

Forensics for Advanced UNIX File Systems

Dr. Knut Eckstein, Senior Scientist*

Advanced UNIX file systems differ substantially from traditional UNIX file systems with respect to their internal layout and data structures. This paper analyzes these differences and their effects on the methods and results of digital forensic media analysis. In addition, it provides results from a prototype implementation of a forensic toolkit for an advanced UNIX file system, IBM's Journaling File System for Linux. Finally a generalized scheme for categorizing file system meta-data is proposed.

I. INTRODUCTION

This paper aims to provide a comparison between traditional UNIX file systems and more recent implementations with regard to digital forensic media analysis. The term "traditional UNIX file systems" refers to the Berkeley Fast File System (FFS) [1] and related implementations such as *ext2fs* on Linux, *ufs* on Solaris and the BSD family, *efs* on IRIX, *hfs* on HP-UX and many more. All of these file systems share the basic design principles with FFS. Naturally, they have evolved over time to add differentiating features.

For the purpose of this paper, the term "Advanced UNIX File Systems" is intended to group a number of file systems that implement design principles not to be found in FFS.

Among those are

- The Veritas File System (VxFS), for AIX, HP-UX, Linux and Solaris [2],
- IBM's Journaling File System (JFS) for AIX and Linux [3,4],
- SGI's XFS for IRIX [5] and Linux [6] and
- ReiserFS for Linux [7].

Many members of this list are sometimes also grouped by the term "journaling file systems", referring to one of their common differences from FFS. This technology, roughly spoken, introduces the concept of database transactions to file systems for higher performance, increased robustness and more rapid integrity checking.

Many or all of the advanced UNIX file systems share the following important enhancements:

- Binary trees (btrees) instead of linear tables
- Variable sized disk allocation units
- On-demand inode allocation

* NATO C3 AGENCY, P.O. BOX 174, THE HAGUE, NL.

- Flexible and extensible internal structures

These enhancements require specific support in forensic software packages: For example, "EnCase", a widely used commercial-off-the-shelf forensic software, offers support for ReiserFS, but none of the other advanced UNIX file systems. For the well-known Open Source software "The Sleuth Kit" (TSK) and its predecessor "The Coroner's Toolkit" (TCT), UNIX support is currently limited to traditional UNIX file systems only¹.

The *ext3fs* file system for Linux [8] has been left out of the list of advanced file systems on purpose. Its current version focuses exclusively on adding the journaling technology to *ext2fs*². Thus by ignoring the disk journal, an *ext3fs* file system can be mounted or analyzed as an *ext2fs* one. EnCase, TSK and other tools are capable of doing that.

This paper focuses on how the technological differences between traditional and advanced UNIX file systems can influence the methods and results of a forensic examination. For the purpose of this research, the author has implemented a prototype module for TSK, which adds support for IBM's JFS file system for Linux to this popular, open-source forensics toolkit. This module enables direct, experimental comparison with traditional file systems already supported by TSK³. JFS was chosen because it implements a large number of the listed enhancements and because it is well documented, as IBM chose to make this file system available to the open source community.

In the following section, traditional UNIX file systems and their forensic analysis will be described. In section III, the new technologies in advanced UNIX file systems will each be examined from a forensic point of view, referring to JFS as a sample implementation. Section III closes with the proposal of a generalized scheme for categorizing file system meta data.

¹ Microsoft's NTFS uses some of the listed enhancements and both EnCase and TSK can analyze NTFS. Carrier [9] discusses the effects of the NTFS structures on forensic media analysis.

² Work is being performed by the developers of *ext3fs* to add several of the other enhancements listed above.

³ The author believes that this is the first open source forensic analysis tool for an advanced UNIX file system such as JFS.

II. FILE SYSTEM FOUNDATIONS

The analysis of traditional UNIX file systems shows a number of important concepts, which will now be discussed in the context of digital forensics.

A. Actual Data vs. Meta-Data

The main purpose of a file system is to efficiently store and retrieve data on secondary storage containers like hard disks or the popular memory “sticks” and many more. To attain this goal, file systems have to store not just the actual data, but also management data like defect lists, allocation tables as well as file and directory names. This data is commonly being referred to as meta-data.

The software that implements a file system is usually part of the operating system. It follows a set of algorithms to create, update or delete meta-data in correspondence to creation, modification or deletion of actual data.

B. Forensic Software

Carrier [10] describes the high-level process of digital forensics through the following steps: Data acquisition from the source, data analysis and extraction of evidence, and preservation and presentation of the evidence. Depending on the type of source, digital forensics may comprise media analysis, code analysis, network packet analysis etc.

