Forensic Science International: Digital Investigation 42 (2022) 301399

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

Ambiguous file system partitions

Janine Schneider*, Maximilian Eichhorn, Felix Freiling**

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

ARTICLE INFO

Article history:

Keywords:
Anti-forensics
Digital evidence
Digital forensics

ABSTRACT

We investigate the problem of creating *ambiguous file system partitions*, i.e., the possibility to have two fully functional file systems within a single file system partition. The problem is different from steganographic data hiding since there is no real distinction between content and cover data, and no translation process may be applied to the content data. Since typical file systems that occur in forensic analysis are usually unambiguous, ambiguous file system partitions may be useful corner cases in forensic tools and processes. We show that it is possible to create ambiguous file system partitions by integrating a guest file system into the structures of a host file system in two cases: We integrate a fully functional FAT32 into Ext3 and HFS+. In a third example we even integrate two guest file systems (HFS+ and FAT32) into a single Btrfs file system partition. We test common forensic tools on these examples and exhibit some deficiencies. Moreover, we develop a taxonomy of ambiguous file system partitions and argue that the existence of essential data at fixed positions still is a way to distinguish host from guest and so to heuristically reduce the ambiguity, without removing it completely.

1. Introduction

In the literature, the technique of *data hiding* usually refers to overlaying (secret) content data with (public) transport data such that the content data is not readily visible. Data hiding has been studied in different contexts, the main research direction being digital steganography. Digital photographic images are particularly good for hiding data because usually they are relatively large and the secret information can be added by slightly manipulating color values within an image (Wiseman, 2018) such that the visual impression to a human is unchanged.

Many techniques have been developed that can arguably hide data in a way such that it is impossible to recover the information without knowing the way the data was masked (Blunden, 2012; Wayner, 2002). Such techniques have also been applied to hide data in file systems. For example, a structure which can be used to hide data is the Master File Table (MFT) of New Technology File System (NTFS) which can be manipulated with the tool FragFS (Thompson and Monroe, 2006). Another example is the tool Slacker (Kennedy et al., 2011) which can be used to hide data in the slack space of NTFS. Similar techniques have been developed for other file systems as well. Göbel et al. (Göbel et al., 2019) identified several locations in the Apple File System (APFS) were data can be hidden. For example, they used the Inode entry versioning mechanism, which is managed by a counter, to hide data within the counter's byte range. They also adapted the exploitation of the nanoseconds part of timestamps already known from Ext4 (Göbel and Baier, 2018) and NTFS (Neuner et al., 2016) for APFS. Furthermore, they identified padding spots as a promising location for data hiding.

So, while data hiding techniques in file systems appear well-developed, the hidden data can only be actually used after applying a reverse transformation. In this paper we investigate whether it is possible to incorporate a fully functional file system within another file system such that both file systems can be used without applying any transformation of the data.

Achieving this would create truly ambiguous file system partitions, i.e., partitions in which two fully functional file systems coexist and for which it is not totally clear which file system is being primarily used. We wanted to determine if it was possible to construct such file system partitions and see how forensic analysis tools would react to the ambiguity. This substantially differs from the data hiding approaches sketched above, and is not to be confused with what is known as steganographic file systems (Anderson et al., 1998; McDonald and Kuhn, 2000; Neuner et al., 2016) that do not make any attempt to conceal that data hiding is happening.

* Corresponding author.
** Corresponding author.
E-mail addresses: janine.schneider@fau.de (J. Schneider), maximilian.eichhorn@fau.de (M. Eichhorn), felix.freiling@fau.de (F. Freiling).

https://doi.org/10.1016/j.fsidi.2022.301399

Ambiguous file system partitions are not only theoretically interesting. They potentially also pose a challenge to current forensic tools that usually depend on the assumption that basic data structures are reliable and that a partition contains exactly one file system. This is in line with Wundram et al. (2013) who created a malicious partition table loop where an extended partition entry pointed to the beginning of the partition table. They used several of these anti-forensic examples to prove that (advanced) forensic tool testing is as relevant as security testing because

> [f]orensic software has to fulfill highest quality criteria both in qualitative and quantitative ways (Wundram et al., 2013)

and that

> such forensic tools are a critical point in any investigation (Wundram et al., 2013).

Therefore, forensic tools must be careful when they rely on the correctness of the data in specific file system structures, especially in the case of ambiguous file system partitions.

1.1. Contributions

In our paper, we examine several different file systems regarding their potential for creating ambiguous file system partitions. We analyze NTFS, FAT32, Ext3, HFS+ and Btrfs and identify the file system structures that are best suited for hiding file system information. In addition, we develop a taxonomy of ambiguous file system partitions, which can be used to classify different combination approaches and evaluate their quality. We also present four file system combinations that show that it is possible to overlay different file systems and that it is possible to hide several "guest" file systems in the data structures of a "host" file system. Furthermore, we analyze three images containing file system combinations with four common open source and two commercial forensic tools and discuss the results. We show that the apparently simple forensic analysis of file systems can be non-trivial and that forensic tools are not prepared for ambiguous file system partitions.

Our goal is to make digital forensics more reliable by reporting potential tool limitations to enable their inclusion in future tool testing scenarios. We would also like to, once again, draw attention to the fact that even in presumably well understood areas of digital forensics there are still uncertainties and assumptions that need to be taken into account and considered in the forensic analysis of data.

1.2. Paper outline

We describe the general layout and the file system structures of prominent file systems in Section 2. The taxonomy is presented in Section 3 followed by the presentation of four example file system combinations in Section 4. In Section 5 we report on the evaluation performed and the results of the forensic analyses of the file system combinations using various forensics tools followed by a discussion about the detection of ambiguous file system partitions in Section 6. Section 7 concludes the paper.

2. Background

We now revisit the main structures of the file systems we use later in this paper. Readers familiar with this topic can skip this section at first reading. The file systems described here were selected based on their layout and properties. Important properties are offsets, number and simplicity of data structures and adaptability and manipulation ability of the structures and their position.

2.1. FAT32

The File Allocation Table (FAT) (Carrier, 2005) file system is a very simple file system developed by Microsoft. Nowadays the FAT file system is mainly used for flash cards and USB drives with low capacity. There are three different versions of the FAT file system: FAT12, FAT16 and FAT32. We will focus on FAT32. The names are referring to the size of the entries in the FAT. The FAT contains the files cluster addresses, the so called *cluster chains* (Carrier, 2005). The FAT is also used to mark clusters as damaged, which will then be omitted during allocation. Another important data structure is the *root directory* (Carrier, 2005), which contains *directory entries* (Carrier, 2005) for every single file and directory within the file system. Each directory entry contains the file or directory name, the size and the starting address. The starting address also refers to the corresponding FAT entry. These data structures as well as other relevant file system information can be found in the *boot sector* (Carrier, 2005). Some of the most relevant boot sector data are summarized in Table 1. Figure 1 shows the typical structure of a FAT32 files system and the location of the different data structures.

