Generic Metadata Time Carving

By:
Rune Nordvik (Norwegian University of Science and Technology (NTNU); Norwegian Police University College), Kyle Porter (NTNU), Fergus Toolan (Norwegian Police University College), Stefan Axelsson (NTNU and Halmstad University), and Katrin Franke (NTNU)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2020
July 20 - 24, 2020

Forensic Science International: Digital Investigation 33 (2020) 301005

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA

Generic Metadata Time Carving

Rune Nordvik [a, b, *], Kyle Porter [a], Fergus Toolan [b], Stefan Axelsson [a, c], Katrin Franke [a]

[a] Norwegian University of Science and Technology, Norway
[b] Norwegian Police University College, Norway
[c] Halmstad University, Sweden

ARTICLE INFO

Article history:

Keywords:
Digital forensics
Carving
Metadata
File system

ABSTRACT

Recovery of files can be a challenging task in file system investigations, and most carving techniques are based on file signatures or semantics within the file. However, these carving techniques often only recover the files, but not the metadata associated with the file. In this paper, we propose a novel, generic approach for carving metadata by searching for equal and co-located timestamps. The rationale is that there are some common metadata for files and directories within each file system. Our generic time carver provides potential timestamp locations for repeated timestamps in each metadata structure, identifying potential metadata for files. A semantic parser then filters the results with respect to the specific file system type. In our experiments, extraction of `MFT` entries in `NTFS` and inodes in Ext4 had near perfect precision for metadata entries with multiple equivalent timestamps, and for such metadata structures we obtained perfect recall for NTFS. For known file systems, we use the information found within identified metadata to recover files, and by recovering files and their associated metadata we increase the evidential value of recovered files.

Introduction

File carving is a technique which identifies and extracts files from unallocated areas based on signatures found within the file content, and not by using file system metadata (Garfinkel, 2007). While extremely useful, file carving has a few challenges. First, investigators need to decide which file type to carve. To decrease the file carving time, investigators often select the file types they assume could be pertinent for the criminal case. For instance, by carving for typical image files in cases related to sexual abuse of children, the investigator limits the ability to identify other file types. Furthermore, not all files have a signature, and will not be found by using file carving. Some carving techniques will carve based on the assumption that files have contiguous blocks, which will fail when trying to carve a fragmented file (Garfinkel, 2007).

Our novel approach does not use file carving, but rather metadata carving. We search for repeated co-located timestamps, based on equality, in a small window to obtain locations of potential timestamps. In this way, timestamps are used as a kind of dynamic signature. Once verifying the timestamp as likely to be legitimate for a particular file system, we use the metadata surrounding it to fully or partially recover the file. Our approach handles both contiguous and fragmented files.

We only recover file or directory metadata from `NTFS` and `Ext4` to demonstrate the usability of the novel approach, but the approach can be extended to recover metadata from other file systems. In order to achieve a realistic scenario we reformat the volume with another file system, effectively "damaging" the previous file system. The tools developed in this paper are prototypes, and the main target group are file system experts with the competence to manually assess file system structures. The tools and the disk images can be downloaded for review at (Nordvik et al., 2020).

To our knowledge, no one has used equality between closely co-located timestamps to identify metadata before as a carving technique. Previous attempts suffer from many shortcomings including the inability to find static signatures for all pertinent metadata structures.

Our approach focuses on metadata structures found in `MFT` entries or inodes carved from unallocated space. There will always be a risk that the blocks (clusters) pointed to by the discovered metadata structures may be overwritten by new or existing allocated files, but this may be identified by examining the allocated file system bitmap for allocated blocks, and by comparing metadata

* Corresponding author. Norwegian University of Science and Technology, Lyngstien 13, 6522, Kristiansund, Norway.
E-mail address: rune.nordvik@phs.no (R. Nordvik).

https://doi.org/10.1016/j.fsidi.2020.301005

Table 1
File Systems with timestamps co-located within metadata structures.

File System	Co-located timestamps	Granularity
NTFS (Carrier, 2005)	4	64 bit - ns intervals since 1.1.1601
ReFS (Nordvik et al., 2019)	4	64 bit - ns intervals since 1.1.1601
APFS (Hansen and Toolan, 2017)	4 (5)	64 bits - ns since 1970
HFS + (Apple, 2004)	4	32 bits - s since 1904
BTRFS (Bhat and Wani, 2018)	3 (4)	64 bits - s since 1970 + 32 bits (ns)
ExFAT (Hamm, 2009)	3	32 bits + UTC offset
FAT (Carrier, 2005)	3	16 bits for time (except accessed), 16 bits for day
UFS1 (Carrier, 2005)	3	32 bits - s since 1970 + 32 bits (ns)
UFS2 (Carrier, 2005)	4	64 bits (ns) since 1970
Ext2/3 (Carrier, 2005)	4	32 bits - s since 1970
Ext4 (Dewald and Seufert, 2017)	4	34 bits - s since 1970 + 30 bits (ns) (Göbel and Baier, 2018)

information with the content of the recovered file. Most file systems have some sort of bitmap system which has bits representing each block (cluster), and allocated blocks have their corresponding bit set [3, p.311].

Our new approach is suited for recovery of metadata and file content from storage devices that have been reformatted with another file system, with the same kind of file system when not assessing the allocated inode/file table, or from generally damaged file systems. The approach is also useful for finding historical metadata structures located on disk that are not contained in `MFT` or inode tables.

Detailed file system structures are described by Carrier (2005). Even though his book does not include details about `Ext4`, it contains most of the basic information from `Ext2` and `Ext3`. Dewald and Seufert (2017) include more details about `Ext4`.

Assumptions

Currently, most file systems will include at least 3 contiguous timestamps. Linux file systems normally use the `MAC` (Modified, Accessed and Changed) timestamps [3, p.297], for instance `Ext2` and `Ext3` use the contiguous `atime` (accessed), `ctime` (inode changed), `mtime` (data modified) and `dtime` (deleted) [3, p.298]. `Ext4` also contains the same contiguous timestamps, but adds the `crtime` (creation) in the end of the inode (Ext4 development team and 20, 2019). `NTFS` and `ReFS` use 4 contiguous timestamps (Creation, Modified, `MFT` modified, and Accessed) in multiple attributes (Carrier, 2005; Nordvik et al., 2019). Table 1 shows a few file systems with closely co-located timestamps.