This paper focuses on the data analysis and the extraction and presentation of evidence performed by forensic *media analysis* software: Given the notion of actual data versus meta-data, typical media analysis tasks can be structured into the following categories:

- Search actual data
 - o Search *allocated* data blocks for specific content, for example key words.
 - o Search partly used *allocated* data blocks for data hidden in their “slack” space.
 - o Search *unallocated* data blocks for the content of deleted files.
- Search meta-data
 - o List *allocated* meta-data in presentation formats specific to forensic analysis, for example a timeline built upon the modified, access, and change (MAC) timestamps of each file in the file system.
 - o Search for *unallocated* meta-data⁴ – for example the name of a deleted file – to aid reconstruction of past events or recovery of deleted files.

C. The Original UNIX File System

A quick look at the disk layout of the original AT&T Unix file system [11] displayed in figure 1 shows that the split between actual data and meta-data was originally performed in a very simple fashion.

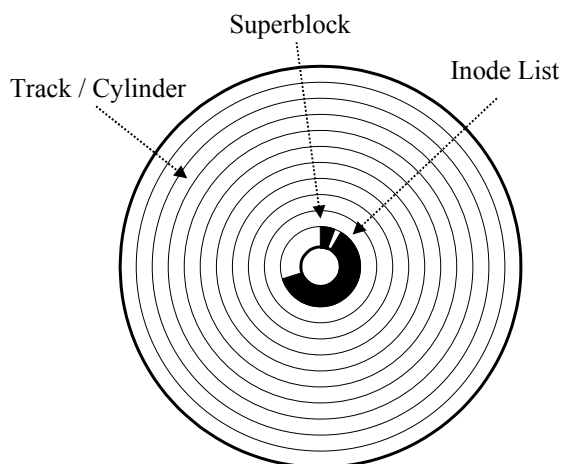
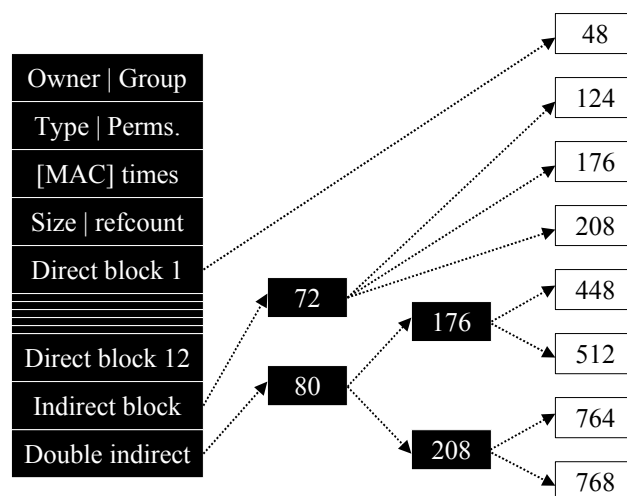


Figure 1: Original UNIX disk layout

Most of the meta-data resides at the beginning of the disk storage space. The super block at the very beginning stores general file system state information like its size, where to find free unallocated space and other information. It is followed by the inode⁵ list that holds per-file meta-data like serial number, timestamps, access control information and disk block allocation lists.

This approach had multiple limitations in terms of performance, for example due to long disk head movements required from the meta-data to the actual data zone on disk. But from a forensic point of view, this initial UNIX disk layout is of course attractive at first sight, because it seems to offer a straightforward separation between meta-data and actual data.

A closer look at the inode structure reveals that this separation is not entirely straightforward, given the concept of indirect, double and even triple indirect disk block addresses.



⁴ I.e. meta data that belonged to actual data that has been deleted.

⁵ “Inode” is the technical term describing the meta-data structure that UNIX operating systems keep per individual file.

Figure 2: Inode Data Structure

Figure 2 illustrates the general inode data structure as well as the concept of indirect block addresses, where data blocks are used to store a list of block addresses, which - depending on the level of indirection - lead to blocks of actual data or again to blocks of disk block addresses. This concept is required to keep the inode size small (128 bytes) as well as the disk block size small, but still allows creation of files larger than the number of block pointers in the inode times the disk block size. So in reality the actual data in the data block area is interspersed with file allocation meta-data.

D. The Berkeley File System

The Berkeley FFS addressed the performance and robustness issues in the original UNIX file system through a number of advanced concepts:

- Cylinder groups
- Block fragments
- Super block backup copies
- File and directory placement policies

The introduction of cylinder groups brought actual data and meta-data into physical vicinity in terms of minimized disk head movements, as can be seen in the simplified illustration of the FFS disk layout in figure 3.

