Digital Investigation 30 (2019) 127–147

Contents lists available at ScienceDirect

Digital Investigation

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

Reverse engineering of ReFS

Rune Nordvik [a, b, *], Henry Georges [a], Fergus Toolan [b], Stefan Axelsson [a, c]

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

ARTICLE INFO

Article history:
Received 23 March 2019
Received in revised form
4 July 2019
Accepted 17 July 2019
Available online 23 July 2019

Keywords:
Digital forensics
ReFS
File system

ABSTRACT

File system forensics is an important part of Digital Forensics. Investigators of storage media have traditionally focused on the most commonly used file systems such as NTFS, FAT, ExFAT, Ext2-4, HFS+, APFS, etc. NTFS is the current file system used by Windows for the system volume, but this may change in the future. In this paper we will show the structure of the Resilient File System (ReFS), which has been available since Windows Server 2012 and Windows 8. The main purpose of ReFS is to be used on storage spaces in server systems, but it can also be used in Windows 8 or newer. Although ReFS is not the current standard file system in Windows, while users have the option to create ReFS file systems, digital forensic investigators need to investigate the file systems identified on a seized media. Further, we will focus on remnants of non-allocated metadata structures or attributes. This may allow metadata carving, which means searching for specific attributes that are not allocated. Attributes found can then be used for file recovery. ReFS uses superblocks and checkpoints in addition to a VBR, which is different from other Windows file systems. If the partition is reformatted with another file system, the backup superblocks can be used for partition recovery. Further, it is possible to search for checkpoints in order to recover both metadata and content.

Another concept not seen for Windows file systems, is the sharing of blocks. When a file is copied, both the original and the new file will share the same content blocks. If the user changes the copy, new data runs will be created for the modified content, but unchanged blocks remain shared. This may impact file carving, because part of the blocks previously used by a deleted file might still be in use by another file. The large default cluster size, 64 KiB, in ReFS v1.2 is an advantage when carving for deleted files, since most deleted files are less than 64 KiB and therefore only use a single cluster. For ReFS v3.2 this advantage has decreased because the standard cluster size is 4 KiB.

Preliminary support for ReFS v1.2 has been available in EnCase 7 and 8, but the implementation has not been documented or peer-reviewed. The same is true for Paragon Software, which recently added ReFS support to their forensic product. Our work documents how ReFS v1.2 and ReFS v3.2 are structured at an abstraction level that allows digital forensic investigation of this new file system. At the time of writing this paper, Paragon Software is the only digital forensic tool that supports ReFS v3.x.

It is the most recent version of the ReFS file system that is most relevant for digital forensics, as Windows automatically updates the file system to the latest version on mount. This is why we have included information about ReFS v3.2. However, it is possible to change a registry value to avoid updating. The latest ReFS version observed is 3.4, but the information presented about 3.2 is still valid. In any criminal case, the investigator needs to investigate the file system version found.

Introduction

Reverse engineering of closed source file systems is a prerequisite for digital forensic investigations (Marshall and Paige, 2018). Hence, when investigating a digital storage medium it is imperative to retrieve the pertinent files from different file systems. Most investigators use digital forensic tools to retrieve this information,

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

https://doi.org/10.1016/j.diin.2019.07.004

and depend on these tools because of large backlogs (Scanlon, 2016). Digital forensic tools do not support all existing file systems, and there might be deviations between which file systems different tools support. Further, different tools might implement support for the same file system differently, and all versions of the file system are not necessarily supported. Because of limited budgets, digital forensic investigators only have access to a limited number of digital forensic tools (Garfinkel, 2010). Without tools to parse the underlying file system in a forensically sound manner, we are left with a large unknown volume. It will still be possible to carve files based on known file internal signatures (headers and/or footers), but this approach will not find the metadata structures of files, and often only partly retrieves the content of fragmented files (Garfinkel, 2007).

Showing both metadata and the corresponding files in the directory structure increases the evidentiary value of the evidence, and the lack of file system parsing support in current digital forensic tools motivated us to reverse engineer the Resilient file system (ReFS). At the start of this research EnCase v.8 had support for ReFS v1.2, and we needed to understand the structures of this file system in order to verify results from this tool.

We found that Paragon Software now supports full forensic ReFS access (Paragon Software, 2019b), but their implementation is closed source.

Objectives

The main research problem is that the low level structures of the new ReFS file system are undocumented, and investigators are left with just a few tools that support parsing. It is best practice to test tools, but this is not feasible for tools that implement ReFS support without describing the structures. Hence the objectives are:

- How can digital forensic investigators verify the reliability of ReFS file system support in existing Digital Forensic tools?
- How will B+ tree balancing impact the recoverability of files in ReFS?
- Can digital forensic investigators be confident in the ReFS integrity protection mechanism of metadata?
- How will ReFS impact the recovery of files compared to NTFS?

We aim to solve the first research question by describing the structures necessary to parse the file system. Knowing the structures will give digital forensic investigators the ability to verify the digital forensic tools that claim ReFS support. The investigator will be able to verify tool results, if they understand the low level structures, by manually parsing the file system on a low level of abstraction. The investigator can do this by using an existing methodology, for instance the Framework of Reliable Experimental Design (FRED) (Horsman, 2018).

The second research question is about B+ tree balancing, which often leaves remnants (Stahlberg et al., 2007), and we aim to verify if B+ tree node remnants in ReFS contain artifacts relevant for file recoverability. An introduction to B+ trees related to file systems has been described by Carrier [5, p.290]. The aim is to identify possibilities for recovering files based on unallocated metadata structures.

The third research question is to test if there are existing tools that can manipulate metadata in ReFS in such a way that impact our confidence in the integrity protection mechanism. Would it be possible to manually manipulate a timestamp using a hex editor, without the file system detecting or fixing it. Will ReFS be resilient for this kind of manipulations?

The fourth research question is about the use of remnants found in unallocated space for recovery of files. This is important to know since it may allow the investigator to restore previous files. We compare this with NTFS, where unallocated records in the $MFT can be recovered as long as they are not overwritten [5, p.328].

In order to answer the research questions we will need to first reverse engineer and interpret the structures used.

Features important for digital forensics

ReFS, Microsoft's newest file system, increases the availability of data (Microsoft, 2018c). If integrity streams for data (file content) are enabled, the integrity of the data is also increased. Unfortunately, integrity streams for data are not enabled by default (Microsoft, 2018b). However, integrity streams for metadata are enabled. This is a very important feature, because it means increased reliability of the metadata.

ReFS still uses the concept of attributes. The attributes found in the NTFS $MFT are similar to the attributes found in ReFS, but not identical (Head, 2015). However, the $MFT is not a part of ReFS (Head, 2015). Instead, the attributes are now located in B+ tree nodes.

If B+ tree reorganizing leaves remnants of confidential information, then these remnants might further support recovery of deleted data, highly relevant to digital forensics. The use of B+ trees in databases implies remnants of privacy data (Stahlberg et al., 2007).

Currently, there are tools used for manipulation of metadata on NTFS, for instance **SetMace** (Schicht, 2014), and we tested this tool on ReFS, but it did not work since there was no $MFT in ReFS. This will be similar for other tools that depend on manipulating the $MFT. As long as no ReFS metadata manipulation tools exist, digital forensic investigators may have increased trust in the validity of the metadata.

In this paper we describe the main structures necessary to manually interpret ReFS, and we have published a prototype tool that is able to parse the structures of ReFS v1.2. We have published the prototype tool under an open source license, and the tool is available from: https://github.com/chef2505/refs. Publishing the tool allows peers to review our interpretation of ReFS, and to test our research reproducibility in order to make it comply with the Daubert criteria (US-Supreme-Court, 1993). Making the prototype tool available as open source also allows other developers to implement support for newer versions. Its purpose is to automate the manual parsing of the file system in order to test our hypotheses. Therefore, the prototype tool is not discussed further in this paper.

Limitations

This reverse engineering of ReFS, a closed source file system, was initially performed without the checked/debug version of Windows 10, and therefore the names of the structures might not correspond with the names given by the file system developers. When finalizing this paper, we debugged the file system using the partly debug/checked version from Windows 10.0.17134x64 bit, including their symbols. We found that the names for structures and structure field names were not included, only the function names. Therefore, the real names of the structures are still unknown.

The selections of file system instances does not handle all possible use cases. Further, the results of this study are only valid for the file system versions described.

We describe only ReFS v1.2 and ReFS v3.2 in this paper. A driver is a software running within the kernel, and as all software it may be updated to newer versions. Microsoft may develop new features that may change the structures described in this paper. During the

finalizing of this paper we found that the latest ReFS version was ReFS v3.4 (Unknown, 2019). We were able to manually parse the ReFS v3.4, using the same structures as those defined for ReFS v3.2.

Organization of this paper

The remainder of this paper describes previous work and our contributions in section 2. Then we describe the main methods we have used to reverse engineer ReFS in section 3. Our results for ReFS v1.2 are described in section 4, and the results for ReFS v3.2 are described in section 5. In section 6 we discuss the results, and finally in section 7 we summarize and describe further work.

Literature review

Little peer-reviewed research on ReFS has been published, and therefore we will describe similar research performed on other popular file systems in addition to previous work on ReFS. This is relevant because there are similarities between file systems.

