Digital Investigation 20 (2017) S83–S91

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2017 Europe — Proceedings of the Fourth Annual DFRWS Europe

AFEIC: Advanced forensic Ext4 inode carving

Andreas Dewald [a, *], Sabine Seufert [b]

[a] ERNW Research GmbH, Heidelberg, Germany
[b] Basys GmbH, Erlangen, Germany

ARTICLE INFO

Article history:
Received 26 January 2017
Accepted 26 January 2017

Keywords:
Digital forensics
Ext4 file system
Data recovery
Open source
Tool

ABSTRACT

In forensic computing, especially in the field of postmortem file system forensics, the reconstruction of lost or deleted files plays a major role. The techniques that can be applied to this end strongly depend on the specifics of the file system in question. Various file systems are already well-investigated, such as FAT16/32, NTFS for Microsoft Windows systems and Ext2/3 as the most relevant file system for Linux systems. There also exist tools, such as the famous Sleuthkit (Carrier), that provide file recovery features for those file systems by interpreting the file system internal data structures. In case of an Ext file system, the interpretation of the so-called superblock is essential to interpret the data. The Ext4 file system can mainly be analyzed with the tools and techniques that have been developed for its predecessor Ext3, because most principles and internal structures remained unchanged. However, a few innovations have been implemented that have to be considered for file recovery. In this paper, we investigate those changes with respect to forensic file recovery and develop a novel approach to identify files in an Ext4 file system even in cases where the superblock is corrupted or overwritten, e.g. because of a re-formatting of the volume. Our approach applies heuristic search patterns for utilizing methods of file carving and combines them with metadata analysis. We implemented our approach as a proof of concept and integrated it into the Sleuthkit framework.

Introduction

Data reconstruction plays an important role in the field of hard disk forensics (Casey, 2011) and it is specific to the used file system (Carrier, 2005). The Ext file system family is encounter as the standard file system on Linux and Android systems (Fairbanks et al., 2010). This paper illustrates an approach that enables reconstructing data without information from the superblock or the group descriptor table of the Ext4 file system.

Motivation

The Ext4 file system is a widely used file system, which is nowadays not only standard among Linux distributions, but is also used on mobile devices (Fairbanks et al., 2010). Ext4 and its predecessors save the metadata in the so-called superblock or the group descriptor table. Without these metadata structures it is difficult to interpret the file system correctly and to reconstruct the data. Of course, the remains the option of file carving, which however will not be able to recover file system metadata and on the other hand (besides specific techniques for some specific file types) is not able to cope with file fragmentation. The aim of this work is to reconstruct data without using metadata structures even in the case of overwriting or modifying the superblock through, for instance, overformatting. For previous Ext versions there are approaches which use the available contents of the metadata structures (Pomeranz). For example, on the Ext3 file system indirect block pointers in inodes – data structures where metadata is to be found on individual files – are saved on content data blocks. By dereferencing these block pointers, it is possible to reconstruct file contents. Referencing to the data contents on the Ext4 file system is handled differently, hence other techniques become necessary. Such a technique is introduced in this paper.

Contributions

This paper introduces an approach which was developed for finding and recovering files from an Ext4 file system. Moreover, the central metadata structures of the file system, such as the superblock and the group descriptor table, do not need to be available for our approach to work. Thus, a possible use case for our approach is

* Corresponding author.
E-mail address: research@andreasdewald.de (A. Dewald).

http://dx.doi.org/10.1016/j.diin.2017.01.003

when parts of the hard disk are overwritten or overformatted resulting in loss of the original metadata structures. In order to reconstruct files on the file system, we use search patterns, as it is known from file carving. However, instead of carving for specific file types, we carve for inodes. As there are no real magic bytes for an inode that can be used for simple carving, we build more complex patterns that identify valid inodes, what to the best of our knowledge has not been done so far. Carved potential inodes are then analyzed and the according files are recovered. By this means, our approach combines techniques form both, file carving, and metadata analysis. This way, not only the file content, but also the original file name and path can be reconstructed. We implemented our approach as a module for the Sleuthkit (Carrier), which is released open source along with this paper (Dewald and Seufert, 2017), where usage and configuration information is included, too.

Outline

After we discussed related work in the next section, in Section Ext4 file system novelties, we explain the novelties of Ext4 compared to Ext3. Section Methodology explains the methodology and implementation of our approach, which is thoroughly evaluated in Section Evaluation. We conclude our paper in Section Conclusion.