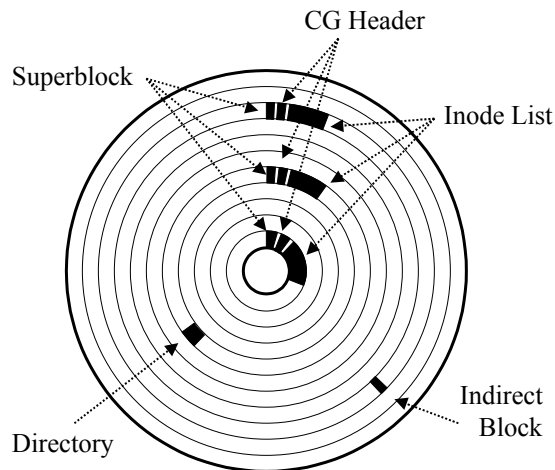


Figure 3: FFS Disk Layout

The concept of block fragments allowed for the increase in logical disk block size while keeping the quota of disk space wasted through “slack” space in incomplete blocks low. A summary list of available fragments is managed as additional meta-data in the header of each cylinder group. With a new default logical block size of 4KB, a maximum of 4GB per file could be addressed with just two levels of indirection, so the costly triple indirection would be used much less frequently than in original UNIX file system.

The introduction of specific file and directory placement policies started to effectively leave room for future growth of these objects. Fragmentation, which means allocation of

a file to non-contiguously numbered logical disk blocks, can thus be often avoided.

From a forensic point of view, this leads to a higher success rate for recovering deleted files. Also, objects that are logically close to each other⁶ will be placed physically close to each other on the disk.

E. Meta-Data Categories

In order to describe FFS’s more complex pattern of meta-data interspersed with actual data, the meta-data in a file system can be grouped into four categories. These categories follow the four sub-layers or functional categories into which a file system can be decomposed [9]: The file system layer, the data unit layer, the inode layer and the human interface layer. These sub-layers or categories exist to describe file systems in generic terms and should not be confused with abstraction layers [10].

For each sub-layer the following paragraphs discuss their data and meta-data parts. Like in other layering schemes, for example the OSI network layer model, actual data (payload) as seen by a lower layer contains meta-data (header information) and actual data when viewed on a higher layer. Unlike the OSI model, this model is not strictly nesting, i.e. not all meta-data on layer $n+1$ is stored inside actual data on layer n .

The file system sub-layer views essentially all of the file system as actual data except parts of the super block, where general meta-data about the file system is stored: The total size of the file system, the logical disk block and fragment size, the total number of inodes, a volume label etc.

At the data unit sub-layer, two representations of actual data exist in FFS: Logical disk blocks and fragments. The main meta-data is the fragment allocation bitmap stored in each respective cylinder group header. The super block stores summary meta-data for unallocated disk blocks; the cylinder group headers store summary information for unallocated fragments.

The inode sub-layer perceives files consisting of logical disk blocks or fragments as actual data. The primary meta-data are the inodes themselves and their corresponding allocation bitmaps, which are stored right after the cylinder group headers and the indirect addressing blocks that are stored in logical disk blocks. The super block stores inode and cylinder group quantity information.

At the human interface sub-layer, which was omitted from the previous discussion of the original UNIX file system, a hierarchical, human-readable namespace is created. This is done by introducing the concept of directories where a directory is comprised of the meta-data by linking the name

⁶ For example two files located in the same directory.

of a file or subdirectory to its corresponding inode serial number⁷.

In FFS, directories are stored in files like actual data. Thus, at the human interface sub-layer, the file type variable in a file's inode is used to distinguish between meta-data stored in directories and actual data stored in so-called "regular" files.

Using forensic software to search through previously allocated meta-data trying to reconstruct past events or recover deleted files, it is interesting to observe that different implementations of FFS treat meta-data differently upon deletion of a file [13,14]

III. NEW FILE SYSTEM TECHNOLOGIES

New file system technologies like those mentioned in the introduction do affect forensic media analysis through the introduction of new meta-data structures, new disk layouts and other innovations. The relevance of these technologies will now be discussed with technical details being derived from their implementation in JFS.

A. Variable-Sized Allocation Units

FFS-style file systems record all file allocation addressing information as block numbers of fixed-length logical disk blocks. They record direct and indirect references to those disk blocks to address every single one of them, even if the file occupies a contiguous sequence of blocks on disk.