Jeff Ham (Hamm, 2009) was the first to publish documentation about the ExFAT file system structures, and this work helped practitioners to understand and analyze this file system. The system is similar to the old FAT systems, but each file has multiple directory entries which describe metadata about files. In addition there is a file allocation table which is used mainly for fragmented files, and a bitmap file for describing which cluster (block) is allocated. Vandermeer et al. (2018) continued research of ExFAT and were able to separate deleted entries from renamed entries by correlation of the FAT table, the bitmap file and the directory entries.

Brian Carrier (2005) has documented the NTFS file system, a work based on the reverse engineering performed by the Linux-NTFS Project. In NTFS everything is a file, and one of the most important system files with forensic value is the master file table ($MFT). All files have at least one entry, even the $MFT itself. NTFS is the main file system on Windows, and is still used on Windows system volumes including Windows 10.

Microsoft (2018a) has built the new Resilient File System (ReFS) on the basis of NTFS. However, instead of using a master file table, they now use a single B+ tree where metadata, bitmaps (allocators), files and folders can be found. Unfortunately, the inner structures are not documented.

Metz (2013) has partly described some of the structures within the ReFS file system, we assume it is for ReFS v1.1 and 1.2, however, the results are preliminary. He identified structures that he referred to as Object identifiers at levels 0,1,2 and 3. These object identifiers are metadata structures that are 16 KiB in size. We have named level 0 as the superblock, level 1 as checkpoint, level 2 as $Object_tree and level 3 as directories which can contain files and subdirectories.

Head (2015) has compared FAT32, NTFS and ReFS and he found that the File System Recognition Structure was used in the ReFS volume boot record. Head also described that there is no $MFT in ReFS, and there are no instances of FILE0 or FILE to indicate any MFT records. Further, Head found attribute style entries, and a $I30 index attribute in every folder. Head also describes that ReFS file content is not saved resident within an attribute. Head also shows that a previous name of a renamed folder was found in an earlier 16 KiB metadata block.

Gudadhe et al. (2015) describe some of the features that ReFS has and describe that ReFS uses B+ trees. Further they describe that a checksum is always used for preserving the integrity of metadata, and that a checksum for preserving file content can be enabled per file, directory, or volume. They do not describe structures or artifacts that might be important for digital forensic investigation.

Ballenthin has published information about ReFS in memory structures (Ballenthin, 2018a) and ReFS on disk structures (Ballenthin, 2018b). Our work started on the basis of this work.

Georges (2018) has published a masters thesis about the reverse engineering of ReFS v1.2, and his interpretation is at a low level. We scrutinize his work, and improve it. Georges describes that he was unable to document the structures of ReFS v3.2, therefore, we continue the reverse engineering of ReFS v3.2.

Paragon Software (Paragon Software, 2019a) was the first to release a ReFS driver for Linux, and it supports ReFS v1.x and ReFS v3.x. They have not released the source code for the driver, and customers need to contact them in order to get information about this driver. They have also included support for ReFS in their digital forensic tool (Paragon Software, 2019b).

Brian Carrier (2005) has also documented Ext2 and Ext3, which use superblocks and group descriptors for file system layout, and inodes for file metadata.

Kevin Fairbanks (2012) has documented Ext4, which is similar to Ext2 and Ext3, but Ext4 has additional features.

Hansen and Toolan (2017) reverse engineered the APFS file system, which enabled investigators to analyse iOS and Mac devices. In 2017, none of the commercial digital forensic tools had support for APFS. APFS also uses inodes for describing metadata about files. The APFS file system uses B+ trees extensively.

Plum and Dewald (2018) continued the work where they proposed novel methods for file recovery in APFS. They utilize known structures in order to recover files.

In October 2018 Apple released the APFS specifications which show the actual structures and their meaning (Apple, 2018a), which also could be used for digital forensic purposes. Apple has also published technical information about the HFS + file system, which also uses B+ trees (Apple, 2018b).

Stahlberg et al. (2007) describe the threat of privacy when using database systems that utilize B+ trees. They propose a system that overwrites obsolete data when the B+ tree is balanced and still in memory. This is relevant for this paper because this gave us the hypothesis that B+ tree balancing in ReFS will leave remnants of confidential information.

Method

We used reverse engineering as a method for finding the structures of ReFS. Initially we used the **diskpart** command or the **format** command in Windows 10x64 (ver 10.0.14393) to format different partitions with ReFS file systems on one disk. We tried to use different cluster sizes when formatting ReFS partitions, however this was not possible for ReFS v1.2. We also created ReFS volumes where we added both small files and large files.

In order to enable formatting of ReFS v1.2 we added a registry hack which allows formatting ReFS over non-mirrored volumes (Winaero, 2018). In Windows 10 Pro we were able to format ReFS v3.2 without any hack. Even when the registry hack was enabled, we were not able to format USB thumb drives using ReFS. We have observed that Microsoft has removed the option to format ReFS in Windows v 10.0.17134x64 Pro, and the previous Registry hacks do not work. We are still able to mount with read and write support in this version of Windows 10. We can still use one of the Windows Server editions to format a ReFS volume, and they can be attached to Windows 10, which automatically update the volume to the latest ReFS version. We are not sure why Microsoft has removed the support for formatting ReFS volumes in Windows 10 Pro.

We used FTK Imager or **ewfacquire** to create forensic images of the disks, and mounted these forensic images using **ewfmount** in Linux.

In order to find the volume boot records we parsed the MBR or

the GPT using a hex viewer/editor. Then we skipped to the sector location where the volume boot record starts in order to interpret the volume (Carrier, 2005). This start sector location is always important when investigating a disk image with multiple ReFS volumes, because the ReFS file system volume uses pointer offsets that are relative to the start of the volume boot record.

ReFS volume boot record

Using knowledge about the NTFS volume boot record, we assumed ReFS should contain information such as sector size, cluster size, volume serial number, the location of the metadata structure for files ($MFT), etc. Since the command line tool **fsutil** can yield important properties for the file system, we used this tool to verify our interpretation of the fields found in the VBR.

B+ tree

We knew from the sparse documentation from Microsoft that they extensively used B+ trees in ReFS (Microsoft, 2018a). Microsoft describes that one single B+ tree structure was used that could include other B+ tree structures (Microsoft, 2018a). Therefore, we started searching for blocks of data that could have a typical node structure. Then we tried to identify the entry point that gives access to the top node of the B+ tree. When analyzing the B+ tree structures we used knowledge about the HFS + file system (Apple, 2018b).

We knew from Stahlberg et al. (2007) that B+ trees in database systems could be a threat to privacy, because balancing B+ trees without wiping the content might leave remnants. For ReFS remnants of previous metadata records, for example records containing attributes, could have an evidentiary value for digital forensic investigation. Therefore, we tried to identify if attributes are intact even when they are not pointed to by pointers in the pointer area of a node.

Keywords or signatures

When we found different keywords or signatures, we searched for information about these to see if they could be connected to known structures.

In memory structures

Often structures in memory are saved directly to disk. We used previous work on ReFS memory structures in order to see if they are also found on disk (Ballenthin, 2018a). We did not perform any in depth kernel debugging or memory analysis, but rather used structures from previous work. The main reason for this was that we were not able to get hold of a checked/debug build of Windows 10. When using partly checked Windows 10, we did not get access to private symbols for the refs.sys driver. We did, however, list public function names that were available in the partially checked Windows 10 version.

Known structures

We used our knowledge of known structures and compared them to patterns discovered in hexdumps in order to give them meaning. Whenever a structure was interpreted, multiple tests were performed in order to try to falsify our interpretation (Popper, 1953). We also compared our observations with the ReFS structures on disk that Ballenthin has published (Ballenthin, 2018b), which also gave us an indication of searching for the entry block 0x1E and the standard block size of an entry block.

Comparing to other solutions

When we started this research only EnCase v7 or newer had support for parsing ReFS v1.2, but no tool was able to parse ReFS v3.2. We used information we observed when opening our forensic images in EnCase v7, and have tried to use the same names on system files as EnCase used.

Automation

Manually performing every test in a hex viewer is time consuming, and therefore we created a prototype tool to parse the structures found. This tool was used in our testing of ReFS v1.2. We have made the tool available as open source for other researchers to validate our work.

Experiments

When we reverse engineered ReFS v1.2, we used several experiments comparing different states of sectors within the file system, and we started by trying to understand the first sector of the file system, the volume boot record. Describing all these experiments is beyond the scope of this paper. Based on observations, we defined research hypotheses that could explain what we observed. Then we performed new experiments trying to falsify the null hypothesis, in order to indirectly get support for our main hypothesis about the meaning of a field.

For example one field that was unknown was the two bytes from 0x28 in the VBR. We observed the values 0x0102. When the ReFS file system was upgraded to version 3.2 we saw that the value was changed to 0x0302. Therefore, we defined the H_1 that this was the field for the file system major and minor version. We tested by formatting new instances of the file system with the old and the new version several times, and always the fields were corresponding to the version of the file system. We also identified that the file system was automatically updated after Microsoft released a new version of the driver. When we used fsutil to verify the file system version, it always corresponded to the value found in the VBR. We had to reject our null hypothesis H_0 that the changes of these values was only a result by chance alone. We did not observe once that the null hypothesis H_0 was true. Therefore, the alternate hypothesis H_1 was indirectly supported. Similar experiments were performed for the other values in the VBR, and values found in other structures.