Objectives

- Can we reliably use time as a generic identifier to carve for file and directory metadata structures in different file systems?
- What is the reliability of recovery of files using the discovered metadata in `Ext4` and `NTFS`?

We aim to identify file or directory metadata structures from different file systems based on a common identifier. In our case, equal and closely co-located timestamps, which will allow for a generic approach for metadata carving. We identify the potential timestamps by using a simple string matching algorithm, and then we interpret the semantics[1] of the expected file system metadata in order to significantly reduce the number of false positives.

Identifying the metadata is not enough in order to connect file content and the metadata, because the content may be overwritten by allocated files. We discuss why it is important to perform a manual assessment of both metadata, content, and context in order to decide if the metadata and the file content can be connected.

Novelty of the new approach

Existing techniques for metadata carving do not use timestamps as a common identifier (dynamic signature) for different file systems. Even Dewald and Seufert (2017) describe that there is no magic signature for inodes in the `Ext4` file system, and they depend on semantics from `Ext4` in order to identify the locations of the inode metadata structures. However, metadata structures can easily be found by using string pattern matching based on equality, but unfortunately with a large number of false positives which have similar properties. Thus, obtaining high recall and low precision for finding file system metadata entries. We also do not depend on a start date or an end date to identify the timestamps. This datetime agnostic nature of our approach allows the support of any file system that has closely co-located timestamps. While we do not depend on other semantics in order to identify the locations of these potential timestamps, we do utilize semantic parsers to validate and reduce the number of false positive hits of file system metadata structures significantly.

Importance for digital forensics

By using the novel generic metadata time carving approach, we do not need to specify which specific file types to carve for. The approach does not consider file types or file signatures; it only carves for metadata structures that potentially can be used for recovery of files. By accurately connecting the metadata to the corresponding file content, we also increase the evidential value of the files recovered, which most existing carving techniques do not accomplish.

Organization of this paper

We have introduced the objectives and the novel generic metadata time carving approach in the introduction section. In the literature review and contributions section we discuss the current state of the art related to file and metadata carving. In the method section we describe the carving algorithms and the methods we used for the experiments, and in the results section we describe our results using precision and recall. Then we discuss our results in the discussion section, and we conclude in the conclusion and further work section.

Related work and contributions

There has been a significant amount of literature published on

[1] Previous authors used “semantic filtering” to describe this, we have chosen to adopt that terminology.

file carving, both using signature based contiguous carving, specific file type semantic carving or other statistical approaches in order to identify and carve for fragmented files. We address literature more specifically related to metadata carving, therefore, a complete list of all file carving literature will be out of scope.

Mueller (2008) introduced the idea of searching for NTFS timestamps as a string in unallocated space, since each timestamp is 64 bits and represents the number of nanosecond intervals since 1.1.1601. He also describes that the timestamps are in groups of 4 contiguous timestamps for each group. He created an EnScript (plugin for EnCase) that searched for the NTFS timestamps and bookmarked them. The EnScript uses a grep search for a particular date range, and it has an option for checking the next 8 bytes in order to only include hits that are followed by another valid timestamp. Use of the consecutive timestamps search ideally reduces the number of false positives, but based on the comments on this blog post it appears as though the consecutive timestamps search does not work correctly.

In order to decrease the number of false positives, our idea is to search for a set of identical timestamps within a small window in order to detect metadata structures that describes files. Only metadata structures with a specific number of equal timestamps will be found. Our approach is more generic since we do not need to know how the timestamp is formatted (other than that they are closely co-located).

McCash (2010) based his work on the EnScript from Muller (Mueller, 2008), and adds the idea of using this information to detect MFT records and their attributes to extract the data content. He also describes that the script can be used to identify directory indexes and Registry key nodes.

Metadata carving

Dewald and Seufert (2017) consider the case in which the Ext4 superblock or group descriptor table is corrupted or overwritten, and they use either metadata mode or content mode for parsing file systems or metadata carving respectively. In content mode their solution is to carve for inodes, which potentially provides the metadata necessary to extract the file content. However, the filename and inode number is not recovered in content mode. Since inodes have no magic bytes (except for extent headers in Ext4), they describe that they carve for them using pattern matching and analysis of the metadata. They conclude that their approach can reconstruct files from Ext4 despite not knowing about the specific structure of the file system. They do, however, describe that they need multiple Ext4 parameters in metadata mode for file system parsing. They describe that these can either be given by the user, or estimated based on the file system size.

Their work shows that carving for metadata structures is already suggested for file recovery. Their metadata mode approach explicitly depends on semantics specific for Ext4 in order to include both metadata and file content, which enables parsing of the file system (not carving). Their carving approach, content mode, is not able to recover filename or inode numbers.

Plum and Dewald (2018) describe carving for APFS container superblocks, volume superblocks, or inode carving. APFS uses multiple container superblocks, and each of them may contain a reference to the previous container superblock. Within each container superblock they find volume superblocks, which describe specific volumes. These can be used to parse a specific volume and recover files from previous states of the file system. They further describe that inodes do not have a specific signature, but they can be carved using a combinations of the object type and subtype inode fields. These inodes can be used to potentially recover files with the connected metadata.

Their approach is similar to the work of Dewald and Seufert (2017), but it differs by depending on specific semantics from APFS. Our generic approach will also work for the APFS inodes, since each inode has a set of contiguous timestamps. However, we have not implemented a semantic parser for APFS.

Work by Garfinkel (2013) describes the Bulk_Extractor tool which parses a large stream of data, using multiple threads, for feature extractions (URLs, e-mail addresses, Google search terms, Exif data, etc), which utilizes optimistic decompression before extracting the features. The features are detected based on rules which consider local context, which improve precision and recall. The features extracted do not need to be found within file entries. As part of the result, histograms of extracted features are created.

Evaluating recovered files

Casey et al. (2019) describe forensic processes such as authentication, classification and evaluation of recovered files. The problem is that different recovery tools do not use the same names for the same thing. They suggest to use Potentially Recovered before the authentication is performed. The authentication process is necessary in order to decide if the file is Fully Recovered, Partly Recovered, only Name and Metadata recovered, or Name Recovered. The decision should be based on confidence level after testing or trying to falsify different scenarios or claims.

