Timestomping NTFS

MSc final research project report

July 7th, 2014

Author
Wicher Minnaard

Supervisors
Prof.dr.ir. C.T.A.M. de Laat
M. van Loosen MSc

University of Amsterdam, Faculty of Natural Sciences, Mathematics and Computer Science
Master's programme in System & Network Engineering

Contents

1. Introduction

'Timestomping' refers to the act of intentionally falsifying timestamps associated with content. Essentially, the goal one hopes to reach with this act is one of the following two:

- Make the content—a contract, a receipt, some message—appear to have been created, modified, copied, accessed, etc. at some other time than it really has been. The content is not hidden—on the contrary, the existence of the content (with its falsified metadata) serves the tamperer's purposes.

- Evade detection of the content, as an anti-forensic measure. This is strongly related to the use of *time lines* in forensic investigations. A time line serves to associate events. For instance, timestamped web browser history can be integrated with file creation events, revealing whether visiting a particular web site could have led to creation of suspicious files, which can then be examined for malware. By tampering with timestamps such time based event correlation is frustrated. Specimens of malware that timestomps their files have appeared in the wild [1].

This is an important distinction to make. In the first case, the content and associated metadata can serve as a key with which to find contradicting sources of time information. In the second case, the key is lost: the timestomping of the malware file metadata has wiped the time-based relation between the web site visit and the creation of the files associated with the malware. Conversely, if the malware itself would be found through some other means, an investigator would not be able to trace it back to the web site visit just by examining the file timestamps.

In the latter and much harder case, it is the detection of the tampering itself that could yield keys on which to base an investigation. Indeed, in the chapter *Defeating Disk Analysis* of the guidebook *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System* [2, p 328], the following piece of advice is offered:

> [...] This highlights the fact that you should *aspire to subtlety*, but when it's not feasible to do so, then you should at least be consistent. If you conceal yourself in such a manner that you escape notice on the initial inspection but are identified as an exception during a second pass, it's going to look bad.
>
> [...] If you can't change time-stamp information uniformly, in a way that makes sense, then don't attempt it at all.

There are many possible sources of inconsistencies. For instance, a file may contain internal metadata on the program used to produce it. If that program simply did not exist at the time recorded in the timestamps, there is a clear inconsistency. The inconsistency does *not* immediately imply tampering—for instance, a system clock may simply have been faulty.

While there are many possible sources of such inconsistencies, the original research documented in this report focuses exclusively on NTFS structures. The goals of the research presented are formulated in terms of these structures. So, before addressing these goals, first a discussion of NTFS, its timestamps, the ways in which they can be tampered with, and related work on tamper detection is required.

Section 2 is the summary result of the preparatory literature study into these four topics. Section 3 then introduces the research questions. Sections 4: Investigating directory index records, 5: Carving directory index entries from MFT slack space, 6: Investigating direct manipulation, and 7: Single-source anomaly detection present the original research and discuss the findings, and Section 8 concludes.

2. Timestamps and timestomping in detail

2.1. Essential NTFS concepts

NTFS has been the default filesystem type for the popular consumer-oriented series of the Windows family of operating systems for over a decade. For this reason, forensic analysis of this filesystem is ground well covered. NTFS is also a complex filesystem. Indeed, Brian Carrier's *filesystem Forensic Analysis* [3] devotes more text to this filesystem than to any other filesystem by a large margin.[1] What follows is a short overview of concepts as presented by Carrier [3], complemented with corollaries. This serves to allow a reader with a basic understanding of filesystems and data structures to navigate the remainder of this report.

MFT The MFT (for **M**aster **F**ile **T**able) is at the heart of NTFS. For each file or directory, it contains an entry. The entry slot size is defined in the bootsector[2] and is commonly 1024 bytes. Slots are enumerable. Not all slots contain entries referring to existing files or directories; some slots are simply not allocated (yet), others are marked as deallocated (and available for reuse), or serve as an extension of an entry in a different slot. Many NTFS elements—such as the transaction log (journal), attribute descriptors, the free space bitmap, and even the MFT itself—are defined as a "file" in the MFT. The operating system is supposed to go to great lengths to keep the MFT contiguous on-disk, but in theory, the MFT can become fragmented, at which point the linear relation between slot number and disk location will break down.

MFT entries An MFT entry is identified by its MFT reference, which is composed of a 16-bit integer (the record sequence number) and a 48-bit integer (the MFT slot number). The concatenation of these is then

[1]To be exact, it devotes 62 pages to FATx, 124 to NTFS, 82 to EXTx, and 58 pages to UFS.

[2]Not to be confused with the classic x86 PC-architecture MBR boot sector, this structure is inside the actual partition on which the filesystem resides.

represented as a 64-bit integer [3, pp 276-277]. An entry may span multiple slots, but usually doesn't. When an MFT entry is (re)allocated, the record sequence number is incremented, thereby rendering references to the entity previously allocated to that slot unresolvable. After the header structure, which encompasses these and some other book-keeping fields, *attributes* follow.

Attributes There are a multitude of attributes which may be mixed and nested to form meaningful data structures. Attribute bodies are preceded by a header of constant format, which includes a declaration of the attribute size. Walking the attributes is a matter of reading headers and using the size declarations to seek to the next header.

The header also declares whether the attribute body is *resident* or *non-resident*. Resident attributes reside in the MFT, while the nonresident attributes reside in the data area of the filesystem. Some attributes are always resident, others are always non-resident, and some can be both. For instance, file content is stored in the DATA attribute. As long as the attribute fits in the MFT entry, that is where file contents will be stored. If it grows too large, the data will be placed in the clusters (groups of sectors; the basic allocation unit of the filesystem) in the data area of the filesystem. Naturally, this then requires a descriptor of these clusters holding file data: another attribute. The Windows NTFS driver stores the attributes (or their headers, in case of nonresident attributes) ordered by their type identifiers.

