Digital Investigation 24 (2018) S111–S120

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2018 Europe — Proceedings of the Fifth Annual DFRWS Europe

Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding

Thomas Göbel[*, 1], Harald Baier [1]

da/sec–Biometrics and Internet Security Research Group, University of Applied Sciences, Darmstadt, Germany

Keywords:
Anti-forensics
Anti–anti forensics
Digital forensics
Data hiding
Steganography
File system forensics
Ext4
Nanosecond timestamps

A B S T R A C T

Ext4 is a popular file system used by Android and many Linux distributions. With its rising pervasiveness, anti-forensic techniques like data hiding may be used to conceal data. This paper analyzes the feasibility of using timestamps of the ext4 file system to hide data. First, we examine the usage, the structure and the capacity of the available timestamps with a special focus on their sub-second granularity. The results reveal that the nanoseconds part of the ext4 timestamps can be used to build a system with steganographic strength. Second, we devise an ext4 anti-forensic technique that offers *secrecy* of the hidden data and easy *usability* in a wide range of scenarios. We provide a set of requirements (e.g., indistinguishability of regular and tampered timestamps) and a proof-of-concept implementation that is able to conceal arbitrary data within the file system timestamps. The evaluation shows that our implementation satisfies our requirements and actually works in practice.

Introduction

The rise of cybercrime as well as the growing amount of anti-forensic tools demand a more intense debate on the problem of *anti-forensics*. This requires a more comprehensive analysis of current anti-forensic threats, in order to gather reliable evidence during digital forensic investigations and to develop *anti-anti-forensic* techniques, i.e., countermeasures and mitigation strategies against the anti-forensic methods, tools and techniques. However, the total number of research papers focused on anti-forensics is limited, as stated by Baggili et al. (2012), and is outnumbered by the number of websites about how to exploit the forensic process, as put forth by Harris (2006).

Conlan et al. (2016) shared an overview, which includes data on 308 collected anti-forensic tools and assigns them to an anti-forensic category. According to the quantitative analysis of the data set, the majority of tools are assigned to the *data hiding* category. This is due to the increasing need for protected information exchange and storage.

Data hiding is possible on different abstraction layers, e.g., the hard disk or the file system layer. Besides the traditional hiding places in file systems, e.g., file and directory slack space (Carrier, 2005), reserved areas (Piper et al., 2005) for future use or leftovers from earlier versions, the misuse of existing data structures is another effective way to hide data in the file system.

Ext4 is a popular file system used by Android and many Linux distributions. With its rising pervasiveness, anti-forensic techniques like data hiding may be used to conceal data. This paper addresses the anti-forensic problem by presenting a new potential hiding place in ext4 volumes. Regarding the forensic investigation, it is important to know about such places, especially due to the extensive use of ext4 in smartphones since Android 2.3 Gingerbread (Ts'o, 2010).

Our approach makes use of a data structure in the inode table. The inode table contains all the metadata of a file or directory (e.g., the file owner, permissions, timestamps). Hidden data that matches the normal internal structures will not be recognized by a digital forensics analysis tool, since there is no inconsistency and thus nothing unexpected (e.g., `if e2fsck` interprets timestamps with hidden data as ordinary timestamps). Without any warnings given by existing forensic tools or the file system check, hidden data is hard to find.

Instead of encrypting the confidential data (e.g., passwords, personal data) with well-known cryptographic techniques, a steganographic approach is considered here, since steganography does not leave any evidence of information exchange and therefore does not arouse the interest of the forensic investigator. If cryptography is used instead of steganography, third parties can recognize an

* Corresponding author.
E-mail addresses: thomas.goebel@h-da.de (T. Göbel), harald.baier@h-da.de (H. Baier).
[1] *URL:* https://www.dasec.h-da.de

https://doi.org/10.1016/j.diin.2018.01.014

existing communication, but can not read the contents (Hassan and Hijazi, 2016). Steganography is often considered when censorships or restrictions of governments and other opponents need to be circumvented.

The use of file system timestamps as a steganographic channel was proposed by Neuner et al. (2016), who showed in case of NTFS, how the unused capacity in timestamps can be used to create a steganographic channel to hide data. As future work Neuner et al. (2016) considered porting their hiding approach to other file systems, but have not yet studied this for ext4.

Contribution

The contribution of this paper is to examine the steganographic capabilities of timestamps in the ext4 file system. More specifically, we make the following contributions:

1. The structure and usage of ext4 file system timestamps are analyzed. We explore their applicability as an effective means for data hiding in environments such as Linux operating systems and Android devices.
2. A proof-of-concept implementation is developed. The most recent version of the source code can be downloaded from the GitHub website https://github.com/dasec/ext4-timestamp-magic. The implementation confirms that data can be securely hidden in ext4 volumes without interrupting the normal system operation, that is, the file system is fully accessible to normal usage. Furthermore, our implementation is robust in the sense that hidden data must be restorable despite user interaction.
3. Our hiding method is evaluated with respect to two aspects: `secrecy` and `usability`. Ideally, manipulated timestamps are indistinguishable from ordinary timestamps. To validate the steganographic strength of this hiding approach, we compare the entropy of timestamps including embedded steganographic information with the entropy of timestamps produced by normal file system operations and show their indistinguishability.
4. A discussion about digital forensic implications of the hiding method described in this paper and the future work.

The rest of the paper is organized as follows: In Section Related work we discuss related work. Section Analysis of ext4 timestamps analyzes the inode structure and the granularity of the available timestamps in the ext4 file system. We then present in Section Methodology our methodology, how to hide and restore data in ext4 volumes. After that, we explain our proof-of-concept implementation in Section Implementation. In Section Evaluation of the hiding technique we evaluate the requirements (e.g., secrecy, usability, indistinguishability) of the proposed hiding technique. Lastly, we present the implications for digital forensic investigations in Section Implications on forensic investigations and conclude our work in Section Conclusion and future work.

Related work

In this section we review related work with respect to anti-forensics in general and data hiding in an ext file system in particular.

Anti-forensics

Solving anti-forensic issues requires a consensus view with a standardized definition and categories of different anti-forensic methods in order to determine mitigation strategies (Harris, 2006). Well known definitions for the term anti-forensics were proposed by Rogers (2005) and by Harris (2006). The most recent definition was provided by Conlan et al. (2016), who summarized previous definitions and defined the term anti-forensics as "any attempts to alter, disrupt, negate, or in any way interfere with scientifically valid forensic investigations.