Method

We use a generic automated approach to identify a potential set of timestamps within a specified threshold. We then record the byte positions in the image file where the set of timestamps were found. The approach is generic because it will identify the metadata structures in any file system that uses two or more timestamps of a user defined size to describe the temporal information of a file or directory. Since our approach is based on identifying equality between sequences of bytes, we do not require a start or end time for the timestamps. This approach will increase the false positive rate, but our semantic parsers attempt to exclude false positives by verifying if each timestamp location has a valid metadata structure for a specific file system.

As a proof of concept, we have added support for the recovery of metadata based on the identified timestamps in the Ext4 and NTFS file systems.

General potential timestamp algorithm description

We first describe the general potential timestamp algorithm at a high-level overview. A motivating factor for this algorithm is that often one or more MAC timestamps are identical. Furthermore, for file system entries in NTFS, ReFS, and ExtX the timestamps are closely co-located together in the metadata structure. Let m be the length in bytes of the potential timestamp, let T be an array of bytes of the data being searched. The user will define the length m (ExtX requires $m = 4$ and NTFS requires $m = 8$), as well as the length k of bytes to be searched after the potential timestamp, which we refer to as the search threshold. The crux of this search approach is that every non-overlapping m bytes in our binary data T is considered a search keyword, and we look for repetitions of this size m byte sequence within the subsequent k byte threshold window following the keyword. If the given byte pattern occurs one or more times within this threshold window, then we have identified a potential timestamp.

The mechanics of the search algorithm are based on the *sliding window* approach as is often found in malware analysis. The search begins at $T[0]$, in which the first m bytes are taken to be a potential

timestamp, which contains the values $T[0:m)$. We then check if this m byte keyword is equivalent to every non-overlapping m bytes in $T[m:(m+k))$, and keep count of how many exact matches have occurred. Given that we are searching for timestamps where at least two of them per metadata structure are equivalent, if no matches are found, we would then advance our search position by m bytes to position $T[0+m]$. The advancement of m bytes assumes that timestamps will always fall on a multiple of m, and we do plan in implementing an *exhaustive* search functionality which checks for timestamps on every one or two bytes. Such skip sizes were chosen to enhance the speed of search substantially, as the current solution for 8 byte timestamps will process a disk image 8 times faster than an exhaustive search alternative that checks for potential timestamps on every single byte. If one or more matches are found we advance our search position by k bytes to position $T[0+k]$. In either, case the entire search procedure is repeated from our new search position. This process is repeated for the entirety of the data T, except last k threshold of bytes. The skip size of k was chosen as it is the minimum size to avoid multiple hits for the same metadata structure. Note, for our current implementation k must be a multiple of m, otherwise the bytes being searched will be misaligned with the actual disk image timestamps. Algorithm 1 provides the pseudocode of the basic potential timestamp carving algorithm.

Algorithm 1. Basic Potential Timestamp Carving Algorithm.

```
Data: Raw disk image T as a byte array
Result: Potential timestamp positions (in bytes)
m # Length of timestamp;
k # Length of search threshold;
h # Theshold of matching timestamps;
i = 0 # Byte location;
bool repeatedBytes = False;
while (i < |T| − k) do
    searchString = T[i : (i + m)];
    decimalDate = stringToDecimal(searchString);
    repeatedBytes = checkRepeatBytes(decimalDate);
    if (!repeatedBytes) then
        matchCount = 0;
        j = i + m;
        while (j < i + m + k) do
            testBlock = stringToDecimal(T[j: j + m]);
            if ((testBlock == decimalDate) then
                matchCount += 1;
            end
            j += m;
            if (matchCount >= (h − 1)) then
                Print Byte Location i;
                j = i + m + k;
                i += (k − m);
            end
        end
    end
    i += m ;
end
```

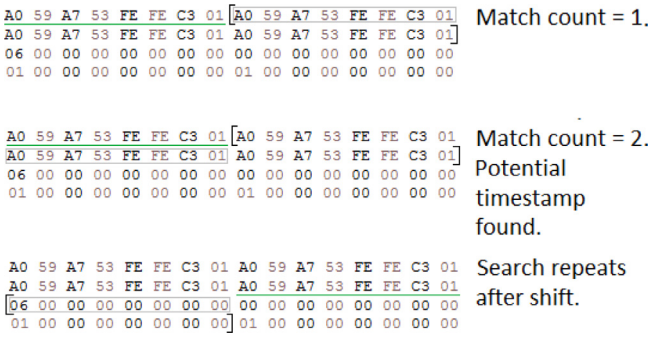

Fig. 1. Visual representation of the search procedure where three matching time stamps are searched for. The underlined byte sequence represents the current byte sequence being tested as a possible timestamp. The subsequent bytes in brackets represent the search threshold for checking matches. The bytes in grey boxes represent checks for matching byte sequences. In the second row, after a second match is found, we advance the search procedure ahead by k bytes, where the process is repeated.

We provide an illustration for further explanation. In Fig. 1, the underlined bytes represent the potential timestamp keyword with $m = 8$, and the brackets represent the threshold of bytes, $k = 24$, being searched for matches.

This general search by itself likely produces a large number of false positives, thus we placed an additional condition to improve the algorithm's precision. We determine if the potential timestamp to be searched for consists of a single repeated byte value, and if so, we skip the search procedure and move forward m bytes. This is to avoid fruitless searches on blocks of repetitive bytes. Examples of such timestamps we wish to avoid are 0x0000000000000000 and 0xFFFFFFFFFFFFFFFF.

Here we approximate the time complexity of the worst case search scenario. We assume that the entire disk image could be read into memory at once to simplify our approximation. In this scenario, we are searching a disk image with no sets of co-located bytes that are repeated two or more times (we get no hits). In this fashion, we cannot perform any byte window skips after searching through our search window threshold. We also perform the most generic type of potential timestamp carving, where we do not consider repetitive byte sequences. In this way, we cannot skip any particular keyword byte sequence, since all byte sequences will be considered to be potentially valid timestamps. Thus, every m sequence of non-overlapping bytes on the array of disk image bytes T will have a search procedure performed on it, in which the entire threshold window of size k is searched.

Given this worst case scenario, the computational time complexity is $O\left(\left\lfloor\frac{|T|}{m}\right\rfloor \times \frac{k}{m}\right)$. The integer of the cardinality of T divided by m is the number of byte sequences that have a search procedure performed on it, and it is multiplied by the maximum number of byte matching checks, the threshold of bytes k divided by m where m is a factor of k.