Directory indices Though an MFT entry contains a reference to its parent directory, the directories themselves also contain references to their children. Additionally, the directories include copies of the filenames and timestamps of the child entries. In principle, the directory tree can be built by using just the first type of reference. A path lookup for, say, /dir1/dir2/dir3/file would then encompass a linear scan of the n entries in the MFT for each path component m, matching by name and parent reference. This would result in $O(mn)$ time complexity which is clearly undesirable. The chosen solution is to use directory indices, which are B$^+$-tree structures not only storing the MFT references to the directory child elements, but also their names—these are the sorting criterion for the tree [3, pp 290-294]. Such a setup features $O(\log(n))$ lookup time complexity (for a directory with n entries). A path lookup operation would thus feature a time complexity in the order of $O(m\log(n))$.[3] The directory indices are built up from one or more attributes, of which one, the INDEX_ALLOCATION attribute, is non-resident.

[3]Though it is possible (using hardlinks) for each directory to contain all files in the filesystem, in reality, the search in each directory will be among n entries with n a far lower number than the total number of files.

2.2. *Timestamps in NTFS*

Modification, Access, Change, Birth

NTFS features modification, access, change and birth time stamps for file and directory entries; MACB timestamps, for short. There is considerable confusion caused by different nomenclature lineages in forensic filesystem analysis. MACB is an extension of the MAC timestamps found on most traditional Unix-like systems, as displayed by the stat command. The terms are used in *Forensic Discovery* by Dan Farmer and Wietse Venema [4], which mainly deals with Unix-like systems, and Carrier's work [3], and in the open source TSK filesystem analysis suite [5] which is closely connected to both of these works. Some other forensic suites apply a different nomenclature. For instance, EnCase (by Guidance Software) uses 'MACE'; for Modification, Access, Creation and "Entry Modified".[4]

Format

Timestamps are stored in 64-bit unsigned little-endian integers, counting the number of 100-nanosecond intervals since January 1, 1601 UTC [6]. On the surface this granularity might seem superfluous—but when analyzing NTFS, it is of great value: as two operations performed subsequently in a low-latency thread may use different timestamps, derived from the system clock closely after one another, a one-second or even a one-millisecond resolution would make it much harder to spot the fact that these are, in fact, two distinct timestamps.

Meaning of the time stamps

The modification timestamp signifies when the content of the file is last updated. The access time refers to the last time of access to the contents, but in versions of Windows from Vista and up, updates to this timestamp are turned off by default. The change time refers to changes to the file metadata—its name, security attributes, and any of the other timestamps.

The birth time refers to the time at which the MFT entry was created. This attribute receives a lot of attention, but it in contrast with the natural world, it actually means very little for a file to be "born"; its content may be completely replaced afterwards and its name, location, and security attributes may change. For practical purposes, its identity may have completely changed since its "birth". The timestamp is not even tied to a particular MFT entry; moving files between file systems can result in copying the birth timestamp to a completely new entry. This begs the question of what the birth time is supposed to signify. Especially when file content is of an incriminating nature, using only the birth timestamp as definite proof of creation of the content at the signified time is highly questionable.

[4]'MACE' has an edge on 'MACB' since in the MACB nomenclature, the terms 'change' and 'modification'—concepts used interchangeably in everyday-language—actually have very different meanings. Some may find the following moniker to be of use when working with the MACB nomenclature: *ch**A**nge* refers to *met**A**data*, *m**O**dification* refers to *c**O**ntent*.

From the operating system perspective, there are choices to be made with regard to the timestamp contents; for instance, what should the value of the modification time be for a file created empty? In 2010, Bang, Yoo and Lee [7] have documented how NTFS timestamps change when performing file operations. To parse the MFT, they use EnCase, TSK and an unpublished tool by Bang. Details of the latter are unknown, but it deserves to be mentioned that the first two are constrained in that they represent timestamps rounded to whole seconds; so reducing the actual precision of the NTFS timestamps by a factor of ten million. As mentioned, such resolution loss is undesirable when investigating the way the operating system sets timestamps.

Location of timestamps: SI, FN, IXFN

An MFT entry for a file or directory stores the MACB timestamps in an attribute of type STANDARD_INFORMATION (hereafter: **SI**). The MAB timestamps can be displayed in the explorer.exe file property dialogues. The change time is usually not shown to the user.

In the same MFT entry, the file name is stored in an attribute of type FILE_NAME (hereafter: **FN**). Multiple FN attributes may exist, since there are four name spaces; for DOS 8.3 names, Windows names, POSIX names, and a shared namespace for filenames compatible with both DOS and Windows. An entry can thus have up to three FN attributes. Each of these attributes stores a set of MACB timestamps. These timestamps are peculiar in that they are updated with the values of the SI timestamps whenever the FN attribute itself changes. As the FN attribute includes a reference to the parent directory, the update also happens when a file is moved to another directory within the same file system [7]. The experiments conducted during this research show that if a file name in a particular namespace is changed, the timestamps of FN attributes in the other namespaces are also updated, resulting in duplicates of (some earlier state of) the SI attribute timestamps in all FN attributes.

The directory indices discussed in Section 2.1, Essential NTFS concepts also store timestamps for each file; the designers have made the choice to use FN attributes to store names. To avoid confusion with the FN attribute proper, these structures will hereafter be referred to by the name **IXFN**.

Of particular interest is the fact that the timestamps are not updated in the same way as for FN attributes stored directly in an MFT entry. Rather, these IXFN timestamps reflect the SI attribute timestamps. This behaviour has been reported on in discussions following a post on the *SANS Digital Forensics and Incident Response* blog [8], and has been confirmed in the experiments conducted in the course of this research project.

2.3. *Timestomping methods*

API access

Traditionally, Unix-like operating systems have allowed read-/write access to modification and access timestamps using the utime family of system calls, through utilities such as touch. The change time is usually not alterable through an API. In contrast, the Windows API does let one change the change time. The ZwSetInformationFile and NtSetInformationFile functions [9] take a FILE_BASIC_INFORMATION structure [10] which has fields for all of the MACB timestamps. Noteworthy is that a caller can instruct the operating system to *not* update some or all of the timestamps for operations performed on the file handle.

At the 2005 Blackhat conference, James Foster and Vinnie Liu (of the Metasploit project) presented their anti-forensic tool timestomp, to which we owe the verb 'timestomping' [11]. It uses these functions to set the SI timestamps. The timestomp utility has been discontinued [12], but the method continues to be popular [2, pp 327-329].