Based on the original, widely accepted anti-forensic taxonomy proposed by Rogers (2005), Conlan et al. (2016) designed a more comprehensive and up-to-date taxonomy to divide anti-forensic techniques into several categories. The extended taxonomy includes five categories, each containing multiple sub-categories: (i) data hiding, (ii) artifact wiping, (iii) trail obfuscation, (iv) attacks against forensic tools and processes, (v) possible indications of anti-digital forensic activity. The hiding method proposed in this paper can be mapped to the category *Data hiding* of the extended taxonomy. In particular, it fits the two sub-categories *Filesystem manipulation* and *Steganographic filesystem*.

Data hiding in the ext file system

Hiding data in file system metadata was carried out by Anderson et al. (1998) along with the development of a steganographic file system. This resulted in StegFS (McDonald and Kuhn, 1999), a steganographic file system based on ext2, which allowed people to deny the existence of hidden data. Various data hiding techniques for ext2 and ext3, as well as suitable countermeasures, have already been discussed by Piper et al. (2005), Berghel et al. (2008), Eckstein and Jahnke (2005) and Grugq (2005). The most recent contribution in this field was a low-level study and comprehensive forensic analysis of the ext4 data structures by Fairbanks (2012). He also mentioned several potential hiding places in ext4, such as group descriptor growth blocks and data structures in uninitialized block groups, but did not study these places in-depth. Another relevant paper is "Anti-Forensic Capacity and Detection Rating of Hidden Data in the ext4 Filesystem". Göbel and Baier (2018) present, analyze, and evaluate different techniques to hide data in the ext4 file system, but did not focus on ext4 timestamps.

Besides that, there are the following helpful references for file system analysis. Wong (2016) provides the ext4 wiki, an extensive reference work for file system analysis. Mathur et al. (2007) published an extensive work explaining the new ext4 file system features, such as nanosecond timestamps. In addition, essential information about ext4 can be found in the source code of the Linux kernel, e.g., the Git repository provided by Ts'o (2017).

Steganography based on timestamps

The paper by Neuner et al. (2016) plays a major role for our approach, since it proposed the applicability of file system timestamps as a steganographic channel for the first time. Based on the central idea of this paper, that is hiding information within NTFS timestamps, we examine whether ext4 offers similar means to conceal data. In contrast to Neuner et al. (2016), we provide the full source code of our PoC implementation. Furthermore, to complete the related work, we calculate the entropy of regular and tampered timestamps, make statements on the steganographic channel capacity (i.e., how much data can be hidden using the proposed technique and which volume size is required) and provide a more efficient hiding algorithm, without having to sort the timestamps first.

Analysis of ext4 timestamps

In this section we analyze ext4 timestamps with the aim to make use of these timestamps as a steganographic channel. We first review the five available timestamps in the ext4 file system and then analyze, which timestamps and which part of them are useful for our approach. The result of this section is that the nanoseconds

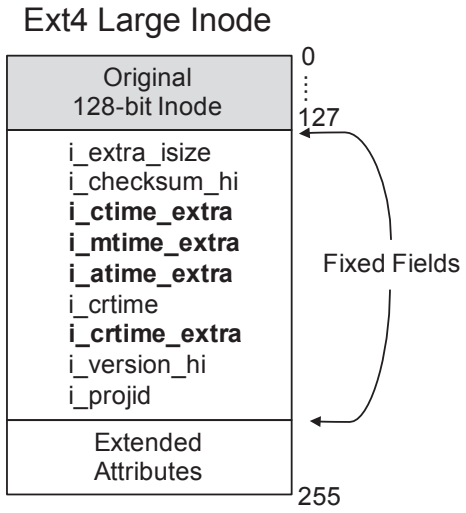

Fig. 1. Layout of the ext4 inode structure. Adapted figure, originally appeared in Mathur et al. (2007).

part of the last access and the creation time are suitable for our steganographic channel, respectively.

Timestamps in ext4

In order to select suitable timestamps for data hiding, we first examine all available timestamps in ext4. The specific structure of an inode table entry is mentioned in the documentation of the ext4 disk layout (Wong, 2016) and depicted in Fig. 1. The first 128 bytes of an ext4 inode are identical to inodes from a former ext version (i.e., ext2, ext3) followed by some extra fixed fields starting at byte offset 128 of an inode data structure.

In what follows, we explain the timestamps provided by ext4 for each file or directory, respectively. We make use of the term *object* to denote a file or directory. Table 1 shows all ext4 timestamps with their respective offset and length within an inode. For each object the following timestamps are provided by ext4: (i) the last modification time of the object (`mtime`), (ii) the last access time of the object (`atime`), (iii) the last metadata change time (`ctime`), e.g., change of ownership, permissions or file size and (iv) the deletion time (`dtime`). These four timestamps are stored in 32-bit integers each and represent the seconds since the Unix epoch (January 1, 1970). In ext4 an additional fifth timestamp was added: (v) the creation time (`crtime`) of the object.

Besides that, the larger inode structure size of 256 bytes in ext4 offers additional space to support nanosecond timestamps (Mathur et al., 2007). Four of five timestamps (except for the deletion timestamp) were extended to 64 bits by adding the 32-bit field `i_[c|m|a|cr]time_extra` respectively, as depicted in Fig. 1. The lower two bits of these four extra 32-bit fields are used to extend the 32-bit seconds field to a 34-bit second field and thus prevent the time overflow in the year 2038 (new overflow date is now 2446-05-10 (Wong, 2016)). The upper 30 bits of the additional timestamp fields are used to provide nanosecond granularity, since 30 bits are sufficient to support timestamps with a precision of one nanosecond. Ext4 is able to fill the extra timestamp fields with nanosecond-precise information since the Linux system clock also provides nanosecond granularity[3].

[3] https://linux.die.net/man/2/clock_getres [Visited on 2018-01-12].

Table 1
Available timestamps in the ext4 inode structure.