After the reverse engineering of the ReFS file system, we performed a number of experiments in order to test if ReFS is resilient to metadata manipulation. First we tested the tool **SetMace** (Schicht, 2014) to see if it succeeds in changing the timestamp of a file located on a ReFS volume. We also tested to manually change the timestamp in a FNA file attribute.

We also checked if there are remnants of attributes not currently in use by the system, and if it is possible to recover data based on information found in these remnants.

Results - ReFS v1.2

In this section we will describe the structures necessary to manually parse the ReFS v1.2 file system. The forensic container file refs-v1_2.E01, available at Mendeley (Nordvik, 2019), allows the reader to follow our examples. These structures are the result of performing reverse engineering on the file system and testing forensic image containers containing ReFS file systems. We present the results as a guided tour through the structures necessary to interpret in order to find the metdatata, files and their contents.

We will start by describing our results for ReFS v1.2. The first

```
typedef struct _FILE_SYSTEM_RECOGNITION_STRUCTURE {
    UCHAR  Jmp[3]; // 3 unsigned chars: 3 bytes
    UCHAR  FsName[8]; // 8 unsigned chars: 8 bytes
    UCHAR  MustBeZero[5]; // 5 unsigned chars: 5 bytes
    ULONG  Identifier; // unsigned long : 8 bytes
    USHORT Length; // unsigned short : 2 bytes
    USHORT Checksum; // unsigned short : 2 bytes
} FILE_SYSTEM_RECOGNITION_STRUCTURE,
*PFILE_SYSTEM_RECOGNITION_STRUCTURE;
```

Fig. 1. File system recognition structure.

thing to note is that ReFS v1.2 always has 64 KiB clusters. The format tools in Windows 10 did not support formatting ReFS v1.2 with cluster sizes other than 64 KiB, even though the documentation (Microsoft, 2018c) claims support for both 4 KiB (default) and 64 KiB cluster sizes. Our observations show that ReFS v3.2 corresponds to the documentation in (Microsoft, 2018c).

ReFS volume boot record

For ReFS, the volume boot record is in the first sector of the file system volume, as it is for FAT32 and NTFS (Carrier, 2005). For FAT32 and NTFS the first 3 bytes are jump instructions to the boot code [5, p. 254]. However, since ReFS v1.2 and ReFS v3.2 can not be booted, the 3 first bytes are just zeros. We also found a C structure from Microsoft (Computing, 2018) defining the fields of the file system recognition structure (FSRS) as can be seen in Fig. 1[1]. Therefore, we got a kick start in interpreting the ReFS VBR. The first 6 fields in Table 1 were found by using Fig. 1. When interpreting the length of the types, we assume a Windows OS.

The names used for the FSRS structure are the names from the developers of ReFS, while the names from byte offset 0x18 are given based on our experiments.

Entry block

During our reverse engineering we found blocks of 16 KiB and blocks of 64 KiB. The blocks of 16 KiB were used for metadata and system files. However, for data streams 64 KiB allocation blocks were used. We named the 16 KiB blocks, entry blocks.

The first entry blocks we identified were file system metadata blocks with pointers to other entry blocks, and finally to the first entry block containing a B+ tree. We will describe how we found these entry blocks in section 4.4. Entry blocks in the B+ tree include nodes that followed a typical B+ tree pattern. We found a similar structure as used by other file systems that use B+ trees. We identified that all these 16 KiB blocks started with a descriptor for the block. The entry block includes a descriptor (size 0x30) and can include one or more nodes as shown in Fig. 2. When using the structure tables presented in this paper, the **E offset** means a byte offset relative to the entry block, while the **R offset** means a relative offset to the actual structure. Whenever the E and R offsets are equal, we will only show the E offset.

The first 8 byte field found at offset 0x0 of the entry block descriptor contains its entry block number. We also found at offset 0x18 a field describing the node identifier (Node ID). However, nodes that contain metadata will typically have the value 0 for the node id.

We found just one or two nodes in an entry block, but it could be more. Each node can have one or more records. A record contains the sub entries of a node. A node describing a directory will contain records of files and sub directories. There will be more than one record per file. In NTFS each file has at least one MFT record, which consists of a number of attributes, while in ReFS each of the attributes are contained within records in directory nodes. A file's standard information attribute is often more than 1000 bytes, which means there are not many files required until the entry block node is full. If an entry block contains a node that runs out of space for new records, then the B+ tree system will utilize a new entry block which consists of a node that has extent records. Extents make it possible for a node to extend its capacity by including records to other entry blocks, and adding more nodes, which for instance allows for more files within a directory node. In this case record 1 will contain an extent pointer to the entry blocks containing the existing node that is running out of space, and record 2 will have another extent pointer that points to a new entry block where the new records can be stored in a new node. We have not tested if the records could be reorganized between the two nodes. The order of records within a node is decided by the order of pointers in the pointer area. This means that records can appear in any order in the hex dump. We detected the extents records when we were experimenting with different numbers of files in the same directory.

Table 1
Structure of the volume boot record.

Offset	Length	Description
0x00	3	Jmp (Jump instructions)
0x03	8	FSName
0x0B	5	MustBeZero
0x10	4	Identifier
0x14	2	Length (of FSRS)
0x16	2	Checksum (of FSRS)
0x18	8	Sectors in volume
0x20	4	Bytes per sector
0x24	4	Sectors per cluster
0x28	1	File system major version
0x29	1	File system minor version
0x2A	14	Unknown
0x38	8	Volume Serial Number

Table 2
Structure of the entry block descriptor.

E offset	Length	Description
0x00	0x8	Entry Block number
0x08	0x08	Unknown
0x10	0x08	Unknown
0x18	0x08	Node ID
0x20	0x08	Unknown
0x28	0x08	Unknown

Superblock

At first we had only identified this superblock as an entry block that points to $Tree_Control. When we reversed engineered ReFS v3.2, we found the string SUPB in the entry block descriptor, which we believe is an abbreviation for superblock. From other file systems, such as ext4, we know the superblock is similar to the volume boot record. We were experimenting by trying to find pointers to the structures within the volume boot record, however we were looking in the wrong place. Microsoft has included these pointers in these superblock entries. We find it strange that they did not included all the information found in the VBR within the superblock.

[1] We added comments to make the structure easier to read for those not familiar with C structures.

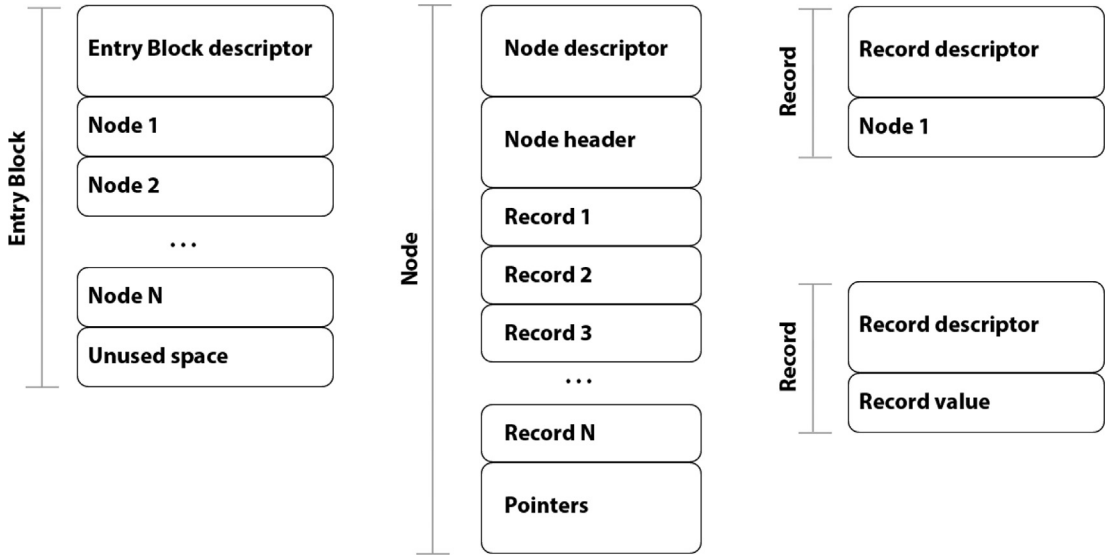

Fig. 2. Standard structures used by ReFS.

Finding the B+ tree check point

In order to find the B+ tree check point, the root of the B+ tree, we first need to find the superblock, then we follow the pointer to the check point. In this section we will try to manually find the starting node for the ReFS B+ tree. EnCase had a system file named $Tree_Control, which we believe is a check point. In ReFS v3.2 Microsoft includes the string CHKP in the entry block descriptor for $Tree_Control. Using the sector offset to this structure, we found the location where this structure started. However, Guidance Software/Open Text does not explain how to find the structure. In order to find out how, we searched for the values in the 0x1E entry block, the superblock, that could point to the check point. Assuming every entry block was 16 Kib, we could find the entry block (EB) offset number of the checkpoint $Tree_Control by multiplying the sector offset with 512 (sector size), and dividing it by 16 KiB (entry block size). Since the EB offset is relative to the start of the VBR, we subtract the VBR sector. The equation used to find the EB offset is shown in Equation (1).

$$\text{EB offset} = \frac{\text{EB sector*Sector size}}{\text{EB size}} - \text{VBR sector} \quad (1)$$

Converting the value to a hex value, made it easy to find the same value in the superblock. By searching for this EB offset hex value (using Big Endian), we always found the superblock in entry block 0x1E. However, it was not just one superblock.