Since there is no API for modifying FN timestamps, Foster and Liu note that forensic tools should be improved to show these, and that those timestamps should be compared to the SI timestamps—since the FN timestamps are derived from SI timestamps, they should never represent a point later in time.

There is a way around the FN timestamp problem. The method is simple: by moving a file, the SI timestamps propagate to the FN attribute. Stomping the SI attribute once more, and subsequently moving the file back, results in SI and FN timestamps all set to values of the falsifier's choosing.

Direct disk access

An adversary with direct disk access can completely influence the information processed by a file system forensic analyst. The only limit to the power of such an adversary is the extent to which he or she understands the system and manages to avoid creating inconsistencies. The SetMACE program by Joakim Schicht modifies SI and FN timestamps by direct disk access from version 1.0.0.7 on [13]. For this, it needs to acquire exclusive disk access—the program operates (with elevated privileges) from user space, unmounts the to be tampered with volume, and does its modifications. Acquiring direct disk access to the volume from which Windows itself is booted is not possible from user space in Windows versions since Vista [14], so to perform timestomping operations with SetMACE on such a volume, it needs to be approached using some other Windows instance. This can be accomplished through booting a portable Windows installation or by simply attaching the disk to some other Windows machine (but not booting from it).

2.4. *Timestomping detection methods*

Detecting timestomping relies on either finding traces of the very act itself, or on finding inconsistencies introduced by the tampering.

The NTFS journal falls in the first category. It yields insight in past operations applied to the filesystem, and can be used to extract the updates made to filesystem timestamps, as shown in 2012 by Gyu-Sang Cho [15]. The journal cannot store all operations, however, so at some point these traces of past operations will vanish.

Other methods rely on finding inconsistencies using causal relationships. In 2008, Svein Yngvar Willassen developed a method to use the record sequence numbers of MFT entries (mentioned in Section 2.1) to find evidence of antedating [16].

He combines these with what is known of the *first available*-type behaviour of the MFT slot allocator to derive a partial ordering for series of files. When a file can be said to must have existed before some other file was created, and that other file has earlier timestamps, there is an inconsistency. Similarly, in 2014, Wicher Minnaard analyzes the Linux FAT32 driver and shows how its *next available*-type allocator allows for deriving file creation order (and thus for finding inconsistencies), not using explicitly defined sequentiality (as with the NTFS sequence numbers), but using the layout of the content of files on-disk [17]. As the Windows NTFS driver uses a *best fit*-type allocator [3, p 313] for file content, such an approach will not work for NTFS.

A straightforward type of inconsistency is when traces of the original timestamps can be recovered. In a 2012 blog post, William Ballenthin and Jeff Hamn present a tool, INDXParse, with which the deallocated space of a non-resident component of directory indices can be searched for IXFN structures of files that have been deleted [18].. Such traces are volatile, but could be used to detect the the double-move trick discussed in section 2.3: Timestomping methods.

3. Research questions

The ultimate goal is to find a reliable way to easily detect any and all timestomping. This, of course, has been attempted before. However, there seems to be some space left for a modest contribution.

1. The INDXParse tool (version 1.2.2) can only process the non-resident INDEX_ALLOCATION attribute. The other directory index attribute where IXFN structures are stored in is the INDEX_ROOT attribute. Therefore it makes sense to find out if former entries of this attribute can somehow be recovered, and if such is the case, then a computer program that does so could be of use to the forensic community.
2. The goal of the direct disk access timestomping utility SetMACE is to leave no trace of its use. Its documentation mentions SI and FN attributes, but does not mention the other place where timestamps are to be found: the IXFN structures. An investigation into whether the IXFN structures can betray the use of SetMACE is due.
3. All of the detection methods discussed in Section 2.4 rely on some secondary source of 'truth' to compare to the possibly falsified timestamps. In their Blackhat presentation, James Foster and Vinnie Liu mention that since the FN timestamps are derived from SI timestamps, they should never represent a point later in time [11]. The question is whether real-world systems hold up to such (supposed) regularities, and if there are any such *self-inconsistencies* that could be used to detect timestomping without relying on some second source of information.

The research is constricted to the behaviour of the ntfs.sys driver, version 6.1.7601.17514, with an internal modification timestamp of "11/23/2010 4:23 AM", operating on version 3.1 of the NTFS on-disk format. That is to say, older or newer driver versions and disk formats may behave differently from the system experimented with.

4. Investigating directory index records

4.1. Test environment

When examining the behaviour of an automated work, such as a filesystem, it is desirable to create a setup that enables as quick a turnaround as possible. The setup employed is Windows (7, "premium", x86-64 architecture, version 6.1.7601) virtual machine running under the Linux KVM hypervisor using Qemu. A 16 MB block device, backed by a memory-backed file in a tmpfs file system, is attached to the guest OS through the the VirtIO mechanism [19]. Write caching for this device (on the host side) is turned off using the cache=directsync parameter. This disk is initialized and formatted with a 1KiB cluster size through the diskmgmt.mmc management console snap-in. The Cygwin environment is deployed to provide a featureful programmable remote shell. Prior to inspecting the on-disk effects of operations, the test volume is unmounted through the offline disk command of the interactive diskpart.exe console application. This serves to flush all changes made in the virtual machine to the file which is backing the block device.

4.2. Creating a parser

There is no scarcity of utilities that can read the MFT structure. However, to truly understand how structures are nested, the 'extractor'-type utilities are insufficient, for what may matter from a forensic perspective is not just the information, but also the way it is laid out. Further requirements are scriptability and the ability to inspect raw values (the raw bytes, or raw types not further interpreted). The Hachoir framework fulfills these requirements [20]. Using a somewhat declarative style of programming, structure parsers can be defined incrementally. The framework forces the user to declare any data to be skipped over; automatically, a map of which bytes have been interpreted and which have not (yet) been is generated. The structure parser can be loaded into a variety of user interfaces to interactively explore data. Every structure, no matter how deeply nested, is named unambiguously in a path-like fashion. These paths can be used to quickly navigate a structure hierarchy. Dependency ordering is automatic; fields are only read when they need to be. For instance, if the on-disk position of some structure depends on the value of an offset field defined elsewhere, the latter will of course need to be parsed first. In the code, this dependency does not require specification beyond simply using the offset field to get to the position of the structure.