Offset	Length	Name	Description
0×8	32 bits	i_atime	Access time
$0 \times C$	32 bits	i_ctime	Inode change time
0×10	32 bits	i_mtime	Modification time
0×14	32 bits	i_dtime	Deletion time
0×84	32 bits	i_ctime_extra	Extra ctime bits
0×88	32 bits	i_mtime_extra	Extra mtime bits
$0 \times 8C$	32 bits	i_atime_extra	Extra atime bits
0×90	32 bits	i_crtime	File creation time
0×94	32 bits	i_crtime_extra	Extra crtime bits

However, end users who access the file system via the ordinary operating system interface can only see timestamps with second granularity. As already mentioned by Neuner et al. (2016), there is an information gap between how modern file systems store timestamps and how end users make use of them. Common Linux file explorers, as well as the command ls -la, do not support nanosecond precision. Except for future or time critical applications, in most cases the additional timestamp precision is not required. Thus a manipulation of the nanoseconds part of an ext4 timestamp is of no relevance for end users.

Analysis of nanosecond timestamps

The above-mentioned four extra timestamp fields are able to conceal 16 bytes of data in each inode table entry. However, if we use the lower two epoch bits to hide information, this leads to dates beyond 2038, which looks suspicious and assists the forensic investigator in disclosing hidden data and the steganographic channel. Fig. 2 illustrates, how the lower two bits are used to extend the original timestamps. To prove this behavior, an access timestamp is intentionally set to the year 2111. Linux commands like debugfs `-R 'stat <inode>' [image]` or `stat [file]` are able to parse manipulated timestamps.

Listing 1. Output of debugfs when parsing ext4 timestamps.

```
debugfs -R 'stat <12>' testimage.dd
ctime: 0x58aed1d5:90fbce3c -- Thu Feb 23 13:13:09 2017
atime: 0x0942d682:00000001 -- Sat Jan 10 15:15:30 2111
mtime: 0x70e87e82:00000000 -- Thu Jan 10 15:15:30 2030
crtime:0x58aed1b7:03d70980 -- Thu Feb 23 13:12:39 2017
```

Listing 1 shows the output of debugfs, Listing 2 shows the output of stat. Unlike debugfs, the stat command does not parse `crtime` and `dtime`. To verify timestamp parsing, we add the lower two bits of the `i_atime_extra` field with the value 01 to the beginning of the `i_atime` field. This yields the hexadecimal value `0 × 010942D682` or the decimal number `4450342530`, respectively. The conversion to a human readable format is done with date `-d @4450342530`. As a result we get `Sat Jan 10 15:15:30 2111`, which corresponds to the output of debugfs. At this point it should be mentioned that the `istat` command (provided by the Sleuthkit[4]) does not take the extra epoch bits into account and therefore incorrectly decodes timestamps beyond the year 2038.

To finally get the nanoseconds part (e.g., of the change timestamp) we need to throw the low-order two bits away, since they are not used for counting nanoseconds, and shift the value `0 × 90FBCE3C` to the right by two bits—this is equivalent to dividing by 4. As a result of this operation we get `0 × 243EF38F` or `608105359`, as shown in Listing 2.

[4] Tested with TSK version 4.4.2.

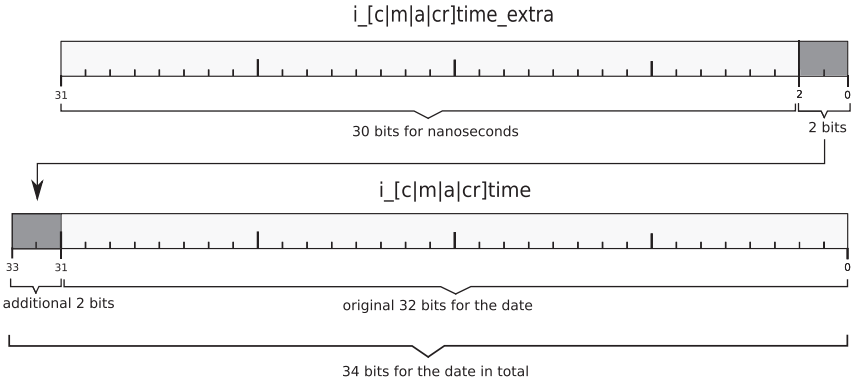

Fig. 2. Usage of the original ext4 timestamps in combination with the additional 32-bit fields.

Listing 2. Output of stat when parsing ext4 timestamps.

```
stat testfile.txt
Access: 2111-01-10 15:15:30.000000000 +0100
Modify: 2030-01-10 15:15:30.000000000 +0100
Change: 2017-02-23 13:13:09.608105359 +0100
```

Timestamps as steganographic carriers

We next turn to the question, which ext4 timestamps are suitable to build a robust steganographic system. As we have seen, it makes sense to conceal data only in the upper 30 bits of the extra timestamp fields because the remaining 34 bits of the timestamps are visible to the end user, i.e., inconsistencies could raise suspicions during the forensic investigation. Depending on the number of (mis)used timestamps, the available hiding capacity per inode table entry ranges from 30 bits, if only one timestamp is used, to 15 bytes, if all four timestamps are used. Since modern file systems typically contain up to millions of files, this gives us a total of a few megabytes, which can be used to hide secret information.

Just as in other file systems, specific timestamps can be overwritten at any time in ext4 due to user interactions (e.g., modification of the file content) or system interactions (e.g., change of file permissions or file size), since timestamps are temporally limited metadata that only describe the current status of a file or directory. Thus, the volatility of timestamps depends on the usage scenario. Some of the timestamps will change over time, while others remain unchanged.

The *creation* timestamp is static and thus suitable for data hiding, since it refers to the unique event when the file is created.

However, the *change* and *modification* timestamps are updated each time the file metadata is changed or the file content is modified. Such an update would destroy the hidden steganographic information. Usually only static information is considered as a carrier for steganographic systems, which is why the change and modification timestamps are not considered here any longer. Nevertheless, it is important to mention that they can be used theoretically, at least as long as neither the file content nor the file metadata changes (e.g., when using only static files, such as JPEG files, without changing the metadata attributes).

The *access* timestamp is updated each time a file is accessed or its directory is opened. However, in order to reduce file system overhead and increase performance, we can disable the recurring updates of access timestamps. This is done in many modern file systems by default (e.g., Microsoft uses this feature since Vista), and is often recommended for SSD storage devices and USB flash drives. In Linux, the following ext4 mount options can be set in the `/etc/fstab` `file`: (i) relatime, which is used on several Linux operating systems by default, will only change the last access time if `mtime` or `ctime` of a file is newer than `atime`, or `if atime` is older than a defined interval (1 day by default). If the mount options (ii) `noatime` and (iii) `nodiratime` are set, neither access to files nor directories updates the access timestamps. This reduces recurring disk access, since the inode tables do not have to be updated every time a file or directory is accessed. The access timestamp thus remains unchanged, in addition to the creation timestamp.