We found three superblocks that had records with pointers to $Tree_Control. One of these three superblocks was always found in entry block 0x1E, and another one was found in the third last entry block in the volume. In addition we found an extra superblock backup in the entry block after the second one. When using **ewfmount** (in order to mount E01 files) and tools like **dd** and **xxd** (a hex viewer), we could show the content of the 0x1E super block. The command used is shown in Listing 1. The start of the VBR volume is at sector 0x800 on the test image, and 0x1E is the entry block we would like to show. In Listing 1 we have used a block size of 512 bytes, and a count of 1 block, which will only show 512 bytes of output. However, in the figures showing hex dumps, we have not included all 512 bytes to make it more understandable. In Equation (2) we show how we compute the sector offset to the sector start of the entry block, which we use in the skip option of the dd command.

$$\text{EB sector} = \text{VBR sector} + \frac{\text{EB number*EB size}}{\text{Sector size}} \quad (2)$$

```
dd if=mount/ewf1 bs=512 skip=$(( 0x800 + ((0x1E * 16384)
    ↪ /512) )) count=1 | xxd
```

Listing 1: Command to show the superblock at 0x1E.

Both the 0x1E superblock and $Tree_Control check point have a special structure. Table 3 shows the structure of these in order to parse the superblock (entry point), and an extract from the hex dump is shown in Fig. 3.

In the superblock 0x1E at offset 0x50 (4 bytes in length) the value 0xA0 was found, which is the byte offset to where we find the first entry block pointer. At the entry block byte offset 0xA0 (8 bytes in length) we find the entry block pointer to $Tree_Control. There is another pointer offset in 0xA8 (8 bytes in length), and we assume this is a pointer to the backup $Tree_Control. The highlighted value found in the hex dump shown in Fig. 3 is 0x1471 (LE) for the first pointer and 0xF3F7 (LE) for the second pointer.

To show the content of entry block 0x1471, we use the same command as shown in Listing 1, but we change 0x1E with the value 0x1471. This will change location to $Tree_Control, which is the checkpoint for our B+ tree structure.

In this section we have shown how to navigate to the entry block that controls the B+ tree. This entry block is a check point to the B+ tree. This entry block is named $Tree_Control because it can be used to navigate the B+ tree.

Top of the node tree

The entry block containing the top node of the B+ tree is called $Tree_Control by EnCase. From this entry block we can find

Table 3
Structure of the superblock.

E Offset	R offset	Length	Description
0x30	0x00	0x10	GUID
0x40	0x10	0x10	Unknown
0x50	0x20	0x04	Offset to first entry block pointer
0x54	0x24	0x04	Amount of entry block pointers
0x58	0x28	0x04	Offset to first record
0x5C	0x2C	0x04	Length of record each record

Fig. 3. Hex dump of the superblock 0x1E.

pointers to six different nodes that have a special purpose, and are shown by EnCase as system files. In section 4.4 we show how to find the $Tree_Control checkpoint manually by using a hex viewer. Now we continue to interpret the top level checkpoint node ($Tree_Control).

We skip the first 0x30 bytes in Fig. 4, which is for the entry block descriptor, and focus on the checkpoint descriptor. This entry block does not contain a typical B+ tree node, because the pointer area at the end is missing.

When using Table 4 we saw, in the hex dump, in Fig. 4 that the offset to the first record was 0x80, and this first record was a record entry for this $Tree_Control check point. Further, we saw from the major and minor version that this was ReFS v1.2. At offset 0x80 we found a 0x18 byte record. The first 8 bytes of this record had the value 0x1417 (LE). This was the same as the entry block number of the $Tree_Control check point as also could be seen in the first 8 bytes of the entry block. This means that each of the records in this checkpoint started with an entry block number to where the record points.

According to byte offset 0x58 the number of records was 0x06. At byte offset 0x5C we found a table of offsets, where each offset was 4 bytes and included byte offsets to each of the additional records in this entry block. In the example hex dump in Fig. 4 we found the following offset values: 0x98, 0xB0, 0xC8, 0xE0, 0xF8, 0x110. The offsets were relative to the start of the entry block. The first record started at offset 0x98, the next at 0xB0, etc. The list below shows what we will find in these records. Each of the records start with an entry block pointer.

- Pointer offset to the $Object_Tree system file can be found in entry block at offset 0x98, and in this example it was pointing to entry block 0x23F (LE). This system file includes nodes and records that have information about directories, and files in sub entry blocks.

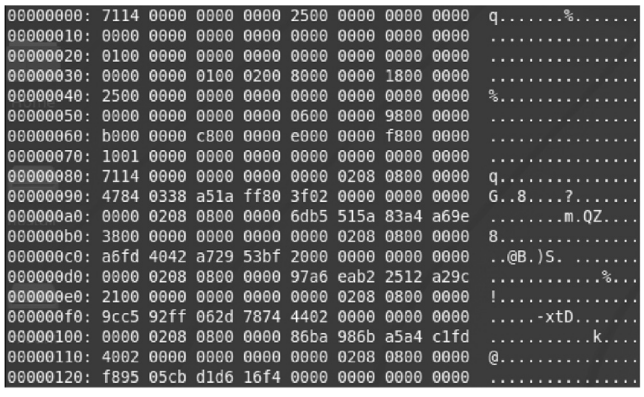

Fig. 4. Hex dump of the first part of check point $Tree_Control.

Table 4
Structure of the tree control checkpoint entry block.

E Offset	R offset	Length	Description
0x30	0x00	0x4	Unknown
0x34	0x04	0x2	Major version
0x36	0x06	0x2	Minor version
0x38	0x08	0x04	Offset to first record
0x3C	0x0C	0x04	Size of a record
0x40	0x10	0x10	Unknown
0x50	0x20	0x8	Unknown
0x58	0x28	0x04	Amount of additional records
0x5C	0x2C	Var	4 byte offset to each records

- Pointer offset to the $Allocator_Lrg system file can be found in the entry block at offset 0xB0, and in this example it was pointing to entry block 0x38 (LE). This system file includes a bitmap of large blocks (512 MiB).
- Pointer offset to the $Allocator_Med system file can be found in the entry block at offset 0xC8, and in this example it was pointing to entry block 0x20 (LE). This system file includes a bitmap of medium sized blocks (64 KiB).
- Pointer offset to the $Allocator_Sml system file can be found in the entry block at offset 0xE0, and in this example it was pointing to entry block 0x21 (LE). This system file includes a bitmap of small sized blocks (16 KiB).
- Pointer offset to the $Attribute_List system file can be found in the entry block at offset 0xF8, and in this example it was pointing to entry block 0x244 (LE). We are uncertain of the meaning of this system file.
- Pointer offset to the $Object system file can be found in the entry block at offset 0x110, and in this example it was pointing to entry block 0x240 (LE). This system file describes the child-parent dependencies, and can be used to rebuild the directory paths for files.

Until now we have used special entry blocks (superblock and check point), but in order to parse the normal nodes we introduce the structure for the standard node descriptor in Table 5 and the standard node header in Table 6. We will use them to interpret the normal nodes which we find in the sub nodes pointed to by $Tree_Control. We can think of $Tree_Control as the top block that maintains control of all the sub B+ trees, or as the root of the single B+ tree.

The standard node descriptor and standard node header can be used from the level we have described as MSB+ and below in the illustration in Fig. 5. The MSB+ is a magic signature for ReFS v3.2, and includes one or more nodes. In ReFS v1.2 the magic signature is not available, but we decided to name Entry Blocks containing nodes for MSB + anyway to make this consistent with ReFS v3.2. The abbreviation, we assume, is for Microsoft B+ tree.

In this section we have shown how to find the starting check point, named $Tree_Control. This checkpoint makes it possible to find all other nodes in the file system. In the illustration in Fig. 5 we have visualized the main structures of ReFS. The superblock was

Table 5
Structure of the standard node descriptor.

E Offset	R offset	Length	Description
0x30	0x00	0x04	Length of Node descriptor
0x34	0x04	0x14	Unknown
0x48	0x18	0x02	Number of extents
0x4A	0x1A	0x06	Unknown
0x50	0x20	0x04	Number of records in node
0x54	0x24	var	Unknown

Table 6
Structure of the standard node header.

E Offset	R offset	Length	Description
0x120	0x00	0x04	Length of Node header
0x124	0x04	0x04	Offset to next free record
0x128	0x08	0x04	Free space in node
0x12C	0x0C	0x04	Unknown
0x130	0x10	0x04	Offset to first pointer
0x134	0x14	0x04	Number of pointers in node
0x138	0x18	0x08	Offset to end of node

found in entry block 0x1E, and we have found the checkpoint $Tree_Control. There was also another $Tree_Control which we assume is a backup. We have already found the pointers to the child nodes of $Tree_Control.

Files and folders

In this section we will show how the B+ tree structures can be parsed to find directories and files. We start by looking at the entry block for $Object_Tree which, in this example, was located at 0x23F. We used the command described in Listing 1, but changed 0x1E with the entry block we wished to investigate. The hex dump of $Object_Tree is shown in Fig. 6. We skipped the entry block descriptor, which was 0x30 bytes. Here we found the node descriptor that started with a length field, which had the value 0xF0. We skipped the node descriptor to make this section easier to read.