As it happens, the Hachoir framework (version 6c52231) already includes a rudimentary NTFS parser. Not many attributes classes are defined; it can parse the bare MFT entry type, and the FN and SI attributes. More importantly, it yields some incorrect results, the cause of which is that the *fixup values* are not applied. As some kind of integrity checking mechanism, fixup values are used in NTFS for structures that may span multiple disk sectors, such as MFT entries. The two bytes at the end of each sector occupied by the structure are *overwritten* with a signature value, which is incremented on every such write. The original values are copied to an array somewhere near the start of the structure, which also includes the signature value used. Before reading the complete structure, it first needs to be 'fixed up' using the original values stored in the array. The idea is that some kinds of corruption can be detected by checking whether the values at the end of the sectors match the stored signature value [3, pp 352-353]. Hachoir did not have provisions for patching up an input stream in a just-in-time fashion. Thus, before starting on improving the NTFS/MFT parser, first an alternate input stream type that does allow patching ahead of the parser is created. This is the InputMmapCowStream, which, as the name suggests, uses a copy-on-write memory-mapped file to this end. It is a simple and safe approach; it does not modify the underlying file, and it lets the operating system do all the bookkeeping work on which byte ranges have been replaced and which have not been. Currently it is limited to 4GiB maps. This seems to be a problem in Python[5] as in straight C code, memory maps addressing more than 2^{32} bytes can be created and used just fine on the 64-bit architecture used in this research. The code has been made available online in the form of a fork of the main Hachoir project [21].

For restructuring, extending and improving the existing NTFS parser code with the main goal of reaching the IXFN structures, Carrier's chapter on NTFS data structures is used. Carrier, in turn, bases this chapter on the documentation of the Linux NTFS project [3, p 396]. Since development of that project has moved into a commercial effort, this documentation is not available online anymore at the project website. Fortunately, the Internet Archive still has a copy of the documentation [22].

The original Hachoir NTFS parser code assumed that the MFT is not fragmented. The code has now been modularized to decouple the parsing of the MFT from the parsing of NTFS, so that the MFT may be extracted and parsed separately from the NTFS partition. Extraction of the MFT from a disk image can be performed using the icat utility from the TSK [5] suite, in the following manner:

```
icat /path/to/ntfs-partition-or-image 0 > mft
```

This works by virtue of the fact that the MFT is the first file in the NTFS filesystem. NTFS suffers from some potentially circular dependencies; for instance, the layout of the MFT is described in the MFT itself. Therefore, to enable bootstrapping of the filesystem, parameters that are essentially variable are in special cases assumed to be static. Such is the case for the location of the MFT entry for the MFT.

The code of the MFT parser is included with the hachoir-cow fork, and available online [23].

4.3. Growth of a directory index

The hachoir-urwid interactive user interface is used to track the on-disk directory index representations as directories created in the virtual machine test environment are grown in size by adding more files using the touch command included with Cygwin. Fig. 1 shows the initial layout of the MFT entry.

Next, files with names G, F, E, D, C, B and A are added to the directory, in that order. The result is the layout as shown in Fig. 2. The index entries are not stored in insertion order, but rather in the order defined by the collation type declared in a field of the INDEX_ROOT attribute. In this case, that seems to be an alphabetical ordering.

No more file references will fit into the root node of the index attribute. Adding a file X will result in a conversion of the index to a B^+-tree proper, as shown in Fig 3. Two new attributes are added: the INDEX_ALLOCATION attribute and the BITMAP attribute. Together, these attributes declare the storage in the data area of the filesystem to be used for further index nodes. From this moment on, the three attributes will slowly grow in size as the tree grows. The INDEX_ROOT is initially empty, with all of the index entries in child nodes, but over time will come to accommodate some index entries of its own. The INDEX_ALLOCATION and BITMAP attributes will grow to contain the references to the out-of-MFT storage units. Of particular interest is the fact that these attributes have only overwritten part of the list-type initial node. Entries B through G are now contained in the space still assigned to the MFT entry, and this space is unavailable as a storage area to anything other than this very MFT entry. As the index grows, it will take up more of the slack, eventually overwriting every last one of the original list-type entries.
Further experiments show that once the conversion has taken place, shrinking the amount of index entries (by deleting files) will not bring the directory back into the single-node strictly-resident state. Therefore, the timestamps in any entry retrieved from slack MFT space carry a special meaning, since we know that they are traces of some of the very first couple of files having been placed in the directory.

[5]To be precise: CPython 2.7.6, GCC 4.7.3, on x86_64 linux-3.15.2 .

Figure 1: Initial layout of an MFT entry for a directory with a 6-letter name.

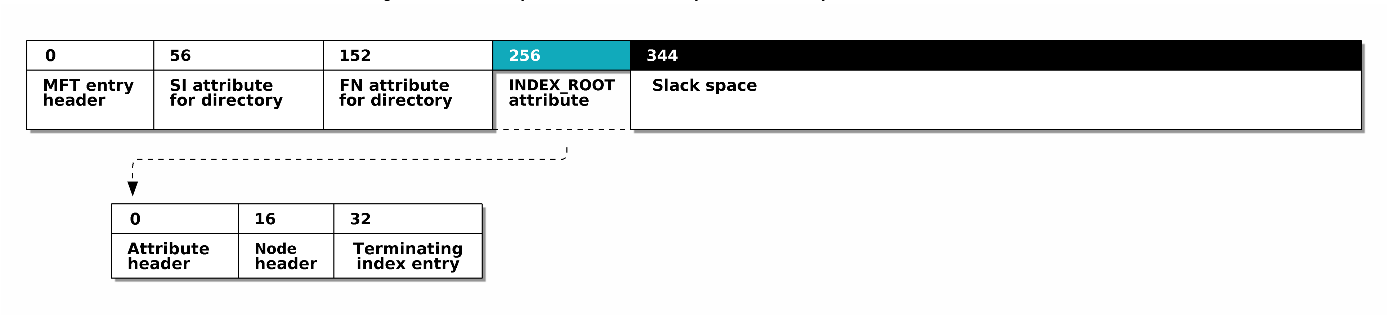


Figure 2: Layout of an MFT entry for a directory with a 6-character name referencing seven files with short names.