JFS and others advanced file systems introduce the concept of variable-sized allocation units. Thereby a contiguous sequence of logical blocks on disk is addressed as a single allocation unit, or "extent". With a maximum extent size of 16 GB in JFS, only very large files and other special cases discussed later on will require the use of multiple allocation extents per file.

```
istat -f linux-ext2 /dev/hdd5 8
inode: 12
Allocated
size: 65536
refcount: 1
[...]
Direct Blocks:
137 138 139 140 141 142 143 144
145 146 147 148 150 151 152 153

Indirect Blocks:
149
```

Listing 1: istat output for a 64KB file in ext2fs

For example, the allocation of a FFS file containing 64 kilobytes of data by default creates 16 allocation references to 16 4K-disk blocks. Since the number of direct block addresses inside an FFS-style is limited to 12, four of these references do not fit inside the inode. Thus additional

meta-data in the form of an indirect block has to be created. It is typically interspersed with the actual data, as can be seen from listing 1, where this file occupies blocks 137 through 153 with 149 being the indirect block.

In JFS, such a file requires a single allocation reference, provided a contiguous area of disk blocks with a total length of 64KB is available on the file system. The TSK prototype module for JFS shows this situation in listing 2 where the file occupies a contiguous sequence of 16 blocks starting at block 51. Up to 16 allocation references can be stored directly inside a JFS inode, as will be explained in more detail in the following section.

From a forensic point of view, variable-sized allocation extents lead to less "indirect-block-style" meta-data having to be allocated on disk. This means more contiguously allocated files, which can be more easily recovered from deletion.

```
istat -f jfs /dev/hdd6 6
inode: 12
Allocated
size: 65536
refcount: 1
[...]
Extents: address(length)
51(16)
```

Listing 2: istat output for a 64KB file in JFS

B. Tree Data Structures

Advanced file systems choose to organize a number of meta-data records in trees data structures. This is done to improve lookup performance compared with the linear lists or tables that FFS maintains. In the case of JFS, trees are used for

- extent allocation information, keyed by file offset,
- directory entries, keyed by name, and
- the allocation bitmap, keyed by disk block address.

1) Trees for Extent Allocation

A tree for extent allocation gets set up when all 16 extent allocation slots inside a JFS inode (as introduced in section A) are occupied. To be able to describe a 17th extent, the status of the slots inside the inode changes from *leaf node* to *internal node*. The former 16 entries inside the inode – together with the new extent descriptor – are placed in a meta-data disk block holding up to 254 entries. This disk block then represents a leaf node in the allocation tree. Inside the inode, a single allocation slot points to that meta-data disk block. Once 16 times 254 entries are occupied, another sub-tree level will be created.

Since JFS tries to allocate files contiguously in a single extent on disk, multiple extents are only needed in specific situations. For example, if no unallocated sequence of disk blocks exists in the file system, which would be large enough to hold the particular file. Another situation are so-called "sparse" files, which provide an efficient means for applications to store a relatively small amount of data spread out over a large range of offsets in a file.

⁷ Operating systems use this number to identify files internally, so they do not rely on file names.

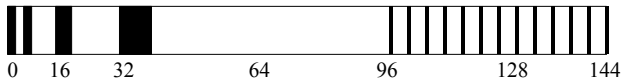


Figure 4a: Example sparse file

Figure 4a shows a sparse file created for demonstration purposes, using a total of 17 allocation extents. This file has allocated 25 disk blocks distributed over an offset range from 0 to 144, leaving 16 “holes” of varying size. A non-sparse file would have to allocate all 145 disk blocks thus wasting considerable space. Figure 4b shows a simplified view of the JFS metadata for that file: The JFS inode, which is 512 bytes in total size, contains at its beginning meta-data fields very similar to an FFS inode: Owner information, type and permission fields as well as timestamps etc. This generic information is followed by a rather large amount of space that can be dedicated to multiple purposes depending on the type of file associated with the inode.

In the case of a regular file, a header precedes a list of extent allocation slots. These are being referred to as “root” node slots, because they are stored directly inside the inode and they form the root of the binary allocation tree. Each slot consists of the logical offset *O* of the extent in the file, the extent’s physical address *A* on disk and the length of the extent. All numbers are given in multiples of the logical disk block size. The logical offset *O* is used as the key to sort the tree.

