Time is on my side: Steganography in filesystem metadata

Sebastian Neuner[a,*], Artemios G.Voyiatzis[a], Martin Schmiedecker[a], Stefan Brunthaler[a], Stefan Katzenbeisser[b], Edgar R. Weippl[a]

[a]*SBA Research, Vienna, Austria*
[b]*Technische Universität Darmstadt, Germany*

Abstract

We propose and explore the applicability of file timestamps as a steganographic channel. We identify an information gap between storage and usage of timestamps in modern operating systems that use high-precision timers. Building on this, we describe a layered design of a steganographic system that offers stealthiness, robustness, and wide applicability. The proposed design is evaluated through theoretical, evidence-based, and experimental analysis for the case of NTFS using datasets comprising millions of files. We report a proof-of-concept implementation and confirm that the embedded information is indistinguishable from that of a normal filesystem use. Finally, we discuss the digital forensics analysis implications of this new information-hiding technique.

Keywords: digital forensics, data hiding, steganography, storage forensics, file system forensics, real-world data corpus

1. Introduction

The need for protected information exchange and storage in the digital world is constantly increasing. Cryptographic techniques can provide information confidentiality, authenticity, and integrity. However, they do leave evidence of the information exchange.

Steganographic techniques are able to hide the existence of information passing through communication channels or resting in storage media for later access. These techniques are useful in a wide range of real-world scenarios, including but not limited to: circumventing censorship and restrictions imposed by governments and other adversaries [1, 2], assisting whistleblowers when disclosing documents [3], and supporting businesses to protect strategic corporate information during transmission [4].

Numerous steganographic techniques have been proposed and analyzed in the research literature [5]. The analysis focuses on criteria such as the achieved secrecy on specific application scenarios, the steganographic channel capacity, and the information channel utilization.

Storage or format-oriented steganographic techniques hide information in logical channels by utilizing redundant or unused fields in format specifications. This includes, among others, the master boot record (MBR) of non-bootable hard disks and the unused disk space caused by the misalignment of hard disk sector size and file size [6].

Modern filesystems support a wealth of operations that span beyond the primitive of mapping files into sequences of hard disk sectors. The filesystem specifications define additional data structures (i.e., "metadata") to describe information like the owner, the access permissions, and the date and time when important file events took place.

In this paper, we propose and explore, to the best of our knowledge for the first time in literature, the applicability of *filesystem timestamps* as a steganographic channel. More specifically, we make the following contributions:

1. We analyze the granularity of the timestamps that modern filesystems implement, and we evaluate their applicability for steganographic applications.
2. We propose the use of timestamps as a means to hide information in `NTFS` and other filesystems with sub-second timestamp granularity.
3. We describe a system design and a proof-of-concept implementation that support different levels of possible capacity to securely hide data on `NTFS` volumes.
4. We validate the proposed system using real-world and synthetic datasets, and we show that the embedded steganographic information cannot be distinguished from the information produced by normal filesystem operations.
5. We discuss the digital forensics implications of this new steganographic method.

The rest of this paper is organized as follows: Section 2 provides a literature review on steganography with emphasis on storage artefacts. Section 3 analyzes the

*Corresponding author
Email addresses: `sneuner@sba-research.org` (Sebastian Neuner), `avoyiatzis@sba-research.org` (Artemios G.Voyiatzis), `mschmiedecker@sba-research.org` (Martin Schmiedecker), `sbrunthaler@sba-research.org` (Stefan Brunthaler), `katzenbeisser@seceng.informatik.tu-darmstadt.de` (Stefan Katzenbeisser), `eweippl@sba-research.org` (Edgar R. Weippl)

use of timestamps on modern filesystems. Section 4 proposes a novel steganographic channel based on file timestamps. Section 5 evaluates the security of the proposed system. Section 6 describes a proof-of-concept implementation aiming at NTFS filesystems. Section 7 discusses the implications on the digital forensics process. Finally, Section 8 concludes the paper and presents the future directions of our work.

2. Background

2.1. Data hiding

Early works on digital steganography focused on hiding data in the clear, deriving and discussing different methods of embedding data, and arguing how steganography is and probably will be used in the present and in the near future [7, 5]. Such works did not anticipate the widespread use of the personal digital devices and the role of the Internet in our daily lives [8, 9].

A considerable amount of research was devoted to embedding unobservable communication within normal network traffic, ranging from the utilization of TCP/IP timestamps [10] to the more general usage of TCP/IP fields [11]. Many implementations of steganography hide encrypted data in innocent-looking network traffic (e.g., `ptunnel` [12]), header fields [13], or use timing intervals and artificial transmission delays for information transmission [14, 15, 16]. While it has been shown that secure steganographic protocols are feasible, we are still lacking functional implementations and widespread use of such tools [17].

A second line of research focused on embedding unobservable information within the contents of stored files, introducing undetectable degradation of multimedia quality (e.g., manipulating the low-significance bits of pixel representation in images [18]), the color palettes in GIF images [19], or (possibly) encoding information in YouTube videos that look like static snow [20].

2.2. Filesystems

A plethora of different filesystems is available, including FAT and NTFS for Microsoft-Windows-based devices, ext4 and btrfs for GNU/Linux systems, and HFS+ for Apple OS X and iOS devices[1]. Most of them store different artefacts at various levels of granularity and detail, collectively known as "filesystem metadata".

Filesystem metadata can be classified in five categories: *file system*, *application*, *file name*, *content*, and *generic* metadata [21]. *File system* metadata are information on how the filesystem is to be read and where the important data structures reside. *Application* metadata are information useful for the application utilizing the filesystem, such as the file owner and the file access permissions. *File name* metadata are information for the human-readable names mapping to logical data locations. *Content* metadata are information about the logical addressing of the files, the file allocation status, and the actual data of the files. *Generic* metadata are information mostly used internally by the filesystem for its operations. This includes information such as the timestamps of various events in the lifecycle of a file.

2.3. Steganography using filesystem metadata

The topic of hiding data in filesystem metadata was heavily discussed in the late 1990s [22]. Back then, export restrictions on the use of strong cryptographic algorithms outside the USA were in place, and there was an increased concern by the public regarding key escrow. StegFS, a steganographic filesystem compatible with the Linux `ext2` filesystem, was developed [23, 24]. This filesystem achieved plausible deniability of the hidden content thanks to its indistinguishability from unused content. This behavior was achieved by applying encryption on the content under the assumption that a good encryption algorithm ensures that encrypted data appear as random data. However, the use of StegFS is not undetectable as the needed filesystem driver is not hidden. Additionally, there is no integrity check of the data. Thus, StegFS cannot recover from any kind of intrusive data modifications.