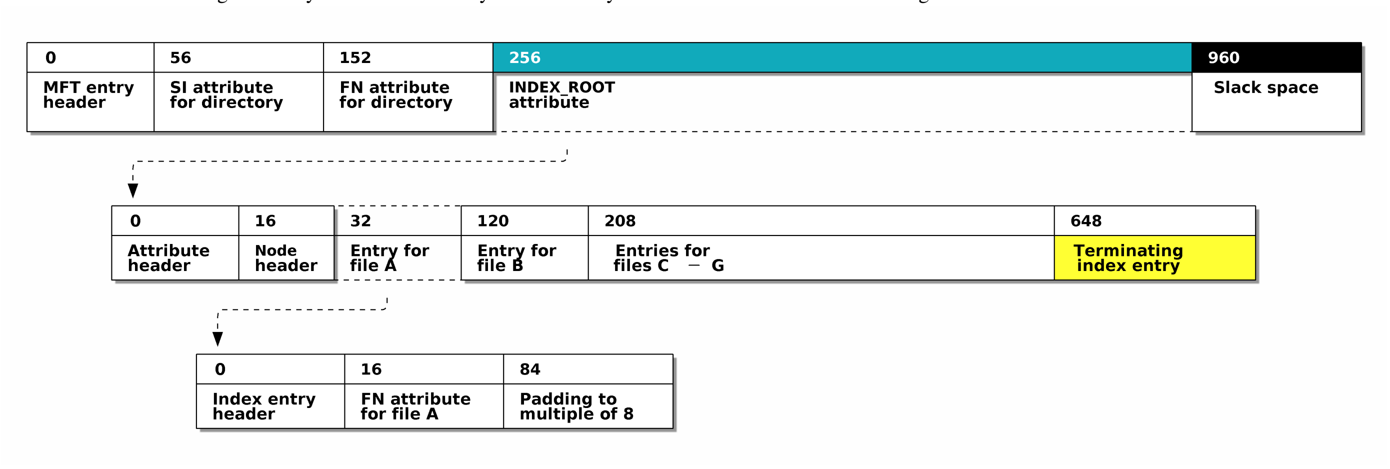


Figure 3: Layout of the MFT entry for a directory after conversion to a B^+-tree proper.

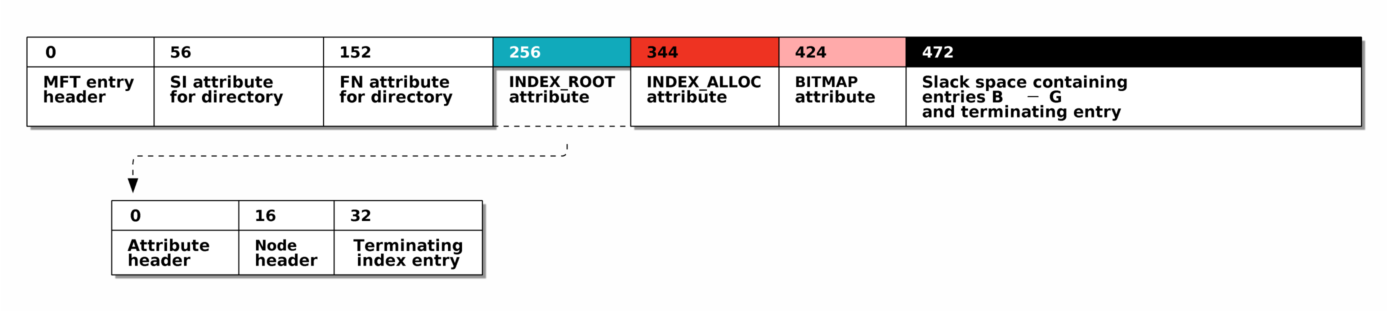
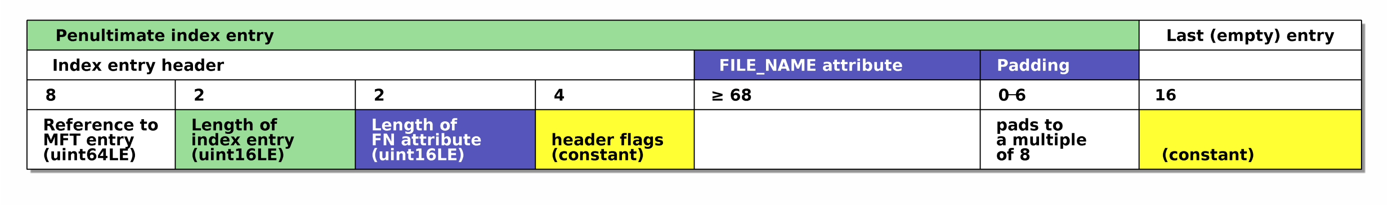


Figure 4: Layout of the end of a list-type INDEX_ROOT attribute. Synchronization points are coloured in yellow. The two declarations of substructure sizes and the structures they refer to are coloured blue and green.



5. Carving directory index entries from MFT slack space

5.1. Method

Fig. 4 depicts the end of a list-type INDEX_ROOT attribute. Shown are the terminating index entry and the preceding index entry. The terminating entry is constant across indices, but not by design. The structure is the same as for any other entry, but by virtue of embedding no IXFN structure, its size is constant, and therefore so are the descriptor of its own length and the IXFN attribute length. Moreover, the reference to the parent directory is zero, and the only flag set is the one that designates the element as the last element in a node. For these reasons there are simply no terminating entry elements that can vary across directory indices.