Practical potential timestamp program details

The general timestamp carving algorithm was implemented in C++, which we refer to as *cPTS*, and is supported by a number of libraries. Since disk images under analysis are likely greater than memory, we use the cross-platform mio memory mapping library.[2] This allows us to read in 1 GB of memory at a time, and read the image as a series of arrays. However, once the potential time stamp

[2] https://github.com/mandreyel/mio.

carver arrives within the last 4096 bytes of the gigabyte in memory, we load a new gigabyte into memory from the search point relative to the disk, as to handle directory entries that are spread across segments. For converting datetime formats into decimal form, we used the Date library.[3] The program outputs a text file list of all the potential timestamp locations (in byte offset) that were found.

Semantic parsers

Identifying closely co-located potential timestamps based on equality will provide generic results, but will contain many false positives due to its genericity. Therefore, parsers which utilize the semantics of the expected metadata structures of specific file systems were developed for more accurate automation. Our Python 3 parsers accept the timestamp locations from the generic timestamp carver and the disk image as input, as seen in Fig. 2. Our experiments utilize this process.

NTFS semantic parser

The `NTFS` script assesses if the potential timestamp is within a Standard Information Attribute (SIA), or a Filename Attribute (FNA). To reduce the number of false positives, we exclude any potential timestamps from before the year 1970 and years beyond 2100. The script outputs metadata information contained in the SIA, FNA, and Data Attribute if possible. When attempting to parse out a full MFT entry, we start with the identification of the SIA. Once we identify the location of the SIA header, we use the length of the SIA to see what the header of the next attribute is, ultimately searching for the Data Attribute. If the next attribute header is not the Data attribute, and the first byte of the attribute type is less than 0x80, we read the length of the attribute and perform another skip down to the next attribute. Though, if the next attribute is an FNA, we will output its metadata information, and add the byte location of its first timestamp to a list of future timestamps to avoid. This is done so we do not get redundant FNA outputs. Encountering at least one FNA is required to read out a potential Data Attribute we encounter. This is repeated until we find the Data attribute, or abandoned if we identify an attribute type where its first byte is greater than 0x80 or if we have searched more than the length of the potential MFT entry. A limitation of this work is that we do not currently perform MFT entry searching starting from identified File Name Attributes, where their timestamps are more likely to be reliably[4] found due to their relatively unchanging nature compared to Standard Information Attributes (Cho, 2013). Another limitation is that we currently do not support Alternate Data Streams. Relevant MFT entry information is output into the file `NTFSResults.txt`, and if the file is resident, we also include the resident file encoded in ASCII.

Ext4 semantic parser

The Ext4 Python 3 script uses the text file produced by the cPTS tool containing the potential timestamp locations, the disk image, the byte position to where the partition starts, and the assumed block size. For conducting a similar search as was done in the `NTFS` parser, these parameters and a default static inode size of 256 bytes are the only assumptions we make.

Like the `NTFS` parser, we use the potential timestamps as anchor points and test for various semantics at local offsets. But now, we also verify information found in likely directory entries. For `Ext4`, we test all possible offsets backwards for file flags of interest: 0x04 for directories, 0x08 for regular files, and 0x0A for symbolic links. For `Ext4` inodes not using extents, we ensure the relative position of bytes 36–39 of the inode are unused. For inodes using extents, we check that the relative position of its extent header magic number is equal to 0xF30A. We then conduct additional tests to increase the likelihood of having discovered an inode, such as using some of Dewald and Seufert's (Dewald and Seufert, 2017) timestamp consistency tests, that the size of the file appropriately fits the sector count, and that the size of the file cannot exceed the size of the disk image. All inodes found to be sufficiently valid have their likely starting points added to a list.

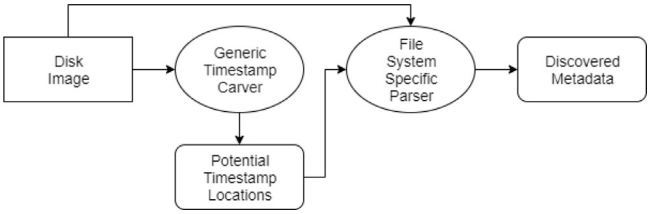

Fig. 2. Diagram for system deployment, used in our experiments.

The great difficulty in performing full file extraction in `Ext4` is connecting inodes and their directory entries when not relying on superblocks, block group descriptor tables, or inode bitmaps. Such connections will need to be made if these metadata structures are irrecoverable. The inode contains the majority of the metadata for the file, but its associated directory entry contains the filename and the inode number. The task was then to connect inodes based on their physical position to their actual inode number. We pursued solving this problem solely relying on information we can find locally within or around a validated inode.

Our solution revolves around using the verified inodes of directories that have not been deleted, as this gives us ground truth information about inode numbers and filenames, including the inode number of the directory itself. Verification is performed by following the directory's extent or direct block pointer to its first directory entry and checking if bytes 4–6 are 0x0C0001 (the length of the entry and the first byte of the length of the name). For `Ext4`, we perform two passes over the disk image. The first pass collects information on valid inodes, creating a dictionary of inode numbers and filenames found in all validated directories, as well as a so-called synchronization list. This synchronization list is a recording of the first validated inode of a directory found per block group, wherein we record the inode's location and inode number. The second pass uses the inode dictionary and the synchronization list to make inode number estimations of validated inodes, outputting the inode number alongside its likely filename.

We can estimate the inode number of potential inodes in two different ways. The first way uses the positions found in the inode synchronization list, where we can then make estimates of the inode numbers of the validated inodes that are in the same block group as the validated directory being used from the inode synchronization list. Assuming that the `Ext4` inodes are a static size of 256 bytes, the following is the equation of the estimated inode number e, where *dn* represents the inode number of the validated directory being used from the inode synchronization list, *vl* represents the validated inode location, and *dl* represents the location of the inode of the same directory obtained from the inode synchronization list:

$$e = dn + ((vl - dl) / 256) \quad (1)$$

[3] https://github.com/HowardHinnant/date.

[4] FNA timestamps are updated mainly on MFT entry creation and on file name change.

Using the inode synchronization list, we can estimate inode numbers prior to encountering the directory its filename and inode number are held in (in case of deletion). During the second pass (while we are estimating inode numbers), when encountering a validated inode of a directory, we update the entry in the inode synchronization list for the current block group we are in. This allows for rather local synchronization of the current inode number.