Related work

Brian Carrier (2005) describes in his book different partition and file systems. Carrier introduces different methods and tools to support the forensic analysis of the different file systems. The Sleuthkit (Carrier) is one of his developments, which provides various command line tools for digital forensics. On the one hand, we complement the work of Carrier, by highlighting the novelties in Ext4, and on the other hand, we implement a prototype of our introduced approach for Ext4 analysis as a plugin for the Sleuthkit Framework.

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

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

The work of Pomeranz illustrates an approach to data recovery on Ext2 and Ext3 file systems that enables the recovery of user data by using indirect block pointers. The author exploits the fact that, typically, the first 48 KiB of a file content are not highly fragmented. Consequently, the first 12 block pointers are usually sequentially numbered, which similar to our approach applies some kind of specific carving. However, this particular search pattern cannot be applied to Ext4 file systems because normally (as we explain later) extent structures are used instead of indirect block pointers in order to reference file content. Nevertheless, the procedure used in our approach is similar to the one mentioned above.

Ext4 file system novelties

In this section, we summarize the relevant information about Ext4, as they are given in Ext4 disk layout (2016). The general layout of Ext4 is very similar to Ext3, but has changed in some ways that we want to focus on now. The Ext layout, in general, is based on sequential blocks of 1024, 2048 or 4096 bytes that are numbered and grouped together in block groups.

Each block group contains metadata that documents its inner structure. The general layout of all block groups is identical and is illustrated in Fig. 1. The superblock contains many essential metadata of the file system, such as the number and size of blocks, number of inodes and reserved blocks, for example. The following group descriptor table contains one group descriptor per block group in the file system and the block bitmap stores the free/used state of each block in the block group in a single bit each. Similarly, the inode bitmap stores the free/used state of each inode (entry) in the inode table. The rest of the block group consists of consecutive data blocks that are used to store data.

Inodes

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

In Ext4, for the sake of compatibility with prior versions, only few changes to the inode structure have been implemented. For example, some of the formerly unused space has been used to introduce new attributes, as shown in Table 1 (To recall the full original structure of Ext3 inodes, refer to Table A.9 in Appendix A).

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

The inode of the example-file on the left side of the figure has 60 bytes to store its extent structure. The size of one extent entry is 12 bytes, thus there can be 5 extent entries stored directly. Each extent structure starts with an extent header with a size of 12 bytes as well. For completeness, the detailed structure of a extent header is shown in Table A.6 in Appendix A. The extent header is followed by extent

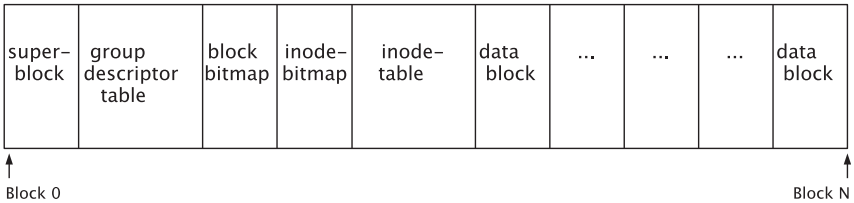

Fig. 1. General block group layout.

Table 1
Additional inode attributes in Ext4.

Offset	Length	Description
36	4	OS specific
40	60	Indirect block pointers or extent Data structure
		…
116	12	OS specific
128	2	Additional size for inode
130	2	Upper 16 bit of inode checksum
132	4	Additional bits ctime
136	4	Additional bits mtime
140	4	Additional bits ctime
144	4	Cr(eation) time
148	4	Additional bits crtime
152	4	Upper 32 bit version number

nodes, that can be either inner nodes of the extent tree (extent indexes) and point to other extent headers, or leaves (extents) that point to data block runs. For completeness, the structure of those entries is shown in Appendix A in Tables A.7 and A.8.

Flex-groups

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

Methodology

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

In order to provide applicability in various use cases, our Sleuthkit module that implements our approach supports two distinct recovery modes: The so called *contentdata mode*, that exclusively recovers the content of regular files, for which only the block size of the Ext4 file system must be provided or detected correctly. And the *metadata mode*, that requires more Ext4 parameters because the necessary inode numbers need to be calculated by the module. Then, file names with their complete file paths can be recovered using directory entries in this mode. The presented method can be divided into the following phases that are presented in the subsequent sections:

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

Initialization

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

- Offset
- File system size
- Block size
- Inode size
- Inode ratio
- Flex group size
- 64 bit mode
- Sparse superblock
- Number of blocks per block group
- Number of blocks and block groups in the file system
- Number of inodes per block group
- Space for growing group descriptor table

The default values for new Ext4 file systems used by mkfs are shown in Table A.10 in Appendix A. Furthermore, there are relevant