Encoding (hiding) information in the order that a filesystem indexes the file names is explored in [25]. The approach is applicable only in the case of a FAT filesystem and cannot be generalized. The file fragmentation is explored in [6]. This approach introduces significant performance penalties, more evident in magnetic storage media, in the form of delays when accessing the file contents. This delay is due to the heavy file fragmentation that is enforced in order to create the steganographic channel. The delay and the heavy fragmentation can serve as indicators for the presence of steganographic information, thus *defeating* the steganography. Furthermore, (automatic) defragmentation of the storage medium can *destroy* the steganographic information.

Application metadata (e.g., the file owner or the file access permissions) can encode only a few bytes of information and the encoding is easily detected. For example, it is technically feasible to attach an arbitrarily large list of user–permission pairs in an NTFS file, even by referencing non-existent users [26]. However, the mismatch of the users mentioned in the system user list and the user–permission pairs, on top of having such long lists in first place, would raise suspicions in a forensics investigator.

The file name cannot be considered as a good candidate for steganographic operations. Indeed, an odd pattern of filenames will look instantly suspicious.

Mixing steganographic information with the actual contents of a file is studied extensively [7, 27, 28, 29, 30]. Format containers for multimedia content (e.g., audio or video) are transparent to and independent of the underlying filesystem that hosts the multimedia file. Thus, a filesystem-level analysis will not be able to disclose the

[1]An exhaustive list is provided in the Wikipedia entry available at `https://en.wikipedia.org/wiki/Comparison_of_file_systems`.

presence of steganographic information in a format container. Also, we note that multimedia transcoding can effectively *destroy* the steganographic information without significantly affecting the original information channel.

Generic metadata, such as temporal information describing the lifecycle of a file, are very sensitive to both the actions of the user and the operating system itself. For example, certain timestamps of file events can be overwritten at any moment while using the filesystem in a normal way. This includes a timestamp of the (last) file modification and (last) access of the file. The fragility of the temporal information might be the reason why, to the best of our knowledge, timestamps have not yet been explored as a steganographic means.

3. Timestamps in modern filesystems

We analyze in the following paragraphs how modern filesystems use timestamps. The assumption we seek to validate is that there is unused (redundant) capacity in timestamps that is sufficient enough to create a logical channel with steganographic strength.

NTFS is the standard filesystem for Microsoft Windows operating systems. NTFS uses the number of 100 nanoseconds passed since January 1, 1601 for its timestamps [31]. The timestamps are saved as 64-bit values in the `$Standard_Information` field of the Master File Table (MFT). Additionally, they are saved in the NTFS attribute `$FILE_NAME`. Each file has four 64-bit timestamps: (i) creation of the file, (ii) last access of the file, (iii) last modification of the file, and (iv) last modification of the corresponding MFT entry.

ext4 is the successor of the Linux standard filesystem `ext3`. Ext4 uses 64-bit values to represent timestamps with nanosecond granularity [32, 33, 34]. Ext4 uses the following four timestamps per file: (i) creation of the file, (ii) last modification of the file, (iii) last access of the file, and (iv) the last attribute modification of the file (e.g., permissions or ownership).

btrfs is the upcoming major filesystem for the Linux operating systems [35]. It is a “copy-on-write” filesystem based on B-trees. All file timestamps in `btrfs` support nanosecond granularity and are saved as 64-bit values [36]. The first 32 bits of the timestamps are the seconds since the epoch (January 1, 1970) and the remaining 32 bits are the nanoseconds since the beginning of the second. The provided per-file timestamps include: (i) creation, (ii) last modification, (iii) last modification of the file's attributes (e.g., permissions or ownership), and (iv) last access.

ZFS is intended to be a highly performing, decentralized filesystem [37]. The following per-file timestamps of ZFS have a nanosecond granularity, saved in 64 bits each: (i) creation, (ii) last modification, (iii) last access, and (iv) the last attribute modification [38].

FAT32 is the predecessor filesystem of NTFS on the Microsoft Windows operating system. FAT32 uses three different timestamps per file: (i) creation, (ii) last modification, and (iii) last access. The first two timestamps are saved as 32-bit values and the last one is saved as a 16-bit value. The difference comes from the fact that the first two timestamps are provided with a granularity of two seconds, whereas the date of last access is provided with a granularity of one day [39].

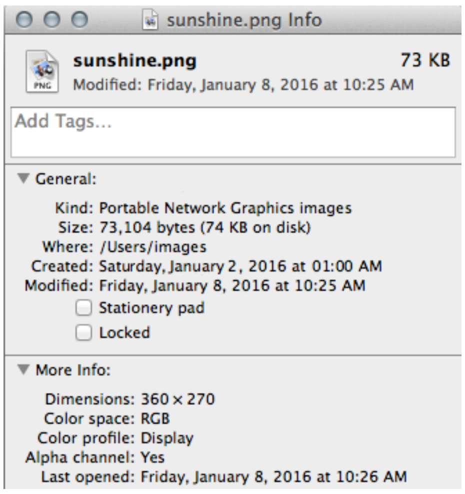
HFS+ is the standard filesystem for the Apple Macintosh and iOS devices. HFS+ uses the following per-file timestamps: (i) creation, (ii) content modification, (iii) last attribute modification, (iv) last access, and (v) the last backup [40]. All of these timestamps have a granularity of one second and are saved as 32-bit values.

ext3 is the successor of the `ext2` filesystem and enhances it by providing journaling capabilities. Ext3 uses three timestamps per file: (i) last access, (ii) last modification, and (iii) last attribute modification. The timestamps have a granularity of one second and are saved as 32-bit values. The use of the undocumented large-size `inode` feature can increase the granularity of the timestamps to one nanosecond [41].

Table 1 summarizes our analysis. We confirm that many modern filesystems use 64-bit values as timestamps and offer sub-second granularity [41]. This statement covers all filesystems that mainstream consumer operating systems use or access nowadays (e.g., Apple OS X, Google Android, GNU/Linux, and Microsoft Windows) with the exception of the HFS+.

Three file timestamps, namely *creation*, *access*, and *modification* are supported by almost all the analyzed filesystems. All three timestamps store date and time information with sub-second granularity (one or 100 ns).

The nanosecond precision is not communicated, explicitly or implicitly, to the end users who access the filesystem. They are confronted with file timestamp information that resolves to a second granularity, as depicted in Figure 1. Thus, there is an information gap between how timestamps are stored and how they are used.



sunshine.png Info

sunshine.png 73 KB
Modified: Friday, January 8, 2016 at 10:25 AM

Add Tags...