2.2. NTFS

The New Technology File System (NTFS) (Carrier, 2005) designed by Microsoft is the successor of the FAT file system and is much more complex. It is the default file system for Windows operating systems since Windows NT. The main advantage over the FAT file system is that NTFS is scalable because of growing and changing data structures and can be applied to storage devices of bigger size. In NTFS everything is a file, which means even system data are stored in files and the data area expands over the whole file system. As in FAT32 the first sector contains the boot sector, which is called *boot file* (Carrier, 2005) in NTFS. Some of the most

Table 1
Relevant FAT32 boot sector data.

Boot sector	
Byte	**Data**
11–12	Bytes per sector
14–15	Size of reserved area
16	Number of FATs
44–47	Root directory location
50–51	Backup boot sector

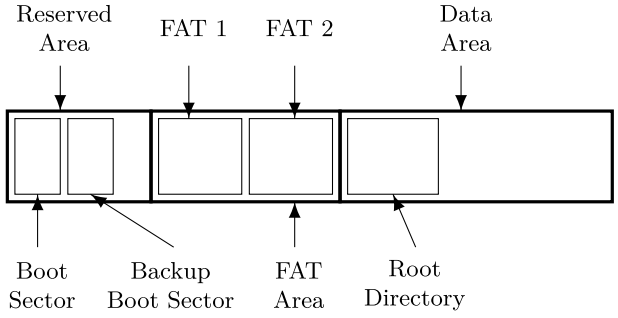

Figure 1. Typical FAT32 data structure layout. The boot sector is located at the beginning of the partition, usually followed by the backup boot sector. The file allocation table and a copy of it are located in the FAT area. This area is followed by the data area which contains the root directory and the actual user data.

Table 2
Relevant NTFS boot file data.

$Boot	
Byte	**Data**
11–12	Sector size
13	Cluster size
14–15	Reserved
16–20	Not used
40–47	Total sectors in FS
48–55	Starting of MFT

Table 3
Relevant data of the Ext3 superblock (top) and group descriptor table (bottom).

Superblock	
Byte	**Data**
4–7	Number of blocks
12–15	Number of unallocated blocks
20–23	Block Group 0 location
24–27	Block size
GDT	
Byte	**Data**
0–3	Block bitmap location
12–13	Unallocated blocks in group

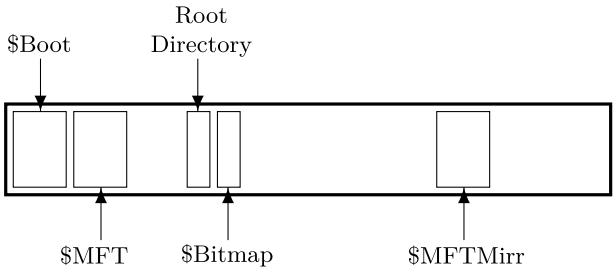

Figure 2. Typical NTFS data structure layout. The boot file is located at the beginning of the partition, followed by the MFT. The root directory is located somewhere in the partition and is followed by the bitmap file. The partition also contains a MFT mirror.

important data contained in the boot file are shown in Table 2, an overview over the entire file system layout is shown in Fig. 2. Another important data structure is the *master file table* (MFT) (Carrier, 2005). The MFT contains an entry for each file and directory, including an entry for itself. . The MFT can be stored anywhere in the data area and thus the boot sector (which contains the starting address of the MFT) is needed to locate the MFT. An MFT file record mainly consists of *attributes* (Carrier, 2005) which store important file information (for example the file name).

The actual file data is stored either in the MFT entry itself or in a cluster run. A cluster run is a sequence of clusters given by the run offset and a run length.

2.3. EXT3

The Extended Filesystem (Ext) is the default file systems of most Linux distributions (Carrier, 2005). Even though the Ext4 file system is the most common one nowadays, Ext3 is structurally very similar to Ext4 and slightly easier to handle. We will therefore focus on the Ext3 file system here. This file system is divided into *block groups* (Carrier, 2005). In Ext3 the main file system information is stored in the *superblock* (Carrier, 2005), comparable to the boot sector in FAT32. The superblock has a fixed offset of 1024 bytes to

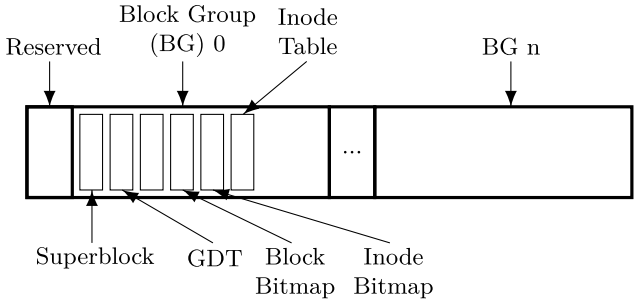

Figure 3. Typical Ext3 data structure layout showing the primary superblock at the corresponding offset followed by the group descriptor table, the block bitmap, the Inode bitmap and the Inode table.

the beginning of the partition and contains general information about the file system, information about the block groups and *inode* (Carrier, 2005) information. Inodes store file and directory metadata in so called *inode tables* (Carrier, 2005), where each block group contains one in the first block of the block group. File and directory names are not stored in inodes. Instead they are stored in *directory entries* (Carrier, 2005) contained in the blocks allocated to the file's parent directory. The *root directory* (Carrier, 2005) contains directory entries for each file and directory that is its immediate child. The *group descriptor table* (GDT) (Carrier, 2005) follows the superblock and contains information about every block group in the file system like the allocation status of each block. A summary of key data from the superblock and the GDT are shown in Table 3. The typical structure of an Ext3 file system can be seen in Fig. 3.

If the superblock is damaged, copies of the superblock are located at predefined positions. These copies can be used in case the primary superblock is corrupted.

2.4. HFS+

Hierarchical File System Plus (HFS+) (Apple Inc) is the successor of Apples Hierarchical File System (HFS) developed for the Mac OS operating system. HFS+ was introduced with Mac OS 8.1 in 1998 and is very similar to HFS.