The second way of estimating inode numbers uses the previous estimations, and the inode dictionary. The first time we make an estimate for a particular inode number, we update its entry in the inode dictionary by adding in the inode's file version number and created time as parameters. If the inode number did not exist in the dictionary prior to the inode estimation, we simply create an entry with these parameters. Once the entry has been updated or created, it cannot change. The file version number and the created time should be relatively unique per inode, and so when parsing future inodes we check if we have already recorded its file version number and creation time in the inode dictionary, and write out the associated recorded inode number and filename.

We output a text file and csv file, *ExtResults*, where we record pertinent inode and directory entry information. We list both the estimated inode number and filename (using the inode number as a key in the inode dictionary), and the recorded inode number and filename (using the file version number as the key in the inode dictionary).

Experimental setup

We used an external USB thumb drive and wiped the partition[5] using the tool `dc3dd v. 7.2.646` (Department of Defence Cyber Crime Center, 2012) in `macOS Mojave v. 10.14` (`Linux` could also be used).

```
sudo dc3dd hwipe=/dev/rdisk8s1 hash=md5
```

Listing 1: Wiping USB thumb drive.

Experiment - NTFS reformatted with exFAT

We formatted the device in `Windows 10` using `NTFS`, where we created 50 files, and for each file type we named them File1, File2, File3,..., File10. Five different file types were used, and there were 10 files for each of these file types, where the extensions were added to the filename. Then we reformatted the file system using `exFAT`. Fragments, or the complete `MFT` table should still be available. Finally, 10 text files were added to the reformatted image.

The files created by the batch file give us a known basis in order to test precision and recall. We know all the file names and content, as the base forensic image (`ntfsbase.dd`) of the partition was created before reformatting it with `exFAT`. After the reformatting and the creation of 10 text files, we created a new forensic image of the partition (`nftsexfat.dd`) using `dc3dd`.

We measured the false positive and false negative rates by comparing the carved metadata results with the filenames we found in `ntfsbase.dd`. A false positive is a hit location not found within metadata describing a file or directory, while a false negative is a set of timestamps not identified as a hit, but which is located within metadata describing a file or directory. Finally, we calculated the precision and recall (Perry et al., 1955) of the methods implemented.

```
cPTS ntfsexfat.dd 8 24 3
```

Listing 2: Command to find possible NTFS timestamps.

We used a timestamp size of 8 bytes, a search threshold of 24 bytes to search for equivalent timestamps, where at least 3 timestamps are equal. The output from Listing 2 was saved to the file cPTS.txt.

```
python3 ntfsParser.py cPTS.txt ntfsexfat.dd
```

Listing 3: Command to identify if the hits are SIA or FNA in a MFT entry, and outputs information from these attributes and from the DATA attribute if possible.

Experiment - previous Ext4 reformatted with NTFS

The next step was to assess if each hit was part of a standard information attribute (SIA) or a file name attribute (FNA). Then the script in Listing 3 identifies the Data attribute and shows the resident data or the non-resident data runs.

Additionally, we tested if `X-Ways`, `EnCase`, and `EaseUS Data Recovery Wizard` (previously named `Recuva`) were able to recover the previous `NTFS` partition or to find unallocated `MFT` entries using the same forensic image.

For the `Ext4` experiment we used `Linux Mint 18.2` and `Windows 10`. In `Linux` we wiped the storage device using the command shred, overwriting using zeros. Then we formatted the storage device with `Ext4`, and mounted it. We created 50 directories with 500 files in each directory. The files were numbered from 1.txt to 25000.txt. The file names correspond with the number of bytes (a's) in each file. The text files were selected because they are more difficult for carving tools to recover, as there is no signature. Therefore, recovery of these text files relies solely on metadata. The file system was unmounted, and a ground truth forensic raw image named `expExt4.dd` was created using `dd`. Then it was mounted to a `Windows 10 OS`, and reformatted with `NTFS` using a 4096 byte cluster size. Then 10 files were created. A raw image was created with the name `Ext4NowNTFS.dd`.

```
cPTS Ext4NowNTFS.dd 4 12 2
```

Listing 4: Command to find possible Ext4 timestamps.

We used a timestamp size of 4 bytes, a search threshold of 12 bytes to search for equal timestamps, where at least 2 timestamps are equal. The output from Listing 4 was saved to the file cPTS.txt.

```
ext4Parser.py cPTS.txt Ext4NowNTFS.dd 0 4096
```

Listing 5: Command to parse Ext4 inodes.

[5] We wiped only the partition, shown in Listing 1, because `MacOS` gave a resource busy message when trying to wipe the complete raw disk.

In Listing 5 we start at byte offset 0 in the image, the block size is 4096 bytes, blocks per group are estimated to block size*8 = 32768.

Limitations

We do not know at the start of the investigation if there has been a previous file system. We suggest to search for known signatures of volume boot records/superblocks, which may document the start of a previous partition. We also suggest to try to recover the partition before attempting our metadata carving approach.

The output results of the prototype should be assessed by file system experts (or expert systems) in order to assess if a file (name, metadata and content) can be fully or potentially recovered. This is called authentication, and it includes an evaluation of the classifier, the results, and finally a confident decision (Casey et al., 2019).

Our prototype tool will not work properly on a manipulated file system where sectors or clusters are removed or added, because the mapping between data runs (extents) and the cluster locations are not in sync. Our approach depends on the existence of metadata structures in the unallocated area of the partition, and we assume that the start of a data run (extent block pointer) is relative to the start of the partition. The prototype also does not consider fixup values found in the last two bytes in each sector in a MFT entry. Since both SIA and FNAs are among the first attributes in a MFT entry, we assume they normally will not be found within a fixup value. We do not consider files that use multiple MFT records if the DATA attribute is not located in the first of these MFT records. The currently implemented semantic parsers do not consider FNAs in directory indexes, but the cPTS tool will locate them. Lastly, we do not consider `Ext4` inode record sizes other than the standard 256 bytes.

We are aware our experiments have a small sample size, and we have not included testing on real forensic images from real criminal cases in order to comply with legislation. We are also aware that we have only tested using specific versions of Linux and Windows, which opens for possible deviations if other OSes are selected. We selected this small sample because it allows us to know the ground truth of the content, which is difficult when using a system volume where the OS is continuously creating and deleting files. Using an unknown source makes it difficult to compute precision and recall, and gives us no control of the different variables that may affect the results.