General:
Kind: Portable Network Graphics images
Size: 73,104 bytes (74 KB on disk)
Where: /Users/images
Created: Saturday, January 2, 2016 at 01:00 AM
Modified: Friday, January 8, 2016 at 10:25 AM
☐ Stationery pad
☐ Locked

More Info:
Dimensions: 360 × 270
Color space: RGB
Color profile: Display
Alpha channel: Yes
Last opened: Friday, January 8, 2016 at 10:26 AM

Figure 1: How file timestamps are displayed to Apple OS X users.

The *creation* timestamp is by and large a static piece of information, as it refers to a unique event, the creation of the file itself. The *access* and *modification* timestamps are updated each time a file is accessed or modified.

Modern operating systems are exploiting latest advances in storage technologies to deliver increased performance and reliability while reducing costs. USB flash drives and SSD storage media are commonplace. In this setting it is advisable, if the application scenario allows so, to reduce filesystem overheads for timestamp housekeeping. This includes disabling the update of per-file access and/or modification timestamps. Such an approach can increase both the performance and the lifetime of a storage medium. Yet, in most consumer-grade usage scenarios, we can expect that only the *access* timestamp remains intact.

The analysis above validates the first part of our initial assumption: there is unused (redundant) capacity in filesystem timestamps. Depending on the filesystem and usage scenario, this capacity ranges from one to nine bytes per file. With modern filesystems hosting hundreds of thousands or even millions of files, this provides enough capacity for storing up to a few megabytes of extra information. In the next sections, we explore how the available capacity can be utilized to create a logical channel with steganographic strength.

Table 1: Characteristics of filesystem timestamps.

Filesystem	File timestamp	Size	Granularity
NTFS	creation	64 bits	100 ns
	access	64 bits	100 ns
	modification	64 bits	100 ns
	modif. of MFT entry	64 bits	100 ns
ext4	creation	64 bits	1 ns
	access	64 bits	1 ns
	modification	64 bits	1 ns
	attribute modif.	64 bits	1 ns
btrfs	creation	64 bits	1 ns
	access	64 bits	1 ns
	modification	64 bits	1 ns
	attribute modif.	64 bits	1 ns
ZFS	creation	64 bits	1 ns
	access	64 bits	1 ns
	modification	64 bits	1 ns
	attribute modif.	64 bits	1 ns
FAT32	creation	32 bits	2 sec
	access	16 bits	1 day
	modification	32 bits	2 sec
HFS+	creation	32 bits	1 sec
	access	32 bits	1 sec
	modification	32 bits	1 sec
	attribute modif.	32 bits	1 sec
	backup	32 bits	1 sec
ext3	access	32 bits	1 sec
	modification	32 bits	1 sec
	attribute modif.	32 bits	1 sec

4. Steganography based on file timestamps

We assume a threat model where the attackers can inspect the file contents and can manipulate the filesystem metadata. Also, the attackers can freely remove, rename, or insert new files in the filesystem, and they accept the associated risk of thereby disclosing their presence.

We aim for a steganographic storage system based on file timestamps, namely TOMS (Time-On-My-Side) that is stealthy, robust, and applicable in a wide range of scenarios. "Stealthy" means that the attacker cannot deduce the presence or absence of steganographic information by examining the timestamps. Thus, the attackers are left only with the option of a denial-of-service attack i.e., to overwrite all timestamps and destroy the steganographic channel, thereby disclosing their presence. "Robust" means that the system can sustain and recover from file manipulation attacks. "Widely applicable" means that the system can be configured to match different operation scenarios, balancing performance and secrecy.

4.1. System design

The aim of the TOMS system is to hide an input (the *message*) inside the metadata of the hosting filesystem (the *carrier*). For the sake of clarity, we assume that the system can identify the necessary space (i.e., the file timestamps to use) and that all the necessary space is already available. We will return on this issue at the end of the design description (cf. Section 4.2).

The design of TOMS follows a layered approach. From top to bottom, the system comprises: (i) a storage container layer for the message, (ii) an error correction layer for redundancy, and (iii) an encryption layer.

4.1.1. Storage container layer

The storage container layer maps the message into the underlying file timestamp metadata elementary storage units. The naïve approach for keeping track which files and directories have been used to embed the information is to keep an *encrypted metadata file* with the absolute paths of the files and directories used. The metadata file approach has the benefit that the correct ordering of the files to extract the information is trivial. Also, this file does not necessarily need to be stored in the same filesystem with the hidden data. Rather, it can be stored in another storage media altogether. This is beneficial, since the very presence of a metadata file inside the examined filesystem is neither elegant nor stealthy, even if its contents are encrypted. On the contrary, such an encrypted file can raise further suspicions.

A second option is to reliably embed and extract information *only* based on the files and their timestamps. This can be realized using *oblivious replacements* on whole filesystems (e.g., an NTFS volume) or on the subfolder level (e.g., an NTFS non-root directory). In this case, all files and directories are sorted by their creation timestamp, either globally (filesystem level) or locally (subfolder level). This ordering defines a (logically) continuous storage space that can be used to write and later read the hidden data.

4.1.2. Error correction layer

The normal use of the storage medium hosting the filesystem as well as the actions of the attackers may remove some of the files stored on the filesystem. Also, the attackers might intentionally change the creation timestamps of some of the files. Such actions, deliberate or not, cause a new ordering of the creation timestamps, which results in the inability to either access the input file segments in the correct order or altogether.

The error correction layer augments the initial representation of the input file with additional information that can cope with the aforementioned issues. As a first step, an error correcting code (ECC) is appended to the representation. The ECC can both detect and reconstruct missing information. As a second step, this layer enforces a *start offset* for the used files. This allows the program to start from a random point in the ordering and use both older and newer files. Thus, not only old files are used to hide information.

The selection of an appropriate ECC is left to the implementation. By and large, an ECC should not introduce significant storage overhead.

4.1.3. Encryption layer

The error correction layer introduces data redundancy. This redundancy comes on top of the structured information needed to represent the links from timestamp to timestamp in order to form a logically continuous storage space. These can be sources that result in timestamps with patterns. If patterns are detected, the whole steganographic system will collapse, since they reveal the existence of hidden information.

The role of the encryption layer is twofold: On the one hand it protects the hidden information from disclosure. Only somebody in possession of the related cryptographic key(s) can access the encrypted and hidden information. On the other hand the confusion and diffusion properties of a (good) secure cipher ensure that hardly any pattern will exist in the output allowing it to appear totally random.

The encryption layer uses symmetric-key cryptographic algorithms to encrypt the information of the two previous layers. Stream ciphers, as for example AES-OFB or RC4, can be used in this layer. The advantage of stream ciphers over block ciphers is that the former do not need to expand the output of the operation and that they can recover to a certain point from errors (e.g., missing timestamps) at a bit rather than a block (i.e., dozens of bytes) level.

