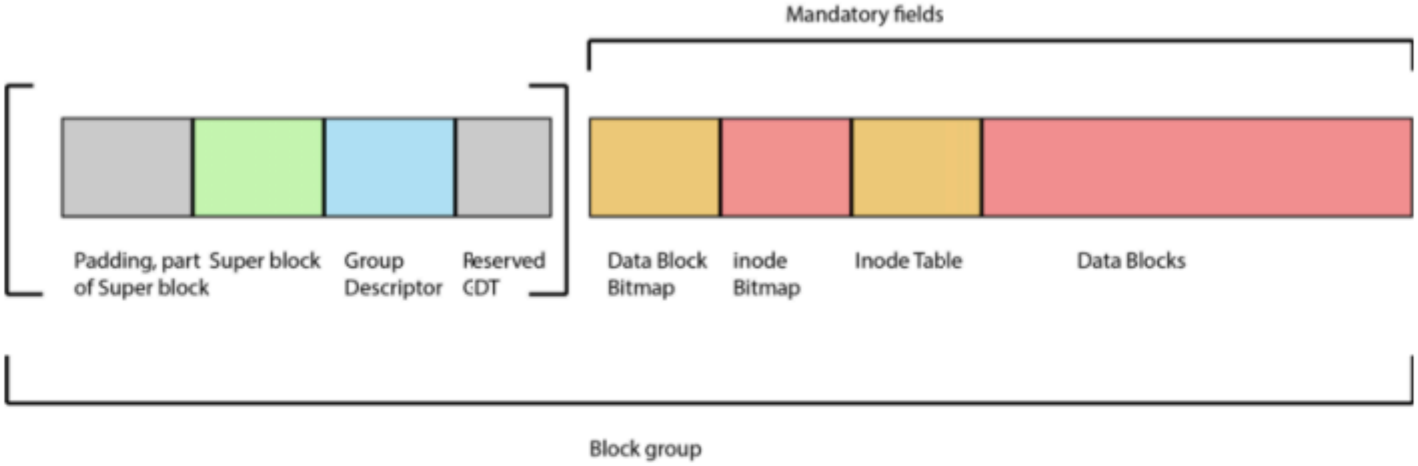


Fig. 2.1: Illustration of the elements of a block group.

Block Group

- The first part of the block group is 1024 bytes reserved that can be used for boot code and form part of the first superblock
- It is not a requirement that each block group has a superblock
- Store superblocks and group descriptors in block group 0, and then in block group 3^x , 5^x , and 7^x
- Any superblock can be used to recover the file system
- It has location addresses, statistics, and checksums about other mandatory elements in the block group

Forensic Tip

- File system feature flags
    - Document the supported features of the file system under investigation. Do not assume all Digital Forensic tools support all features.

Superblock

- File system uses superblocks: to describe important structures of the file system:

  – the number of total inodes and blocks

  – how many inodes and blocks that are free

  – the size of inodes and blocks

  –information about file system checks, which OS the file system was created on

  – features the file system supports

  –a unique UUID (Universally Unique Identifier) for the file system volume

Temporary data about the File system

- superblock contains its temporary information:
  - when it was created
  - last mounted
  - last written to
  - first time an error found place
  - the time of the last error

- All timestamps found in the superblock:
  - are described as seconds since 1970
  - are defined as 32-bit fields that must be interpreted as little-endian

Forensics Tip

- From a mobile forensic expert view, it will be important to know when the file system was created since we can expect to find user allocated files created between the file system creation time and before the file system last written time

Why these temporary data are useful for forensics?

- If find the allocated files outside this time range, then this can be explained by one or more theories (hypotheses):

    - Files that are part of the OS system installation process may keep one or more of their original timestamps.
    - Apps may keep the timestamps when extracting container files, depending on how they extract the files.
    - The cell-phone has lost its date/time due to power failure.
    - The user could have reset the cell-phone date/time manually.
    - Someone has manipulated the timestamps using a tool.

The importance of Time stamps

- It is also possible to find previous files from before the file system was created, unallocated, from a previous file system.
- All these different theories about the reasoning for why we can find timestamps out of range is not complete and which theory(hypothesis) is the most likely should be tested.

Forensics Tip

- Use Experiments
  - Scientifically testing theories (hypotheses) is part of Digital Forensics.
  - When it comes to file systems this can be done by performing experiments.
  - Do not base your investigation on assumptions!

Exampl

	Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	UTF-8
0000	000D780400	E0	2C	04	00	50	AF	10	00	00	00	00	00	DA	39	00	00	P
0010	000D780410	CC	0F	04	00	00	00	00	00	02	00	00	00	02	00	00	00	
0020	000D780420	00	80	00	00	00	80	00	00	70	1F	00	00	E7	0E	BD	5E	p ^
0030	000D780430	E7	0E	BD	5E	18	00	1B	00	53	EF	01	00	03	00	00	00	^ S
0040	000D780440	F0	88	5B	49	00	4E	ED	00	00	00	00	00	01	00	00	00	I N
0050	000D780450	00	00	00	00	0B	00	00	00	00	01	00	00	2C	00	00	00	,
0060	000D780460	42	02	00	00	7B	00	00	00	F1	CD	2A	39	FB	43	5B	2C	B { 9 C[,
0070	000D780470	98	32	59	3F	38	42	51	C0	73	79	73	74	65	6D	00	00	2Y?8BQ system
0080	000D780480	00	00	00	00	00	00	00	00	2F	73	79	73	74	65	6D	00	/system
0090	000D780490	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00A0	000D7804A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00B0	000D7804B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00C0	000D7804C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00D0	000D7804D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00E0	000D7804E0	08	00	00	00	00	00	00	00	00	00	00	00	AD	D9	4F	AB	
00F0	000D7804F0	0D	DE	53	3D	9B	36	DC	F3	00	C3	02	E3	01	01	00	00	= 6
0100	000D780500	4C	04	00	00	00	00	00	00	F0	88	5B	49	0A	F3	01	00	L I
0110	000D780510	04	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	@
0120	000D780520	00	80	07	00	00	00	00	00	00	00	00	00	00	00	00	00	
0130	000D780530	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0140	000D780540	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	04	
0150	000D780550	00	00	00	00	00	00	00	00	00	00	00	00	20	00	20	00	
0160	000D780560	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0170	000D780570	00	00	00	00	04	00	00	00	49	17	09	00	00	00	00	00	I
0180	000D780580	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0190	000D780590	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01A0	000D7805A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01B0	000D7805B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01C0	000D7805C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01D0	000D7805D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01E0	000D7805E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
01F0	000D7805F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Mount time (s_mtime) 2020-05-14 09:27:03