Since we want to hide information permanently, it is of interest what happens to the timestamp values when the file is deleted. While the fractional seconds in the `i_ctime_extra and i_mtime_extra fields` get a new value that corresponds to the time (nanoseconds) when the file was deleted, the `i_atime_extra and i_crtime_extra values` are not affected by the delete operation. So if we only use access and creation timestamps for the steganographic system, deleting a file does not directly lead to problems when restoring hidden data. Nevertheless, the inode of the deleted file gets unallocated in the inode bitmap and thus can be used at any time to write new data—at least then the hidden content is overwritten and the recovery process depends on an appropriate error correction method.

Methodology

The anti-forensic technique proposed in this paper aims to conceal data in ext4 timestamps that serve as steganographic carriers. In this section we provide details about our methodology. The detailed process and its implementation will be presented in Section Implementation.

A user of such an anti-forensic technique is primarily interested in the fact that the attacker cannot decide whether data is hidden in the timestamps or not. In addition, the user would also like to be able to use the anti-forensic technique with little technical knowledge. Therefore, we develop a steganographic storage system with high *secrecy* and easy *usability*.

Secrecy means the hardness of extracting the concealed artifacts within the timestamps. We make use of a two layer secrecy approach which bases on both steganography and cryptography. First, steganography is able to obfuscate the existence of the information storage by concealing the data as non-obvious information inside an honest-looking carrier (the timestamps) until the data is re-accessed (Hassan and Hijazi, 2016). This depends on the fact that the information hidden within timestamps can not be distinguished from timestamps created with normal system usage. We call this requirement *indistinguishability*. Second, cryptography provides additional protection, if the steganographic layer is discovered. We make use of an efficient symmetric stream cipher to

encrypt all information stored within timestamps. The symmetric key is derived from a password, which the user has to provide for each encryption or decryption, respectively.

Usability addresses two aspects. First, a good runtime efficiency of the information hiding and recovery process. Second, a reliable robustness of the hidden data during and after user manipulation. To be applicable in a wide range of scenarios, we assume a file system that is fully accessible, i.e., the user can access files and directories at any time, as well as create, rename or delete files and directories.

System requirements

In this case, the file system metadata, in particular the nanoseconds part of the timestamp, will be used as a carrier to conceal data. For the sake of clarity, no data structure other than the timestamps will be used to hide and restore data, i.e., the implementation does not use an additional file that keeps track of the inodes in which the data is hidden because such a file would be suspicious, even if it is encrypted.

To meet the requirements, we expect the system to automatically determine suitable timestamps and verify that enough timestamps are available in the current file system. Our implementation uses only allocated inodes to hide data because the parts of an inode table of unallocated inodes contain only zeros. Data hidden in unallocated inodes would raise suspicions during the investigation.

For additional security of the sensitive data, the steganographic technique is used in conjunction with encryption. Therefore, the secret message hidden in the proposed steganographic channel will hardly be exposed during the forensic investigation in an ordinary case.

Beyond that, the information will be stored in multiple steganographic carriers and thus the message will be distributed across the file system, making the forensic investigation even harder. Access to the hidden data is only possible if the cryptographic key for the encryption and the exact location within the file system is known. Both depend on the password the user enters.

Storage unit for data to be hidden

Fig. 3 depicts the additional 32-bit wide timestamp fields `i_[a|cr]time_extra`. This figure illustrates the data structure we use to hide the secret information, that has the size of 2.30 bits without using the lower two bits. Any such data structure consists of the access and creation timestamp of one inode table entry. Since we want to hide data that is much larger than the available capacity of a nanosecond timestamp object (60 bits = 7.5 bytes), the input message is split into multiple parts. These parts are called chunks in the following. For each chunk to be hidden, we use one of the structure shown in Fig. 3.

During the information hiding process, four bytes of data are written to the respective offset of both the access and creation timestamp. It is therefore necessary to ensure that only a maximum of 6 instead of 8 bits are used in the fourth and eighth byte of a chunk. This means we have to set the lower two bits to zero (timestamps are read in little-endian), otherwise this would raise the previously mentioned problem of dates beyond the year 2038 (cf. Section Analysis of ext4 timestamps).

Information processing

The correct sequence of writing and reading the individual chunks in and out of the filesystem must also be considered. Thus we need an ordered list of inodes that can be used as carriers.

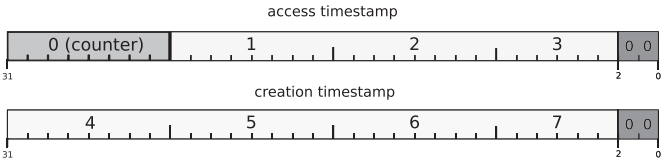

Fig. 3. Overview of how data is stored in the nanoseconds part of both the ext4 timestamp field `i_atime_extra` and `i_crtime_extra`.

Neuner et al. (2016) ensure a logical order by extracting and sorting all files by their creation time during the information hiding and recovery process. This approach does not seem plausible regarding the runtime efficiency, since extracting metadata first and sorting all files and directories can be complex in a large file system with lots of data. Instead, it makes sense to use file system data structures, which almost automatically provide the right order. In the case of NTFS, the sequential processing of the MFT records appears plausible. Similar to the MFT records, the consecutive inode numbers in the inode tables can be used in ext4 to find the next carrier when hiding and recovering data.

To detect the next valid timestamp during restore, there must be a reference point (e.g., a bit pattern or a counter). Therefore, each chunk is prepared with one byte that serves as a counter, as depicted in Fig. 3. This means that all timestamps which are used for data hiding are numbered. Since unallocated inodes are skipped, the implementation can use the counter to determine which of the following inodes has a valid value and thus contains hidden data. Due to the index byte, 6.5 bytes per inode remain for the user data. The smallest counter value 0×00 additionally contains the number of used inodes and the length of the entire message, including the error correction code. This information is essential for recovering hidden data. The use of only one index byte limits the data to be hidden to a maximum size of 255 bytes. According to Neuner et al. (2016), we reset the counter to 0×00 when an overflow occurs, i.e., different counters in several inodes may share the same value. In case of an overwritten start inode, the 256th subsequent inode may alternatively be used to obtain relevant information, such as the message length.