4.2. Information representation: the case of NTFS

We can now describe how the TOMS components work together to hide a message in the file timestamps. In the following, we will use the NTFS filesystem as an example. However, the description is valid for any other filesystem with similar characteristics. Figure 2 depicts the NTFS `inode` data structure used to represent various filesystem objects, including a file and a directory. In the following, we will use the term "file" to refer to NTFS `inodes`.

Two file timestamps can be used by TOMS in the case of NTFS: the *creation* and the *last access*. Each timestamp uses 24 bits to represent its nanoseconds part. Thus, a total of six bytes per file can be used to hide information. This constitutes the elementary storage unit (ESU) for the TOMS system. We assume that the size of the input (steganographic) message, M, is much larger than the size of an ESU. First, an error correcting code function is applied to the input message, $E = ECC(M)$. Then n, the number of ESUs needed to store E, is prepended ($n||E$).

4.2.1. Information hiding

The information hiding process works as follows. The encoded message (E) is fragmented into n blocks of five bytes each ($B_1, B_2, \ldots, B_n$). Then, every block is prepended with one byte that is used as a block index ($i \in \{1, \ldots, n\}$). The special value of `0x00` for the index byte is used to prepend a block of five bytes that contains the number of needed ESUs, n. The resulting structure is a linked list of six-byte blocks: $(0, n), (1, B_1), \ldots, (n, B_n)$. This structure is then encrypted with a stream cipher and a secret key, producing an output list of six-byte blocks: $C_0, C_1, \ldots, C_n$.

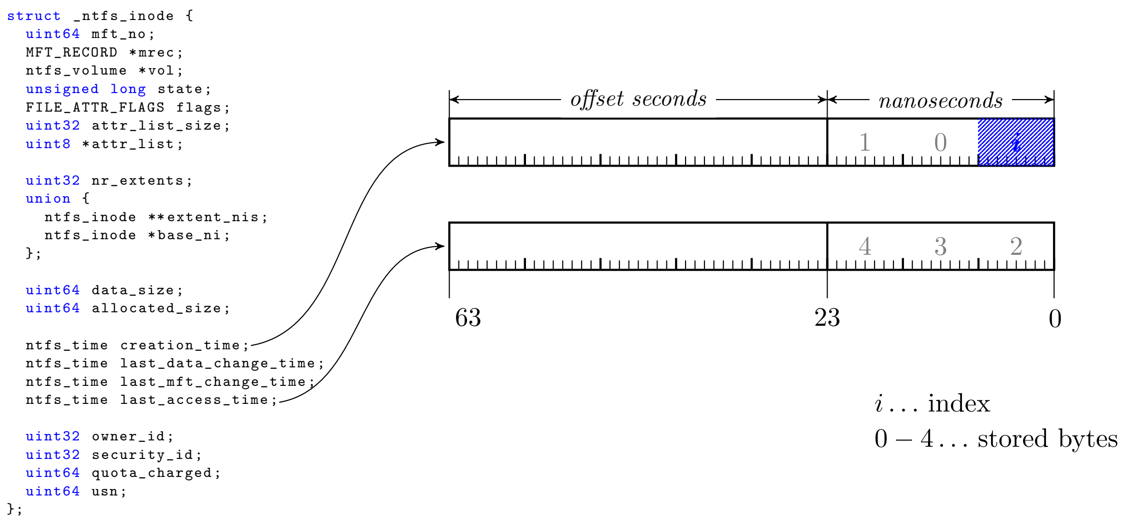
TOMS constructs the list of candidate files that they can be used as carriers. The list, F, is ordered based on the creation timestamp of each file, and a start offset, s, is chosen randomly. The ordered list of files, $F_s = \{f_s, f_{s+1}, \ldots, f_{s+n}\} \subseteq F$, will be used as the carrier. TOMS then proceeds and replaces the nanoseconds part of the creation and access timestamps of each file in F_s with the six-byte encrypted chunk C_i.

4.2.2. Large message handling

Using just one byte as index limits the length of the hidden message (E) to only 255 bytes. We overcome this limitation by allowing multiple index bytes to share the same value (overflow upon reaching the value `0xFF` and restart numbering from `0x01`). Whenever an overflow occurs, an ESU is consumed in order to store the length of the whole message, using again an index byte of `0x00`. Thus, every ESU with an index byte holding the value `0x00` contains the *total* length of the message.

4.2.3. Recall of hidden information

The information recall process works as follows. The timestamps for *all* the files in the filesystem are extracted, sorted by their creation time, and then saved as a list G. For every list entry, the nanoseconds part of the creation and the access timestamps are decrypted by applying the same stream cipher and key material used during the information hiding process. If the decrypted first byte of the creation timestamp equals the index byte value `0x00`, the respective timestamps are added in an offset list L and the number of ESUs, n, is recovered. Then, the next n list entries are processed, recovering the respective index (i.e., `0x01`, `0x02`, ..., `n`) and the structure H. Next, the error correction code function is applied on H, recovering the original (hidden) message M.



```
struct _ntfs_inode {
  uint64 mft_no;
  MFT_RECORD *mrec;
  ntfs_volume *vol;
  unsigned long state;
  FILE_ATTR_FLAGS flags;
  uint32 attr_list_size;
  uint8 *attr_list;

  uint32 nr_extents;
  union {
    ntfs_inode **extent_nis;
    ntfs_inode *base_ni;
  };

  uint64 data_size;
  uint64 allocated_size;

  ntfs_time creation_time;
  ntfs_time last_data_change_time;
  ntfs_time last_mft_change_time;
  ntfs_time last_access_time;

  uint32 owner_id;
  uint32 security_id;
  uint64 quota_charged;
  uint64 usn;
};
```

Figure 2: Overview of storing data in the nanoseconds part of the timestamp fields.

5. Evaluation of the TOMS system

In this section, we evaluate the design principles of the TOMS system. Our evaluation is based on theoretical, experimental, and evidence-based analysis of the steganographic strength of TOMS under the assumed threat model (cf. Section 4).

5.1. Stealthiness

"Stealthiness" describes the degree to which the very existence of hidden information is disguised, irrespectively of the ability to recover the hidden message(s). We analyze the two factors defining the stealthiness of the TOMS system in the following.

5.1.1. Timestamp handling by operating systems

The use of the encryption layer ensures that the steganographic information are not recoverable without having access to the key material of the stream cipher used. Furthermore, a good stream cipher ensures that each output bit will be a one or a zero with equal probability. But how do modern operating systems handle the timestamp information in first place? Do they fill these data structures with sub-second-precise information or do they opt for a different strategy? If the former, what is the precision of the provided time information? We sought the answer to these questions using three approaches.