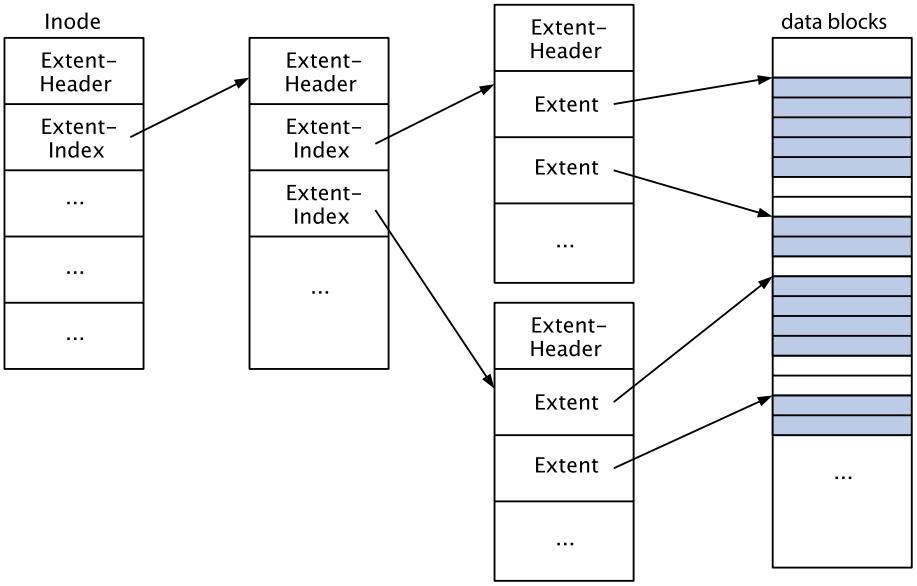

Fig. 2. Extent data structure of Ext4.

values with mkfs defaults that are independent of the file system size. Those are:

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

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

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

Inode carving

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

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

1. $\texttt{mtime} \leq \texttt{ctime}$
2. $\texttt{dtime} = 0 \vee (\texttt{dtime} \geq \texttt{mtime} \wedge \texttt{dtime} \geq \texttt{ctime})$
3. mtime, ctime and dtime must be a valid timestamp

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

Directory tree

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

Directory entries are searched linearly whereby directories not starting with entries for '.' and '..' can be discarded. The inode number and file name of a directory entry is saved along with its parent inode number. This reference pattern constitutes a logical tree from which the complete file path can be deduced. However, inode numbers of regular files are not inherently known and the module must map physical inode addresses to inode numbers as follows: Using the physical address a of a potential inode, the corresponding inode number can be calculated with $\frac{a-s}{i}$, with s being the physical address of the beginning of the inode table and i the size of the inode. Any valid inode number is an integer. Since inode numbers start with '1', the calculation needs to be extended to $\frac{a-s}{i}+1$.

Since the length of the inode table must be a multiple of the inode size, the end of the inode table e can be calculated by $e = s + n_{i,BG} \cdot i$, with $n_{i,BG}$ being the number of inodes per block group. If the size of the flex groups n_f is greater than one, the end of an inode table in the beginning of a flex group can be calculated by $e = s + n_f \cdot n_{i,BG} \cdot i$. With that in mind, the criterion for a physical address of a valid inode can be described by $s \leq a < s + n_f \cdot n_{i,BG} \cdot i$.

As already mentioned, physical addresses a of inodes within an inode table must fulfill $\left(\frac{a-s}{i}+1\right) \in \mathbb{N}$. Considering flex groups, the beginning of an inode table s can only be situated in the first block group of a flex group. The block group bg_a of the address a must be determined with $bg_a = \left\lfloor \frac{a}{b \cdot n_{BG}} \right\rfloor$, b being the block size and n_{BG} the number of blocks per block group. Consequently, another condition for a valid inode within an inode table is $\frac{bg_a}{n_F} \in \mathbb{N}_0$.

The beginning address of the computed block group can be determined by $bg_a \cdot b \cdot n_{BG}$. Between the beginning of an inode table and this address, there can be a copy of the superblock, the group descriptor table, its growth blocks and the inode and block bitmaps. Depending on the flex group size n_F the bitmaps contribute an offset $o_i = 2 \cdot n_F$. The combined size of the superblock, the group descriptor table and its growth space equates to $1 + \left\lceil \frac{d \cdot n_G \cdot 1024}{b} \right\rceil$ but is limited to 1024 blocks, with d being the size of a group descriptor and n_G the count of block groups in the file system. This leads to $o_s = \min\left\{1024, 1 + \left\lceil \frac{d \cdot n_G \cdot 1024}{b} \right\rceil\right\}$.