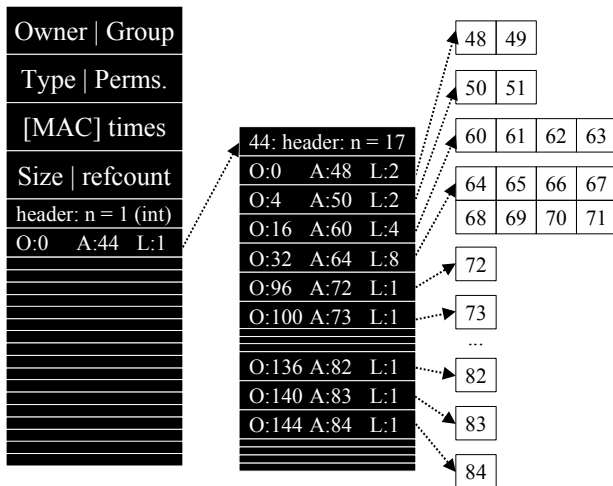


Figure 4b: JFS sparse file meta-data

In this case, only one internal slot exists at root level inside the inode, indicating that allocation information for this file with respect to logical offset zero can be found at disk block number 44. This block is a leaf node in the allocation tree and contains a header plus up to 254 slots. In this example, nine slots are in use, pointing to nine allocation extents on disk, each between one and eight blocks long.

From a forensic point of view it is important to note that extent allocation entries are not deleted upon file deletion, nei-

ther inside the inode nor in meta-data disk blocks. So unless the inode or the former meta-data disk block is being re-used, this information can still be used for “undeletion” of that file. Effective success of the undeletion will of course depend on whether the blocks holding the actual data have been re-used in the meantime or not.

Listing 3 shows that the meta-data structures of the sparse file depicted in figure 4 are still intact after the file has been deleted. The TSK prototype module for JFS developed by the author has been used to successfully undelete sparse files like this one.

```

istat -f jfs /dev/hdd6 15
inode: 15
Not Allocated
size: 593920
refcount: 0
[...]
Indirect Extents: address(length)
40(1)

Extents: address(length)
48(2) 50(2) 60(4) 64(8) 72(1) 73(1) 74(1) 75(1)
76(1) 77(1) 78(1) 79(1) 80(1) 81(1) 82(1) 83(1)
84(1)

```

Listing 3: istat output for deleted sparse file

2) Binary Trees for Directory Entries

Directories in JFS are not stored like regular files, but use a binary tree schema similar to that of allocation extents for regular files. An inode can store up to eight root node slots internally, and up to 123 slots can be stored in a meta-data disk block.

Since each slot is of limited, fixed size, multiple slots may have to be used to store long file or directory names. For example, the eight slots inside an inode can be used to store eight short names or just two long names spanning multiple slots each. The latter situation is depicted in figure 5a.

After the standard inode data fields, the directory-specific data structure starts with a list header. Inside this header *ne=2* specifies that two entries are stored in this directory. They can be located by using the lookup table immediately following the list header. The variable *free_c=2* indicates that two slots are still free and *free_l=7* points to the seventh slot as the starting point of the list of free slots.

The first field of the lookup table points to slot one as the first directory entry. This entry points to inode number four with an entry name that is 28 characters long, the first eleven of which are **this_is_a_v**. Furthermore, *n=2* indicates that this directory entry is continued in slot two. In contrast to slot one, slot two uses a different data structure that could be described as a “continuation entry”. Here *n=3* points to another continuing slot and *c=1* specifies that this continuation entry spans a single slot. With no inode number to record, the continuation slot has room for 15 instead of 11 characters. Slot three is the final slot for this directory entry, which is indicated by *n=-1*.

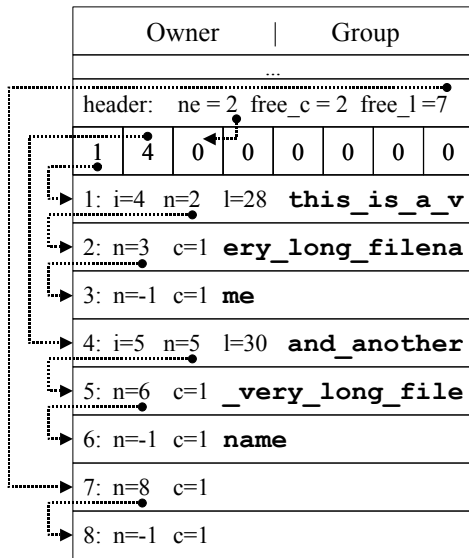


Figure 5a: JFS directory holding two long filenames

Slot four is referenced by the lookup table as the starting slot for the second directory entry. It starts a very similar sequence of chained slots to store another filename, which is 30 characters long.