Write time (s_wtime) 2020-05-14 09:27:03

Time of last check (s_lastcheck) 2008-12-31 15:00:00

Creation time (s_mkfs_time) 2008-12-31 15:00:00

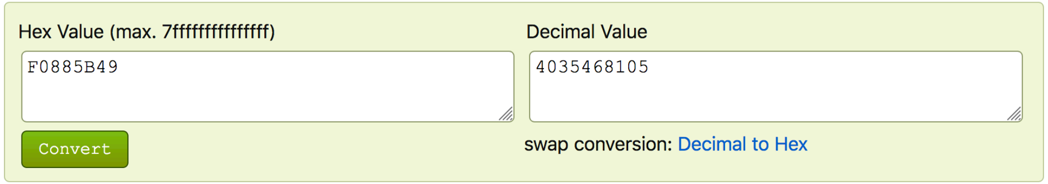
First time for error (s_first_error_time) N/A

Time of most recent error (s_last_error_time) N/A

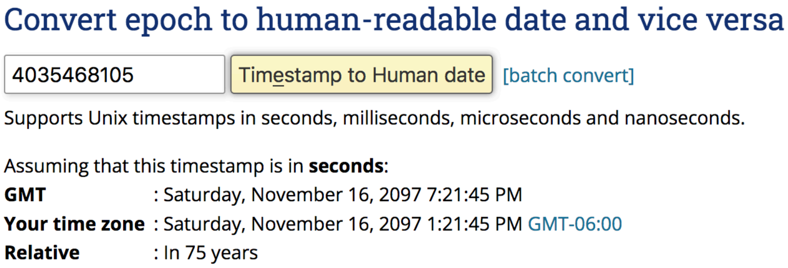
Fig. 2.2: Timestamps found in the Ext4 superblock.

Hex Date to Human Date

- First convert hex to timestamp
  - https://www.binaryhexconverter.com/hex-to-decimal-converter



- Then convert timestamp to human date
  - https://www.epochconverter.com/



Hex Date to Human Date

Or convert hex to human date directly

- https://www.epochconverter.com/

E70EBD5E | Convert hex timestamp to human date

GMT: Sunday, November 2, 2092 10:21:18 PM

Your time zone: Sunday, November 2, 2092 4:21:18 PM GMT-06:00

Decimal timestamp/epoch: 3876502878

Supported features

The superblock defines the features supported in three different 32 bit fields;

- 0x5C s_feature_compat
- 0x60 s_feature_incompat
- 0x64 s_feature_ro_compat

If the feature that is unrecognisable found in:

S_feature_compat : mounting the file system should be ok (read/write)

S_feature_incompat : the file system should not be mounted

S_feature_ro_compat : the file system should be mounted as read only

Feature flags in the Ext4 superblock

- supported by the file system driver version that created the file system

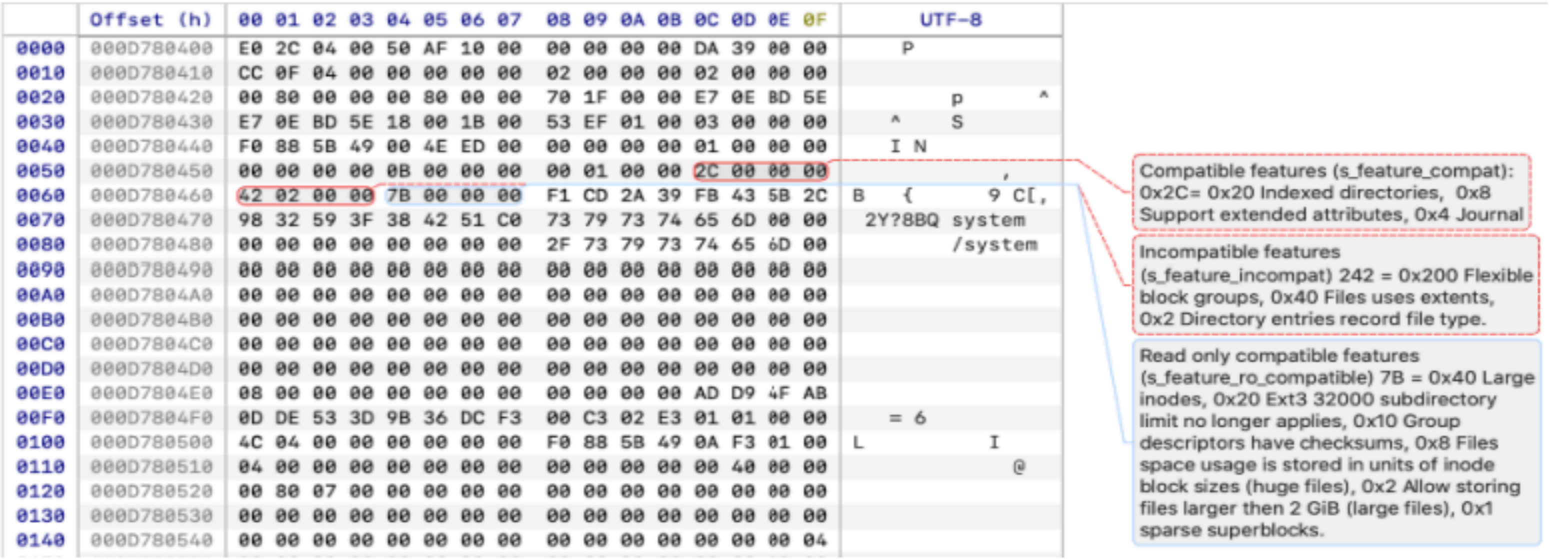


Fig. 2.3: Feature flags in the Ext4 superblock.

Tip

- Features have an impact
- The features described in the superblock impacts how the inodes and directory entries should be interpreted.

Compatible features

Table 2.1: Compatible features

Value	Description
0x1	Directory preallocation (COMPAT_DIR_PREALLOC)
0x2	Could mean the fs supports AFS magic directories. (COMPAT_IMAGIC_INODES)
0x4	Has a journal (COMPAT_HAS_JOURNAL)
0x8	Supports extended attributes (COMPAT_EXT_ATTR)
0x10	Has reserved GDT blocks for filesystem expansion (COMPAT_RESIZE_INODE)
0x20	Has directory indexes (COMPAT_DIR_INDEX)
0x40	"Lazy BG". Not in Linux kernel (COMPAT_LAZY_BG)
0x80	"Exclude inode". Not used. (COMPAT_EXCLUDE_INODE)
0x100	"Exclude bitmap". Not used (COMPAT_EXCLUDE_BITMAP)
0x200	Sparse Super Block, v2 (COMPAT_SPARSE_SUPER2)

Compatible Features

- not every compatible feature is relevant
- the flag COMPAT_SPARSE_SUPER2 is especially important when locating the backup superblocks
    - If the main one is partly corrupted or manipulated, the second block can be used
    - If the COMPAT_SPARSE_SUPER2 flag is set, the super block fields_backup_bgs, found from superblock byte offset 0x24C, points to the two block groups that contain backup superblocks
- It may seem strange that one field can point to two blocks, but this is because the field is an array of two 32 bits elements.