Results

The `cPTS` command took 13 s to run on a 2 GiB byte `dd` image file. The `ntfsParser.py` took less than 1 s, while the `ext4parser.py` took 8 s. This is faster than the runtime performance of Dewald and Seufert's (Dewald and Seufert, 2017) tools, however they additionally exported the file content automatically.

NTFS metadata carving

For each discovered MFT entry, we know the SIA is associated with the FNA because of the distance between them is less than 1024 bytes, which could visually be verified or falsified by interpreting the byte location for the SIA and FNA timestamp hits. The Data attribute belongs to the FNA because we skipped to the next attribute until we found the unnamed Data attribute, which we found within the next 1024 bytes. This means we have the name, metadata and the content, and since we have the data runs, we know this record is non-resident and potentially recoverable. In order to test if the original content can be connected to the metadata, we need to extract the content and perform hypothesis testing. Extraction of the content based on known data runs is described by Carrier (2005).

In Table 2 we focus on files/directories and Standard Information Attributes (SIAs). We know each base MFT entry has one SIA. Since we have 79 files (50 files and the 29 system generated files and directories) in our experiment, we know there must be 79 SIAs in the MFT table. We also know there must be 79 SIAs in $LogFile[6] and 4 SIAs in $MFTMirr [3, p.303]. This gives a total of 162 SIAs. We found all 162 SIAs, and only one of the hits was a false positive. We did not have any false negatives for SIAs. Our computation of precision and recall is shown below.

$$Precision = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} = \frac{162}{163} = 0.9939$$

$$Recall = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = \frac{162}{162} = 1$$

For our simple experiment, it was easy to verify the found MFT records with the known base. However, with a forensic image from a real criminal case hypothesis testing must be performed in order to verify that the data runs found in the MFT entries still can be connected to the file content, and that they are not completely or partly overwritten by another file.

Hits from $LogFile

In addition to entries in the MFT Table and the MFTMirr, we found MFT entries within the unallocated $LogFile, but in this case they did not include data runs or resident data. It was interesting to observe that our created files had a Data attribute within the $LogFile with the size of only 0x18 bytes, which only contains the attribute header. This means that we cannot fully recover files from all MFT entries found in $LogFile.

Ext4 metadata carving

We were able to recover all inode metadata entries that were not overwritten by `NTFS`, but reformatting `Ext4` with NTFS in our experiment wiped approximately 20257 inodes from the inode table. When using flex groups, the inode tables are all located continuously in the first block group (Dewald and Seufert, 2017), instead of being divided into its corresponding block group. Our current observations show that our approach supports both types, and does not depend on information from the block group descriptors. For each extent found in an inode, an extraction of the file content is easily performed by extracting from the extent start block as well as the number of blocks contained in this extent. A deleted inode will normally have the extents zeroed out, making the inode connection to the file content infeasible. However, since we find hits for duplicate inodes in different physical locations, we may be able to recover the file content by using extents found in these duplicates. We did not extract the files from the overwritten `Ext4` image due to their quantity, and thus are only discovering potentially recovered files.

Table 2
Precision and Recall for finding MFT records in ntfsexfat.dd

	TP	FP	FN	Precision	Recall
SIA matches	162	1	0	0.9939	1

[6] The only file system transactions we performed were creating files.

In order to measure precision and recall, we created the same Ext4 dd images again following the same method as previously described, but in addition we added the real inode number as an attribute to each inode for the 25000 text files and 50 directories we created in the experiment using the Linux attr command. This way we could compare our estimated inode number and the recorded inode number with the real inode number included in the attribute.

For the original and reformatted images, we conducted two precision and recall experiments. The first calculates the precision of attributing our recorded or estimated inode number to the true inode number of the inode, and we also calculate the recall of finding our known files with correct attribution. For calculating recall, if at least one inode per inode number was found and we correctly estimated or recorded its inode number, it counted as a true positive, where the duplicate inodes (with respect to its true inode number) were removed. For these precision and recall calculations, we only considered inodes we extracted that contained their true inode number as an attribute. Table 3 shows our results for the non-reformatted version of the dd file. A similar process was done for the reformatted dd file, as seen in Table 5, except rather than calculating the recall we simply record the number of discovered inodes per method of inode number estimation. This was done as we cannot be certain how many false negatives were due to our methods or to the file system reformatting (we cannot find something that is overwritten). Note that since at least 20257 of 25050 inodes were wiped from the table, and we recover 5755 inodes, that we are potentially recovering at least 963 previously overwritten inodes.

The second experiment calculates the precision of our method to correctly classify inodes, whether they were from known files or not. False positives in this case are inode hits that contain junk information. Table 4 shows the precision from the non-reformatted experiment, and Table 6 shows the precision of the reformatted experiment. In both experiments, we obtained 100% precision. We cannot calculate the recall in this case, since we cannot know how many inodes (from the inode tables and copies throughout the disk) exist on the image.

Commercial tools

The results of the tool testing described in this section are shown in Table 7.

NTFS reformatted with exFAT

We created a case in X-ways v 19.8 and imported the file ntfsexfat.dd. Using the feature refine volume snapshot, we selected the particularly thorough file system data structure search, and checked search FILE records everywhere. The X-ways manual describes that this search should be able to find MFT entries from unallocated space. However, the tool did not find any of the MFT entries from the previous NTFS file system.

X-Ways also has a function that should be able to scan for lost partitions, but this feature was not able to detect the previous partition. We searched for MFT entries, but X-Ways did not find the MFT entries from the previous partition.

Table 3
Precision and Recall for finding and attributing iNode numbers for known files in expExt4Attr.dd

	TP	FP	Precision	Recall
Recorded inode matches True inode	77481	0	1	1
Estimated inode matches True inode	27336	50145	0.3528	1
Est or Rec inode Matches True inode	77481	0	1	1

Table 4
Precision of iNode classification for non-reformatted image.

TP	FP	Precision
77675	0	1

Table 5
Precision and Files Found for finding and attributing iNode numbers for known files in Ext4AttrNowNTFS.dd

	TP	FP	Precision	Files Found
Recorded inode matches True inode	15544	41692	0.2716	4848
Estimated inode matches True inode	7091	50145	0.1239	5755
Est or Rec inode Matches True inode	16553	40683	0.2892	5755

We also tried to carve for file content (except text files), and X-Ways was able to carve the contiguous files, but not the tiff files that had two fragments.