From byte offset 0x120 we found the node header which included information about the length of the node header (0x20 bytes), offset to the next free record, free space in node and offset to first record pointer (0x354C (LE)) This is a relative offset from the start of the node header. To find the entry block byte offset we added 0x120. Therefore, the pointer area started at offset 0x366C. We also found the number of pointers (this should be equal to the amount of allocated records), which was 9. These pointers are 4 bytes in size and pointed to records in this node, and the offset was relative to the node header (we needed to add 0x120 in this case). In the node header we also found an offset to the end of the node. We skipped to the first record by moving to offset $0x20 + 0x120 = 0x140$. This record is followed by the fourth record $0x70 + 0x120 = 0x190$.

We could have used the pointers from the pointer area to find all allocated records, which we actually should do for any node. However, we only showed two of the records here. It is the pointers that decide the order of records, we can not assume that all records are in sequence or that all records are in use.

At offset 0x140 in Fig. 6 we found the start of the records area. Each record was 0x50 bytes. All the records were related to directories, either directories the user created or system created directories. These records could be analyzed using Table 7. The first record was a pointer to $Volume (node id 0x500), which contained information about the volume and also included a timestamp for volume creation, which could have a value for the investigation. The fourth record contained a pointer for the root directory (node id 0x600). At this level in the B+ tree we did not see the names of the nodes, but most of the names could be found within the entry blocks pointed to by the records at this level. We observed that records with node IDs from 0x700 and above were normal directories either created by the user or system created directories. We have illustrated this in Fig. 7. All the records with a node id within the 0x500 range were metadata directories, and not shown by File Explorer when parsing the root directory. All node IDs of root sub-directories were in the 0x700 range.

Volume information

We used Table 7 to interpret the records in the entry block $Object_Tree. The record for the 0x500 node id was found in entry

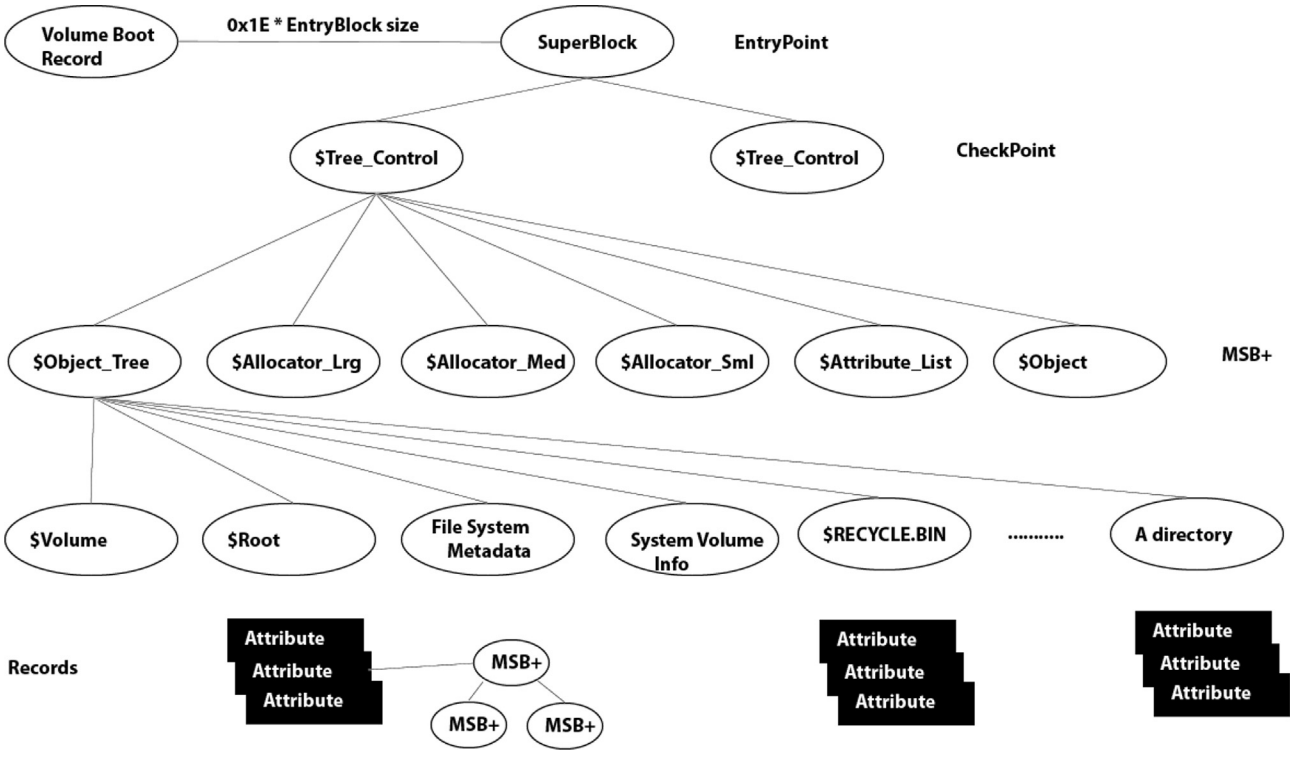

Fig. 5. Illustration of the ReFS structure.

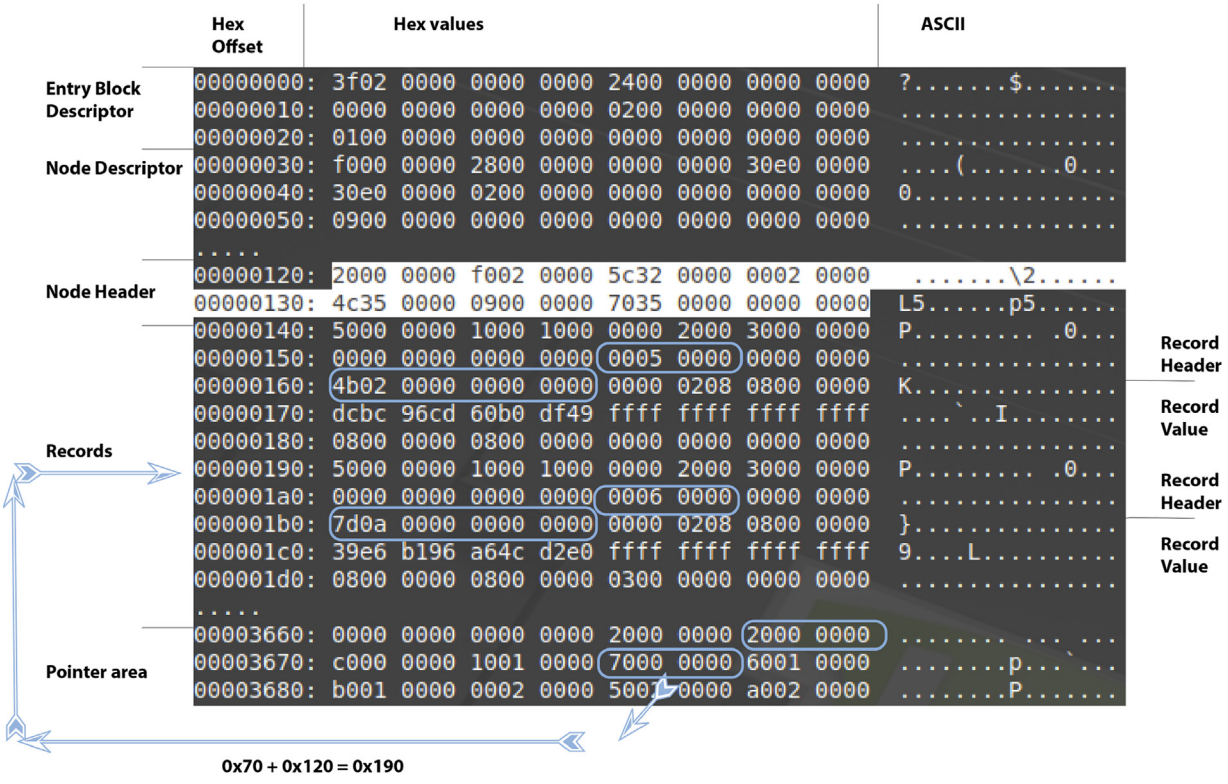

Fig. 6. Hex dump of the $Object_Tree.

Table 7
Structure of the $Object_Tree entry block.

E Offset	R offset	Length	Description
0x140	0x00	0x4	Record size
0x144	0x04	0x06	Unknown
0x14A	0x0A	0x2	Record header size
0x14C	0x0C	0x2	Record value size
0x14E	0x0E	0x2	Unknown
0x150	0x10	0x8	Unknown
0x158	0x18	0x4	Node ID
0x15C	0x1C	0x4	Unknown
0x160	0x20	0x8	Entry block number
0x170	0x30	0x8	Checksum
0x178	0x38	0x18	Unknown

block offset 0x140, and from offset 0x160 in Fig. 6 the entry block pointer could be found. In this case it pointed to the entry block 0x24B (LE), which described the $Volume directory. In the hex dump shown in Fig. 8 we show parts of the 0x24B entry block (the $Volume directory). We could see from the entry block descriptor at byte offset 0x18 that this was the entry block for the 0x500 node id. From the start of the node descriptor at offset 0x30 we saw that the node descriptor was 0xE8 (LE) in size. Adding 0x30 to this gave the start of the node header at offset 0x118. In entry block byte offset 0x128 we found the value 0x3570 (LE), which was the relative byte offset to the record pointer area. We added 0x118 to this and found the entry block offset 0x3688. We could see the first pointer pointed to $0x208 + 0x118 = 0x320$, and the second record pointer pointed to $0x20 + 0x118 = 0x138$. We have focused on this second record.