Compatible Features

- Another important flag is the COMPAT_HAS_JOURNAL.
- If this flag is set, recovery of data from the journal should be possible.
- when the journal is full, it will start writing transactions from the beginning of the journal file, effectively overwriting previous transactions.

Compatible Features

- Some of the flags are supported by Linux, but not necessarily used.
- For instance, the COMPAT_DIR_PREALLOC allows for pre-allocating a specific number of blocks to directories, defined in fields_prealloc_dir_blocksat superblock byteoffset 0xCD.
- This field is currently not used by the Linux kernel

Compatible Features

- The flag COMPAT_RESIZE_INODE does not have a descriptive name, since it describes the number of blocks reserved for the extra Group Descriptor Table (GDT).
- Some of the flags are supported by Linux, but not necessarily used.
- Ex:COMPAT_DIR_PREALLOC
  - allows for pre-allocating a specific number of blocks to directories,
  - defined in field s_prealloc_dir_blocks at superblock byte offset 0xCD.
  - The field is currently not used by the Linux kernel

Incompatible features

- If the kernel does not understand one of the flags in this 32 bit field, it should not mount or repair the file system.
- Table 2.2 can be used to interpret this field
- Figure 2.3 demonstrates an example of interpretation

Table 2.2: Incompatible features

Value	Description
0x1	Compression. Not implemented. (INCOMPAT_COMPRESSION)
0x2	Directory entries record the file type (INCOMPAT_FILETYPE)
0x4	Filesystem needs journal recovery. (INCOMPAT_RECOVER)
0x8	Filesystem has a separate journal device. (INCOMPAT_JOURNAL_DEV)
0x10	Meta block groups. See the earlier discussion of this feature. (INCOMPAT_META_BG)
0x40	Files in this filesystem use extents. (INCOMPAT_EXTENTS)
0x80	Enable a filesystem size over 2^{32} blocks. (INCOMPAT_64BIT)
0x100	Multiple mount protection. Prevent multiple hosts from mounting the filesystem concurrently (INCOMPAT_MMP)
0x200	Flexible block groups (INCOMPAT_FLEX_BG)
0x400	Inodes can be used to store large extended attribute values (INCOMPAT_EA_INODE)
0x1000	Data in directory entry. Feature still in development (INCOMPAT_DIRDATA)
0x2000	Metadata checksum seed is stored in the superblock (INCOMPAT_CSUM_SEED)
0x4000	Large directory >2GB or 3-level htree (INCOMPAT_LARGEDIR)
0x8000	Data in inode. Small files or directories are stored directly in the inode i_blocks and/or xattr space. (INCOMPAT_INLINE_DATA)
0x10000	Encrypted inodes are present on the filesystem (INCOMPAT_ENCRYPT)

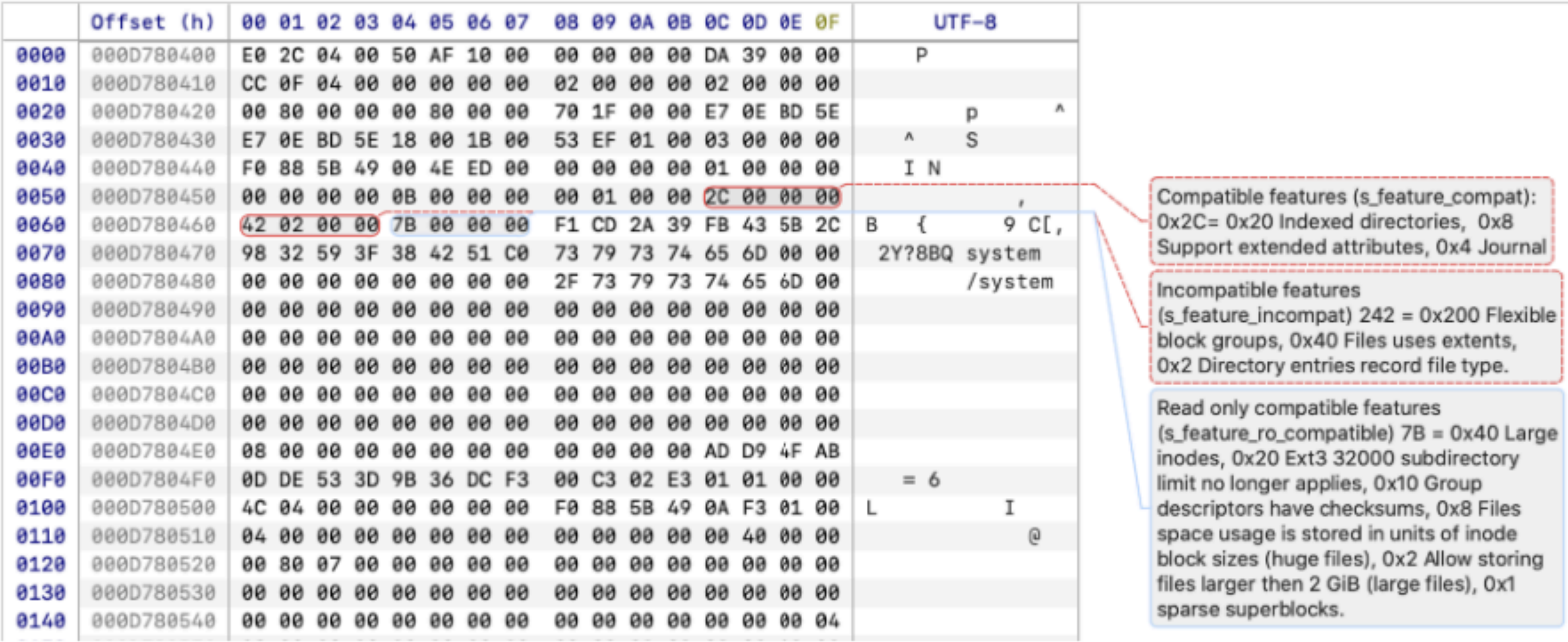


Fig. 2.3: Feature flags in the Ext4 superblock.

Flexible Block Groups

- Flexible block groups are a unique way of organizing block groups into a set of flex groups

- The first block of a flex group (Fig. 2.4) will include:
  - the bitmaps and the inode table for all groups within all the flex groups
  - the other groups may contain super blocks
  - group descriptors depending on the sparse superblock feature
  - will include data blocks

- The group descriptor is used to define where the bitmaps and inode table should be located, which enables flex groups, meaning they all point to the same bitmaps and inode locations as the first group descriptor.