Since computers generally store data bytewise in memory, the data must be processed in a way that the two low-order bits contain zeros. Therefore, when reading the input, the data is first divided into several 7-byte chunks, of which two chunks generally overlap in 1 byte. This means the program reads 14 bytes of data of which 2.7bytes overlap in 1 byte, i.e., the last byte of chunk1 matches the first byte of chunk2 and so on. This step is relevant for subsequent operations, since we can discard 4 bits of data in each chunk using shift operations and logical bitwise operators without losing any relevant data. The processing that is done every time data in the size of two chunks is read is shown in Fig. 4 and Fig. 5. After preprocessing, the data matches the structure shown in Fig. 3. Therefore, we have to insert four zero bits (2.2bits) in each 7 byte chunk as low-order bits to get a valid date. Lastly, the resulting data structure (chunk) includes the zero bits and the index byte and has a total size of 8 bytes.

Encryption

Before embedding the information into the timestamps, the additional encryption is performed. This approach is a kind of fall-back security if the steganographic channel in the timestamps is disclosed. Using a counter results in patterns distributed across the inode table, which may reveal the existence of hidden information. The encryption provides randomly occurring timestamps, which

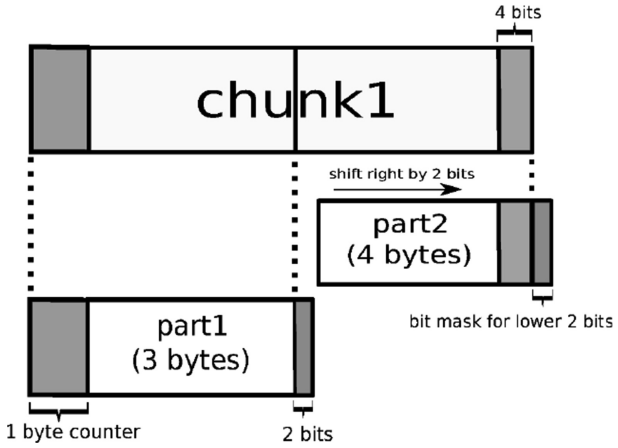

Fig. 4. First step in preprocessing a chunk.

neither allow the recognition of patterns, nor disclose hidden information without access to the cryptographic key. Encryption in form of a stream cipher is best suited for our needs. Unlike a block cipher, a stream cipher can be used without having a fixed size for the input block, i.e., the stream cipher is capable of any number of plaintext characters. Furthermore, using a stream cipher guarantees that the length of the input and output message is consistent, since each character of the plaintext is linked bitwise with the respective character of the key stream. Besides, the stream cipher is less vulnerable to a missing timestamp (30 bits) than the block cipher where a block of multiple bytes would not be recovered.

During information hiding and recovery a SHA-256 hash value is calculated that depends on the password the user entered. The generated hash value is responsible for the first inode number in which data is hidden. This allows the program to start hiding data from a random inode number, i.e., the data thus is hidden within different inodes depending on the password. If not enough inodes are allocated in the file system, the program starts in inode 12, since this is the traditional first usable inode number.

Error correction code

In general, when using file system internal data structures to conceal data, the file system can interfere with the hidden content.

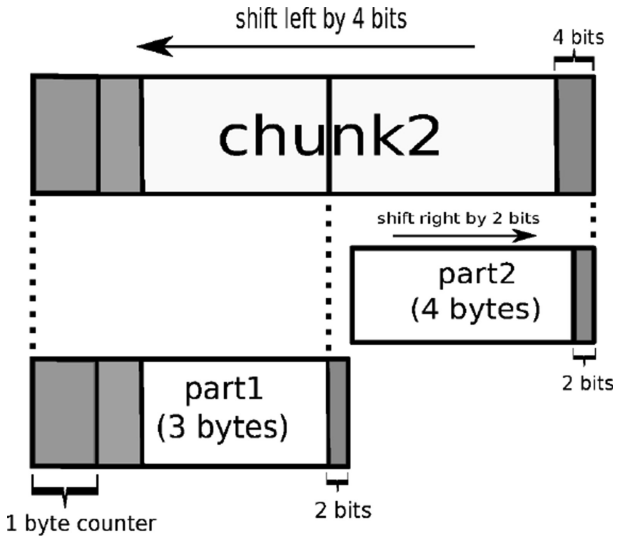

Fig. 5. Second step in preprocessing a chunk.

In a fully accessible file system, it is normal that file content and metadata is modified. By removing old files and creating new files, hidden information can be deleted at any time. In order to recover the information of modified or deleted timestamps, we use an error correction code (ECC) as integrity check to provide some redundancy of the hidden content. Since ECC is not in the focus of this research, an open-source Python library of the Reed Solomon algorithm is used. Since the amount of available memory within the timestamps is already limited, the ECC should not cause a significant overhead. In this case, the ECC adds 40 bytes of redundancy to each 215 byte of the incoming message, i.e., a redundancy of about 15 percent. This ensures that the redundant information is distributed over the entire message and thus over several timestamps, instead of storing the complete redundancy within few contiguous inode table entries.

Implementation

In this section we present in detail the process to hide and recover data in ext4 timestamps.

Relevant information for the data hiding process

The hiding and recovery process relies on various information from the superblock and group descriptor table.

First, to calculate the block number of the inode tables and inode bitmaps of the individual block groups, the total block count, the number of blocks per group, and the size of a group descriptor in bytes are read from the superblock. To get the total number of block groups in the file system, we need to divide the number of blocks by the number of blocks per block group.

$$nr\_of\_block\_groups = \frac{s\_blocks\_count\_lo}{s\_blocks\_per\_group} \tag{1}$$

We then iterate over the block group descriptor table with `nr_of_block_groups` iterations. For each block group we store the offset of the inode table and the offset of the inode bitmap.