This record has two main parts. The first part is the record header, and the second part is the record value. Records found within entry blocks for directories contain attributes. We verified the use of attributes by performing kernel debugging, where we found functions such as:

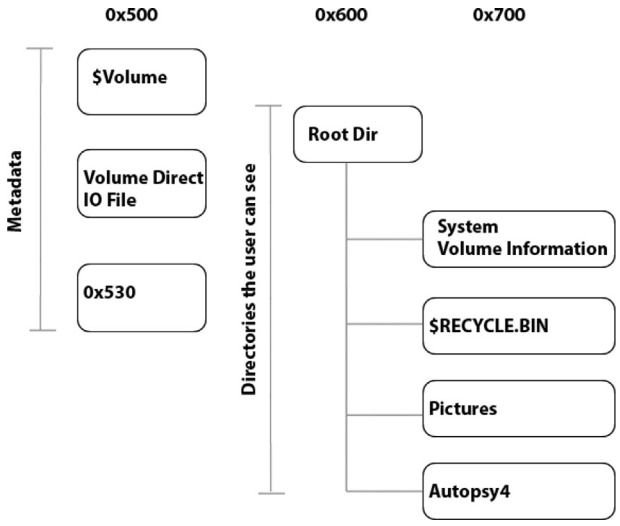

Fig. 7. Level of directories found in $Object_Tree.

- ReFS!RefsCreateAttributeWithValue
- ReFS!RefsFoundAttributeName

Unfortunately, the attribute header and the attribute value might be different based on the type attribute.

Using Table 8 we see that this attribute is used for the attribute type 0x20050000, and the total size of the attribute is 0x1E8.

The value of the attribute only has a few bytes of information, and Table 9 is used to interpret this attribute value. The timestamp found at offset 0x1E0 was when this volume was created (Sun, 21 Oct 2018 09:29:39 UTC). At offset 0x1F0 we found the last time the volume was mounted (Sun, 21 Oct 2018 10:55:48 UTC).

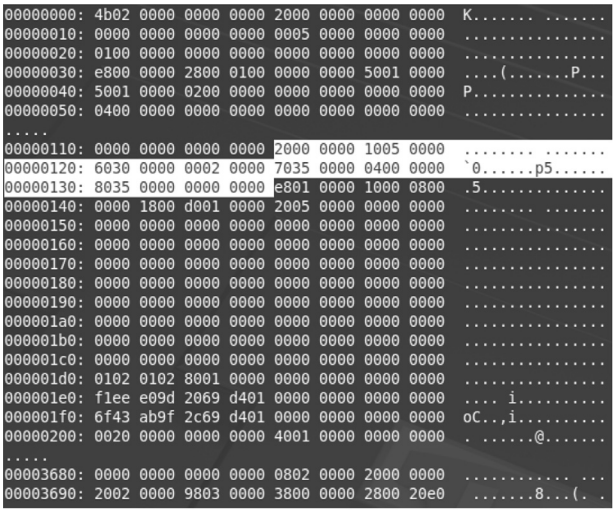

Fig. 8. Volume 0x500 directory entry block.

Root directory

We use Table 7 when interpreting the fourth record in the $Object_Tree entry block, shown highlighted in the hex dump in Fig. 9. The Node ID value was 0x600 (which we found at record offset 0x18 or entry block offset 0x1A8). In all our experiments we found the root directory when parsing the 0x600 node id. The entry block pointer was found at relative record offset 0x20 or entry block offset 0x1B0. In this example it pointed to the entry block 0xA7D (LE).

Therefore, we show the 0xA7D entry block by using the same command syntax as we have used previously.

Fig. 10 shows a few important parts of the 0xA7D Node ID 0x600 (the root directory in this case). The first 0x30 bytes contain the entry block descriptor. We used Table 2 to interpret the entry block descriptor. The next part of the hexdump was the node descriptor, which can be parsed using Table 5. We found that the node descriptor structure was 0xE8 in size. The node did not have any extents, and it contained 7 records. In order to find the start of the node header, we added the size of the entry block descriptor to the size of the node descriptor.

Table 9
Structure of the attribute value in $Volume.

E Offset	R offset	Length	Description
0x150	0x00	0x80	Unknown - not used
0x1D0	0x80	0x1	Major version
0x1D1	0x81	0x1	Minor version
0x1D2	0x82	0x1	Major version
0x1D3	0x83	0x1	Minor version
0x1D4	0x84	0x0C	Unknown
0x1E0	0x90	0x8	Volume created
0x1E8	0x98	0x8	Unknown
0x1F0	0xA0	0x8	Volume last mounted
0x1F8	0xA8	0x8	Unknown

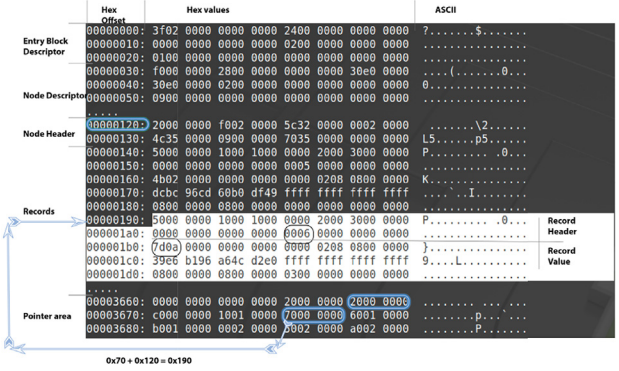

Fig. 9. $Object_Tree record 0x600.

We parsed the node header from offset 0x118 using Table 6, and found that it is 0x20 bytes in length, and the offset to the first record pointer in the pointer area was 0x3564. Since this is a relative offset from the start of the node header, we added 0x118 so the entry block byte offset is 0x367C. Each of the pointer offsets are 4 bytes in size, and in this case there were 7 record pointers. The first pointer was 0x20, and the next was 0xA60. In order to find the correct byte offset in the entry block we added 0x118. Therefore, we got the entry block offset 0x138 for the first record and 0xB78 for the second one. The fifth record is shown in the hex dump in Fig. 10 and it started at entry block offset 0x668. We saw it had the

$$\begin{aligned}\text{Offset Node Header} &= \text{Size of EB descriptor} + \text{Size of node descriptor}\\ \text{Offset Node header} &= 0x30 + 0xE8 = 0x118\end{aligned} \tag{3}$$

The offset to the node header is important because the pointers found in the pointer area are all relative to the start of the header node.

Table 8
Structure of the attribute header in $Volume.

E Offset	R offset	Length	Description
0x138	0x00	0x04	Attribute Size
0x13C	0x04	0x02	Offset to next part of header
0x13E	0x06	0x02	Used size of header from next part of header
0x140	0x08	0x02	Flags? (0x04 = Deleted)
0x142	0x0A	0x02	Size of attribute header
0x144	0x0C	0x02	Size of attribute value
0x146	0x0E	0x02	Unknown/Reserved?
0x148	0x10	0x04	Attribute Type

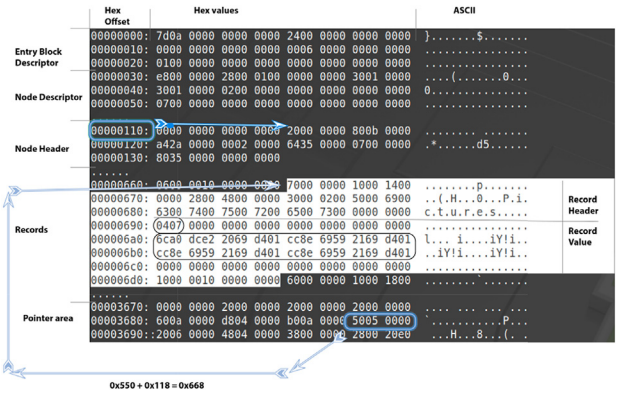

Fig. 10. Hex dump of entry block 0xA7D (Root directory).

directory name Pictures, and its node id was 0x704. In addition we saw 4 timestamps. Each timestamp is 8 bytes in length, and they should be interpreted in a similar manner to the timestamps in the $SIA attribute in NTFS. The order of these timestamps are File Created, File Content Changed, File Metadata Changed, and File Accessed.

We are now ready to parse a directory entry, and we will start with the attribute describing the current directory.

Standard information attribute (SIA)

We used the first pointer and looked at offset 0x138, which is shown in the hex dump in Fig. 11, we saw that the structure is large, it contained 0x428 (LE) bytes and it had the attribute type 0x10000000 (BE), which is the same identifier as used in NTFS for standard information attribute (SIA). We used Table 13 to interpret the SIA attribute header. After the attribute header, we found the value area, which in this case was 0x410 bytes and contained a B+ tree node.

We can use our knowledge of B+ tree nodes to interpret this structure. Microsoft described that a B+ tree could have an embedded B+ tree structure (Microsoft, 2018a). This was exactly what we saw here. We have tried to illustrate this in Fig. 12. At offset 0x150 we found the SIA node descriptor. Table 10 was used to interpret this. Interpreting the SIA node descriptor showed that the created timestamp was Sun, 21 Oct 2018 09:29:51 UTC, and the other timestamp was Sun, 21 Oct 2018 11:09:56 UTC. This entry did not have any logical or physical size, because there was no corresponding file on the disc.