The first 1024 bytes of the file system are reserved independently of the block size. However, if the block size is 1024 bytes, that reserved area makes up one whole block shifting all addresses by one block. This offset is represented by o_r. The beginning of the inode table can thus be computed by Equation (1).

$$s = (bg_a \cdot n_{BG} + o_s + o_i + o_r) \cdot b \tag{1}$$

With the additional conditions in $a \in [s, e[$ and $\frac{a}{i}+1 \in \mathbb{N}$ and $\frac{bg_a}{n_F} \in \mathbb{N}_0$, Equation (2) describes the mapping of an address to its inode number:

$$\begin{aligned} o_s &= \min\left\{1024, 1 + \left\lceil \frac{d \cdot n_G \cdot 1024}{b} \right\rceil\right\} \\ o_i &= 2 \cdot n_F \\ o_r &= \begin{cases} 1 & \text{if } b = 1024 \\ 0 & \text{otherwise} \end{cases} \\ bg_a &= \left\lfloor \frac{a}{b \cdot n_{BG}} \right\rfloor \\ f(a) &= \left(\frac{a-s}{i} + n_{i,BG} \cdot bg_a + 1\right) \end{aligned} \tag{2}$$

Using these formulas the mapping from physical addresses to inode numbers can be performed. Together with the information gathered from the directory tree, file names can be associated to inodes. Hence, the metadata mode is able to reconstruct the whole directory structure of the file system. If a directory is irreparable, its children's file paths cannot be reconstructed. Then, their content is saved in files named after their inode addresses.

The contentdata mode cannot map physical addresses to inode numbers due to the lack of necessary Ext4 parameters. This fact

excludes the directory tree phase from the contentdata mode. All recovered files are named after their inode address and saved in a flat hierarchy.

Regular files

Whether or not the file path has been reconstructed for a given regular file, its content is now recovered. In Ext4 file systems, file content is spread across the volume in a way managed by so-called extents, whereas Ext2/3 file systems use indirect block pointers. The presented module does not support the latter, which is already well-covered by older tools, and ignores inline files and directories.

Since the file system journal can contain copies of existing inodes and because the contentdata mode does not consider inode table boundaries, duplicate inodes with different physical addresses can be found. The lack of inode numbers necessitates an inode equality criterion with which it can be assured that two compared inodes are the same. Therefore, the file size and the extent structures of the inodes are compared, as they identify same content.

Files without content

Similarly to the directory tree phase, this phase is optional and builds upon the results of the directory tree and content data phases and is thus only available in the metadata mode. Files found in the directory tree phase but not in the inode carving phase cannot be recovered with respect to their content but indeed to their file name. Hence, they can be written as empty files with their original name and complete file path. Examples include symbolic links or device files.

Evaluation

In the previous section, we introduced an approach for reconstructing inodes based on search patterns of different inode attributes. All inodes that match the patterns are considered potential inodes and using their extent tree, the file content can be reconstructed. In this section, we evaluate the quality of each of those search patterns, as well as the completeness and correctness of our tool.

Dataset

To evaluate our approach and implementation, we built a dataset with different hard disk images as follows. We used different file system sizes to cover the different configurations that occur when formatting volumes of different sizes with default values. Further, we built test cases where we deleted specific files or changed file system parameters that might influence the success of the approach. The entire list is shown in Appendix in Table B.11. For the images where we deleted files, we distinguished the two cases where files have been deleted directly vs. moved to trash first. Further, we produced a case where we deleted all files and copied new ones on the file system to check whether the former files can be recovered. We also compared Ext4 file systems with enabled and disabled journal and deleted files. To create a more realistic example image, we created one that contains an entire Ubuntu Linux installation with various files that have been moved, deleted and modified and also containing symbolic links, device files and other non-regular file types to cover a broad spectrum. Finally, we also created images where we overformatted the existing Ext4 file system with NTFS or again Ext4. For each of those cases, we compared both, standard and quick formatting. Since the size of an image is not as important as the configuration via mkfs, the dataset contains mainly small images.

Search patterns and selectivity

In the first step, we want to check, how well the different patterns perform. For this test, we chose the image with installed Ubuntu and various cases. We used the Sleuthkit tool fsstat to provide a ground truth about the number of reserved inodes. Considering the first 10 inodes to be reserved, there remained 201.269 files, of which we checked how many are regular files and directories. Table 2 summarizes the number of identified potential inodes for each pattern, which we discuss in the following sections.