5.1.2. Code audit

As a first approach, we performed a code audit of the NTFS-3g implementation of the NTFS filesystem [42]. This is the default driver for accessing NTFS volumes from within the Linux and Apple OS X operating systems, and it is an open source code. A similar code audit for the NTFS implementation of the Microsoft Windows operating system was beyond our reach, since the source code is not publicly available. The code audit revealed that the NTFS-3g fills the related timestamp structures with nanosecond-granular information provided by the Linux system clock which also has a nanosecond granularity.

5.1.3. Synthetic data

The second approach was to create synthetic data for experimentation, which is online at the authors website together with the most recent version of the code used for this purpose[2]. We created files in batches using a Python script on a Linux system accessing an NTFS volume via NTFS-3g. Each batch created 100,000, one million, or ten million different files. Half of the files were created with a random delay of one to two seconds between each creation. The other half of the files were created with zero delay, i.e., as fast as the computer system could handle the requests.

Our "synthetic" dataset contains 117 million files. We collected the three timestamps (create; access; and modify, all equal to each other) for this dataset as well. We conducted an exploratory data analysis to determine if the timestamp distribution was uniformly distributed. Results for the Kolmogorov-Smirnov goodness-of-fit test for uniformness indicated that the timestamp distribution did not deviate significantly from a uniform distribution ($D = .99, p = 2.2 \times 10^{-16}$).

The source code audit and the experiments validate that the Linux operating system uniformly uses the full spectrum of the 24-bit sub-second granularity to store the timestamp information.

5.1.4. Real data

The third approach was to collect evidence from Microsoft-Windows-based systems that are actively used to perform day-to-day tasks ("real-world systems"). We therefore collected the file timestamp information from a sample of 70

[2]https://www.sba-research.org/dfrws2016/

filesystems (NTFS volumes) from multiple Microsoft Windows computers available at our research lab. On average, each filesystem of our sample contained about 290,000 files and 40,000 directories; the largest one contained over 2.2 million files and directories. The majority of the sampled filesystems (n=63) were actually "system volumes", i.e., they contained the files of the Microsoft Windows operating system (e.g., those files commonly found in the `C:\Windows` directory) and (most likely) of the majority of installed software (e.g., those files commonly found in the `C:\Program Files` directory). Only seven systems contained more than one NTFS volume (i.e., "non-system volumes"). Such volumes are often used as storage for work or personal data (e.g., documents, spreadsheets, and pictures). In total, our "real-world" dataset contains the timestamps of 22,261,386 files and directories.

We analyzed the timestamps contained in this dataset and we noticed some irregularities in their distribution. Creation timestamps that are filled with zeroes in their nanoseconds parts were disproportinally more frequent than expected. This is the case when files are migrated into NTFS volumes from FAT32 filesystems. The latter use a granularity of two seconds at best, hence the zeroes. This assumption was empirically tested and further confirmed by Microsoft's documentation regarding timestamp changes [43].

5.1.5. Time of filesystem inspection

In the previous paragraphs, we saw that the timestamps can be used as stealthy information carriers, since the sub-second information follows a uniform distribution, as does the output of a stream cipher encryption. Before replacing any timestamps, one should consider if and how often the filesystem is inspected by an attacker. As an example, we consider the case of (operating) system files. These files are installed once and are seldom, if ever, touched again (e.g., only when operating system updates are installed). Thus, if their timestamps are proactively collected, any future modification by the TOMS system will be easily detected.

In a forensics analysis scenario, we can assume that the investigator will inspect the metadata *after* the message was hidden in the timestamps. We can also assume that the investigator does not have access to earlier versions of the filesystem metadata information. Thus, existing timestamps can be utilized to hide steganographic information. In a scenario where the filesystem can be proactively inspected, already existing files might not be good candidates for carriers. Thus, only new files (i.e., generated after the last inspection) can be utilized to hide steganographic information.

We assume for our subsequent analysis a more conservative scenario, in which the filesystem is *proactively* inspected. In this case, one should exlude all system files and all files with timestamps containing zeroes in the subsecond part. Applying this to our initial real-world dataset, it resulted in an almost 50% drop of available files, down to 11 million files spanning 70 NTFS volumes.

5.2. Robustness

"Robustness" refers to the ability of the TOMS system to cope with changes in the filesystem contents. The information hiding and recall procedure of TOMS is straightforward when the initial ordered list of files F_s remains intact between information hiding and information recall(s). In the following, we analyze how the TOMS system defends against actions that result in modifications of F_s.

5.2.1. Encrypted metadata

This is the simplest of the the proposed storage container layers. The ordered list of files F_s is not affected by operations on the filesystem level (assuming that these operations do not touch the timestamps). Should some files have been removed from the filesystem, or some timestamps are updated, the encryption and the error correction layers may be able to recover the lost information thanks to the use of the stream cipher and the ECC. If and how the recovered information is stored back to the timestamps (e.g., insert new files, re-encode information, or even fix the "corrupted" timestamps) is a decision to be made taking into account the severity of the errors and the assumed time and frequency of inspection.

5.2.2. Oblivious replacements

In this approach the ordered file list F_s results from sorting the timestamps that are provided by the filesystem. Thus, it might be the case that the TOMS system unknowingly uses a different list F_s' for information recall instead of the one originally used for information hiding. If some files were removed between information hiding and recall(s), the same arguments as in Section 5.2.1 apply.

We assume that some additional files, F_a, are included in the $F_s' = F_s \cup F_a$ list, and that the computing system has a proper clock. If oblivious replacement is applied globally (filesystem level), the TOMS system will always recover the correct F_s. This is feasible because the file creation timestamp is immutable on an NTFS volume, i.e., it does not change when the file is copied, renamed, or moved *within* the same NTFS volume [21]. Thus, even if files are moved across different NTFS folders, their creation timestamp will not change. Also, these additional files F_a will have more recent creation timestamps than those already contained in the original F_s and allow therefore a clear separation of the two sets.

If oblivious replacement is applied locally (subfolder level), then it is possible that the ordering of F_a is intermixed with the ordering of F_s. This is the case where files are moved to the specific subfolder containing F_s; as mentioned earlier, the file creation timestamps are immutable. This situation is accommodated by the use of the encryption and the error correction layers. Assume that a file $f_m \in F_a$ is inserted in the ordered list F_s. First, the ESUs of f_m must decrypt correctly and not be rejected by the encryption layer. Then, the value of the index byte contained in the (erroneously) decrypted ESUs must match the currently expected sequence number in order to not be rejected by the error correction layer. Finally, the payload information contained in the ESUs must pass through the ECC. Only then, these information are accepted as valid. If the processing of f_m fails, the error correction layer provides the necessary protection to recover from the error. Thus, the two layers provide an adequate defense (up to a certain point) against such (deliberate or not) insertion attacks. The amount of redundant information handled by the ECC defines this protection level. An oblivious replacement at the subfolder level requires a stronger ECC compared to the filesystem level.