Second, to later hide the data in the right place, we need to calculate the block number of the access and creation timestamps for each inode, respectively. The calculation is done as follows:

1. Calculate the block group of the respective inode:

$$block\_group = \frac{(inode\_nr - 1)}{s\_inodes\_per\_group} \tag{2}$$

2. Calculate the index of the used inode within the block group:

$$index = (inode\_nr - 1) \bmod s\_inodes\_per\_group \tag{3}$$

3. The program uses the previously stored list including the offsets of the inode tables to get the right offset of the inode table of the current block group that is stored in `offset_inodetable`.
4. The exact block number of the inode within the inode table is determined using the calculated index and the size of an inode:

$$offset\_inode = index \cdot s\_inode\_size \tag{4}$$

5. Using the aforementioned values, the block size in the file system and the fixed offset of the access and creation timestamps within the inode structure (cf. Table 1), finally the exact offsets

of the two timestamps can be calculated individually for each inode:

$$offset\_access = offset\_inodetable \cdot s\_log\_block\_size + offset\_inode + 140 \quad (5)$$

$$offset\_creation = offset\_inodetable \cdot s\_log\_block\_size + offset\_inode + 148 \quad (6)$$

Third, to check the allocation status of an inode, the program uses the inode bitmap of the corresponding block group. Therefore, the block group of the respective inode is calculated. The respective inode bitmap offset of the current block group is taken from the previously created list and stored in `offset_inodebitmap`. The byte- and bit-offset within the inode bitmap is determined as follows:

$$byte\_offset = ((inode\_nr - (offset\_inodebitmap \cdot s\_inodes\_per\_group)) - 1) \quad (7)$$

$$bit\_offset = ((inode\_nr - (offset\_inodebitmap \cdot s\_inodes\_per\_group)) - 1) \quad (8)$$

In addition, a loop with the total inode count of iterations determines whether sufficient inodes in the inode bitmaps are allocated. If there are not enough allocated inodes available, the program informs the user and stops.

Information hiding

The hiding process in our PoC implementation works as follows. First, the user starts the Python program and provides (i) a file to be hidden and (ii) a key to encrypt the data. The program then performs the following steps:

1. ECCs are calculated and concatenated with the message or data provided: $E = ECC(M)$.
2. The software determines the corresponding device file of the current path where the program gets started.
3. It reads necessary data from the superblock and the block group descriptor table, and thus calculates block numbers of inode tables, inode bitmaps and offsets of the access and creation timestamps.
4. It calculates the first inode that is used to hide data depending on the entered password (derived from the hash value of a cryptographic hash function) and determines the total number of allocated blocks. Therefore, the program verifies the allocation status of each available inode.
5. The number n of required inodes depends on the total length of the message including the error correction codes: $size = len(E)$. The data is then split in n chunks of 7 bytes each $(B_1, B_2, \ldots, B_n)$, of which two successive chunks overlap one byte.
6. Each chunk gets a one byte large counter prefix that works as a block index ($i \in \{1, 2, \ldots, n\}$). The counter starts again at 0×01 after reaching the value 0xFF. Starting with the chunk number 0×00, an additional chunk is inserted repeatedly every 256 chunks. This extra chunk contains the total size and the number of used inodes n. This results in chunks with a size of 8 bytes: $(0x00, n, size), (0x01, B_1), \ldots, (n, B_n)$.
7. All chunks are encrypted with a symmetric stream cipher using the previously entered password i.e., $C_1 = enc(0x01, B_1)$, $C_2 = enc(0x02, B_2), \ldots$ This results in encrypted chunks with a size of 8 bytes $(C_0, C_1, \ldots, C_n)$.
8. Each encrypted chunk is processed so that the lower two bits of the access and creation timestamp only contain zeros. After shift operations each chunk contains 6,5 bytes user data, 1 index byte and 4 zero bits i.e., each timestamp conceals 30 bits of the payload.
9. If necessary, the last chunk is padded with null bytes.
10. If enough inodes are allocated, the first part of a chunk is written into the access and the second part is written into the creation timestamp. Therefore, the offsets of both the access and creation timestamps are determined. Since we aim for a robust steganographic system, only allocated inodes, that are not set to zero, are used in the hiding process.

Recovery of hidden information

When restoring the hidden information, the user simply enters the password that was used to encrypt the data. The program then performs the following steps:

1. The program determines the device file associated with the path where the program gets started. This has to be the path where data is hidden.
2. It reads necessary data from the superblock and the block group descriptor table and thereby calculates all relevant, previously mentioned values.
3. Depending on the hash value of the entered password, the first inode containing hidden information is determined.
4. If the first inode starts with the value 0×00, this provides the number of used inodes n and the total size of the hidden message including ECC.
5. According to the value n, the program is able to read the right access and creation timestamps. It only processes timestamps that correspond to the next larger index value in this order: 0×01, 0×02, ..., n.
6. Previous shift operations are reversed. The original data is written back into the gaps of the chunks that have been created to prevent dates beyond 2038.
7. The padding in the last chunk is removed.
8. All chunks are decrypted with the symmetric stream cipher using the previously entered password, e.g., $(1, B_1) = dec(C_1)$.
9. All leading counter bytes as wells as chunks that contain the total size of the message are removed. In addition, the overlaps of two chunks are removed.
10. The original file is restored by calling the ECC function: $M = ECC(E)$.

File system check to repair invalid inode checksums

In ext4, all major data structures use metadata checksums. This also applies to the inode table, since each inode has its own checksum (Wong, 2016). Overwriting the nanosecond timestamps causes the original inode checksums to no longer match the contents of the inode structures. Therefore, we need to repair the inode checksums (matching the new content), otherwise this would raise suspicions during the forensic investigation. In ext4, the file system check tool `e2fsck` can be used to repair inconsistencies, such as wrong metadata checksums.

However, the used `e2fsck version`[5] contains a function which assumes garbage in the case of multiple incorrect inode checksums (Wong, 2017). For this reason, the tool asks the user if the corresponding inode entries should be removed. This would also remove the hidden content. The error message prevents the file system from being automatically repaired. Therefore, the source code of `e2fsck` was modified and recompiled[6]. The command e2fsck -p -f [image] then repairs all inconsistent inode checksums without additional user interaction. Another forced file system check does not give any further warnings. The volume can be mounted without problems.

As future work, we consider to integrate the CRC32C algorithm, that is used to calculate checksums within kernel code, directly into our PoC implementation.

Evaluation of the proposed hiding technique

The previous concept of our developed PoC implementation is now evaluated in terms of the following relevant aspects: *indistinguishability*, *correctness*, *robustness* and *available capacity*.

Indistinguishability

In this section, we evaluate if data hidden in the nanoseconds parts of ext4 is distinguishable from that of normal system usage. In case of the steganographic timestamps we expect a uniform probability distribution of the timestamps, since we use a stream chipher to encrypt the data before embedding it into the timestamps, which provides ones and zeros with equal probability.