HFS + divides the available space into equally sized *allocation blocks* (Apple Inc). HFS + has seven major file system structures. These structures are referred to as *volume header*, *allocation file*, *extents overflow file*, *catalog file*, *attributes file*, *startup file* and *alternate volume header* (Apple Inc). The space between the file system structures is free space or can be filled with file data. The typical HFS + layout is shown in Fig. 4.

The base information of HFS+ is stored in the volume header. The position of this header is fixed, and the header always has an

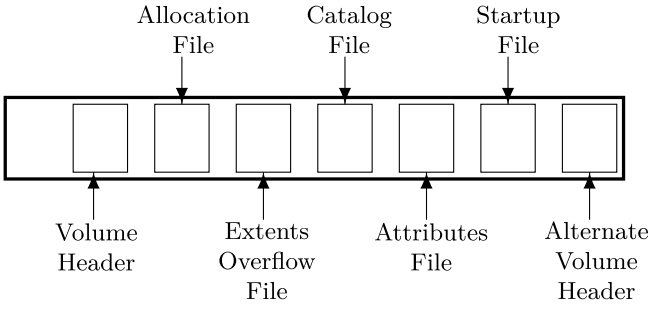

Figure 4. HFS + data structure layout. The volume header is located at the beginning of the partition, followed by the allocation file, extents overflow file, catalog file, attributes file, the startup file and the alternate volume header.

Table 4
Relevant HFS + volume header structure data.

Volume Header	
Byte	**Data**
40–43	Allocation block size
112–191	Allocation file
192–271	Extents overflow file
272–351	Catalog file

offset of 1024 bytes. Details on the data contained in the volume header are listed in Table 4. There is also a copy of the volume header at the end of the file system called *alternate volume header* (Apple Inc).

The volume header not only contains the basic file system information but also the location and size of the file information data structures (allocation file, extends overflow, catalog file, attributes file and startup file). These files can be located anywhere between the volume header at the start and the Alternate volume header at the end of the file system.

We will now go into more detail about file information structures.

The allocation file is a bitmap that stores the allocation status of each allocation block within the file system. If a block is allocated, the corresponding bit is assigned the value one. Accordingly, the bitmap can be used to read out the allocation status of each allocation block. Since the allocation file itself can extend over several allocation blocks, there may be bytes in the file whose bits are not assigned to any blocks. Such unallocated bits must correspond to the value zero.

The extents overflow file is structured as a B-tree and stores lists of extents. Extents are contiguous allocation blocks. In case a file has more than eight extents, the list of extents is stored in the extents overflow file. An extent list can also point to another extent list to form a longer list. The extents overflow file additionally contains information about damaged blocks. If a block is marked as damaged, it is also listed as allocated in the allocation file.

The catalog file is also structured as a B-tree and contains information about folders and files.

The attributes file is currently not used and is reserved for future implementations.

The startup file provides space for a boot loader and its information. It can be used to boot from operating systems not belonging to the Apple universe.

2.5. BTRFS

The B-tree file system (Btrfs) is a file system developed for Linux which combines copy-on-write (CoW) functionalities with a logical volume manager. Therefore, Btrfs file systems can extend over multiple partitions on multiple hard disks.

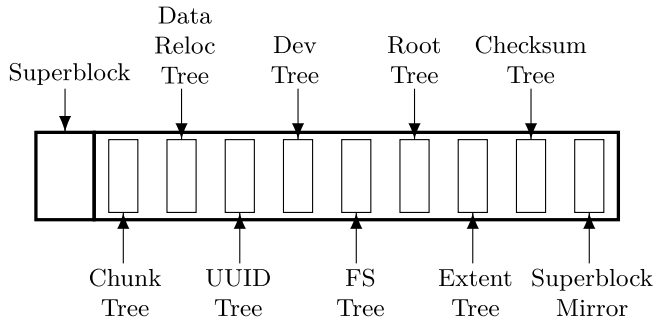

Figure 5. Example Btrfs data structure layout. The figure shows the respective root nodes of the different file system data structures, whereby the position of the data structures is not fixed.

Table 5
Relevant data of the Btrfs superblock (top) and an extent_item (bottom).

Superblock	
Byte	**Data**
0–31	Checksum of superblock
48–55	Physical address of this block
80–87	Logical address: root tree
88–95	Logical address: chunk tree
96–103	Logical address: log tree
144–147	Sector size
148–151	Node size
152–155	Leaf size
156–159	Stripe size
196–197	Checksum type
Extend Item	
Byte	**Data**
0–7	Number of references to this extent
8–15	Transaction ID that allocated this extent
16–23	Item type
24-x	Extent records

A general overview over the file system layout is shown in Fig. 5. It uses B-trees for organizing file system data (Btrfs wiki a). Essential B-trees are the *file system (FS) tree*, *extent tree*, *root tree*, *chunk tree*, *checksum tree* and *device tree* (Btrfs wiki b; Hilgert et al., 2018). In addition to these essential trees, there are also the *data reloc tree*, *log tree* and *universally unique identifiers (UUID) tree* (Btrfs wiki a). The only data structure that is not organized as a B-tree is the so called *superblock* (Btrfs wiki a), which stores basic file system information. The position of the superblock is fixed to an offset of 65.536 bytes from the start of the file system. Details of the superblock data are listed in Table 5. Similar to Ext3, Btrfs stores several copies of the superblock. As Table 5 shows, the root nodes of the three B-trees root tree, chunk tree and log tree are specified in the superblock. Since Btrfs uses a mapping from physical to logical addresses for internal addressing the logical addresses of the root nodes are given.

The chunk tree manages the mapping of the address spaces. These system chunks allow the initial mapping for the root node of the chunk tree.

The root tree contains the references to the B-trees not contained in the superblock. Therefore, it can be used to locate the root nodes of the other trees.

Btrfs uses checksums to preserve the integrity of the data. These checksums are found in the checksum tree.

The device tree is responsible for the reversed mapping of physical to logical addresses. This B-tree is used when devices are removed from the Btrfs file system.

The FS tree references the actual file data. There is a separate FS tree for each subvolume created. The objects of an FS tree are indexed with Inode numbers. The FS trees also contain information about the folder structures and metadata.

The extent tree primary stores information about the allocation status of blocks, but also the corresponding B-tree the block belongs to. Similar to HFS+, Btrfs extents are contiguous chunks of storage allocated for some unspecified usage. There are two different types of extents that are stored in the extent tree; *block group extents* and *data extents* (Btrfs wiki b).

Block group extents provide information about which logical addresses are valid and useable and general information about the block groups.