We continue by interpreting the SIA node header, which gives the pointer to the pointers area. The pointer area is not shown in the hex dump in Fig. 11. The pointer area was located at $0x270 + 0x1F8 = 0x468$. We found two pointers here. The first pointed to record 1 at entry block offset $0xE0 + 0x1F8 = 0x2D8$ and record 2 started in entry block offset $0x20 + 0x1F8 = 0x218$.

We start by interpreting the first SIA record, it started in entry block offset 0x2D8 and was 0x88 bytes in size. The attribute header was 0x20 in size and the attribute value was 0x64 in size. The main

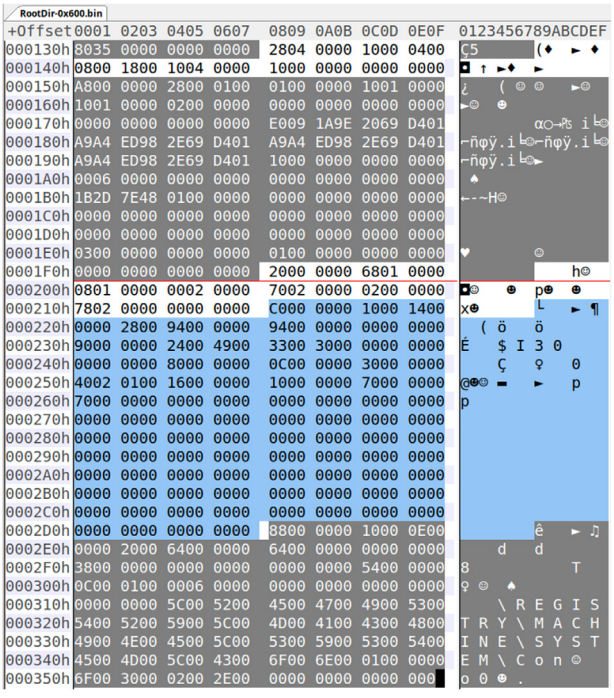

Fig. 11. Node within SIA attribute value.

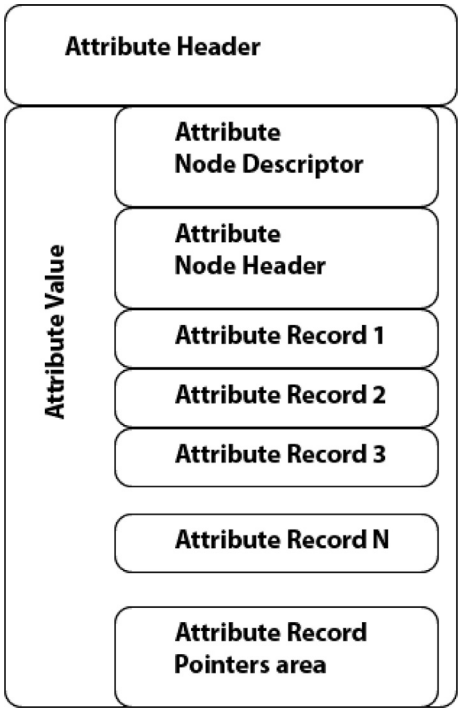

Fig. 12. Node inside an attribute value.

Table 10
Structure for the SIA/FNA attribute node descriptor.

E Offset	R offset	Length	Description
0x150	0x00	0x04	Size of structure
0x154	0x04	0x02	Offset to first timestamp
0x156	0x06	0x22	Unknown
0x178	0x28	0x08	Created
0x180	0x30	0x08	Modified
0x188	0x38	0x08	Metadata Modified
0x190	0x40	0x08	Last Accessed
0x198	0x48	0x8	Unknown (Attrib Flags?)
0x1A0	0x50	0x08	Parent Node ID
0x1A8	0x58	0x08	Child id (start on number 0)
0x1B0	0x60	0x08	Unknown
0x1B8	0x68	0x08	Logical file size (FNA)
0x1C0	0x70	0x08	Physical file size (FNA)
0x1C8	0x78	0x08	Unknown
0x1D0	0x80	0x08	Extra timestamp (FNA)

content was the text string: \REGISTRY\MACHINE\SYSTEM\Con.

In Fig. 11 we have highlighted the second record with a blue background, and our hypothesis was that this was a directory index with a pointer to a directory index entry block. It is the text $I30 that hints about an index root attribute, which was also used in NTFS to index the files in a directory. Here the attribute type is 90000000 (BE) and the attribute header is 0x28 in size, and the attribute value size is 0x94. In entry block byte offset 0x250 we find the two byte value 0x240. However, we observed that this value did not change for other instances of ReFS, and the $Object entry block was located in other entry blocks. Therefore, we had to falsify our first hypothesis. We found that a node that describes a directory, actually is the directory index.

Child attribute

Another attribute we found was what we called the child attribute, which is shown in the hex dump in Fig. 13. It has attribute

```
00000b70: 0000 0000 0000 0000 5000 0000 1000 1800  ........P.......
00000b80: 0000 2800 2800 0000 2000 0080 0000 0000  ..(.(... .......
00000b90: 0006 0000 0000 0000 0100 0000 0000 0000  ................
00000ba0: 0000 0000 0000 0000 0c00 1a00 5300 6d00  ............S.m.
00000bb0: 6100 6c00 6c00 4600 6900 6c00 6500 2e00  a.l.l.F.i.l.e...
00000bc0: 7400 7800 7400 0000 7000 0000 1000 1400  t.x.t...p.......
```

Fig. 13. Hex dump of a child attribute.

Table 11
Structure for the child attribute header.

E Offset	R offset	Length	Description
0xB78	0x00	0x04	Attribute Size
0xB7C	0x04	0x02	Offset to next part of header
0xB7E	0x06	0x02	Used size of header from next part of header
0xB80	0x08	0x02	Flags (0x04 = Deleted)
0xB82	0x0A	0x02	Size of attribute header
0xB84	0x0C	0x02	Size of attribute value
0xB86	0x0E	0x02	Unknown/Reserved
0xB88	0x10	0x04	Attribute type
0xB8C	0x14	0x02	Unknown
0xB90	0x18	0x08	Parent Node Id
0xB98	0x20	0x08	Child number

Table 12
Structure for the child attribute value.

E Offset	R offset	Length	Description
0xBA0	0x00	0x08	Unknown
0xBA8	0x08	0x02	Offset to file name relative to value start
0xBAA	0x0a	0x02	Size of file name
0xBAC	0x0c	var	Filename

id 0x20000080 (BE), and it has a parent node id, child number and the file name. We used Table 11 to interpret the values. The size is 0x50 (LE) and the header is 0x28 bytes and the parent node in this case is 0x600, which means that this entry is a child of the root directory. It is child number 1 in this directory.

Using Table 12 we can see that the name is SmallFile.txt.

File name attribute (FNA) for directories

The File Name attribute is known from NTFS, but as we will see there are two types in ReFS. We will use the fifth pointer as an example. The fifth pointer has the value 0x550 (LE), and adding 0x118 gives 0x668.

In the hex dump shown in Fig. 10 we see that at byte offset 0x668 a directory entry starts. The relevant part of this record is shown in Fig. 14.

We use Table 13 when parsing the directory attribute header. The size of the complete attribute is 0x70 (LE), and the attribute id is 0x30000200 (BE). This looks similar to the File Name attribute in NTFS, which is 0x30000000 (BE). We have observed that all directories (not files) in our experiments have 0x30000200 (BE), while files have 0x30000100 (BE). Therefore, we have decided to use the name File Name attribute for both. The name of this entry was Pictures, which was one of the directories we created.

If we look at the attribute value, using Table 14, it starts with the node id 0x704. This must be correlated with the pointer we found to the node id 0x704 in $Object_Tree B+ tree node. Therefore, we

```
000660h 0600 0010 0000 0000  7000 0000 1000 1400  ♠ ►        p   ► ¶
000670h 0000 2800 4800 0000  3000 0200 5000 6900    ( H   0 ☻ P i
000680h 6300 7400 7500 7200  6500 7300 0000 0000  c t u r e s
000690h 0407 0000 0000 0000  0000 0000 0000 0000  ♦•
0006A0h 6CA0 DCE2 2069 D401  CC8E 6959 2169 D401  lá▄Γ i╚◎╠ÄiY!i╚◎
0006B0h CC8E 6959 2169 D401  CC8E 6959 2169 D401  ╠ÄiY!i╚◎╠ÄiY!i╚◎
0006C0h 0000 0000 0000 0000  0000 0000 0000 0000
0006D0h 1000 0010 0000 0000  6000 0000 1000 1800  ►  ►     `   ► ↑
```

Fig. 14. Hex dump of a FNA directory attribute.

Table 13
Structure of the directory attribute header.

E Offset	R offset	Length	Description
0x668	0x00	0x04	Attribute Size
0x66C	0x04	0x02	Used size of header from next part of header
0x66E	0x06	0x02	Used size of header from next part of header
0x670	0x08	0x02	Flags (0x04 = Deleted)
0x672	0x0A	0x02	Size of attribute header
0x674	0x0C	0x02	Size of attribute value
0x676	0x0E	0x02	Unknown/Reserved
0x678	0x10	0x04	Attribute type
0x67C	0x14	var	Directory name

Table 14
Structure of the directory attribute value.