It is of interest now, how ext4 handles the sub-second precision and whether normal timestamps differ from timestamps with hidden content. Therefore, we calculate the information entropy for both a variety of original and modified nanosecond timestamps. High information entropy corresponds to maximum uncertainty. This is what we assume, at least in timestamps including encrypted content.

Entropy is defined as:

$$H(x) = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i) \tag{9}$$

where $p(x_i)$ is the relative frequency of the output x_i with the additional definition of $p(x_i)\log_2 p(x_i) := 0$ in case of $p(x_i) = 0$.

First, we analyze the distribution of the nanoseconds part of the timestamps produced by normal system operations. In order to determine representative values for the timestamp distribution, we need to create as much files to get reasonable expected values (we set the expected value to 100 in case of the uniform distribution).

As we expect a deviation in the least significant part of the timestamps, we only consider the lower 10 bits of the nanosecond timestamps in our evaluation instead of the full spectrum of 30 bits. These bits include values in the range of 0–1023, i.e., values between 0 ns and slightly more than 1 μs could have passed within the current second since the Unix epoch. If we get a distribution in these bits that is nearly uniform, we can conclude that the full spectrum of timestamps with sub-second granularity also has a uniform distribution, since the lower 10 bits have the highest accuracy in the time specification.

To collect enough timestamps, we created (i) 100,000, (ii) 250,000 and (iii) 500,000 files in multiple iterations (fifty times in total) on a Linux system with an ext4 test volume using a Python script. Theoretically, we want to provide all possible values in the given range of the lower 10 bits, so we use a random delay of 1 ns to 1 μs between each creation of a file. This helps us to obtain realistic test values, in contrast to a file creation without delay, in which files are created as fast as the computer can handle it and several equal timestamps could appear.

We now collect the lower 10 bits of both the 30-bit wide access and creation timestamps of our created files. These two timestamps are equal to each other when the files are created, while they are different after data is hidden. We then build the absolute frequency, i.e., the number of times each nanosecond value occurs in all created files. We then normalize the absolute frequency by the total number of collected timestamps to get the relative frequency. With the previously calculated probability of each value, we now calculate the entropy. Since our experiments are with respect to the lower 10 bits, 10 is the maximum of entropy (in case the timestamps are uniformly distributed). Thus, slightly less than 10 bits is a more realistic value. To compare the entropy of original timestamps to the modified ones, we conceal our test file within the timestamps. After that, we calculate the entropy again.

We repeat the process of (i) creating files with random delay, (ii) calculating the entropy, (iii) hiding data within timestamps and (iv) calculating the entropy again multiple times to get meaningful values. In our experiments we hide a bitmap file with a size of 357,574 bytes (including ECC) which needs at least 55,228 allocated inodes.

The result is summarized in Table 2 in terms of minimum, maximum, standard deviation and the arithmetic mean results of the calculated entropy values for both original timestamps and tampered timestamps. We highlight two aspects: First, the timestamp distribution of the nanoseconds part itself (i.e., without hidden information) does not significantly deviate from a uniform distribution. Second, the entropy of timestamps with hidden information indicate that these timestamps are indistinguishable from that of normal file system operation since we use a stream cipher before embedding the content.

Correctness

To verify the integrity of the recovered data, SHA-256 is used as a hash function. That is, we calculate the hash value of the input file before hiding it, and calculate it again after restoring the data. If both values match, we are sure the same data is restored as previously hidden.

Further tests are performed with the above-mentioned Linux commands stat and debugfs, to verify that the dates (especially the year) have no unintended high values (cf. Section Analysis of ext4 timestamps). Verifying the correctness of the nanosecond timestamps can also be done manually by using a hex-viewer. The binary representation of 4-byte extra timestamp values containing the embedded information shows that the two low-order bits generally contain zeros. The conversion of timestamps can be done using date command, as shown in Section Analysis of ext4 timestamps. The hex-viewer also demonstrates that plain text patterns, in particular an ascending counter value, are no longer visible due to the encryption.

In addition, `e2fsck` verifies that the file system in which the data is hidden is in a consistent state. Since we repaired the inode checksums of all tampered inodes, the file system check does not give any further warnings.

Robustness

If the volume is mounted with the `noatime` option (which implies nodiratime), the embedded information can only be

[5] Tested with `e2fsck` version 1.43.4.

[6] Comment out the method `EXT2_SF_WARN_GARBAGE_INODES` in the `pass1.c` file.

Table 2
Minimum, maximum, standard deviation and arithmetic mean of the calculated entropy results.

	100,000		250,000		500,000	
	w/o data	w/data	w/o data	w/data	w/o data	w/data
MIN	9.771057856	9.808216785	9.795085524	9.964931144	9.975373588	9.986241694
MAX	9.986066967	9.94436697	9.994354388	9.987314605	9.995437745	9.993833209
SD	0.03894002	0.027292105	0.054713306	0.005360409	0.002860247	0.001523064
AVG	9.943489772	9.90690734	9.967231224	9.979583372	9.992873314	9.99153063

overwritten manually because the timestamps are no longer updated. Overwriting the access timestamps is done by mounting the image without the necessary mount option. Since we accessed multiple carrier files via cat the access timestamps were updated. In addition, several files were removed manually using the rm command. New files were created by the touch command. `stat` shows that access and creation timestamps were changed.

The maximum number of overwritten timestamps depends on the number of additional redundancy bytes. If a high amount of overwritten information is expected (e.g., on a multi-user system), the redundancy level can be increased. However, if the case of overwritten timestamps cannot occur, the error correction method can be skipped completely which results in an increased amount of concealable data. In the PoC implementation redundant information is intentionally distributed over the entire file system. Depending on ECC and file system usage, redundancy could also be stored in a fixed location (e.g., inodes of invariable/non-erasable files) so that it can not be lost.

Available capacity

The amount of usable space depends on the total size of the file system and the number of allocated inodes. To analyze the number of available inodes, we created ext4 test volumes with the size of 1 GB, 10 GB, 50 GB, and 100 GB, respectively using `mkfs.ext4` with default settings. Table 3 shows the number of available inodes in the images.