Data extents are always allocated from within block group extents and represent the sequences of bytes that contain data (or metadata) managed by the filesystem.

Data extents are lists of bytes containing data (then called extent_item) or metadata (then called metadata_item).

3. Taxonomy

We now present a taxonomy of ambiguous file system partitions that can be used to classify various file system combination possibilities. The taxonomy has three dimensions: Placement, Stability, and Accessibility. A combination is a valid combination in the sense of the taxonomy if two or more file systems exist on one partition simultaneously. For simplicity, in the following we consider two file systems x and y co-existing in a single file system partition.

3.1. Placement

This dimension refers to the smallest and largest sector numbers that the file systems x and y utilize in the partition. Disregarding symmetric cases, there are three possible cases:

1. Overlapping: starting sector $x <$ starting sector y, starting sector $y <$ ending sector x and ending sector $x <$ ending sector y
2. Subset: starting sector $x <$ starting sector $y <$ ending sector $y <$ ending sector x
3. Identical: starting sector $x =$ starting sector y and ending sector $x =$ ending sector y

File systems x and y are *overlapping* if the two file systems only partially superimpose each other. For example, the first sector of x may be located at the beginning of the partition and the first sector of y has an offset to the beginning of the partition.

File system y is a *subset* of x if y is completely embedded into x.

File systems x and y are *identical* with respect to placement if both file systems are completely superimposed. Here the data structures of the two file systems must be integrated into each other.

We postulate the amount of overlap correlates with the difficulty of creating the given placement. In this sense, identical placement is hardest to achieve, followed by subset. An overlapping placement is easiest. For sake of ambition, we disregard the easy case when the two file systems lie sequentially one behind the other.

3.2. Stability

This dimension refers to the ability to use both file systems without interference. We say that a file system x *respects* another file system y if the usage of x does not interfere with the current or future usage of y. If x does not respect y, then the usage of x eventually makes it impossible to use y or overwrites data of y. Stability has three cases:

1. Mutual disrespect: file system x does not respect y and file system y does not respect x.
2. Directed respect: file system x respects y but y does not respect x.
3. Mutual respect: file system x respects y and y respects x.

Since any form of disrespect can be achieved by simply formatting one file system over the other, we believe that mutually respecting file system combinations are the most challenging variant of stability.

3.3. Accessibility

This dimension refers to the ability to use the file systems in a default way, i.e., without needing special parameters to find the (beginning of the) file system in the partition. Accessibility has two variants:

1. Unmodified: both file systems do not need special parameters to be found and used.
2. Modified: at least one file system needs special knowledge from the outside to be found and used.

It is clearly easier to combine file systems with modified than with unmodified accessibility.

4. Combining filesystems

We report on how we constructed four examples where several file systems are placed on top of each other within a single partition. The main idea in the construction process was to use one file system as “host” and then place other file systems as “guests” into unused areas or allocated areas with non-essential data. Additionally, we used the options to mark specific blocks as “allocated” or “damaged” to protect areas in the partition that could potentially cause conflicts if they were written. Thereby, we could prevent the file systems from overwriting each others’ data during use. The challenge was to arrange essential data that could not be relocated in a mutually compatible way.

Table 6 shows an overview of the file system combinations as well as their placement in our taxonomy.

4.1. Example A: NTFS and FAT32

Because of its flexible structure, NTFS is an obvious candidate for combination with another file system. Accordingly, we first combined NTFS and FAT32 because FAT32 is well suited as a guest file system due to its simple data structures and because the size of the reserved area can easily be adjusted. Since the MFT has no fixed start address and the FAT can be moved arbitrarily, the boot sectors are most important. Note that even though it is theoretically possible to arbitrarily move the MFT, this is not trivial in practice. Since both boot sectors are very similar but have several unused or non-essential data areas, we superimposed both boot sectors. The combination scheme can be seen in Fig. 6.

To superimpose the two boot sectors, both boot sectors must first be compared with each other and the essential data identified. The comparison is shown in Fig. 7, where essential data is shown in boldface. The green areas show where the combination can be performed successfully, the red areas show where the combination cannot be performed because the values in both boot sectors are mandatory.

Table 6
Overview of the different file system combinations used for creating the ambiguous file system partition examples.

Name	Host file system	Guest file system(s)	Placement	Stability	Accessibility
A	NTFS	FAT32	partially identical	mutual respect	modified
B	Ext3	FAT32	overlapping	mutual respect	unmodified
C	HFS+	FAT32	overlapping	mutual respect	unmodified
D	Btrfs	HFS+, FAT32	overlapping	directed respect	unmodified

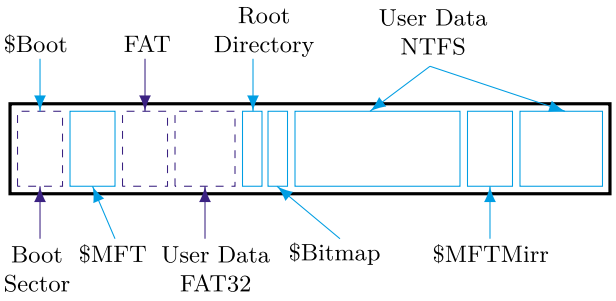

Figure 6. Sketched combination layout of NTFS and FAT32. NTFS structures are depicted in light blue and FAT32 structures are depicted in dashed dark blue.

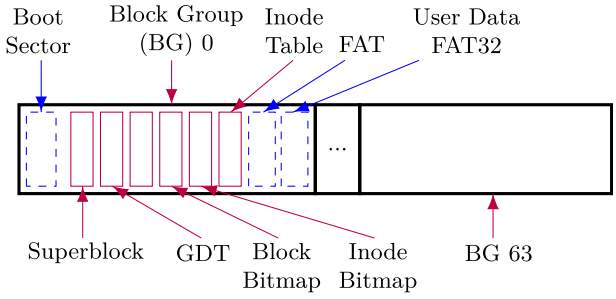

Figure 8. Sketched combination layout of Ext3 and FAT32. Ext3 structures are depicted in red and FAT32 structures are depicted in dashed dark blue.