5.3. Applicability

"Applicability" refers to the degree to which the TOMS system can be utilized in various application scenarios. The layered design of TOMS provides an initial indication of its wide applicability. The TOMS system supports three different storage layers and is agnostic to the ECC used as well as to the stream cipher. Furthermore, TOMS can be easily applied to any modern filesystem that supports sub-second timestamp granularity; while the basic description supports two timestamps often found in modern filesystems (namely, *creation* and *last access*), there is no design constraint regarding the number of timestamps or the use of filesystem-specific timestamps (e.g., *last attribute modification* for `ext4`). The design of the TOMS system allows to explore various performance tradeoffs in order to match the secrecy requirements of the selected application. We discuss these tradeoffs in the following paragraphs.

The application scenario defines the use of existing files or opts to create new ones to embed a steganographic message. In the latter case, it is advisable to generate small files that act as carriers (e.g., files in the range of few thousands bytes).

The use of an ECC introduces an overhead of 10-20%. If the risk of information loss can be sustained, the use of an ECC can be omitted altogether.

The selection of the storage container type is important. If an encrypted metadata file is used, one must decide if the contents of the file should be embedded in the filesystem or stored elsewhere. The resulting size of this metadata file can be a decisive factor. When embedding about 1.5 MB of data into 175,000 timestamps, the corresponding metadata file takes about 215 KB of disk space. A benefit of this approach is that there is no need to store index bytes to rebuild the ordered list of carrier files and recover the hidden information. Also, file reordering is not a threat in this case (unless someone is tampering with the metadata file) and thus the performance requirements for an ECC are more relaxed (or can be omitted altogether).

The oblivious replacement approach mandates the use of index bytes. Each ESU uses one index byte per five payload bytes (ratio of 1:5). If only one timestamp is available for each file, the ratio becomes 1:2, which may cause a lot of overhead. On the other hand, if three timestamps are available, this ratio becomes 1:8, which is quite efficient. Compared to an encrypted metadata approach, oblivious replacement needs between 12.5% (two timestamps) and 20% (three timestamps) more files in order to store the same amount of hidden information.

6. Experimental system validation

We developed a proof-of-concept implementation of the TOMS system for the experimental validation of our steganographic proposal. The implementation targets the NTFS filesystem and is based on the Python language version 2.7 for flexibility and increased portability. Our implementation can be delivered as a stand-alone executable and does not require the installation of special software or any modifications of the Linux kernel. It realizes the layered design described in Section 4 and can be easily ported to work with any filesystem that uses a nanosecond timestamp granularity.

The development and experimentation platforms are based on Xubuntu Linux 15.04 64-bit, running kernel version 3.19.0-25, the latest stable one at the time of writing. The underlying disk on which the operating system is installed is a solid state disk (SSD) for faster I/O access. As NTFS is Microsoft-proprietary, we opt for NTFS-3g in its current version. The steganographic executable application takes care of all information management tasks. The application is assumed to have full access to the NTFS volume (filesystem).

The application supports the use of two- and three-file timestamps. The file *creation* and *last access* timestamps are not modified by the operating system: starting with Microsoft Windows Vista, the default value of `NtfsDisableLastAccessUpdate` is set to one [44]. The corresponding mount option in Linux is `noatime`; in most of the popular Linux distributions this option is not activated by default. The file *last modified* may be modified under normal use, so it is up to the users to decide if they enable it (and pay attention not to destroy the related information during the normal use of the filesystem).

6.1. Information hiding and recall

The typical work flow for information hiding is as follows: the user starts the Python application and provides (i) the message to be hidden, (ii) a key to encrypt the message, (iii) the method for hiding (metadata file, oblivious replacements on volumes, oblivious replacements on subfolders), and (iv) the number of different timestamps to use (either two: *creation* and *access* or three: *creation*, *access*, and *modification*). Once the necessary information are collected, the application performs the following steps: (i) it concatenates the message with the error correction code, (ii) adds the index bytes to the resulting data (if

Listing 1: Embedding data in timestamps.

```
def hide(path, msg, key):
    rs= calcReedSolomon(msg)
    m= msg · rs
    C= chunk(m, 5)
    index= 0
    temp= ∅
    for c ∈ C:
        s= c
        if index = 0 or index % 255 = 0:
            s= length(m)
        temp= temp · index · s
        index++
    files=sort(recEnumFiles(path),by=creation_time)
    offset= calcRandomOffset()
    while offset:
        files.pop()
    em= encrypt(temp, key, mode=RC4)
    C= chunk(em, 6)
    for c ∈ C:
        f= files.pop()
        f.creation_time.nsec= c[0:3]
        f.access_time.nsec= c[3:6]
```

Listing 2: Extracting data from timestamps.

```
def extract(path, key):
    F= sort(recEnumFiles(path), by=creation_time)
    em= ∅
    for f ∈ F:
        c= f.creation_time.nsec · f.access_time.nsec
        em= em · c
    m= decrypt(em, key, mode=RC4)
    C= chunk(m, 6)
    i,l= 0,0
    for (i,c) ∈ enumerate(C):
        if c[0] ≠ 0x00:
            continue
        l= int(c[1:6])
        break
    S= sort(C[i:l], by=first_byte)
    temp= ∅
    for (i,c) ∈ enumerate(S):
        t= c[1:6]
        if c[0] ≠ i:
            t= 0x00 × 5
        temp= temp · t
    return decodeReedSolomon(temp)
```

the chosen hiding method is not the metadata file), (iii) encrypts the data with the stream cipher, and (iv) embeds the encrypted data into the timestamps. On the information recall path, the user enters the encryption key and the application displays the decrypted message.

Listing 1 and Listing 2 outline in pseudocode the two work flows for information hiding and recall respectively.

6.2. Metadata file information protection

All information processed by the application are held in memory (RAM) and are encrypted with AES-256-CBC using a user-provided key. This is a precautious measure in order to protect against extraction of the plain metadata file during a forensics analysis of the storage medium, e.g., in the slackspace of the hard disk [45].

After the information has been embedded, the metadata file is built from the information kept in RAM so far. Before writing this information to the disk, it is compressed using `gzip` and encrypted with the AES algorithm using a user-provided password. Our implementation supports the use of different passwords for the metadata file and the actual data.

We take care not to accidently write the unencrypted metadata file to the disk, as this could leave persistent traces which particular files were modified. During the embedding process the information resides unencrypted in the RAM, and we did not implement countermeasures to prevent the operating system to store the corresponding memory pages on the disk, e.g., due to paging or hibernation. However, our application supports the encryption of information on a per-path basis, right after embedding the information in order to minimize the time the unencrypted information resides in the RAM, at the cost of creating a much larger metadata file due to the lack of compression.