Table 4 shows that the proposed hiding technique quickly reaches its limits, this applies especially to large files. The previously hidden bitmap file required 55,228 inodes, which already consumes 90 percent of the maximum available inodes in the 1 GB image. A file with 1 MB already requires 183,190 inodes, i.e., a 3 GB image filled with data would be needed. For larger files, the amount of required inodes quickly exceeds one million. Therefore, this anti-forensic technique is not designed to hide large files, such as image or video files. Instead, small text files can be hidden, e.g., multiple files stored in a zip file which include passwords, personal data or other confidential data can be hidden at one time.

Implications on forensic investigations

Since we use encryption, entropy is not usable as a countermeasure. But if one decides to disable the cryptographic layer, a statistical analysis can be used for pattern recognition. Our implementation is a prototype. When developing a productive solution, there should be no installation files, execution traces or other artifacts which prove that an anti-forensic tool has been used, i.e., starting the Python program might appear in the history file if one does not consider to circumvent this. The language interpreter can also leave appropriate hints (e.g., log files). It is advantageous to check whether the chronological sequence of timestamps makes sense, i.e., forensic investigators should keep an eye on invalid states of timestamps, e.g., access or modification timestamps which take place before the creation time of a file, as well as change and modification timestamps which take place just a few nanoseconds after the creation time. In our implementation we got rid of such inconsistencies by appropriate adjustments of timestamp values which have a lower precision than nanoseconds. Furthermore, a proper anti-forensic tool should not store data anywhere else than in the memory (i.e., no swapping on the hard disk). If backups of the volume exist, different timestamps can be used as indication of hidden information. In addition, further file system internal data structures could be examined for their usefulness in this context (e.g., the journal). If operating system files were used as carriers, a timeline of the respective timestamps can be used to compare the chronological sequence of the timestamps of the installation files with the tampered creation timestamps in order to prove inconsistencies.

Table 3
Number of available inodes within several test images.

Volume size	Total number of available inodes
1 GB	61,057
10 GB	610,801
50 GB	3,055,617
100 GB	6,111,233

Conclusion and future work

In this paper, we analyzed the feasibility of using timestamps of the ext4 file system that serve as steganographic carriers to conceal data. We developed an anti-forensic technique that offers secrecy of the hidden data and easy usability, even for users with little technical knowledge. We presented in detail the process of information hiding and recovery in the case of ext4. The evaluation shows that our PoC implementation satisfies our requirements and actually works in practice. Thus, hiding information within ext4 timestamps with sub-second granularity is feasible, also in a file system that is fully accessible. We evaluated our methodology in terms of calculating the entropy for both regular and tampered timestamps, verifying the correctness and robustness of the anti-forensic technique with appropriate Linux commands, such as the file system check, and presenting the maximum available capacity to conceal data. Due to the use of a two layer secrecy approach which bases on both steganography and cryptography, the embedded content is statistically indistinguishable from regular timestamps.

As future work, we consider the combination of this method with other ext4 anti-forensic techniques to achieve a higher

Table 4
Required allocated inodes depending on the input.

Size of the input	Required allocated inodes
1 KB	186
100 KB	18,325
500 KB	91,595
1 MB	183,190
5 MB	915,923
10 MB	1,831,846
50 MB	9,159,219

capacity to hide larger data. Therefore, different hiding methods can be used at the same time, as described by Göbel and Baier (2018). Besides, we consider to integrate the calculation of the metadata checksums directly into the PoC implementation and thus avoid the separate calculation using `e2fsck`. Furthermore, the evaluation of the number of timestamps, which can be overwritten without disturbing the steganographic system, could be completed. In addition, a transfer of this hiding approach to other file systems with nanosecond granularity is conceivable, e.g., to the upcoming major Linux file system btrfs.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the funding program Forschung an Fachhochschulen (contract number 13FH019IB6) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de).

References

Anderson, R., Needham, R., Shamir, A., 1998. The steganographic file system. In: Information Hiding. Springer, pp. 73–82.

Baggili, I., BaAbdallah, A., Al-Safi, D., Marrington, A., 2012. Research trends in digital forensic science: an empirical analysis of published research. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 144–157.

Berghel, H., Hoelzer, D., Sthultz, M., 2008. Data hiding tactics for windows and unix file systems. Adv. Comput. 74, 1–17.

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

Conlan, K., Baggili, I., Breitinger, F., 2016. Anti-forensics: furthering digital forensic science through a new extended, granular taxonomy. Digit. Invest. 18, S66–S75.

Eckstein, K., Jahnke, M., 2005. Data hiding in journaling file systems. In: DFRWS.

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

Göbel, T., Baier, H., 2018. Anti-forensic capacity and detection rating of hidden data in the ext4 filesystem. In: Advances in Digital Forensics XIV. Springer.

Grugq, T., 2005. The art of defiling: defeating forensic analysis. Blackhat. Brief. 2005. Las Vegas, NV. . (Accessed 8 October 2017).

Harris, R., 2006. Arriving at an anti-forensics consensus: examining how to define and control the anti-forensics problem. Digit. Invest. 3, 44–49.

Hassan, N.A., Hijazi, R., 2016. Data Hiding Techniques in Windows OS: a Practical Approach to Investigation and Defense. Syngress.

Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L., 2007. The New ext4 Filesystem: Current Status and Future Plans.

McDonald, A.D., Kuhn, M.G., 1999. Stegfs: A Steganographic File System for Linux, in: International Workshop on Information Hiding. Springer, 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. Digit. Invest. 18, 76–86.

Piper, S., Davis, M., Manes, G., Shenoi, S., 2005. Detecting hidden data in ext2/ext3 file systems. Adv. Digit. Forensic 245–256.

Rogers, M., 2005. Anti-forensics. Center for Education and Research in Information Assurance & Security (CERIAS). Department of Information and Computer Technology, Purdue University.

Ts'o, T., 2010. Theodore Ts'o. Blog - Android will be Using ext4 Starting with Gingerbread. https://thunk.org/tytso/blog/2010/12/12/android-will-be-using-ext4-starting-with-gingerbread. (Accessed 1 October 2017).

Ts'o, T., 2017. Git - Ext4 Filesystem Tree. https://git.kernel.org/pub/scm/linux/kernel/git/tytso/ext4.git. (Accessed 1 October 2017).

Wong, D.J., 2016. Ext4 Disk Layout - Ext4 Wiki. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout. (Accessed 1 October 2017).

Wong, D.J., 2017. e2fsck: Offer to clear Inode Table Blocks that are Insane. https://patchwork.ozlabs.org/patch/375833. (Accessed 15 October 2017).