Each pattern is tested on each physical address and can accept or decline this address as potential inode. Accepted addresses that lie within an area of an inode table at a valid offset and can be reconstructed successfully are listed as hits in the table. The reference for this is the total number of files in the file system mentioned above. Structures that might be a valid inode, but reside outside an inode table are called table misses. Those could be false positives, but could also be copies of inodes for example in the file system journal. Similarly, address misses are potential inode addresses that do not lie at a 128 byte inode boundary.

We do not reject both kind of misses in the first place, as they do not necessarily mean false positives. The sum of hits, t-misses, and a-misses is the number of accepted addresses for the pattern and the selectivity of each pattern is this number compared to the total number of possible addresses. In the next sections, we discuss the results for each pattern in detail.

Access rights

While for access rights, there is no illegal combination, a search pattern can either be defined to search for files with specific rights, or to cover the most common access right combinations. The pattern can be freely adjusted in the configuration file of our tool, and the selectivity of the pattern strongly depends on this choice. For this experiment, we chose the most common combinations of access rights in a usual Linux system as shown in Table 3.

This pattern accepted only 142.855 valid inodes out of 201.269, which however contained almost all regular files and folders.

Timestamps

The timestamp pattern consists of two parts: On the one hand, we validate the inner consistency of the different timestamps as explained in Section Inode carving, and on the other hand, the investigator might configure a relevant timeframe for his case. For this experiment we chose the timeframe from 2015-01-01 00:00:00 GMT to 2016-01-01 00:00:00 GMT, in which the system was set up and all actions have been performed. Amongst the potential inodes have been 7.140 false positives, so that the number of hits as shown in Table 2 reduces to 65.170.

With respect to only the inner consistency, we obtained 209.481 hits, 5.708.816 t-misses and 1.416.653.929 a-misses. 8.213 of the hits have been identified as false positives, however, all other valid 201.264 have been found.

Table 2
Search patterns selectivity. 'k' stands for thousand, 'M' for million and 'G' for billion; t-miss are table misses and a-miss are address misses. Hits compared to all misses is the selectivity.

Pattern	Hits	t-miss	a-miss	Select
access rights	151 k	1.02 M	99.9 M	0.6%
time interval	72.3 k	207 k	238 k	0.003%
time consist	209 k	5.71 M	1.42 G	8.89%
link count	201 k	29.8 M	7.59 G	47.6%
extent flag	166 k	13.4 M	3.28 G	20.6%
extent header	166 k	515 k	53.2 k	0.004%
file type	151 k	4.51 M	905 M	5.7%

Table 3
Access right patterns chosen for the experiment.

Owner	Group	All
r - -	r - -	r - -
rw -	r - -	r - -
rw -	rw -	r - -
rwx	r - x	r - x
rwx	rwx	r - x
rwx	- - -	- - -
rw -	- - -	- - -
rwx	r - x	- - -
rwx	rwx	- - -

Number of hard links

We check if the internal number of hard link counter of a potential inode is larger than 0. Besides the special inodes 7 and 8, which have to be counted as false positives, all the 201.269 are accepted by this check.

Extent flag and header

Although they are very closely related, we evaluated the pattern for the extent flag and extent headers separately. The extent flag as a single bit stores only little information compared to the 2 byte magic number of an extent header. Thus it is not very surprising that its selectivity is very high. The extent header pattern is the only pattern, that produces far more table misses that address misses. This is due to the fact that not only every root node of an extent tree stores an extent header, but every inner node, too. Thus this pattern also matches various content data blocks. However, all regular files and folder have been accepted by this pattern, along with 8.074 false positives. When deleting a file in Ext4, the extent references are getting zeroed, but the extent flag and the extent header is not affected, so that inodes of deleted files are accepted by this pattern, too.

File type

As in our experiment, we restricted ourself to only regular files and folders, we also adjusted the file type pattern accordingly. Besides 8.068 false positives, this pattern matches on all 142.919 inodes that we searched for. We want to recall that the type field of 4 bit stores 7 different valid file types, so that there remain 9 illegal values that allow to exclude false positives even without restricting reconstruction to specific file types.

Pattern combination

To summarize, the 3 most restrictive patterns in our experiments are the extent headers, timestamp intervals, and access rights. Because the two last mentioned include some semantic filtering (that might be very valuable in a real case, but need to be adjusted to the specific case), we did not include them in the next experiments to verify the correctness of our tool.

Completeness and correctness

For the evaluation of completeness and correctness of our tool, we tested and compared both operation modes of our tool. To identify reconstructed files and verify their correct recovery, we calculated both, MD5 and SHA256 hashes and compared them to the list of files that have been originally placed on the hard disk. In order to be considered correctly recovered, in the contentdata mode, the files need to be reconstructed with the correct hash values, while in the metadata mode, besides the hash values, the file name and path have to be correct, too.