Table 2: Time to embed and extract information on filesystem.

Space used	15%	30%	50%
Timestamps needed	78,687	157,325	264,193
Time to embed [sec]	74.78	76.17	76.33
Time to extract [sec]	20.19	36.92	60.29

6.3. Performance

Two of the main considerations of steganographic systems are the undetectability and the confidentiality of the hidden data [46]. The performance of the system is also an important factor with respect to applicability.

We performed a series of experiments to gain insights regarding the performance of TOMS when embedding and extracting information using volume-wide oblivious replacement. Table 2 summarizes our findings. The reported figures are the averages of ten consecutive executions of hiding (embedding) and recall (extracting). The amount of space used to embed data is reported as a percentage of the overall available storage space provided by the ESUs (i.e., 6 bytes per file). The time needed to hide (embed) the information is almost constant, irrespective of the data volume. On the other hand the time to recall (extract) the information is almost linear to the percentage of embedded data. In both cases the time ranges in dozens of seconds, which might be considered too high. Upon closer inspection, it appears that the calculation of the Reed-Solomon ECC dominates the processing time for both. However, since the file metadata are extracted from the MFT, which resides in the RAM, the average time to extract is lower than the average time to embed. This lower time is caused by performing the vast majority of filesystem operations within the RAM instead of directly accessing the hard disk.

6.4. Effect on actual filesystem operation

As a final consideration, we examined if the filesystem remained operational for normal use after manipulating the stored file timestamps. We mounted and unmounted the NTFS volumes that were modified by our proof-of-concept implementation using the drivers provided by Linux, Microsoft Windows, and Apple OS X operating systems. We did not notice any problems in using the volumes, and no error messages were logged by the operating systems. We also performed regular file operations in the volumes and did not notice any issues. Recall of the steganographic information after the regular use succeeded without any problems as well.

The analysis validates our initial assumption: typical usage scenarios of modern filesystems allow to persistently store additional information in file timestamps without affecting their normal use.

7. Implications for forensics analysis

Responsible research in steganography involves both developing new techniques for information hiding and detection (steganalysis). The issue of detection is increasingly important for digital forensics examiners, as criminal activities through digital means ara becoming prevalent [47].

Embedding information in file timestamps is feasible. As discussed in Section 5, these information are indistinguishable from that of normal operations, provided that a stream cipher is used in the encryption layer. Thus, a statistical analysis of file timestamps should be incorporated in the forensics examination procedures as a first line of defence. This analysis can provide hints for the presence of information hidden in the timestamps, if one opts to disable the encryption layer of the TOMS system.

There may be additional artefacts and implementation details which can assist a forensics investigator in disclosing the presence of hidden information. A fully-functional TOMS demands careful implementation and operation decisions. The developer must ensure that the application leaves no installation or execution traces that could reveal its presence. If a backup image of the filesystem contents at an earlier time is available to the examiners, they can compare the timestamps regarding unjustifiable modifications, especially for the case of the creation timestamps. Furthermore, the generation of new files to use them as carriers must be justifiable from a modus operandi point of view. For example, if modification timestamps are utilized, it must be justifiable why they differ from the creation timestamps – this would be suspicious if it occurs in the same second for a large batch of files. It is advantageous to check for modus operandi violations during the forensics examination process.

Another approach for the investigation procedure is the correlation of an installation timeline for an operating system and its well-known application files (e.g., the Microsoft Office suite). If such files are used to hide information and if they share the same timestamp up to the second part, an installation timeline can reveal that the creation order of some files does not match the expected one.

The filesystem data structures are not the only place where timestamps are stored. If an operating system records file-related events in its system logs with nanosecond precision, a digital forensics investigator can perform a correlation analysis between these two information sources in order to detect unjustifiable mismatches. Operating systems also use transaction logs (journals) for recovery processes (e.g., the NTFS Transaction Log `$LogFile` [48]). These can also be used for correlation analysis. Wiping out such log files or carelessly modifying them can raise further suspicions and assist aiding the investigation along.

In the case of NTFS, an informed decision in our proof-of-concept (PoC) implementation was to modify *only* the filename attribute of the MFT. However, the same information are maintained in the standard information attribute as well. Thus, an investigator can compare the two attributes and detect the use of the PoC implementation.

8. Conclusions and future work

In this paper, we proposed and explored the applicability of file timestamps as a steganographic channel. Based on our analysis of how modern operating systems store timestamps for file events in filesystem data structures and how they are displayed to the users, we reveal a redundant space to hide information. We described how this space can be utilized as a steganographic channel using a layered design that offers stealthiness, robustness, and wide applicability. We evaluated our design through theoretical, evidence-based, and experimental analysis in the case of the NTFS filesystem with datasets containing millions of files: the hidden information are statistically indistinguishable from timestamps produced during normal use. We also validated the applicability of our proposal through a proof-of-concept implementation targeting the NTFS filesystem. Finally, we discussed the implications of this new steganographic technique for digital forensics analysis.

As future work, we consider it interesting to confirm our findings with datasets provided by other researchers and practicioners in the field. Regarding our evaluation on the NTFS filesystem, we plan to explore implications of the TOMS system on the NTFS $LogFile. Also, to extend the validation to other filesystems, such as the ext4 that is used natively by the Linux operating system but also showing the robustness of TOMS during heavy usage of the underlying filesystem. Another direction is to explore additional filesystem artefacts and data structures beyond file timestamps that exhibit similar characteristics regarding time handling. It is also useful to extend the study towards devices that provide timing information with smaller precision: such devices may not offer the timing granularity that results in uniformly-distributed timestamps.

Acknowledgements

The research was supported by the Austrian Research Promotion Agency (FFG) through the Bridge Early Stage grant P846070 (SpeedFor) and the COMET K1 program.

References

[1] M. Akgül, M. Kırlıdoğ, Internet censorship in Turkey, Internet Policy Review 4 (2).

[2] D. Anderson, Splinternet behind the great firewall of China, Queue 10 (11) (2012) 40:40–40:49.

[3] G. Greenwald, No place to hide: Edward Snowden, the NSA, and the US surveillance state, Macmillan, 2014.

[4] I. Cox, M. Miller, J. Bloom, J. Fridrich, T. Kalker, Digital watermarking and steganography, Morgan Kaufmann, 2007.

[5] E. Zielińska, W. Mazurczyk, K. Szczypiorski, Trends in steganography, Communications of the ACM 57 (3) (2014) 86–95.

[6] H. Khan, M. Javed, S. A. Khayam, F. Mirza, Designing a cluster-based covert channel to evade disk investigation and forensics, Computers & Security 30 (1) (2011) 35–49.