Elements of a Flex Group

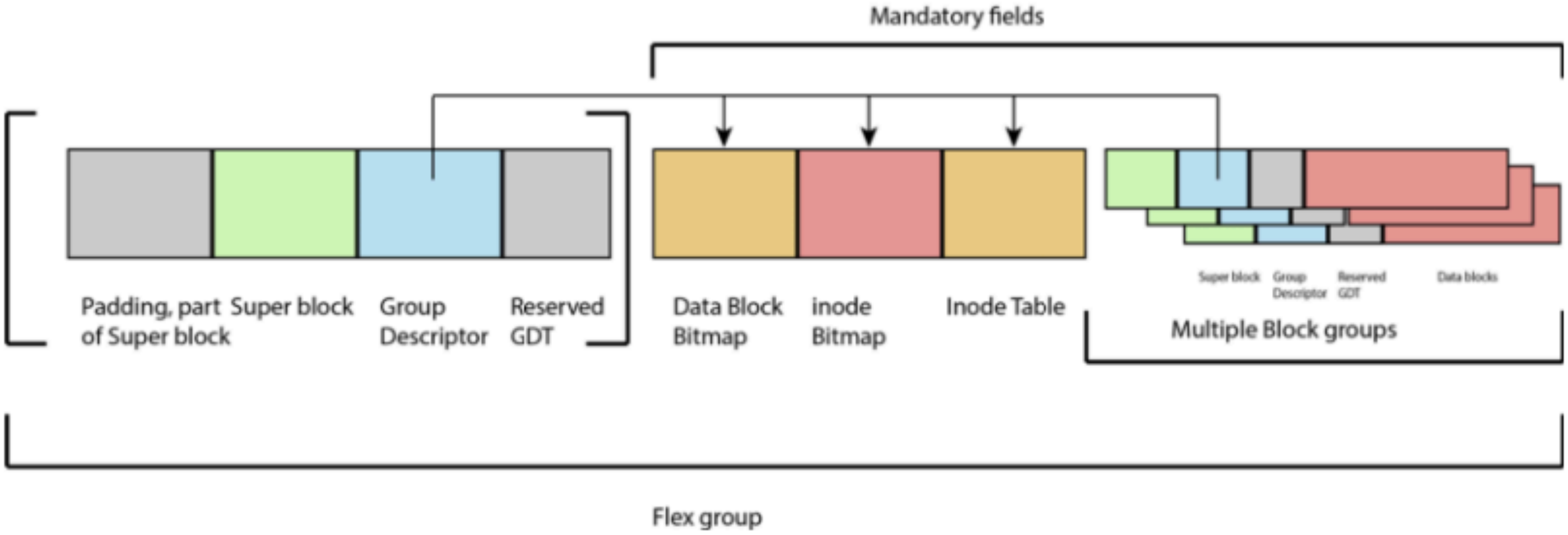


Fig. 2.4: Illustration of the elements of a flex group.

Metadata location

- The investigator should assume and test:
    - the data block bitmap
    - node bitmap
    - inode table

- The metadata is near co-located in the beginning of the file system
- When flex groups are not used, it will be necessary to parse all superblocks and group descriptors in order to identify all bitmaps and the complete inode table.

Block Group and Flex Groups

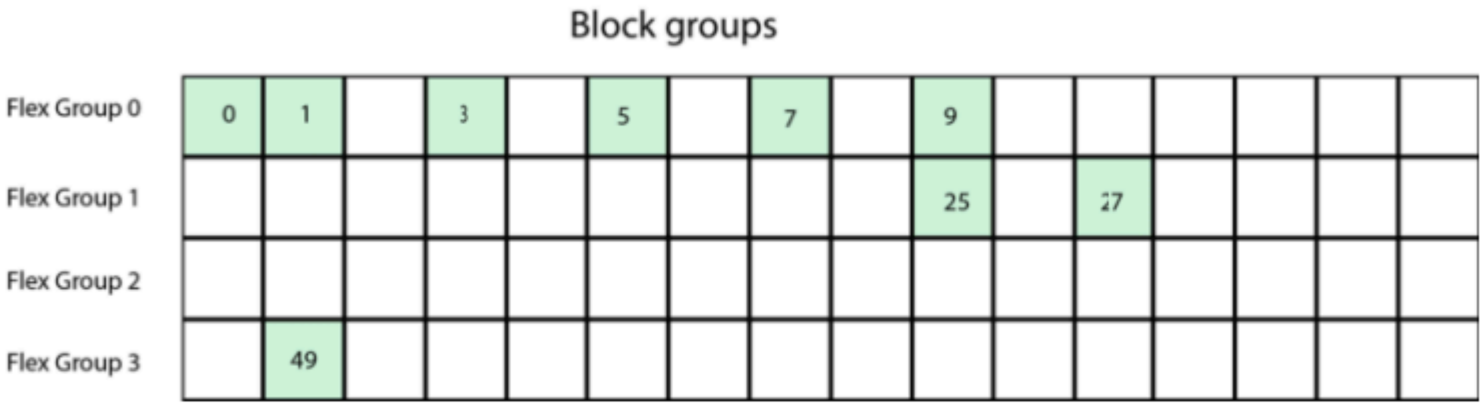


Fig. 2.5: Illustration of superblocks and group descriptors in flex groups or not flex groups when also the RO_COMPAT_SPARSE_SUPER is in use. Based on [5].