While such a 16-byte anchor point seems luxurious with the goal of creating a carver in mind, the reality is that this anchor point mostly consists of zeroes—only two bytes are nonzero. Scanning backwards from this anchor, we encounter another anchor point. This is the header flag set of the preceding entry, and it is no better than the footer anchor point: four bytes, all zero. As the default state[6] of bytes on storage media and in paged userspace memory is zero (NULL)[7], sequences of zeroes are, by themselves, of very low utility as anchor points. However, we know that it should appear at least 72 bytes (68 bytes for the minimum length of a IXFN attribute, padded to a multiple of 8) before the footer. Before the header flags should be two 16-bit unsigned little-endian integers, both of which allow to confirm the distance between the header anchor and the footer anchor *independently*—so decreasing chance of a random match considerably.

Once the penultimate entry is found, its start can be used as an anchor point instead of the 16-byte empty entry, and from there a new backward scan for the header flag anchor point can be attempted. This repeats until the slack buffer is exhausted.

The method is implemented in the ixcarve module, part of the source code accompanying this report [24].

5.2. Evaluation

Described in section 4.3: Growth of a directory index, the mechanism by which indices grow raises the supposition that there is a decent chance of finding stale index entries in MFT slack space. A test run of the ixcarve carver will make clear to what extent this will be the case.

5.2.1. Test images

As results may vary through differences in operational history of filesystems, some background information on subject systems is provided. The following two filesystems are used:

[6]As presented to the OS.

[7]That we speak of ‘zeroing’ instead of of ‘fiving’ or ‘eighting’ also raises the expectation for null values to be quite common.

System A. This filesystem was created December 31st, 2012. It resides in a dualboot system which is infrequently booted to Windows (7, “premium”, x86-64 architecture, version 6.1.7601). Likely, the filesystem has not seen as much use as other NTFS filesystems of similar age. Moreover, the Windows driver does not enjoy exclusive use of the filesystem; it is (infrequently) accessed through the ntfs3g NTFS userspace driver on Linux.

System B. This filesystem was created May 16th, 2014. The driver and operating system are the same as for System A. The filesystem is part of the virtual machine used for research, as described in Section 4.1: Test environment.

5.2.2. Running the carver, results

The script dumptrel-ts.py [24] reads the MFT using the parser described in Section 4.2: Creating a parser, and dumps timestamps and some other useful information to a SQLite database. In addition it carves the slack space of each MFT for derelict index entries. Table 1 shows a breakdown of the results. System A yields 7072 carved records, from 12% of the directories. System B yields 4393 records, from 7% of its directories.

Table 1: MFT slack index entry carving results.

	System A		System B	
Total amount of directories	32803	100%	30425	100%
yielding 1 carved record	1784	5.4%	1214	4.0%
yielding 2 carved records	1150	3.5%	292	1.0%
yielding 3 carved records	495	1.5%	337	1.1%
yielding 4 carved records	362	1.1%	386	1.3%
yielding 5 carved records	11	–	8	–

No further investigation into the causes of the differences between A and B is made; they may or may not be related their respective ages. For now, it is sufficient to say that the method indeed unlocks a non-negligible amount of time information.

False positives. Even though the carver features strict input constraints, it is good practice to examine results for false positives. For System B, the carved results were filtered based on their MFT reference. 76 results had references that could not be resolved in the current state of the MFT. However, the four timestamps for each of these FN attributes were close to each other (in 64 results, the change time was within one second after the birth time), and all change times had reasonable values between May 16th and June 23rd 2014. It is highly likely that the results are valid, but belong to directories that themselves no longer exist (and of which the MFT entries have been reallocated). In other words, there are assumed to be no false positives in System B, and there are no particular reasons to assume that there would be many false positives on other systems.

6. Investigating direct manipulation

SetMACE, version 1.0.0.9 [13] is run on the test VM (see Section 4.1) to examine its behaviour using the MFT parser (see Section 4.2). First, a test was run to see if the utility indeed correctly overwrites the SI and FN timestamps in-place, without leaving traces such as a reordering of the attributes. This appears to be the case, but a bug was revealed through trying to use a timestamp with value 5498:01:10:21:10:24:730:3440, which should have resulted in the 64 bit unsigned little-endian integer 1229782938247303441 being written to disk in its byte representation of 0x1111111111111111. Instead, the lower value of 0x00C1814E1BE8B401 is written to disk. This converts to September 14th, 1990, just before midnight (UTC).

Of more significance is the discovery that SetMACE leaves the copy of the SI timestamps in the IXFN structure intact. This is not only apparent when using the parser, but can be witnessed without the use of special utilities using plain cmd.exe. The initial situation is as follows, with a small 11-byte file testfile with timestamps in 2014.

```
C:\cygwin64\home\boer\setmace>dir F:\testdir
 Volume in drive F is En thee efkes
 Volume Serial Number is 0246-9FF7

 Directory of F:\testdir

07/07/2014  02:50 PM    <DIR>          .
07/07/2014  02:50 PM    <DIR>          ..
07/07/2014  02:50 PM                11 testfile
               1 File(s)             11 bytes
               2 Dir(s)       6,567,936 bytes free
```

Next, SetMACE is run to modify the date to just before the turn of the last millennium (CET, converted to UTC):

```
C:\cygwin64\home\boer\setmace>SetMace64.exe F:\testdir\testfile
     -z "1999:12:31:21:59:59:999:9999" -x
Starting SetMace by Joakim Schicht
Version 1.0.0.9

Record number: 39 found at disk offset: 0x000000000000DC00
Success dismounting F:
Success writing timestamps
Job took 0.03 seconds
```

Another directory listing shows the timestamps from the IXFN structures for the testfile file. These timestamps have *not* been changed. A remount of the volume brings no change in this situation, and neither does rebooting the virtual machine.

```
C:\cygwin64\home\boer\setmace>dir F:\testdir
 Volume in drive F is En thee efkes
 Volume Serial Number is 0246-9FF7

 Directory of F:\testdir

07/07/2014  02:50 PM    <DIR>          .
07/07/2014  02:50 PM    <DIR>          ..
07/07/2014  02:50 PM                11 testfile
               1 File(s)             11 bytes
               2 Dir(s)       6,567,936 bytes free
```

Next, the content of the testfile file is read and displayed in the terminal.

```
C:\cygwin64\home\boer\setmace>more F:\testdir\testfile
le content
```