FAT32				NTFS			
eb	58	90	6d	eb	52	90	4e
6b	66	73	2e	54	46	53	20
66	61	74	**00**	20	20	20	**00**
02	**08**	**00**	**40**	**02**	**08**	**00**	**00**
01	**00**	**00**	**00**	**00**	**00**	**00**	**00**
00	f8	**00**	**00**	**00**	f8	**00**	**00**
3f	00	10	00	3f	00	10	00
00	08	00	00	00	08	00	00
df	**f7**	**ff**	**00**	**00**	**00**	**00**	**00**
e0	**3f**	**00**	**00**	**80**	**00**	**80**	**00**
00	**00**	**00**	**00**	**de**	**f7**	**ff**	**00**
02	**00**	**00**	**00**	**00**	**00**	**00**	**00**

Figure 7. Comparison between the first 48 byte of the FAT32 boot sector (left) and NTFS boot file (right). Bold data are essential. Green areas mark where a successful combination is possible and red color denotes areas where a combination is not possible.

This is for example the case for the byte 14–15 and 16, which refer to the size of the reserved area and the number of file allocation tables of the FAT32 file system. In NTFS these bytes are unused but required to be zero. This contradiction cannot be resolved.Since it is not possible to overlay the boot sectors of NTFS and FAT32 and moving the FAT32 boot sector is not trivial, no successful combination could be achieved in this case.

Regarding our taxonomy, the superimposition of the boot sectors would have been necessary to achieve (at least partially) identical placement. Moving the FAT32 boot sector results in modified accessibility. We believe that it is not possible to create an ambiguous file system partition with both NTFS and FAT32 that has both identical placement and unmodified accessibility.

4.2. Example B: Ext3 and FAT32

The combination of Ext3 and FAT32 appears convenient because the superblock of Ext3 has a fixed offset of 1024 bytes, which provides enough space for another data structure to be placed before. Therefore, the Ext3 file system serves as the host file system for the combination with FAT32. Fig. 8 gives an overview of the construction which we will now explain in detail.

We began by first creating two virtual hard disks with a size of 8 GiB each. A GUID Partition Table (GPT) with one entry was created on both hard disks with `gdisk`. The partitions extend over the entire available storage space. One partition was formatted as Ext3 the other one as FAT32. When formatting as FAT32, the option `-f 1` was used to specify that the file system should contain only one FAT. This step simplifies the construction. Furthermore, the sector size was set to 2048 bytes.

We then combined the data from both disks as follows: The boot sector of the FAT32 was written directly into the Ext3 file system using `dd`. To embed the FAT, the first free block in the Ext3 file system had to be determined by using for example `dumpe2fs` and afterwards the FAT can be written to the corresponding offset.

After that, all necessary structures of both file systems were present on the target partition. However, some adjustments still had to be made. First, the FAT32 boot sector had to be edited. For simplification the backup boot sector position value is set to `0x00` (indicating no backup boot sector). Furthermore, the size of the reserved area had to be adjusted. The offset of 17434 blocks (Ext3) corresponds to 34868 sectors (FAT32) resulting in a size of `0x8834`. If the sector size is too small, this number is too large to be stored in the corresponding data structure, since only two bytes are available for the value.

Furthermore, the blocks used for the FAT and the actual user data had to be marked as allocated in Ext3. Therefore, all data blocks of block group 0 were marked as allocated in the allocation bitmap of block group 0. The total number of free blocks in the superblock had to be reduced accordingly. This value also had to be changed in the group descriptor table of block group 0.

Next, the two file systems must be prevented from overwriting each other when used. For this, the storage must either be divided equally or assigned flexibly. In principle, overwriting is prevented by marking the respective blocks as allocated or damaged (depending on the file system). To divide the storage equally, for example, all blocks with an even index can be assigned to the FAT32 file system and all blocks with an odd index to the Ext3 file system. The corresponding blocks must then be marked accordingly in the respective file system. This has the advantage that one does not have to worry about the storage management when using both file systems. On the other hand this kind of storage division would be noticeable during a forensic analysis because of a high number of damaged blocks. If the storage is allocated flexibly, storage management becomes more complex, since each time a new block is used, the block must be marked as unavailable in the corresponding file system. The advantage is that this type of storage management is less conspicuous. Therefore, we decided to use a flexible storage assignment. With the flexible allocation of the storage first the respective file system structures must be protected against overwriting. To protect FAT32 data from being overwritten by the Ext3 file system, the group descriptor table, the superblock (and their copies) and the respective block bitmaps had to be manipulated. Vice versa, clusters occupied by the Ext3 file system had to be marked as bad in the FAT. Overall, the file system combination uses overlapping placement, but the resulting combination shows mutual respect and unmodified accessibility which means that mounting is trivial in the sense that no offset has to be specified and only the file system type has to be given to determine which file system to mount. For read-only operations, we expect the OS

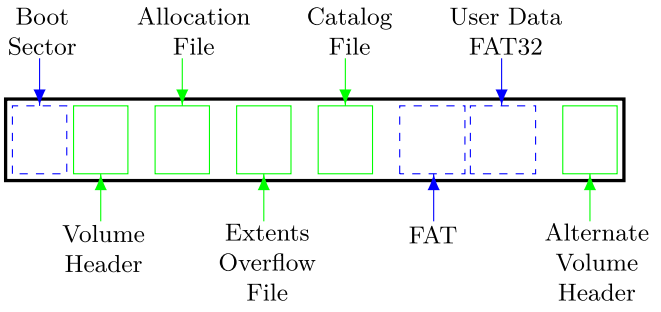

Figure 9. Sketched combination layout of HFS+ and FAT32. HFS + structures are depicted in green and FAT32 structures are depicted in dashed dark blue.

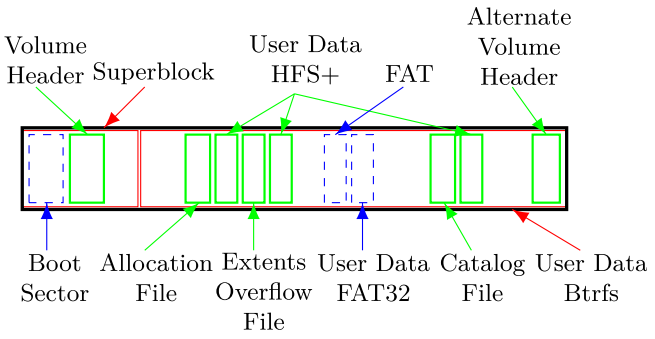

Figure 10. Sketched combination layout of Btrfs, HFS+ and FAT32. Btrfs structures are depicted in red, HFS + structures are depicted in thick green and FAT32 structures are depicted in dashed dark blue.

functions should work without any intervention on both the guest and host file system. However, for write operations, the data structures have to be adjusted if new data is added to guarantee the mutual respect since we use a flexible storage assignment strategy. Creating OS drivers to perform these adjustments automatically was out of scope for this initial study.