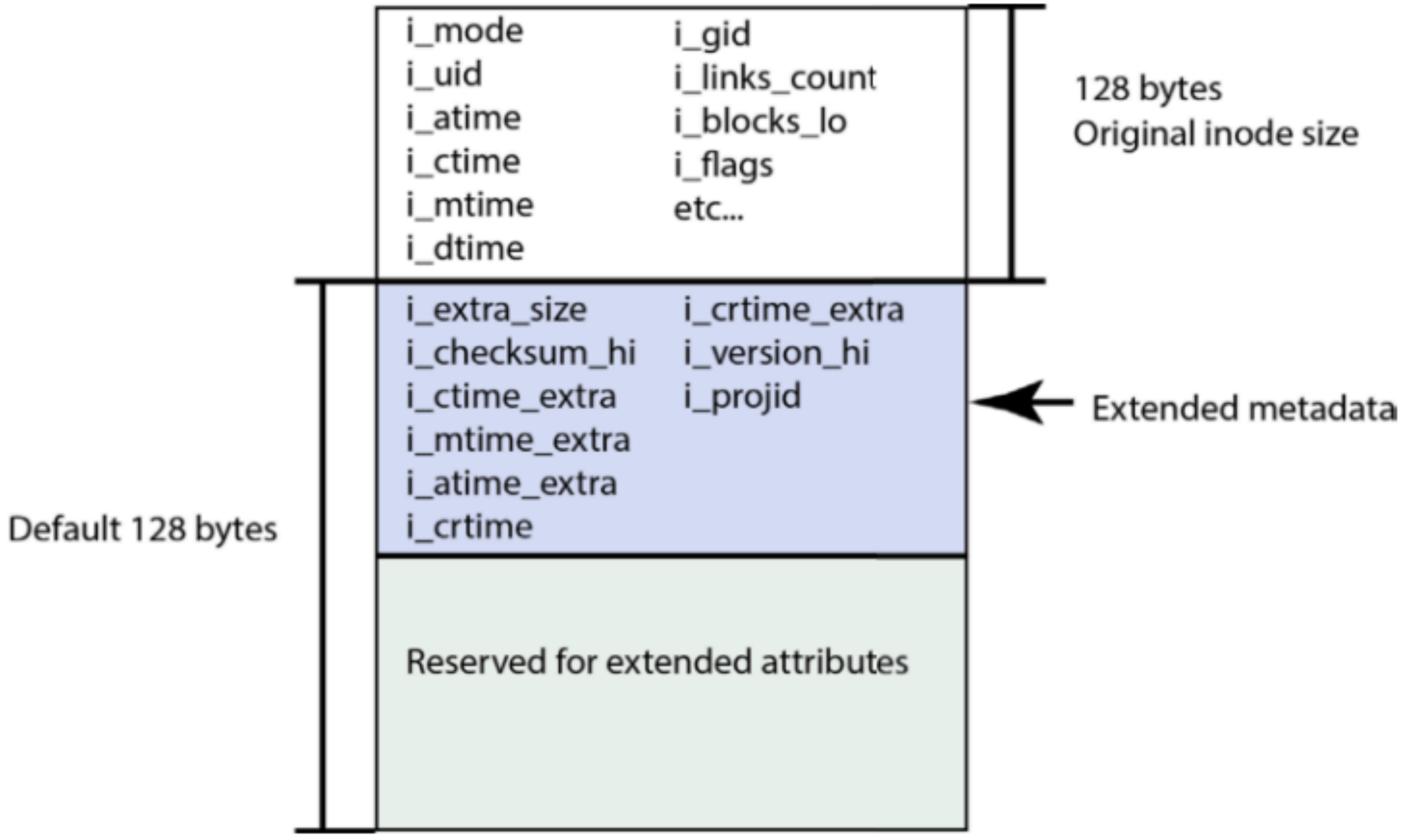


Fig. 2.6: Illustration of usage of extended metadata in inodes.

Metadata

- The default inode size was 128 bytes in Ext2 and Ext3, but in Ext4 it was typically 256 bytes

- The option RO_COMPAT_EXTRA_ISIZE means that extended metadata is utilised

- The part after the extra inode size is still reserved for extended attributes

Metadata

- Figure 2.6 demonstrates that the extra metadata is found directly after the first 128 bytes of an inode

- The extended attributes follow after the extended metadata.

- If extended metadata is not in use, then this area will be reserved for extended attributes.

Metadata

- Using checksums of metadata is a measure to protect the metadata from being used if corrupted or manipulated.
- Check if the file system contains snapshots (a previous state of the file system)
- The verity feature (RO_COMPAT_VERITY) may be interesting for the investigator, which means verity inodes may exist on the file system
    - These inodes have content that is read-only, and can be verified using a Merkle tree-based hash.

Merkle Tree Hash

- A Merkle tree-based hash means that the file is divided into blocks that are hashed.
- Then these hashes are concatenated and represent larger blocks of data and rehashed.
- This continues until there is one large block left, representing the complete file, which is hashed. It is this final hash the read-only file is verified against.

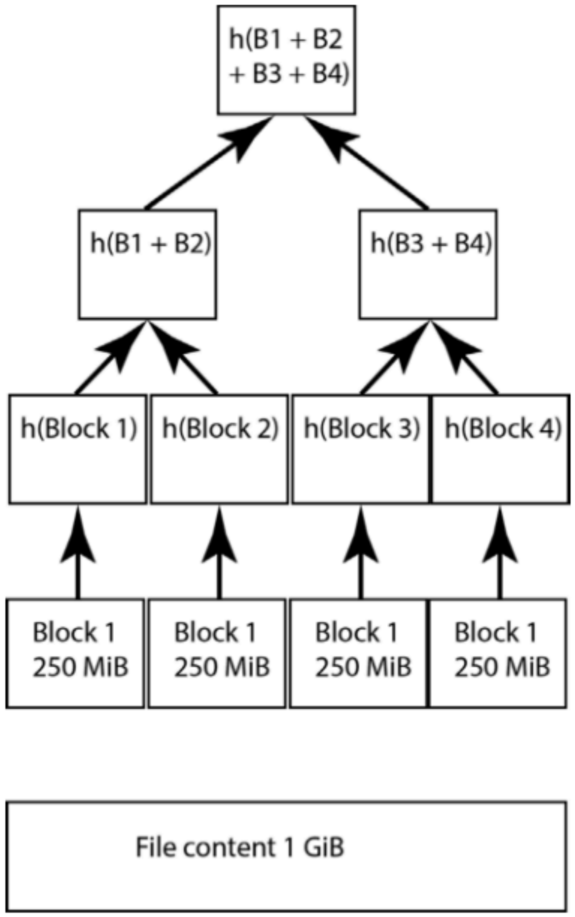


Fig. 2.7: Illustration of Merkle tree hash.

Merkle Tree Hash

- Figure 2.7 illustrates the Merkle tree hashes

- In order to verify the public key, you only need the hashes of the first and second half of the public key and that you can compute half of the public key by knowing their quarters.

Compatible Features

- The feature flag RO_COMPAT_READONLY means that this file system should only be mounted as read only.

- Most implementation of Ext4 file system drivers complies with this setting,

- but there may exist driver implementations or tools who allow writing to the file system even if it is set to read only.

Tip for Forensics

- Test if a file system is read only
- The investigator should perform experiments to test if it is possible to write to an identical copy of the read only file system using the same driver or tools found on the device under investigation.

Compatible Features

Table 2.3: Read only compatible features

Value	Description
0x1	Sparse superblocks. See the earlier discussion of this feature (RO_COMPAT_SPARSE_SUPER)
0x2	This filesystem has been used to store a file greater than 2GiB (RO_COMPAT_LARGE_FILE)
0x4	Not used in kernel or e2fsprogs (RO_COMPAT_BTREE_DIR)
0x8	This filesystem has files whose sizes are represented in units of logical blocks, not 512-byte sectors (RO_COMPAT_HUGE_FILE)
0x10	Group descriptors have checksums (RO_COMPAT_GDT_CSUM)
0x20	Indicates that the old ext3 32,000 subdirectory limit no longer applies (RO_COMPAT_DIR_NLINK)
0x40	Indicates that large inodes exist on this filesystem (RO_COMPAT_EXTRA_ISIZE)
0x80	This filesystem has a snapshot (RO_COMPAT_HAS_SNAPSHOT)
0x100	Quota (RO_COMPAT_QUOTA)
0x200	This filesystem supports "bigalloc", extents are tracked in units of clusters (of blocks)(RO_COMPAT_BIGALLOC)
0x400	This filesystem supports metadata checksumming. (RO_COMPAT_METADATA_CSUM)
0x800	Filesystem supports replicas. This feature is neither in the kernel nor e2fsprogs (RO_COMPAT_REPLICA)
0x1000	Read-only filesystem image; the kernel will not mount this image read-write and most tools will refuse to write to the image (RO_COMPAT_READONLY)
0x2000	Filesystem tracks project quotas (RO_COMPAT_PROJECT)
0x8000	Verity inodes may be present on the filesystem (RO_COMPAT_VERITY)

Large Files

- Table 2.3 demonstrates that if the file system has large files, the superblock will use options like:
  - RO_COMPAT_LARGE_FILE(exist files larger than 2 GiB),
  - RO_COMPAT_BIGALLOC(extents are using clusters instead of blocks) and
  - RO_COMPAT_HUGE_FILE(file size is shown in logical blocks instead of sectors).
- Large files can be of investigative value since they may contain videos, file system containers, encrypted files, etc.
- It is important that these files are investigated.