After this, a directory listing *does* show the timestamp values written by SetMACE:

```
C:\cygwin64\home\boer\setmace>dir F:\testdir
 Volume in drive F is En thee efkes
 Volume Serial Number is 0246-9FF7

 Directory of F:\testdir

07/07/2014  02:50 PM    <DIR>          .
07/07/2014  02:50 PM    <DIR>          ..
31/12/1999  11:59 PM                11 testfile
               1 File(s)             11 bytes
               2 Dir(s)       6,567,936 bytes free
```

Further experimentation using the Cygwin utilities show that this also happens when using stat, which does not read file content, but only its metadata. There must be some mechanism that detects inconsistencies with the parent directory IXFN structures upon reading MFT entries, and fixes them, regarding the SI attributes as authoritative. So while SetMACE can be said to create inconsistencies between SI/FN timestamps on the one hand, and the timestamps in IXFN structures on the other hand, they are corrected by some mechanism—in the NTFS driver, presumably—as soon as entries are accessed. The implication is that it is what happens *after* the timestomping with SetMACE has taken place, that will determine whether it can be easily detected by comparing SI and IXFN timestamps. This once more points to the importance of following best practices for conservation of digital evidence, especially in dealing with situations in which timestomping is a possibility.

7. Single-source anomaly detection

This section details research into the detection of inconsistencies by using just the timestamps that can be retrieved from the MFT, i.e. without resorting to some other source of time or sequentiality information. The filesystem used to investigate wildtype timestamps is the System B filesystem as introduced in Section 5.2.1. Timestamps are extracted to a database using the dumptrel-ts.py script, and post-processing is performed using the tsfipri.py script [24]. The database for System B is made available online [25].

7.1. *Testing assumptions*

7.1.1. *FN timestamps should be less or equal to SI timestamps*

As FN timestamps are updated with SI values, they should invariantly be of lesser or equal value as SI timestamps. For System A, there are 116658 files for which all timestamps are retrieved, and for System B there are 60489 such files. Only 23% of the files on System A conform to this "invariant", and on System B, the fraction is even lower: 3%. Clearly, the invariant does not hold. In fact, the most common relations with respect to FN and SI MACB values are

$$M_{SI} = M_{FN}, A_{SI} = A_{FN}, C_{SI} < C_{FN}, B_{SI} = B_{FN}$$

on System A (19%), and

$$M_{SI} < M_{FN}, A_{SI} < A_{FN}, C_{SI} = C_{FN}, B_{SI} < B_{FN}$$

on System B (57%).

7.1.2. *File change time should be greater than parent directory birth time*

A file cannot be placed in a directory that does not yet exist. When placing a file in a directory, its change time will be updated. It may be updated again for various reasons, but that newer timestamp value should never be smaller than its previous value. Testing this invariant yields 83 nonconforming files on System B. 66 file change times lag more than 5 hours behind their parent directory birth time; and of those, 26 lag by more than a week: too long for it to be caused by non-slewing clock synchronisation. Thus the inconsistency does not occur often—the false positive ratio is low enough to further examine the results by hand—but on the other hand, we know nothing of the false negative ratio. This will depend on a tamperer being unwise enough to antedate files beyond the parent directory birth timestamp.

7.2. *Classifying wildtype timestamps*

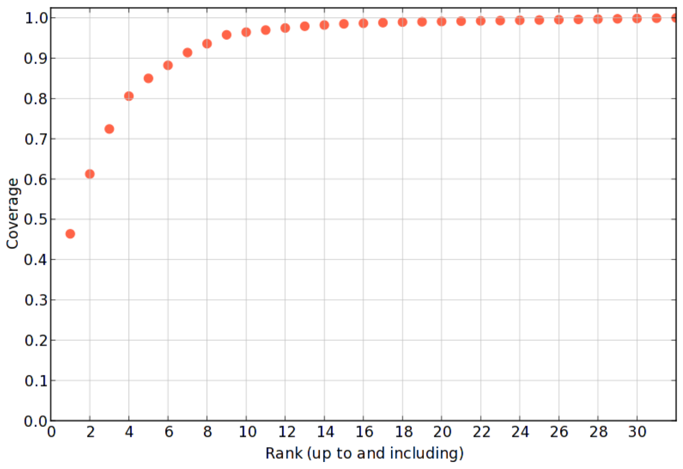
The previous experiment shows that inconsistencies occur on filesystems even if no intentional timestomping has been taking place. This begs the question: what *are* the regular timestamp configurations? To answer this question, a simple fingerprint is derived from the SI and FN MACB timestamps. The timestamps are sorted first alphabetically, by timestamp name. After this, a stable sort by increasing value is performed. Relations between each value and the next one are then expressible as either equality or less-than. A fingerprint may thus take the following form:

$$\text{ffna} = \text{ffnb} = \text{ffnc} = \text{ffnm} = \text{fsia} = \text{fsib} < \text{fsim} < \text{fsic}$$

There are $\sum_{x=1}^{7} \binom{8-1}{x-1} = 55581$ unique configurations that these fingerprints can express.[8] The fingerprints encountered on System B are not evenly spread over this space; some fingerprints are much more common than others. Figure 5 shows the cumulative distribution for the 1360 allocated .exe files on System B. The distribution is highly skewed, one particular fingerprint covers almost half of the files, and seven fingerprints are sufficient to cover 90% of of the files. What follows is a long tail of one-offs. Again, none of these can be considered to be a true positive. While this shows that some fingerprints will be very uncommon, the utility of that knowledge for detecting timestomping is limited. As a selection criterion, it will suffer from a high false negative ratio. It may, however, be of some use when ranking an otherwise bland result of running a filter on the executables, such as a known-good list. As the fingerprints differ by type of file, and will likely differ by how they have been handled, they could also be of some value when attempting to find subgroups within collections of hundreds of pictures, some of which have been handled differently than others.

[8]Each binomial coefficient expresses the number of ways in which the eight time stamps can be subdivided between a certain number of inequality relations. There can be a maximum of seven inequality relations. See http://en.wikipedia.org/wiki/Stars_and_bars_(combinatorics).

Figure 5: Cumulative distribution of fingerprints of .exe files on System B.