Slot seven is the first free slot and points to slot eight which through $n=-1$ indicates that this is the end of the list of free slots. If those two slots were also used, an additional directory entry would cause the creation of a two-level tree in the same fashion as discussed for extent allocation slots in the previous subsection. A single “internal” slot would be occupied inside the inode, using a third data structure containing a name string as a search key and an extent allocation record to point to the meta-data disk block holding the directory entries.

After deletion of the file pointed to by the first directory entry, the respective slots are added to the “free list”. Figure 5b shows the resulting changes in the header, the lookup table and the slots themselves: The number of free slots has grown to five and slot one now constitutes the start of the free list, which continues through slot two and three to seven and eight.

From a forensic point of view, it is interesting to note that name strings in slots, which have been freed up, are not being erased. Thus a forensic tool can potentially recover them. A closer look at slot one in figure 5b though reveals that free slots are always handled as “continuation nodes”. Node one therefore gets “converted” from a directory entry type node to a continuation node during the removal of the directory entry. During this conversion, the lower order word of the inode number gets overwritten with the $n=2$ $c=1$ value pair, thus partly erasing this very interesting piece of information.

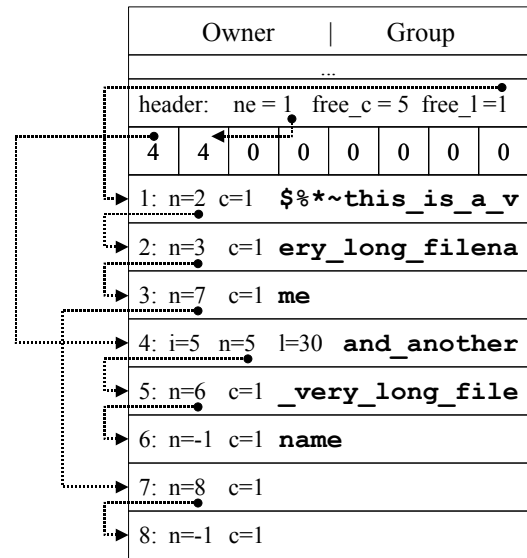


Figure 5b: JFS directory after deletion of a long filename

Therefore the implemented JFS forensic tool is only able to list names of deleted files or subdirectories, but cannot directly associate this meta-data with the actual data, which may also still exist on disk. The seemingly “garbage” string now displayed in the string section of node one can be converted back to its original meaning of $l=28$, $n=2$ plus the high-order word of the inode number.

3) Forensic Recovery of Tree Structures

Given that both directory and extent allocation btree nodes in meta-data disk blocks are not erased immediately, chances are that even if the inode itself gets re-used, the “external” meta-data in the separate disk block is not yet overwritten. Therefore, in analogy to the original “Lazarus” program [12n] that attempts to classify actual data blocks which had been “recovered”, this paper proposes the addition of a small “meta-lazarus” utility that scans unallocated data blocks for extent, directory or bitmap tree headers trying to identify and recover additional meta-data.

The question whether the btree type meta-data structures are more or less volatile than the classic FFS indirect blocks has to be addressed in future, long term tests and observations. Given that the JFS btree extents follow a more elaborate internal structure than just a list of disk block addresses, identification and verification of this meta-data may be slightly more straightforward.

C. Dynamic Inode Allocation

FFS-style file systems pre-allocate a fixed number of inodes to static areas on disk while the file system is being created. Increasing the number of inodes to be able to store more files on a file system can thus only be achieved by recreating the file system from scratch. To avoid that situation, FFS tends to create “10 times more inodes than will ever be needed”, resulting in a potentially large waste of disk space.

Also, pre-allocating fields of inodes at the beginning of each cylinder group means, that an FFS file system cannot contiguously allocate a file on a disk which is larger than a single cylinder group.

For these reasons, advanced UNIX file systems have introduced the concept of dynamic inode allocation. Instead of fixed areas on disk, all inodes are stored in a special file, only visible to the operating system. This file holds arrays of 32 inodes each in separately allocated 4KB extents. The file grows or shrinks whenever an array/extent is added or removed.

To still allow for efficient placement policies preserving locality between actual data and meta-data, JFS has replaced the concept of a cylinder group with the notion of an allocation group. The allocation groups ensure that the 4KB extents of the inode file are placed close to the actual data extents of the corresponding 32 files. While similar in concept, allocation groups do not require the storage of additional meta-data at fixed positions on disk, thus allowing the contiguous allocation of files spanning even multiple allocation groups.