We evaluated two images of two different categories: First, we evaluated our tool on the real case images to verify correctness and completeness by comparing the results of our tool the known ground truth. Second, we wanted to evaluate, if we further can recover files from file systems that have been overformatted with our approach. To this end, we evaluated our tool on such cases.

In the next two sections, we present the results of those two categories. Table B.12 in Appendix B shows detailed information about the evaluation dataset with the total number of inodes, files and directories to provide a ground truth on what potentially could be recovered, for reference.

Real world cases

In case of the `Ubuntu_Pics_complete.img`, the metadata mode reconstructed empty files with the correct file names and paths for all non-regular files, all files and folders that have been placed correctly, as well as the `/lost+found` folder created by the OS. In the contentdata mode, folders are not reconstructed, but all the placed files have been recovered correctly (with their inode address as file name).

For the images `floppy.img`, `small.img`, and `default.img`, our tools reconstructed all regular files and in the contentdata mode in addition the journal. Further, files have been recovered from the journal, which we count as false positives, here.

The image `floppy_deleted.img` has been created from `floppy.img` by sending all files on the image to trash. Thus, the trash contains all the original files, as well as metadata files that document the original paths and times of deletion, which all have been recovered by our tool.

We went one step further in the image `floppy_deletedTrash.img`, that was created from the previous one by emptying the trash. From the OS perspective, there is no file left on the file system. In metadata mode, our tool was only able to recover the empty root and trash folder. However, in the contentdata mode, the tool reconstructed all the files correctly (of course, without original file names) from inodes found in the journal, along with 15 duplicates of some files.

In the next step, we deleted all files from all previously described images (via sending to trash and emptying trash). We then copied totally different files to the resulting new images. The metadata mode recovered all newly written files correctly from all images. In addition, from this image `small_newFiles.img`, the contentdata mode was able to recover one file form the original file system, whose file content has not been overwritten by new files, and its inode resided in the old journal.

Finally, we evaluated the case `Ubuntu_Pics_complete.img`, where the two modes of our tool provided very different results, because the journal contained lots of copies of changed inodes, which led to lots of additional files that could have been recovered in the contentdata mode. On the other hand, the contentdata mode does not reconstruct multiple empty files (since they look identical), but the default Ubuntu installation contained lots of empty files that were not recovered in this mode.

Overformatted file systems

The following test cases are meant to evaluate, if it is possible to reconstruct files from Ext4 file systems that have been overformatted in different ways. As basis for this scenarios, we used the image `small.img`, which provides an upper boundary of what files can be recovered.

First, we overformatted the original file system with Ext4 (`small_newExt4.img`) and NTFS (`small_NTFS.img`) in full formatting mode. In those cases, the blocks have been zeroed and we were not able to recover any files. However, Ext4 as well as NTFS provide an option for so called fast formatting, where no blocks are zeroed while formatting and only newly used blocks are overwritten. Table 4 lists the number of identified inodes and files from the fast overformatted file systems and compared results from both operation modes.

Table 4
Number of found inodes after selection of the overformatted dataset.

Image name and mode	All inodes	Regularfiles	Folders
small_fastExt4.img	**128**	**121**	**7**
accepted		171	77
metadata mode		118	5
contentdata mode		119	0
small_fastdiffExt4.img	**128**	**121**	**7**
accepted		228	90
metadata mode		0	0
contentdata mode		125	0
small_fastNTFS.img	**128**	**121**	**7**
accepted		348	94
metadata mode		121	7
contentdata mode		124	0

Table 5
Runtime for analyzing the image and recovering the files.

Image name	Amount	Size	Time
small.img			
metadata mode	128	128.5 MB	5.6s
contentdata mode	124	134.1 MB	5.4s
default.img			
metadata mode	5102	5.3 GB	2m 36s
contentdata mode	4868	5.4 GB	2m 28s

The image `small_fastExt4.img` has been created by using the same default parameters when overformatting the volume as used for the original formatting. In this case (and due to this parametrization), in the metadata mode our tool was able to recover all but 3 of the original files and all but 2 folders. The contentdata mode additionally recovers the journal from inode 8 as a file.

In case of the image `small_fastdiffExt4.img`, where different non-default parameters have been used for reformatting, none of the original inode table resided in areas that have not been overwritten, and thus the metadata mode was not able to recover files. However, the journal has not been overwritten, so that in the contentdata mode, our tool was able to recover 125 files, of which 52 are totally unmodified. The others have been partly overwritten.