4.3. Example C: HFS+ and FAT32

The third combination consisted of HFS + as host and FAT32 as guest. The schematic representation of the combination can be seen in Fig. 9. We now explain how it was constructed.

Similar to the Ext3 superblock, the major file system structure in HFS + has an offset of 1024 bytes from the beginning of the partition. Similar to Example A above, the combination of HFS+ and FAT32 is a natural option.

The preparatory procedures are analogous to the combination of Ext3 and FAT32 described above: We prepared two virtual hard disks, each with a formatted partition. In this case, too, it was advisable to specify the option `-f 1` when formatting with FAT32. Only the sector size differed from the previous construction. In this case, a sector size of 4096 bytes was selected.

The boot sector of FAT32 was written directly into the HFS + file system. In order to place the FAT, the HFS + file system was checked for larger unused areas. The largest available storage area starts after the last file system structure. In order to be able to store data in HFS + later, an unused area following the last file system structure should be kept free. Based on these considerations, we chose an offset of 40960 blocks for the FAT.

After that, the FAT32 boot sector had to be edited. As before, the backup boot sector position value is set to `0x00` and the reserved area is adjusted. Since the block size of the HFS + file system corresponds to the sector size of the FAT32 file system, this results in an offset of 40960 sectors.

Next, the data has to be protected against mutual overwriting. As in example B we used a flexible assignment strategy. For FAT32, the process already described for this purpose above was used. In HFS+ the allocation file is used to achieve protection.

In this file system combination both file systems use overlapping placement. As in Example B, both also mutually respect each other, but in order to allocate more space some data structures have to be adjusted. Accessibility is unmodified.

4.4. Example D: BTRFS, HFS+ and FAT32

We now show that using the construction principles above it is even possible to combine *three* file systems in a single partition with unmodified accessibility. The chosen file systems were HFS+, FAT32 and Btrfs. We now explain how this can be achieved. As an overview, Figure 10 shows how the file systems structures were overlaid.

Since Btrfs has the most complex data structures, we chose it as the host file system and integrated HFS+ and FAT32 data structures into it. For preparation, three hard disks with one partition each were constructed. As in the other examples above, we used virtual hard disks with a GPT containing one partition entry.

When formatting one of the three partitions with Btrfs, no additional options were specified. To create the HFS + file system, however, we created file system structures with an initial size of 64 blocks. This can be done by passing special values to the command `mkfs.hfsplus`, e.g., using the options `-c b=64,c=64,e=64`, whereby `b` corresponds to the allocation file, `c` corresponds to the catalog file and `e` corresponds to the extents overflow file. The FAT32 file system is created with a single FAT and a sector size of 4096 bytes. For this, the options `-f 1` and `-S 4096` are passed to the command `mkfs.fat`.

In the next step, the file system structures of HFS+ and FAT32 are written to a buffer using the tool `dd`. From the HFS + file system, the structures are alternate volume header, catalogue file, extents overflow file, allocation file and volume header. For the FAT32 file system, only the boot sector and the FAT are required.

After writing the relevant file system structures of HFS+ and FAT32 into the buffer, we made some necessary adjustments directly to the cached files. In order to do this, it is necessary to determine the future positions of the individual structures in the Btrfs file system.

The individual positions of the HFS+ and FAT32 file system structures are shown in Table 7. Since the Btrfs superblock has a large offset, the volume header of the HFS + file system and the FAT32 boot sector can be integrated without additional offsets. The positions of the volume header and boot sector correspond to their positions in their respective file systems. The alternate volume header can also be written without an additional offset. To determine the new positions of the allocation file, extents overflow file, catalogue file and the FAT the Btrfs file system has to be checked for larger areas of free space. We created the mapping in Table 7

Table 7
Addresses HFS+ & FAT32 structures.

New address	Old address	Structure
0x0	0x0	boot sector
0x400	0x400	volume header
0x254 e000	0x1000	allocation file
0x258 e000	0x4 1000	extents overflow file
0x400 1000	0x2 0000	FAT
0x1c00 0000	0x8 1000	catalog file
0x1 ffef ba00	0x1 ffef ba00	alternate volume header

according to these localized areas, taking into account the properties and usage of the respective data structures.

Due to the new addresses, corresponding adjustments had to be made. In the FAT32 boot sector, the size of the reserved area and the sector that contains the backup boot sector had to be changed. The size of the reserved area can be calculated by the position of the FAT (`0x400 1000`) and the size of the sectors (4096 byte).

Furthermore, the volume header and, analogously, the alternate volume header had to be modified. The bytes `0x80-0x83`, `0xd0-0xd3` and `0x120-0x123` must be edited relative to the beginning of the volume header. These bytes indicate the start blocks of the allocation file, extents overflow file and catalogue file. Note that the new values for the HFS + file system data structures are stored in big-endian.

After these adjustments, some changes had to be made to guarantee a robust usage of HFS+ and FAT32. In the FAT, all clusters occupied by Btrfs and HFS + structures are marked as damaged. Furthermore, all allocation blocks occupied by Btrfs and FAT32 structures are marked as allocated in the HFS + allocation file bitmap. Also, all allocation blocks before the catalog file were marked as allocated to simplify matters.

Afterwards, the HFS+ and FAT32 file system structures were written to their new addresses in the Btrfs file system with `dd`, specifying the respective offsets. For a robust usage of all three file systems, the data structures and user data of the different file systems must be protected from being overwritten by one another. Again, the flexible assignment strategy was chosen. We already discussed this for HFS+ and FAT32. To protect HFS+ and FAT32 from being overwritten by Btrfs some byte ranges have to be manually marked as allocated. First, an extent_item or metadata_item must be created in the extent tree. If an extent_item is to be created, the correct link to the corresponding B-tree must be created. Since file system errors can occur with a non-valid referencing, a valid entry must also be created in the referenced B-tree. If all B-trees involved have valid entries, the checksums in the checksum tree must be updated.

This is the most complex of the presented combinations. Overlapping placement was used to accommodate the structures of all three file systems on one partition. Regarding stability, there is a mutual respect between HFS+ and FAT32, and HFS+ and FAT32 also respect Btrfs. Unfortunately, although the Btrfs data structures were known to prevent overwriting of the other file systems, manually changing the nodes and checksums in Btrfs is anything but trivial and we were not able to adjust the data by the time this paper was written. Therefore, Btrfs does currently not respect HFS+ and FAT32. Accordingly, this combination is only partially stable in the sense of the taxonomy. At least all three file systems are accessible in an unmodified fashion.