In general, dynamically allocated inode meta-data is expected to be more volatile than statically allocated meta-data [9]. This is because statically allocated inodes have a “guaranteed” position on disk, which can never be overwritten by actual data. Dynamically allocated inodes on the other hand are stored in disk extents, which can also be occupied by actual data. Statistical long-term measurements are required to determine in how far allocation policies are keeping inode meta-data extents and actual data extents “apart”.

D. More Flexible Structured Meta-Data

Many advanced UNIX file systems generalize the concept of storing meta-data like inodes in a special file to other types of meta-data like block allocation maps, access control lists, the disk journal, etc. This enables a number of advanced maintenance operations, which are difficult or impossible to implement for traditional UNIX file systems, for example on-line resizing of a file system.

Regarding the effect on forensics, the statement from the previous sections regarding volatility and the requirement for a “meta-lazarus” tool equally apply here.

JFS and VxFS go even beyond the generalized concept of storing meta-data in special files. They introduce a “meta-file system”⁸ whose files hold meta-data from the file system sub-layer and the data unit sub-layer as well as one or more actual file systems, which are mounted by the operating system. These multiple file systems or “file sets” are implemented following the Open Systems Foundation’s specification of the “Distributed File System” (DFS). Since

⁸ JFS refers to it as the “aggregate file system”, in VxFS it is called “structural file system”

all file systems exist inside the same disk partition or “aggregate”, they can share meta-data at the data unit sub-level, for example the block allocation bitmap.

E. Journaling

Journaling introduces the concept of atomic database transactions to file system operations. The operating system keeps a disk journal of recent transactions.

The disk journal is being “replayed” when recovering from a system crash, to verify which transactions were committed to the file system and which ones still have to be executed. This means that recovery time is proportional to the size of the journal, whereas traditional recovery has to analyze all meta-data in a file system for inconsistencies, and thus is proportional to the size of the file system.

Depending on its implementation, journaling can be performed on meta-data operations only, or on both meta-data and actual-data.

From a forensic point of view, the disk journal by its very nature is a very interesting source of information. It may reveal attempts of data erasure or it may provide multiple generations of content of a single meta-data container, which has been reused multiple times. For example, listing 4 shows the disk journal entry⁹ for the creation of the directory entry for the 64KB sample file from listing 2.

```
logrec d 16 Logaddr= x 30c4 Nextaddr= x 3058 Backchain = x 3058
LOG_REDOPAGE (type = d 2048) logtid = d 1
data length = d 72 fileset = d 16 inode = d 2 (x 2)
type = d 20 REDOPAGE:BTROOT_DTREE
l2linesize = d 5 pxd length = d 1 phys offset = x 1c (d 28)
0x80689e0 00000000 00000000 00000000 00000000 .....
0x80689f0 83030303 02000000 01020400 00000000 .....
0x8068a00 00000100 06000000 FF093600 34004B00 .....6.4.K.
0x8068a10 66006900 6C006500 2D003200 6D006500 f.i.l.e.-2.m.e.
0x8068a20 05000000 02000100 .....
```

Listing 4: JFS disk journal entry

The log record shows that inode 2 was modified. By convention, this is the inode for a file system’s root directory. The underlined hex dump translates into $i=6, n=-1, l=9$, meaning that a name which is 9 characters long points to inode number 6. The following hex codes translate into the filename **64Kfile-2** which was placed in the root directory of this file system. This example shows, that in general a parser can be implemented that translates the contents of the disk journal into a presentation format which is compatible with the TSK forensic toolkit, for example in a time-line similar to the output of the “mactime” program.

F. Generalized Meta-Data Presentation

Looking back at the concept of meta-data categories, one observes that JFS and FFS store meta-data in very different

⁹ This cleartext log representation was created using the jfs_logdump utility provided by IBM.

data structures at the data unit sub-layer, the inode sub-layer and the human interface sub-layer.

In many cases, a forensic analyst may not be too familiar with the most intrinsic implementation details of the individual file system he is working with. But he may still want to know whether the data unit he is looking at is actual data or meta-data. The fact that advanced file systems store a lot of meta-data in files could thus lead to misinterpretations.

Therefore this paper proposes four generic presentation views for the four functional sub-layers inside a file system. Using these views, a digital forensic media analysis tool can provide consistent analysis across different file systems.

1) File system sub-layer

The file system sub-layer views the whole file system as actual data, barring a relatively small portion of meta-data containing general size and state information about the file system. In case of JFS, VxFS and similar file systems, meta-data in this view would encompass general size and state information about one or more file systems and the "meta file system".