The group descriptor

- The group descriptor describes information about a particular group

- for instance, the locations of the block bitmap, inode bitmap, and the inode table.

- In order to find the group descriptor, we need to know the block size, as shown in Figure 2.8.

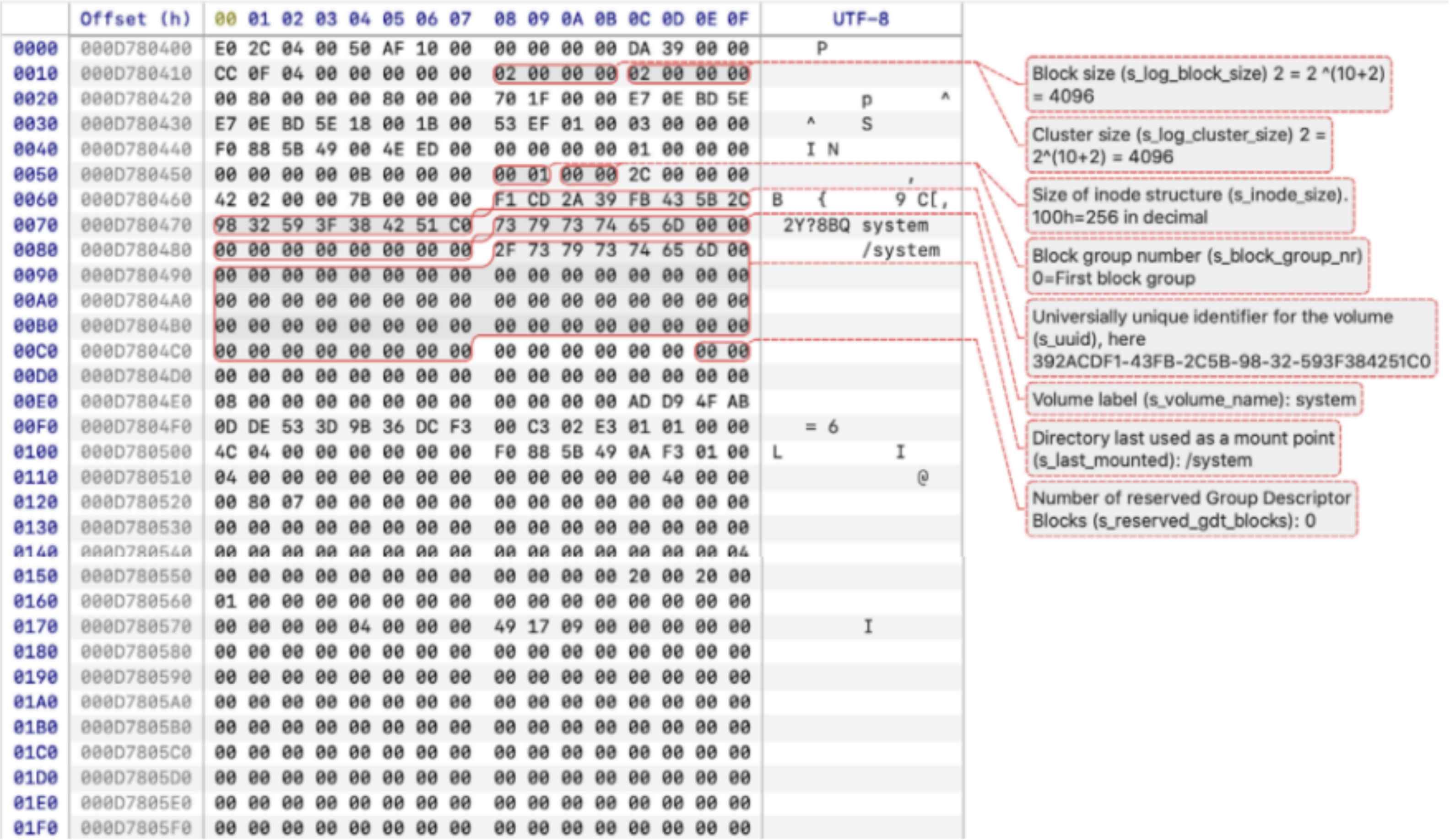


Fig. 2.8: Information about blocks in the Ext4 superblock.

Group descriptor

- The value in this field is 2, and the formula we need to use is

$$10^{(10+s\_log\_block\_size)}$$

- We can find the group descriptor in the block following the superblock.
- In order to find the group descriptor, we, in this case, move 4096 bytes, one block, forward from the start of the superblock, from byte offset 0, not from 1024

Group descriptor

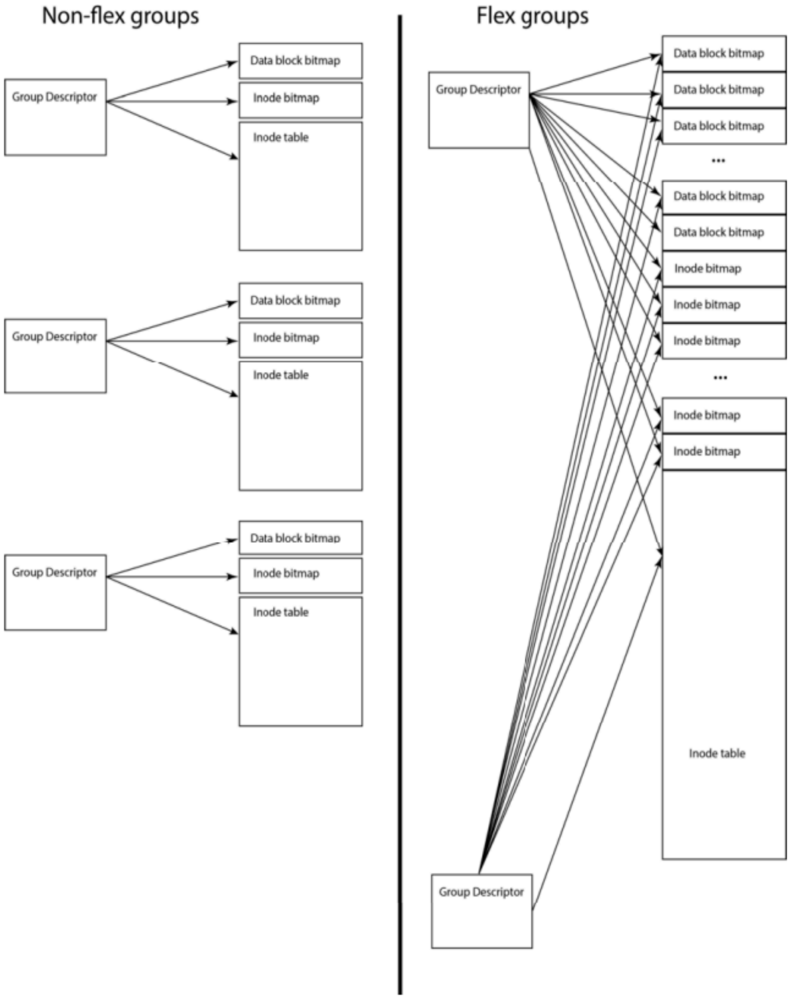


Fig. 2.9: Different designs for Group Descriptors

Group descriptor

- If the Ext4 has the 64-bit feature (INCOMPAT_64BIT) enabled, then the location of the bitmaps and the inodes table has two fields each.