In the image `small_fastNTFS.img`, the structures that have been created for the NTFS file system did not overwrite the area of the original inode tables, so that all inodes have been intact and both operation modes were able to recover all the original files, of which only some have modified content, where some blocks have been used by the new file system.

Runtime performance

Table 5 shows a short runtime evaluation, that shows that both implemented methods show similar speed, while the contentdata mode is slightly faster. Both approaches take roughly about 30 s per GigaByte image size. As all entire image carving approaches the runtime is linear dependent from the image size.

Conclusion

In the previous sections, we presented an approach which was developed for the reconstruction of files on Ext4 file systems. This approach does not require the extraction of the most central metadata structures, such as superblocks or group descriptor tables. Instead, inodes are identified using a search pattern by combining a file carving method and a type of metadata analysis followed by the reconstruction of the corresponding files. The proof of concept implementation resulted in a fully working module for the Sleuthkit framework released open source along with this paper (Dewald and Seufert, 2017).

Contribution

It can be concluded from the findings of this work that, by using the described approach, files can be reconstructed from Ext4 file systems even without knowledge about the particular structure of the file system. By separating the inode search from the inode reconstruction, it is possible to find inodes when there is no given information about the file system layout available. Neither the file system size, nor the examined Ext4 file system's offset to the begin of the partition are necessary in order to locate inodes (and thus gain first information about potential files).

The effectivity of the different patterns vary depending on the use case. For instance, deleted files cannot be recovered from a hard disk if the file system does not use a journal. Similarly, the amount of restorable files from overformatted disks depends on the specific circumstances, such as the file system used for overformatting and the formatting settings. In the content data mode, there is no possibility to distinct recovered inodes from inode tables and the file system journal, which leads to duplicate files. However those can be successfully detected and removed. This way, deleted files can be reconstructed, if a copy of the corresponding inode is found in the journal. Reconstruction in metadata mode discards inode hits that are not within an inode table, thus recovery of deleted files is not possible in this mode, but this approach leads to less false positives and is able to recover file names and directory structures of the files.

Limitations

There are some limitations related to our approach, but also some that are related to the implementation and can be addressed by future development. One of these limitations is the dependency on the standard formatting of the Ext4 file system. This can be addressed by manually providing the parameters that differ from a standard configuration. However, we were not able to observe such a case in the wild, as normally file systems are created by the standard operating system tools, that to not change such parameters as the number of inodes per inode table or the block size to other ones as the default values.

One other limitation is that deleted files can only be reconstructed when the content data mode is used and only on a file system that keeps a journal. Even in this case, only deleted files that have their last change documented in the journal can be recovered.

Other limitations result not only from the described approach, but also from the developed forensic tool. For instance, the file reconstruction is limited to regular files. By applying the metadata mode directories can indeed be read and interpreted, however a file system cannot be completely reconstructed because other file types are not supported, e.g. symbolic links or device files.

Future work

In future work, we plan to improve the used search patterns and maybe add further ones that might be helpful in very specific cases. In order to provide backwards compatibility to Ext2/3 the development new search patterns is required, too. Due to the fact that the extent header pattern is one of the most effective patterns, an alternative must be found for Ext2/3, where no extents are used.

Likewise, further options for saving content data for reconstruction should be considered, to maintain compatibility with systems migrating from Ext3 to Ext4. Not only indirect block pointers that emerge after the migration from Ext2/3 to Ext4 should be taken into account, but also inline files and directories.

With our current implementation it is not possible to combine metadata and content data modes. Certainly, it would be desirable

if outcomes could be generated in a single program run, benefiting from the advantages of both modes, which currently needs to be done manually by running the module twice. Thereby, it would be possible to both generate the directory hierarchy through the metadata mode and reconstruct files, whose inodes can be found exclusively in the journal.

Appendix A. Ext structures

Table A.6
Ext4 Extent header in extent tree.

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

Table A.7
Ext4 Extent entry for inner node in the extent tree.

Offset	Len.	Description
0	4	ID of first block of own tree-part from beginning of the file
4	4	Lower 32 Bit of block address of child entry
8	2	Upper 16 Bit of block address of child entry
10	2	Unused

Table A.8
Ext4 Extent entry for a leaf in extent tree.

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

Table A.9
Structure of Inodes in Ext2/3, offsets and lengths given in bytes.