2) Data unit sub-layer

The data unit sub-layer presents addressable data units as actual data. The meta-data can be generalized to allocation management data structures which may contain data unit allocation bitmaps and/or lists of unallocated or partially allocated data units, bad data units etc.

3) Inode sub-layer

The inode sub-layer contains as actual data the content of an arbitrary file as a continuous stream of bytes mapped to one or more data units. The meta-data on this sub-layer is comprised of file properties visible to the user i.e. timestamps, file size and type as well as access control information. Other meta-data like the inode allocation status is kept private inside the sub-layer.

4) Human Interface Sub-layer

On the human interface sub-layer, the content of regular files¹⁰ is interpreted as actual data, whereas directories containing file or subdirectory names are considered meta-data. Also, symbolic links in general can be seen as meta-data on this sub-layer.

These generalized presentation views can also be used to describe the nature of special cases like

- "fast" symbolic links, where the link target is stored inside an inode, thus combining inode sub-layer and human interface sub-layer meta-data in the single data unit that holds that particular inode

- inline files, where the file content is stored inside the inode, thereby combining inode sub-layer meta-data and actual data in a single data unit.

IV. CONCLUSION

The prototype implementation of JFS support for TSK has shown that even though JFS uses very dynamic data structures, a substantial amount of forensically relevant information can be recovered from the file system. This includes the "undeletion" of files, provided that the corresponding data units have not been reused elsewhere in the file system.

Given the many different ways in which file systems implement the storage of meta-data and actual data, a generalized presentation scheme structured into four sub-layers is suggested for forensic toolkits in general.

Future work on the JFS support for TSK should include long-term analysis of meta-data volatility, including a tool to identify unallocated extents of meta-data and a disk journal parsing utility.

V. REFERENCES

- [1] McKusick, M., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX", *ACM Transactions on Computer Systems* Vol. 2, No. 3, August 1984, 181-197.
- [2] Pate, S. D. "UNIX Files Systems, Evolution, Design and Implementation", *Wiley Publishing*, 2003, 189-212
- [3] Best, S., Gordon, D., Haddad, I. "IBM's Journaled File System", *Linux Journal*, Vol. 2003, Issue 105, January 2003.
- [4] Best, S., Kleikamp, D., "JFS Layout", *IBM developerWorks*, May 2000, Available at: www.ibm.com/developerworks/opensource/jfs.
- [5] Sweeney, A., Doucette, D. et al, "Scalability in the XFS File System", *Proceedings of the 1996 Usenix Annual Technical Conference*, San Diego, Cal., January 1996.
- [6] Mostel, J., et al, "Porting the SGI XFS File System to Linux", *Proceedings of the 2000 Usenix Annual Technical Conference*, San Diego, Cal., June 2000.
- [7] MacDonald, J., Reiser, H., Zarochentcev, A., "Reiser4 transaction design document", April 2002, Available at: www.namesys.com/txn-doc.html.
- [8] Tweedie, S., "Ext3 Journaling File System", *Presentation at the Ottawa Linux Symposium*, 2000.
- [9] Carrier, B. "An Investigator's Guide to File System Internals", *FIRST Conference on Computer Security, Incident Handling & Response*, June 2002, Available at: www.first.org/events/progconf/2002/d1-02-carrier-slides.pdf
- [10] Carrier, B. "Defining Digital Forensic Examination and Analysis Tools using Abstraction Layers", *International Journal of Digital Evidence*, Volume 1, Issue 4, Winter 2003.
- [11] Bach, M., "The Design of the UNIX Operating System", Prentice-Hall Software Series, 1986
- [12] Venema, W., Farmer, D., Series of articles starting with "Forensic Computer Analysis: An Introduction", *Dr. Dobbs Journal*, September 2000
- [13] Farmer, D., Venema, D. "Forensic Discovery", *FIRST Conference on Computer Security, Incident Handling & Response*, June 2002, Available at: www.first.org/events/progconf/2002/d1-01-farmer-slides.pdf
- [14] Carrier, B. "Performing an Autopsy examination on FFS and ext2fs partition images", *SANSFIRE 2001 Conference*, August 2001, Available at: www.cerias.purdue.edu/homes/carrier/forensics/docs/autopsy_sansfire2001.pdf

¹⁰ Depending on the type of operating systems, a number of additional file types exist, such as FIFOs, Sockets, block and character device files. Their classification into data or meta-data needs to be discussed in more detail.