- The first fields should describe the lower bits for the location, while the last describes the upper bits

- These fields should describe the block location of the block bitmap.

	Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	UTF-8
0000	000D781000	02	00	00	00	12	00	00	00	22	00	00	00	00	00	F5	08	"
0010	000D781010	7D	02	04	00	00	00	00	00	00	00	00	00	F5	08	1B	E7	}
0020	000D781020	03	00	00	00	13	00	00	00	19	02	00	00	00	00	66	1F	f
0030	000D781030	0A	00	04	00	00	00	00	00	00	00	00	00	66	1F	2E	BB	f .
0040	000D781040	04	00	00	00	14	00	00	00	10	04	00	00	00	00	70	1F	p
0050	000D781050	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	8C	88	p
0060	000D781060	05	00	00	00	15	00	00	00	07	06	00	00	00	00	70	1F	p
0070	000D781070	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	70	C3	p p
0080	000D781080	06	00	00	00	16	00	00	00	FE	07	00	00	00	00	70	1F	p
0090	000D781090	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	CF	80	p π
00A0	000D7810A0	07	00	00	00	17	00	00	00	F5	09	00	00	00	00	70	1F	p
00B0	000D7810B0	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	22	5F	p "_
00C0	000D7810C0	08	00	00	00	18	00	00	00	EC	0B	00	00	00	00	70	1F	p
00D0	000D7810D0	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	67	5D	p g]
00E0	000D7810E0	09	00	00	00	19	00	00	00	E3	0D	00	00	00	00	70	1F	p
00F0	000D7810F0	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	46	39	p F9
0100	000D781100	0A	00	00	00	1A	00	00	00	DA	0F	00	00	00	00	70	1F	p
0110	000D781110	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	04	FD	p
0120	000D781120	0B	00	00	00	1B	00	00	00	D1	11	00	00	00	00	70	1F	p
0130	000D781130	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	FB	B3	p
0140	000D781140	0C	00	00	00	1C	00	00	00	C8	13	00	00	00	00	70	1F	p
0150	000D781150	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	7A	CE	p z
0160	000D781160	0D	00	00	00	1D	00	00	00	BF	15	00	00	00	00	70	1F	p
0170	000D781170	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	A5	2F	p /
0180	000D781180	0E	00	00	00	1E	00	00	00	B6	17	00	00	00	00	70	1F	p
0190	000D781190	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	D6	14	p
01A0	000D7811A0	0F	00	00	00	1F	00	00	00	AD	19	00	00	00	00	70	1F	p
01B0	000D7811B0	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	2A	C6	p *
01C0	000D7811C0	10	00	00	00	20	00	00	00	A4	1B	00	00	00	00	70	1F	p
01D0	000D7811D0	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	DD	89	p `
01E0	000D7811E0	11	00	00	00	21	00	00	00	9B	1D	00	00	00	00	70	1F	! p
01F0	000D7811F0	00	00	05	00	00	00	00	00	00	00	00	00	70	1F	CF	FA	p

Location to data block bitmap (0x2)

Location to inode bitmap (0x12 or 18 in decimal)

Location to inode table (0x22 or decimal 34)

Fig. 2.10: Group descriptors in a flex group

Group descriptor

- A very similar copy of this group descriptor block is found in all other group descriptor blocks.
- However, bg_flags values may deviate.
- It is important to understand that not all block groups have superblocks or group descriptors if either the superblock RO_COMPAT_SPARSE_SUPER or the COMPAT_SPARE_SUPER2 feature flag is set.
- The field bg_flags can have any combination of these values :

  - 0x1 Inode table and bitmap are not initialized
  - 0x2 Block bitmap is not initialized
  - 0x4 Inode table is zeroed (on initialisation)

Group descriptor

Table 2.4: Group descriptor

Offset	Size	Name	Description
0x0	0x4	bg_block_bitmap_lo	Location to data block bitmap
0x4	0x4	bg_inode_bitmap_lo	Location to inode block bitmap
0x8	0x4	bg_inode_table_lo	Location to the inode table
0xC	0x2	bg_free_blocks_count_lo	Free blocks in block group
0xE	0x2	bg_free_inodes_count_lo	Free inodes in block group
0x10	0x2	bg_used_dirs_count_lo	Used directories in block group
0x12	0x2	bg_flags	Important for bitmaps and inode tables
0x14	0x4	bg_exclude_bitmap_lo	Location of snapshot exclusion bitmap
0x18	0x2	bg_block_bitmap_csum_lo	Data block bitmap checksum
0x1A	0x2	bg_inode_bitmap_csum_lo	Inode bitmap checksum
0x1C	0x2	bg_itable_unused_lo	Unused inodes in group

Universal Unique Identifier

- In the superblock the field, s_uuid, assigns a unique identifier for the file system volume.

- This should be unique for every instance of a volume created, however, if we flash a partition, the target may be assigned the same UUID for its file system as the original source.

Universal Unique Identifier

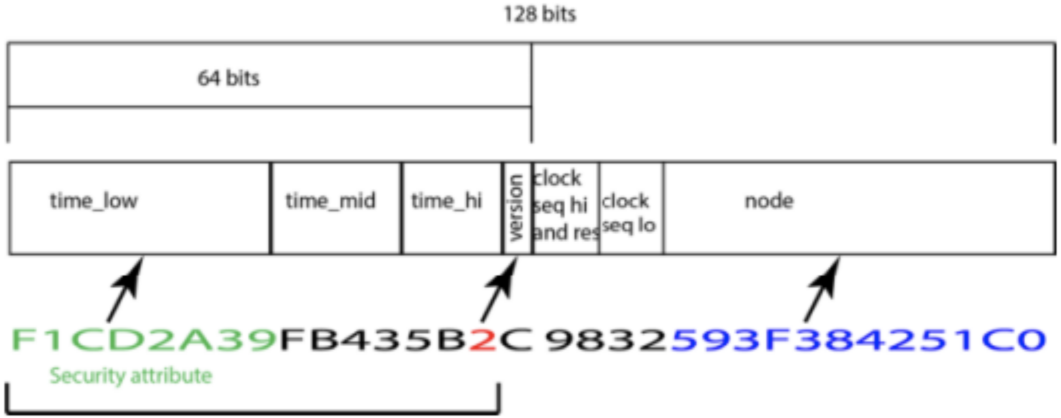


Fig. 2.11: Structure of the UUID v.2

Universal Unique Identifier

- The structures of the UUIDs are defined in RFC4122, and the one used here is version 2 as shown in Figure 2.11.