5. Evaluation

We evaluated the file system combinations B, C and D in two different ways: We first tested whether the file systems could in fact functionally co-exist in one partition by mounting them individually and performing file system operations on them. Second, we evaluated whether common tools used by digital forensic examiners were able to deal with the different file systems in useful ways.

5.1. Functional tests

To test the functionality of each file system within the combinations, we mounted each file system individually, created several files, marked the corresponding blocks as damaged and unmounted the file system again. After creating files in all file systems individually, we mounted each file system again and verified that the files were indeed accessible and readable. The created files were 307.5 KiB, 324.7 KiB and 300.0 KiB in size and thus expanded over more than one block. The verification was done by checking the md5 sum of the files. None of the files were fragmented. All images had a size of 8 GiB, whereby the individual files systems, FAT32, Ext3, HFS+, and Btrf extended over the whole partition thus they also were 8 GiB, respectively. We performed each step sequentially for the respective file system before moving on to the next file system. This was true in all cases. We also manually checked whether file system data was overwritten when creating new data in the other file system. This was not the case with any combination. We therefore conclude that as long as the concrete file system type is known, the file system can be used in a standard fashion without interference with the other overlaying file systems.

Next, we analyzed the disk images with various forensic tools to see how they deal with the file system ambiguity.

5.2. Tool tests

For the tool testing part of the evaluation we used five common forensic tools namely; TestDisk (TestDisk - CGSecurity) (Linux Version 7.0–3), The Sleuth Kit (Carrier) (Linux Version 4.6.5–1), Autopsy (Basis Technology) (Linux Version 2.24–3 and Windows Version 4.19.3), X-Ways Forensics (X-Ways AG) (Windows Version

Table 8
Overview of evaluation results performed with multiple forensic tools for all three successfully created ambiguous file system partition disk images. The table shows which file systems and which files were detected by the respective tool for the different file system combinations and whether there were any indications of ambiguity. The disk image names are referring to Table 6, where image B contains a combination of Ext3 and FAT32, image C contains a combination of HFS+ and FAT32 and image D contains a combination of Btrfs, HFS+ and FAT32.

Tool	Disk image	Recognized FS	Files found	Notes
TestDisk	B	Ext3	n/a	Indication of FAT and damaged fragments
	C	HFS+	n/a	HFS + not supported, Indication of FAT
	D	HFS+	n/a	Indication of Btrfs and FAT
The Sleuth Kit	B	ExtX or FAT	Depending on selected file system	Does not commit to a file system
	C	HFS or FAT	Depending on selected file system	Does not commit to a file system
	D	HFS or FAT	Depending on selected file system	Btrfs not supported, Does not commit to a file system
Autopsy	B	ExtX or FAT	Depending on selected file system	Does not commit to a file system
	C	HFS or FAT	Depending on selected file system	Does not commit to a file system
	D	HFS or FAT	Depending on selected file system	Does not commit to a file system
X-Ways Forensics	B	Ext3	Only host files	
	C	HFS+	Only host files	
	D	HFS+	Only host files	
Magnet AXIOM	B	Ext3	Only host files	
	C	HFS+	Only host files	
	D	HFS+	Only host files	

20.0 SR-4 x64) and Magnet AXIOM (Forensics) (Windows Version v4.8.1.22785). Table 8 summarizes the results. Some of the used tools (TestDisk, Autopsy and X-Ways Forensics) have integrated file carving functions. By using them all of the created test files can be recovered independent of the file system type or the file system combination, since file carvers usually do not rely on file system structures. Instead they search for file type patterns therefore the underlying file system is not relevant for recovering files with this technique. We therefore only use the created test files as an indication whether the file system can be used without overwriting data, the data is correctly written to the file system and as indication if the tools used for the evaluation are able to read the recognized file system as intended.

5.2.1. TestDisk

For combination B TestDisk was able to identify and analyze the host file system (Ext3) while FAT32 was not recognized. During the usage of the deeper analysis function, messages were shown that indicated the existence of a FAT. However, these messages were no longer displayed after the completion of the analysis. The tool also indicated partitions with overlapping fragments, but when looking at these fragments, an error message was displayed, and the fragments were described as damaged.

For combination C and D TestDisk was only able to identified HFS + while a further analysis of this file system was not possible because of missing HFS + support. Also for these two combinations, a message was displayed during deeper analysis indicating the existence of a FAT. For combination D a Btrfs file system label was displayed when the overlapping partitions were shown.

Since TestDisk is not intended to do so no files could be viewed or analyzed.

5.2.2. The Sleuth Kit

For all three combinations The Sleuth Kit showed the message cannot determine file system type (<file system A> or <file system B>). The Sleuth Kit is therefore unable to determine the file system but gives the user the possibility to choose from the most probable file systems. If one of the offered file systems is then passed as an option to The Sleuth Kit, all file system data is displayed correctly. Since The Sleuth Kit does not support Btrfs this file system could not be identified and was not displayed as an file system type option.

Depending on the chosen file system the corresponding files could be viewed and analyzed.

5.2.3. Autopsy

Since the file system analysis of Autopsy is based on The Sleuth Kit, the results look very similar to the results of the evaluation with The Sleuth Kit. Accordingly, Autopsy was unable to automatically detect the file system type. For Linux the underlying terminal window displayed the error messages we originally observed with The Sleuth Kit. However, the messages were not displayed in the graphical interface.

Similar to The Sleuth Kit depending on the chosen file system the corresponding files could be viewed and analyzed.

5.2.4. X-Ways Forensics

For all three file system combinations X-Ways Forensics was not able to recognize any of the embedded file systems. Also, X-Ways Forensics was only able to recognize Ext3 and HFS+, i.e., Btrfs was not recognized at all. By using the standard configuration (using file system structures to reconstruct user data) X-Ways Forensics was therefore only able to detect the test files in Ext3 and HFS+.

5.2.5. Magnet AXIOM

Similar to X-Ways Forensics Magnet AXIOM was not able to recognize the embedded file systems for any of the three file system combinations. The tool was also not able to identify Btrfs and only showed files from the Ext3 and HFS + file systems.

6. Recognizing ambiguous file system partitions

The question that naturally arises is of course how can ambiguous file system partitions be detected by investigators and forensics tools.

One point we already mentioned is file carving. This allows all files to be reconstructed regardless of which file system they are part of. Depending on the size of the storage, however, this can take a significant amount of time and, in addition, the file system itself is not analyzed so only the file data itself is acquired, as opposed to metadata including MAC times and file paths.