Offset	Len.	Description
0	2	Mode (file type and access)
2	2	Lower 16 bit user-ID
4	4	Lower 32 bit file size
8	4	Atime
12	4	Ctime
16	4	Mtime
20	4	Dtime
24	2	Lower 16 bit group ID
26	2	Link count
28	4	Sector count
32	4	Flags
36	4	Unused
40	48	12 direct block pointers
88	4	1 indirect block pointer
92	4	1 double indirect block pointer
96	4	1 triple indirect block pointer
100	4	Generation number
104	4	Extended attributes (file-ACL)
108	4	Upper 32 bits file size/directory-ACL
112	4	Fragment block address
116	1	Fragment index in block
117	1	Fragment size
118	2	Unused
120	2	Upper 16 bit user-ID
122	2	Upper 16 bit group-ID
124	4	Unused

Table A.10
Default parameters as chosen by mkfs for Ext4.

mkfs Type	Parameter	Value
floppy (to 3 MiB)	block size	1024 Byte
	inode size	128 Byte
	inode ratio	8192
small (to 512 MiB)	block size	1024 Byte
	inode size	128 Byte
	inode ratio	4096
default (to 4 TiB)	block size	4096 Byte
	inode size	256 Byte
	inode ratio	16,384
big (to 16 TiB)	block size	4096 Byte
	inode size	256 Byte
	inode ratio	32,768
huge (from 16 TiB)	block size	4096 Byte
	inode size	256 Byte
	inode ratio	65,536

Appendix B. Detailed evaluation results

Table B.11
Dataset specification.

Image name	Größe	mkfs-Type
floppy.img	2,3 MB	floppy
small.img	230 MB	small
default.img	12 GB	default
floppy_deleted.img	2,3 MB	floppy
floppy_deletedTrash.img	2,3 MB	floppy
floppy_newFiles.img	2,3 MB	floppy
small_newFiles.img	230 MB	small
default_newFiles.img	12 GB	default
default_sameFiles.img	12 GB	default
default_withoutJournal.img	6 GB	default
Ubuntu_Pics_complete.img	16,0 GB	default
small_newExt4.img	230 MB	small
small_diffExt4.img	230 MB	small
small_fastExt4.img	230 MB	small
small_fastdiffExt4.img	230 MB	small
small_NTFS.img	230 MB	small
small_fastNTFS.img	230 MB	small

Table B.12
Evaluation dataset details.

Image name and mode	All inodes	Regular files	Folders
floppy.img	**6**	**4**	**2**
metadata mode		4	2
contentdata mode		6	0
small.img	**128**	**121**	**7**
metadata mode		121	7
contentdata mode		124	0
default.img	**5.102**	**4.866**	**236**
metadata mode		4.866	236
contentdata mode		4.868	0
floppy_deleted.img	**13**	**8**	**5**
metadata mode		8	5
contentdata mode		9	0
floppy_deletedTrash.img	**2**	**0**	**2**
metadata mode		0	2
contentdata mode		9	0
floppy_newFiles.img	**28**	**25**	**3**
metadata mode		25	3
contentdata mode		27	0
small_newFiles.img	**349**	**340**	**9**
metadata mode		340	9
contentdata mode		355	0
default_newFiles.img	**9.062**	**7.773**	**1.289**
metadata mode		7.773	1.289
contentdata mode		7.664	0
default_withoutJournal.img	**2**	**0**	**2**
metadata mode		0	2
contentdata mode		0	0
Ubuntu_Pics_complete.img	**201.269**	**127.159**	**15.760**
metadata mode		127.159	15.760
contentdata mode		168.035	0

References

Carrier, B. The sleuth kit, TSK. http://www.sleuthkit.org/sleuthkit/ (Online).

Carrier, B., 2005. File System Forensic Analysis, vol. 3. Addison-Wesley Reading.

Casey, E., 2011. Digital Evidence and Computer Crime: Forensic Science, Computers and the Internet. Academic Press.

Craiger, P., 2005. Recovering digital evidence from linux systems. In: Advances in Digital Forensics. Springer, pp. 233–244.

Dewald, A., Seufert, S., 2017. Ext4 file recovery. (Accessed 5 January 2017), URL https://www1.cs.fau.de/content/ext4-file-recovery.

Ext4 disk layout, 2016. (Accessed 26 December 2016) URL https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

Fairbanks, K.D., 2012. An analysis of ext4 for digital forensics. Digit. Investig. 9, S118–S130.

Fairbanks, K.D., Lee, C.P., Owen III, H.L., 2010. Forensic implications of ext4. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, ACM, p. 22.

Lee, S., Shon, T., 2014. Improved deleted file recovery technique for ext2/3 filesystem. J. Supercomput. 70 (1), 20–30.

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

Pomeranz, H. Ext3 file recovery via indirect blocks.