[7] S. Katzenbeisser, F. Petitcolas, Information hiding techniques for steganography and digital watermarking, Artech House, 2000.

[8] E. Franz, A. Jerichow, S. Möller, A. Pfitzmann, I. Stierand, Computer based steganography: How it works and why therefore any restrictions on cryptography are nonsense, at best, in: Information Hiding, Springer, 1996, pp. 7–21.

[9] R. J. Anderson, F. A. Petitcolas, On the limits of steganography, IEEE Journal on Selected Areas in Communications 16 (4) (1998) 474–481.

[10] J. Giffin, R. Greenstadt, P. Litwack, R. Tibbetts, Covert messaging through TCP timestamps, in: Privacy Enhancing Technologies, Springer, 2003, pp. 189–193.

[11] S. Murdoch, S. Lewis, Embedding covert channels into TCP/IP, in: Information Hiding, Springer, 2005, pp. 247–261.

[12] D. Stodle, Ping tunnel, online at `http://www.cs.uit.no/~daniels/PingTunnel/`.

[13] J. Rutkowska, The implementation of passive covert channels in the Linux kernel, in: Chaos Communication Congress, Chaos Computer Club eV, 2004.

[14] K. S. Lee, H. Wang, H. Weatherspoon, PHY covert channels: can you see the idles?, in: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), USENIX, 2014, pp. 173–185.

[15] V. Berk, A. Giani, G. Cybenko, Detection of covert channel encoding in network packet delays, Tech. Rep. TR2005-536, Department of Computer Science, Dartmouth College, USA (2005).

[16] W. Mazurczyk, J. Lubacz, LACK –a VoIP steganographic method, Telecommunication Systems 45 (2-3) (2010) 153–163.

[17] N. Hopper, L. von Ahn, J. Langford, Provably secure steganography, IEEE Transactions on Computers 58 (5) (2009) 662–676.

[18] K. Bailey, K. Curran, An evaluation of image based steganography methods, Multimedia Tools and Applications 30 (1) (2006) 55–88.

[19] J. Fridrich, R. Du, Secure steganographic methods for palette images, in: Information Hiding, Springer, 2000, pp. 47–60.

[20] A. Williams, Transfer data via YouTube, online at `https://hackaday.com/2015/08/23/transfer-data-via-youtube/`.

[21] B. Carrier, File system forensic analysis, Addison-Wesley Professional, 2005.

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

[23] A. D. McDonald, M. G. Kuhn, StegFS: A steganographic file system for Linux, in: Information Hiding, Springer, 2000, pp. 463–477.

[24] H. Pang, K.-L. Tan, X. Zhou, StegFS: A steganographic file system, in: Proceedings of the 19th International Conference on Data Engineering, IEEE, 2003, pp. 657–667.

[25] J. Aycock, D. M. N. de Castro, Permutation steganography in FAT filesystems, in: Transactions on Data Hiding and Multimedia Security X, Springer, 2015, pp. 92–105.

[26] M. Perklin, ACL steganography: Permissions to hide your porn, online at `https://www.defcon.org/images/defcon-21/dc-21-presentations/Perklin/DEFCON-21-Perklin-ACL-Steganography-Updated.pdf`.

[27] A. Cheddad, J. Condell, K. Curran, P. Mc Kevitt, Digital image steganography: Survey and analysis of current methods, Signal processing 90 (3) (2010) 727–752.

[28] B. Li, J. He, J. Huang, Y. Q. Shi, A survey on image steganography and steganalysis, Journal of Information Hiding and Multimedia Signal Processing 2 (2) (2011) 142–172.

[29] R. Amirtharajan, J. Qin, J. B. B. Rayappan, Random image steganography and steganalysis: Present status and future directions, Information Technology Journal 11 (5) (2012) 566–576.

[30] M. Hussain, M. Hussain, A survey of image steganography techniques, International Journal of Advanced Science and Technology 54 (2013) 113–124.

[31] Microsoft Developer Network, File Times, online at `https://msdn.microsoft.com/en-us/library/ms724290/`.

[32] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, The new ext4 filesystem: current status and future plans, online at `https://www.kernel.org/doc/mirror/ols2007v2.pdf` (2007).

[33] K. D. Fairbanks, An analysis of ext4 for digital forensics, Digital Investigation 9 (2012) S118–S130.

[34] M. Kerrisk, The Linux programming interface, No Starch Press, 2010.

[35] O. Rodeh, J. Bacik, C. Mason, BTRFS: The Linux B-tree filesystem, ACM Transactions on Storage (TOS) 9 (3) (2013) 9.

[36] S. Poeschel, J.-H. Gim, Btrfs on-disk format, online at `https://btrfs.wiki.kernel.org/index.php/On-disk_Format#Basic_Structures`.

[37] O. Rodeh, A. Teperman, ZFS –a scalable distributed file system using object disks, in: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003), IEEE, 2003, pp. 207–218.

[38] Sun Microsystems Inc., ZFS on-disk specification (2006).

[39] S. Garfinkel, Digital forensics XML and the DFXML toolset, Digital Investigation 8 (3) (2012) 161–174.

[40] A. Burghardt, A. J. Feldman, Using the HFS+ journal for deleted file recovery, Digital Investigation 5 (2008) S76–S82.

[41] E. Antsilevich, Capturing timestamp precision for digital forensics, Tech. Rep. JMU-INFOSEC-TR-2009-002, Department of Computer Science, James Madison University, USA (2009).

[42] Tuxera Inc., Open source: Ntfs-3g, online at `http://www.tuxera.com/community/open-source-ntfs-3g/`.

[43] Microsoft, Description of NTFS date and time stamps for files and folders, online at `https://support.microsoft.com/en-us/kb/299648`.

[44] U. Hermann, Access Time Update, online at `http://forensicswiki.org/wiki/MAC_times#Access_Time_Update`.

[45] M. Mulazzani, S. Neuner, P. Kieseberg, M. Huber, S. Schrittwieser, E. Weippl, Quantifying windows file slack size and stability, in: Advances in Digital Forensics IX, Springer Berlin Heidelberg, 2013, pp. 183–193.

[46] T. Morkel, J. H. Eloff, M. S. Olivier, An overview of image steganography, in: The Firth Annual Information Security South Africa (ISSA 2005), electronically, 2005, pp. 1–12.

[47] Europol, 2015 Internet Organized Crime Threat Assessment (IOCTA), online at `https://www.europol.europa.eu/content/internet-organised-crime-threat-assessment-iocta-2015`.

[48] G.-S. Cho, A computer forensic method for detecting timestamp forgery in NTFS, Computers & Security 34 (2013) 36–46.