EnCase v8.08 was not able to find the MFT records from the previous NTFS partition in ntfsexfat.dd. We selected the Full Investigation pathway, which includes the relevant Recover Folders (which should locate hidden files in FAT and NTFS volumes), and the Windows Artifact Parser with MFT Transactions selected. According to the EnCase manual, the Recover Folder option should be able to recover NTFS files from unallocated clusters. Since EnCase is a closed source tool, we do not know how this is implemented. EnCase did find the Backup VBR (Volume Boot Record) when we searched for it using the Partition Finder. However, it was not possible to recover the partition in disk view.

We tried using EnCase to carve for picture files (bmp, jpg, png, and tiff). The content of all 10 bmp files were found, but also 178 extra false positives. Each of the 10 jpg file contents were found, and with no false positives. We missed one of the png files, but found the others. EnCase did not find any of the tiff files. This is because EnCase searched for the tiff signatures 49492A000A, 49492B00, 4D4D002A, 4D4D002B, while the tiff file created in Windows 10 had a signature 49492A008E.

We tested the storage device we performed the experiment on with EaseUS Data Recovery Wizard v11.15, and it was able to identify all the 50 files and their content. Since this tool is closed source, we assume they performed a partition recovery by using the VBR backup. For partition recovery they only showed the size, the date created and the path. It also carved for files, but the carved files did not include the metadata. Lastly, we found the tool could not correctly carve the fragmented files or the text files.

Ext4 reformatted with NTFS

Carving for ASCII text files is not supported by EnCase v8.08, and therefore carving for the ASCII text files in the previous Ext4

Table 6
Precision of inode classification for reformatted image.

TP	FP	Precision
57427	0	1

Table 7
Tool testing - Carve for metadata from previous file system when reformatted with another file system.

	EnCase	X-Ways	EaseUS	Bulk_Extr	cPTS
NTFS metadata	N	N	Y?	Y	Y
Ext4 inode	N	N	N	N	Y

file system is not possible. However, we tried to carve for 10 different supported file types and measured the run time to be 27 s on a 2 GiB disk image. Next we tried 100 different supported file types, which took about 1 min. Finally, we selected all 349 supported file types, but we canceled the progress after 6 h.

We performed a test using the tool EaseUS Data Recovery Wizard v11.15, but it did not find any of the 25000 text files within the experiment storage media. All the 16 files it listed were allocated NTFS files, and it did not find any previous Ext4 partition.

Additional testing

In order to identify an expected number of false positives using our tools, we performed additional testing on a real 16 GiB raw image from a cell phone that did not have any MFT records and an 80 GiB Windows 10 machine with no Ext4 inodes. Searching for inodes in the Windows image we found 3 false positives, and searching for MFT entries on the cellphone image we found 8 false positives.

Lastly, we tried running Bulk_Extractor on our reformatted images, wherein it found all of the filenames of the MFT entries in both images, but did not recover any of the metadata information from Ext4.

Discussion

In our experiments, we reformatted one file system with a different file system. If we know it has been reformatted, we can first try to recover the previous file system by assessing the backup VBR or superblocks. This may potentially recover the file system, and we can accurately find files that the other file system did not overwrite. However, if the other file system has overwritten parts of the previous file system, then we may need to use our approach to find the parts that are not overwritten.

Discussion related to NTFS

For the timestamp hits where we found NTFS MFT entries with a resident data attribute, we can reliably connect the metadata and the file content (Casey et al., 2019). This is because the resident data is found within the MFT entry. Normal file carving will not find small ASCII text files that are resident in MFT entries, because these files have no signature within their content.

MFT entries could potentially be found in multiple sources; memory dumps, unallocated space, in the allocated system files like $MFT, $MFTMirr, $LogFile, hiberfil.sys, etc. The allocated $MFT should normally be accurate if not manipulated.

Our approach does not need a complete MFT table, nor a complete MFT entry. For instance, we do not use the MFT entry header at all. However, we rely on that the SIA, FNA or Data attributes are co-located within the size of a typical MFT entry. This allows recovery of partly overwritten metadata. Currently, we do not search for a Data attribute if the SIA is not found, but we plan to change this dependency in later releases of the tool. This change will allow detection of FNAs in index entries.

We need to use the $Bitmap of the new file system to identify which clusters/blocks are in use. If a data run found in a recovered metadata structure uses one or more of the clusters allocated by a file in the new file system, we must assume that the file content is partly overwritten.

It is important that the investigator is knowledgeable about the file systems found when using our approach. First of all, our approach uses the data runs found within the NTFS metadata, and the first data run for the MFT entry is relative to the start of the file system, and we need to use the correct cluster size used by the previous NTFS file system. Furthermore, subsequent data runs are relative to the previous run (Carrier, 2005, p.258).

Since we are proposing a generic approach, we cannot automate the extraction of files without considering the specific context, which requires context based recovery tools or manual expert assessment. For instance, the storage media could have first had an NTFS file system, and then been reformatted with NTFS or another file system. Then of course the context is different, and must be taken into consideration.

We have shown that popular digital forensic tools, such as the current versions of X-Ways or Encase, do not necessarily find the MFT entries when the NTFS system is reformatted to exFAT. This may incorrectly cause the investigator to utilize file carving, which of course does not include the metadata, but only the file content. Such actions would result in missing pertinent files, and partly recovered fragmented files.

Discussion related to Ext4

If a user deleted files using the command line rm tool, or by emptying the trash, some of the important fields in the deleted inodes are set to 0, for instance the total size, the link count, the number of extents, and the extents fields. The timestamps for changed, modified and deleted are set to the deletion time, while the accessed and created are not changed. However, since we may find duplicate inodes in locations outside the inode tables, we may find previous versions of a deleted inode, which can allow us to recover the content and the metadata.

Addressing our statistics and current challenges

Statistics: Our high precision and recall does not indicate that our tool will find nearly all metadata entries without error, but it indicates that it will work well *given* that the metadata structures include repeated timestamps. When creating the disk images, files were guaranteed to have at least two or three identical timestamps. If our current solution is applied on a realistic disk image, the percentage of identified metadata entries should effectively be the same as the percentage of metadata entries on the disk that have the identical timestamps.

Metadata Remnants: Our approach does not differentiate between MFT records/inodes found in the MFT/inode table and the instances found in the journal or elsewhere. This is not a limitation, this is a feature since remnant from metadata structures describing files can be scattered across the file system.