Investigators and forensic tools should therefore check certain data structures and look if there are any irregularities (deviations from what can be considered as ordinary). Since the structures of the host and the guest file systems have to be manipulated to combine them, a good starting point would be to check the position and size of certain data structures, for example a very large reserved area for a FAT file system. Another approach is to check if the number of damaged sectors or blocks exceeds a threshold number or percentage, as this mechanism is used to prevent data from being overwritten in the guest file system. Looking for unusual file system labels, such as a Btrfs label in an Ext file system, could detect another red flag. Sectors or blocks that should be empty, for example because of fixed offsets, but are not might also indicate an embedded file system. A final red flag to check is whether the allocated areas fit the files created within the file system.

It is likely that other irregularities will arise with the regular use of an ambiguous file system partition, but detection could become more difficult with the creation of ambiguous file system partitions more sophisticated than the proof of concept presented here.

7. Conclusion

We studied the possibility to combine several file systems in one partition and thereby create an ambiguous file system partition. We also developed a taxonomy that allows to classify and evaluate the combination of file systems to create such partitions. We presented some construction strategies for ambiguous file system partitions and successfully created three examples disk images containing two or more fully functional file systems which can be mounted without passing an offset. Data written to these file systems is protected from being overwritten through a manual storage assignment. This shows that ambiguous file system partitions can actually be constructed, and that this possibility depends on the specific properties of these file systems. Furthermore, we showed that some forensic tools correctly identify the ambiguity and pass it on to the analyst. Other tools are not prepared for such a setting and ignore one or even both file systems or could not communicate the ambiguity clearly enough. Therefore, ambiguous file system partitions remind us that there is no such thing as a “clear file system interpretation”, but that each data interpretation is based on a rational decision by an analyst.

In future work other combinations of file systems can be attempted. The most difficult combination of parameters in our taxonomy (identical placement, mutual respectful stability and unmodified accessibility) is yet to be achieved. The evaluation of the stability of file system combinations can also be intensified, e.g., by writing large files to each partition. Also, it may be possible to automatically create ambiguous file system partition examples.

Furthermore, the storage assignment could be automated to allow seamless use of the ambiguous file system partition by ordinary users where data in guest file systems would be hard to detect.

All ambiguous file system partition example images are available under https://github.com/janineschneider/Ambiguous-File-System-Partitions.

Acknowledgments

We thank Frank Adelstein, Jenny Ottmann, Jan Gruber, Marion Liegl and the anonymous reviewers for their helpful comments on previous versions of the paper. Work was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 “Cybercrime and Forensic Computing” (grant number 393541319/GRK2475/1-2019).

References

Anderson, R., Needham, R., Shamir, A., 1998. The steganographic file system. In: Aucsmith, David (Ed.), Information Hiding. Springer Berlin Heidelberg, pp. 73–82.

Apple Inc. Technical note tn1150 - hfs plus volume format. https://developer.apple.com/library/archive/technotes/tn/tn1150.html.

Basis Technology. Autopsy - digital forensics. https://www.autopsy.com/.

Blunden, B., 2012. The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System, second ed. Jones and Bartlett Publishers, Inc., USA.

Btrfs wiki. On-disk format. https://btrfs.wiki.kernel.org/index.php/On-disk_Format.

Btrfs wiki. Trees. https://btrfs.wiki.kernel.org/index.php/Trees.

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

Carrier, B.. The sleuth kit (tsk) and autopsy: open source digital forensics tools. https://www.sleuthkit.org/.

Göbel, T., Baier, H., 2018. Anti-forensics in ext4: on secrecy and usability of timestamp-based data hiding. In: Proceedings of the Fifth Annual DFRWS Europe, 24, pp. S111–S120. https://www.sciencedirect.com/science/article/pii/S174228761830046X.

Göbel, T., Türr, J., Baier, H., 2019. Revisiting data hiding techniques for apple file system. In: ‘Proceedings of the 14th International Conference on Availability, Reliability and Security’, ARES ’19. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3339252.3340524.

Hilgert, J.-N., Lambertz, M., Yang, S., 2018. Forensic analysis of multiple device btrfs configurations using the sleuth kit. In: Proceedings of the Eighteenth Annual DFRWS Europe, 26, pp. S21–S29. https://www.sciencedirect.com/science/article/pii/S1742287618301993.

Kennedy, D., O'Gorman, J., Kearns, D., Aharoni, M., 2011. Metasploit: the Penetration Tester's Guide, first ed. No Starch Press, USA.

Magnet Forensics. Magnet Axiom | Digital Investigation Platform | Magnet Forensics. n.d. https://www.magnetforensics.com/products/magnet-axiom/.

McDonald, A.D., Kuhn, M.G., 2000. Stegfs: a steganographic file system for linux. In: Pfitzmann, A. (Ed.), Information Hiding. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 463–477.

Neuner, S., Voyiatzis, A.G., Schmiedecker, M., Brunthaler, S., Katzenbeisser, S., Weippl, E.R., 2016. Time is on my side: steganography in filesystem metadata. In: Proceedings of the 16th Annual USA Digital Forensics Research Conference, 18, pp. S76–S86. **URL:**. https://www.sciencedirect.com/science/article/pii/S1742287616300433.

TestDisk - CGSecurity. https://www.cgsecurity.org/wiki/TestDisk.

Thompson, I., Monroe, M., 2006. FragFS: an Advanced Data Hiding Technique. In: Presentation at BlackHat Federal 2006. URL: https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Thompson/BH-Fed-06-Thompson-up.pdf.

Wayner, P., 2002. Disappearing Cryptography: Information Hiding: Steganography and Watermarking, second ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. second ed.

Wiseman, S.R., 2018. Poison pixels: combatting image steganography in cybercrime. In: Presentation at RSA conference 2018, San Francisco, April 16-20, Moscone Center. https://published-prd.lanyonevents.com/published/rsaus18/sessionsFiles/8741/HTA-W02_Poison-pixels-Combatting-Image-Steganography-in-Cybercrime.pdf.

Wundram, M., Freiling, F.C., Moch, C., 2013. Anti-forensics: the next step in digital forensics tool testing. In: Morgenstern, H., Ehlert, R., Freiling, F.C., Frings, S., Göbel, O., Günther, D., Kiltz, S., Nedon, J., Schadt, D. (Eds.), Seventh International Conference on IT Security Incident Management and IT Forensics, IMF 2013, Nuremberg, Germany, March 12-14, 2013. IEEE Computer Society, pp. 83–97. https://doi.org/10.1109/IMF.2013.17.

X-Ways AG. X-ways forensics: Integrierte software für computerforensik. http://www.x-ways.net/forensics/.