- It uses a 60-bit timestamp (in which the four least significant bytes are overwritten with a security attribute) with an Epoch from 15th of October 1582, and a node identifier (MAC address) at the last 6 bytes.

Ext4 - Metadata Category - The inode

- The index node (inode) is defined in the structure ext4_inode, which defines most of the metadata related to a file, except its file name.
- Previous versions of Ext used a 128-byte size inode, while the Ext4 standard uses 256 bytes.
- However, the first 128 bytes are backwards compatible with previous versions of Ext.
- The information in this section is based on the Ext4 source code and the interpretation found at Kernel.org

Inode offset table

Table 2.5: Inode offset table

Offset	Size	Name	Description
0x00	0x2	i_mode	User privileges and type of file
0x02	0x2	i_uid	Lower 16 bits of the owner id
0x04	0x4	i_size_lo	Lower 32 bits of the file size
0x08	0x4	i_atime	Last access time
0x0C	0x4	i_ctime	Last inode change time
0x10	0x4	i_mtime	Last data modification time
0x14	0x4	i_dtime	Deletion time
0x18	0x2	i_gid	Lower 16 bits of group id
0x1A	0x2	i_links_count	Number of hard links pointing to this file
0x1C	0x4	i_blocks_lo	Lower 32 bits of 512 byte blocks this file uses
0x20	0x4	i_flags	Inode flags
0x24	0x4	i_osd1	For Linux this is the inode version
0x28	0x3C	i_block[]	Block map or Extent tree.
0x64	0x4	i_generation	File version for NFS
0x68	0x4	i_file_acl_lo	Lower 32 bit address of extended attribute block
0x6C	0x4	i_size_high	Higher 32 bit address of file size
0x70	0x4	i_obso_faddr	Obsolete fragment address
0x74	0xC	i_osd2	OS descriptor 2
0x80	0x2	i_extra_isize	Size of the used are of inode - 128
0x82	0x2	i_checksum_hi	Upper 16-bits of the inode checksum
0x84	0x4	i_ctime_extra	Extra change time bits
0x88	0x4	i_mtime_extra	Extra modification time bits
0x8C	0x4	i_atime_extra	Extra access time bits
0x90	0x4	i_crtime	File creation time, in seconds since the Unix Epoch
0x94	0x4	i_crtime_extra	Extra file creation time bits
0x98	0x4	i_version_hi	Upper 32-bits for version number
0x9C	0x4	i_projid	Project ID

User privileges and type of file

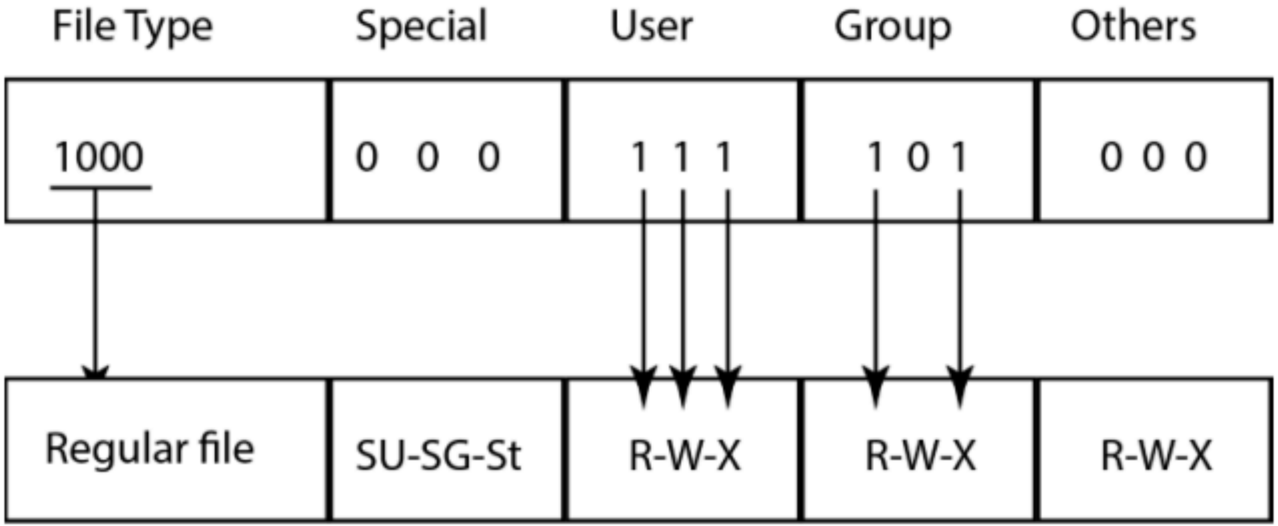


Fig. 2.12: File type and privileges.

User privileges and type of file

- As illustrated in Figure 2.12 the i_mode field name 12 least significant bits are used for user privileges.

- These privileges are important when investigating a file or directory since it explains ownership and user privileges.

- However, it is also important to understand that these privileges may be changed if the user has the privileges to do so

Inode file types

Table 2.6: Inode file types

4 MSb	Meaning
0001	Special FIFO file (named pipe)
0010	Character device
0100	Directory
0110	Block device
1000	Regular file
1010	Symbolic link
1100	Socket

Temporary metadata describing inodes

- Almost every inode has fields describing important timestamps.

- For backward compatibility, these are located from hex offset 0x08 from the start of the inode, and are 32-bit integers describing the number of seconds since 1970 (Unix Epoch).

- However, extra 32 bit fields in the inode use the least significant 2 bits to expand the timestamp to 34 bits.

- The remainder of the 30 bits is used for nanoseconds granularity.

2.4.5 Links count –Blocks used by a file

- The number of 512 byte blocks (sectors) used by a file is defined in the i_blocks_lo field. However, if the inode i_flags has the EXT4_HUGE_FILE_FL file option set and the the superblock has the huge file feature enabled then the field i_blocks_hi needs to be added using this formula.

$$(\text{i\_blocks\_lo} + \text{i\_blocks\_hi} \ll 32)$$

EXT4_HUGE_FILE_FL inode

- If the i_flags has the EXT4_HUGE_FILE_FL inode but file system does not have the huge file feature then field i_blocks_hi needs to be added using this formula.

$$\text{i\_blocks\_lo} + (\text{i\_blocks\_hi} \ll 32)$$

Ext4 - Application Category