Virtual Machines: We assume that not all virtual storage in a Virtual Machine is wiped on creation. This means there could be remnants from metadata from previous host file systems if the area assigned to the virtual storage has previously been allocated to the MFT/inode table or to a previous journal. Our approach will also find these timestamp locations.

Our approach can also be used to identify metadata currently not linked to existing file content, which is important for event reconstruction.

Conclusion and further work

The aim of this research was to answer the following research questions.

- Can we reliably use time as a generic identifier to carve for file and directory metadata structures in different file systems?
- What is the reliability of recovery of files using the discovered metadata in Ext4 and NTFS?

We have shown that a set of similar timestamps can be used as a form of dynamic signature (magic identifier), and we carve for these by using a simple byte matching algorithm. Then we use file system semantics in order to interpret metadata structures, and manually extract the resident or non-resident files. Finally, a file system expert evaluates the classification, authentication and makes a decision for final classification of the manually recovered files.

We argue that a manual evaluation of the reliability of the connection between the metadata and file content is necessary, and that this assessment is context based and should be manual for non-resident file content. The manual assessment could, however, be supported by automated tools.

Connecting the inode number and the file name is challenging in `Ext4` when an inode table is partly wiped. However, connecting the inode metadata with its corresponding file content is still possible even without the correct inode number or file name. On the non-reformatted `Ext4` image, we were able to achieve perfect precision and recall when attributing inode numbers to the extracted inodes of the files and directories we placed on the image. For the reformatted `Ext4` image, it was possible to achieve greater than 28% precision in correctly attributing inode numbers to the extracted inodes of the files and directories we placed on this image. Of the known 25050 known files and directories in the original image, we were able to recover inodes for 5755 of them. Since at least 20257 of the 25050 inodes were wiped from their inode tables, this means that we are potentially recovering at least 963 inodes.

When accurately connecting the metadata to the file content, we increase the evidential value of the evidence. We should not only use file carving when searching for files in unallocated space, since there may be pertinent metadata structures within unallocated space. As long as metadata structures exist in unallocated space, our generic metadata time carving approach combined with the semantic parsers can be used to connect metadata to file content. Knowledge of the file system context is necessary in order to assess the accountability of the connection between the metadata carved and the file content recovered.

When extracting inodes, our method had 100% precision for both the original `Ext4` image and the reformatted image. Even though our tool outperforms commercial tools given our specific experimental setup, our tool should still be considered a proof of concept prototype.

Support for file systems other than `Ext4` and `NTFS` is left for further work. Automation of file recovery is possible, but requires context aware features. Further research is needed in order to improve the accuracy of connecting `Ext4` inode number and file name to the inode entry, especially in the context of partially wiped inode tables.

Acknowledgement

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project “Ars Forensica - Computational Forensics for Large-scale Fraud Detection, Crime Investigation & Prevention”, grant agreement 248094/O70.

References

Apple, 2004. HFS plus volume format. https://developer.apple.com/library/archive/technotes/tn/tn1150.html. Last visited 2019-12-20.

Bhat, W.A., Wani, M.A., 2018. Forensic analysis of B-tree file system (Btrfs). Digit. Invest. 27, 57–70. http://www.sciencedirect.com/science/article/pii/S1742287618302135.

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

Casey, E., Nelson, A., Hyde, J., 2019. Standardization of file recovery classification and authentication. Digital Investigation. http://www.sciencedirect.com/science/article/pii/S1742287618304602.

Cho, G.-S., 2013. A computer forensic method for detecting timestamp forgery in ntfs. Comput. Secur. 34, 36–46. http://www.sciencedirect.com/science/article/pii/S0167404812001721.

Department of Defence Cyber Crime Center, 2012. Dc3dd. Last visited: 2019-09-19. https://sourceforge.net/projects/dc3dd/files/dc3dd/7.2.646/.

Dewald, A., Seufert, S., 2017. Afeic: Advanced Forensic Ext4 Inode Carving. Digital Investigation 20, p. S83. – S91, dFRWS 2017 Europe. http://www.sciencedirect.com/science/article/pii/S1742287617300270.

Ext4 development team, 2019. Ext4 header file. Last visited: 2019-09-11, code from the master development on github. https://github.com/torvalds/linux/blob/master/fs/ext4/ext4.h.

Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12. http://www.sciencedirect.com/science/article/pii/S1742287607000369.

Garfinkel, S.L., 2013. Digital media triage with bulk data analysis and bulk_extractor. Comput. Secur. 32, 56–72. http://www.sciencedirect.com/science/article/pii/S0167404812001472.

Göbel, T., Baier, H., 2018. Anti-forensics in ext4: on secrecy and usability of timestamp-based data hiding. Digit. Invest. 24, S111–S120. http://www.sciencedirect.com/science/article/pii/S174228761830046X.

Hamm, J., 2009. Extended FAT file system. Last visited 2018-09-16. https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf.

Hansen, K.H., Toolan, F., 2017. Decoding the APFS file system. Digit. Invest. 22, 107–132. http://www.sciencedirect.com/science/article/pii/S1742287617301408.

Mccash, J., 2010. Timestamped registry & NTFS artifacts from unallocated space. https://digital-forensics.sans.org/blog/2010/05/04/timestamped-registry-ntfs-artifacts-unallocated-space.

Mueller, L., 1, 2008. Search for windows 64 bit timestamps. http://www.forensickb.com/2008/01/search-for-windows-64-bit-timestamps.html.

Nordvik, R., Georges, H., Toolan, F., Axelsson, S., 2019. Reverse engineering of ReFS. Digit. Invest. 30, 127–147. http://www.sciencedirect.com/science/article/pii/S1742287619301252.

Nordvik, R., Porter, K., Toolan, F., Axelsson, S., Franke, K., 2020. cPTS carve for potential timestamps. Last visited 2020-03-20. https://github.com/RuneN007/cPTS.

Perry, J.W., Kent, A., Berry, M.M., 1955. Machine literature searching x. machine language; factors underlying its design and development. Am. Doc. 6 (4), 242–254. https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.5090060411.

Plum, J., Dewald, A., 2018. Forensic APFS file recovery. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. ARES 2018. ACM, New York, NY, USA, p. 47. https://doi.org/10.1145/3230833.3232808, 1–47: 10.