8. Conclusions

With regard to the three research questions, we can conclude as follows:

1. Carving for derelict IXFN structures in MFT slack space yields good results, in a quantitative sense. Since the traces are related to the time that a directory first grew beyond a couple of entries, types of investigation other than timestomping investigations may be better served by this interesting type of trace.
2. A volatile type of inconsistency caused by direct-disk manipulation of SI and FN timestamps has been discovered.
3. Determining generic self-inconsistencies for timestamps has not been found to be a worthwhile pursuit. The confidence with which we reason about the result of file operations on timestamps is not transferable to systems of which we do not know which operations have been performed on it. With an API to arbitrarily modify timestamps with, there can only be an open world assumption. It is, however, unquestionably the case that in certain scenario's, with specific applications performing specific operations, there can be easily detectable self-inconsistencies–except that these will not be *true* self-inconsistencies, since in such cases, a second source of truth is implicitly being used: the specific behaviour of some application. Fingerprinting time relations might be of some use, but it is hard to see how it could outdo well-established clustering algorithms.

The way forward seems to lie in developing a consistency checker which integrates the types of inconsistencies that we *do* understand well–this includes Willassen's causal relations, and more mundane checks such as for discrepancies between multiple FN attributes in different namespaces, and possibly the invariants that have shown to yield some false positives, such as the file-change-time-before-parent-directory-birth-time inconsistency type.

[1] Elia Floro, Trojan.Ruindem Technical Details (December 2005).
URL http://www.symantec.com/security_response/writeup.jsp?docid=2005-120710-4752-99&tabid=2 1

[2] Bill Blunden, The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System, 2nd Edition, Jones & Bartlett Learning, 2012. 1, 2.3

[3] Brian Carrier, File system forensic analysis, Addison-Wesley Boston, 2005. 2.1, 2.2, 2.4, 4.2

[4] Dan Farmer and Wietse Venema, Forensic discovery, Addison-Wesley Upper Saddle River, 2005. 2.2

[5] Various contributors, The Sleuth Kit (TSK) & Autopsy: Open Source Digital Forensics Tools.
URL http://sleuthkit.org/ 2.2, 4.2

[6] Microsoft Corporation, Windows Dev Center-Desktop: File Times.
URL msdn.microsoft.com/en-us/library/windows/desktop/ms724290.aspx 2.2

[7] Jewan Bang and Byeongyeong Yoo and Sangjin Lee, Analysis of changes in file time attributes with file manipulation, Digital Investigation 7 (3-4) (2011) 135–144. doi:10.1016/j.diin.2010.12.001. 2.2, 2.2

[8] Chad Tilbury, NTFS $I30 Index Attributes: Evidence of Deleted and Overwritten Files (September 2011).
URL http://digital-forensics.sans.org/blog/2011/09/20/ntfs-i30-index-attributes-evidence-of-deleted-and\discretionary{-}{}{}overwritten-files 2.2

[9] Microsoft Corporation, Windows Dev Center-Desktop: ZwSetInformationFile routine.
URL http://msdn.microsoft.com/en-us/windows/desktop/ff567096 2.3

[10] Microsoft Corporation, Windows Dev Center-Desktop: FILE_BASIC_INFORMATION structure.
URL http://msdn.microsoft.com/en-us/windows/desktop/ff545762 2.3

[11] James Foster and Vinnie Liu, Catch me, if you can...
URL http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-foster-liu-update.pdf 2.3, 3

[12] H.D. Moore, Re: timestomp exe?
URL https://dev.metasploit.com/pipermail/framework/2011-July/007887.html 2.3

[13] Joakim Schicht, SetMACE 1.0.0.9 (December 2013).
URL http://reboot.pro/files/file/91-setmace/ 2.3, 6

[14] Microsoft Corporation, KB942448: Changes to the file system and to the storage stack to restrict direct disk access and direct volume access in Windows Vista and in Windows Server 2008.
URL http://support.microsoft.com/kb/942448 2.3

[15] Gyu-Sang Cho, A computer forensic method for detecting timestamp forgery in NTFS, Computers & Security 34 (0) (2013) 36 – 46. doi:10.1016/j.cose.2012.11.003. 2.4

[16] Svein Yngvar Willassen, Finding evidence of antedating in digital investigations, 2012 Seventh International Conference on Availability, Reliability and Security (2008) 26–32. doi:10.1109/ARES.2008.149. 2.4

[17] Wicher Minnaard, The Linux FAT32 allocator and file creation order reconstruction, Digital Investigation 11 (3). doi:10.1016/j.diin.2014.06.008. 2.4

[18] William Ballenthin and Jeff Hamm, Incident Response with NTFS INDX Buffers - Part 3: A Step by Step Guide to Parse INDX (October 2012).
URL http://www.symantec.com/security_response/writeup.jsp?docid=2005-120710-4752-99&tabid=2 2.4

[19] Tim Jones, Virtio: An I/O virtualization framework for Linux (January 2010).
URL http://www.ibm.com/developerworks/library/l-virtio/index.html 4.1

[20] Victor Stinner, Hachoir wiki: Features (July 2009).
URL https://bitbucket.org/haypo/hachoir/wiki/Features 4.2

[21] Wicher Minnaard, Commits in COW-fork of Hachoir (July 2014).
URL https://bitbucket.org/blinkingtwelve/hachoir-cow/commits/all 4.2

[22] Richard Russon and Yuval Fledel, NTFS Documentation (December 2006).
URL http://web.archive.org/web/20061209150816/http://www.linux-ntfs.org/content/view/104/43/ 4.2

[23] Wicher Minnaard, Victor Stinner, mft.py source code (July 2014).
URL https://bitbucket.org/blinkingtwelve/hachoir-cow/src/tip/hachoir-parser/hachoir_parser/file_system/mft.py 4.2

[24] Wicher Minnaard, NTFSix (July 2014).
URL http://nontrivialpursuit.org/ntfs/ntfsix.tar.gz 5.1, 5.2.2, 7

[25] Wicher Minnaard, System B database (July 2014).
URL http://nontrivialpursuit.org/ntfs/systemB.sqlite.xz 7