E Offset	R offset	Length	Description
0x690	0x00	0x08	Node ID
0x698	0x08	0x08	Unknown
0x6A0	0x10	0x08	Created
0x6A8	0x18	0x08	Modified
0x6B0	0x20	0x08	Metadata Modified
0x6B8	0x28	0x08	Last Accessed
0x6C0	0x30	0x10	Unknown
0x6D0	0x40	0x08	Unknown

now know that 0x704 has the directory name Pictures. In addition we found 4 timestamps, and these should be interpreted as 100 ns intervals since 1.1.1601 UTC (FILETIME). This directory was created at Sun, 21 Oct 2018 09:31:47 UTC. While the other timestamps all had the time Sun, 21 Oct 2018 09:35:06 UTC. It is out of scope for this study to identify which actions update the timestamps.

File name attribute (FNA) for files

In order to explain the FNA file attribute, we use the illustration in Fig. 15. Let us assume Record (attribute) 2 is the FNA file attribute. The first part is the FNA File attribute header. We used Table 13 when we interpreted the FNA File attribute header, which started at entry block offset 0x738. The hex dump is shown in Fig. 16.

Files have the attribute type 0x30000100. This structure is larger than the one used for directory FNAs. The size of the complete structure was, in this case, 0x440 (LE) and the name was SmallFile.txt.

This attribute value contains its own B+ tree node. It starts with

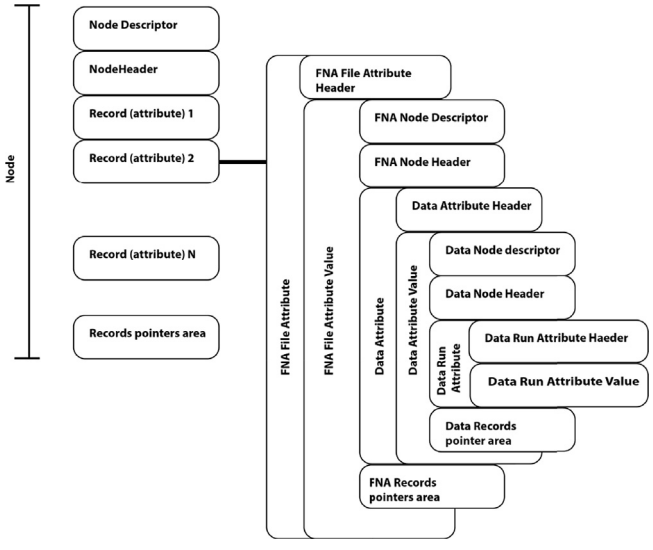

Fig. 15. Illustration of FNA file attribute.

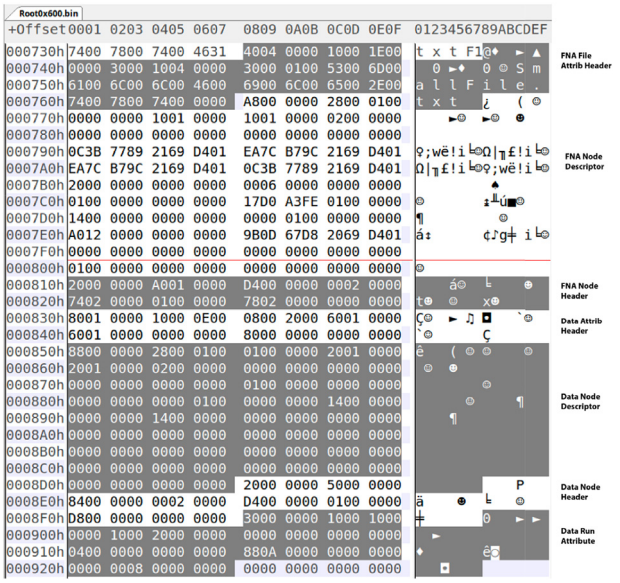

Fig. 16. Hex dump of a FNA file.

a FNA node descriptor which is 0xA8 in size, and we use Table 10 to interpret it. We have four timestamps in this FNA node descriptor. The created value was Sun, 21 Oct 2018 09:36:26 UTC. At the entry block offset 0x7E8 (relative offset 0x80) we found a fifth timestamp (we have not seen this fifth timestamp in all our ReFS volumes). In this case this timestamp had a value before the other four timestamps, Sun, 21 Oct 2018 09:31:29 UTC. We do not know what this timestamp represents, it may be a remnant from a previous attribute.

The parent Node Id found in 0x7B8 was 0x600, and child number found in 0x7C0 is 0x1. The data content in 0x7D0 is logical 14 bytes, and we see from 0x7D8 that it uses 0x10000 bytes (128 sectors, or 64 KiB). This means a file uses at least one 64 KiB cluster. In ReFS the FNA file attribute will only provide the pointer to the file, independent of the file size. These pointers are found within the internal Data Run Attribute (see Fig. 15).

The next part in this attribute value was the FNA node header, which started at offset 0x810 bytes. Here we found the offset to the FNA node pointer area. The first and only pointer started at entry block offset $0x274 + 0x810 = 0xA84$. This pointer area is not shown in the hex dump, but it contained the pointer record at offset $0x20 + 0x810 = 0x830$.

Record 1, at entry block offset 0x830, is 0x180 in size and the attribute type is 0x80000000. NTFS uses this value as a data attribute. This record header is 0x20 in size (highlighted with white background). The next part was the record value, which was 0x160 in size. This attribute value was also a B+ tree node, which we call the Data Node!

The Data Node descriptor started at entry block offset 0x850, and was 0x88 in size. We observed within the node descriptor at entry block offset 0x884 (4 bytes length) that we had the value 0x10000, which is the number of bytes within the 128 sectors allocation size, and at entry block offset 0x88C (4 bytes length) we had the value 0x14, which was the logical size of this file.

At offset 0x8D8 we found that the Data Node header had the Data Records pointer area in $0xD4 + 0x8D8 = 0x9AC$. Here we found the pointer $0x20 + 0x8D8 = 0x8F8$.

Here we found the Data Run attribute (record) (see Table 15). The data run started at entry block offset 0x8F8 and was 0x30 in size. The Data Run attribute header was 0x10 in size. We skipped

Table 15
Structure of the Data attribute header.

E Offset	R offset	Length	Description
0x830	0x00	0x04	Attribute Size
0x834	0x04	0x02	Offset to next part of header
0x836	0x06	0x02	Used size of header from next part of header
0x838	0x08	0x02	Flags? (0x8 = B+ tree node)
0x83A	0x0A	0x02	Size of attribute header
0x83C	0x0C	0x02	Size of attribute value
0x83E	0x0E	0x02	Unknown/Reserved
0x840	0x10	0x04	Size of attribute value
0x844	0x14	0x04	Unknown
0x848	0x18	0x04	Attribute type
0x84C	0x1C	0x04	Unknown

Table 16
Structure of the data run.

E Offset	R offset	Length	Description
0x8F8	0x00	0x04	Size of data run record
0x8FC	0x04	0x02	Size of header
0x8FE	0x06	0x02	Offset to value
0x900	0x08	0x02	Unknown
0x902	0x0A	0x02	Size of data run
0x904	0x0C	0x04	Offset to data run
0x908	0x10	0x08	Unknown (rel entry block pos)
0x910	0x18	0x08	Number of entry blocks
0x918	0x20	0x08	Entry block start
0x920	0x28	0x08	Unknown

0x8 bytes in the Data Run attribute value, and found that the Data Run used 0x04 (same as 1 cluster or 128 sectors) entry blocks. At entry block offset 0x918 we found the entry block where the data run started, here 0xA88. We used Table 16 to interpret the data run. To recover the contents of the file, we used the command shown in Fig. 2. The content of this file was exactly the same as the content of the file SmallFile.txt when opened in File Explorer.

```
dd if=mount/ewf1 bs=512 skip=$(( 0x800 + ((0xA88 * 16384)
    ↪ /512) )) count=128 of=SmallFile.txt
```

Listing 2: Command to extract a file.

Traces of deleted files

When we created the image used as an example in this paper, we also deleted a file from the root directory. The deleted file had the name DeletedFile.txt. We found one child attribute within the entry block of the root directory, as shown in the hex dump in Fig. 17.

We did not find any pointer to this record in the pointer area at entry block offset 0x367C in the hex dump shown in Fig. 10. What is interesting is that there were two pointer fields before the first pointer. These fields have the same value as the first pointer, here 0x20 (LE). This is an indication that there had been more records in this node. We may even be able to recover the content by searching for patterns of data runs that are not part of any active records. With active records, we mean a record that has a valid pointer in the pointer area.

In the hex dump found in Fig. 18 we first saw a normal FNA directory attribute which was 0x78 (LE) bytes in size. However, directly after this we saw a timestamp, which we assumed was from the Last Access timestamp of a FNA file attribute. We

```
000010e0: 5800 0000 1000 1800 0400 2800 3000 0000  X.........(.0...
000010f0: 2000 0080 0000 0000 0006 0000 0000 0000   ...............
00001100: 0200 0000 0000 0000 0000 0000 0000 0000  ................
00001110: 0c00 1e00 4400 6500 6c00 6500 7400 6500  ....D.e.l.e.t.e.
00001120: 6400 4600 6900 6c00 6500 2e00 7400 7800  d.F.i.l.e...t.x.
00001130: 7400 6f00 6e00 ffff 0000 0000 0000 0000  t.o.n...........
```

Fig. 17. Traces of a deleted file.

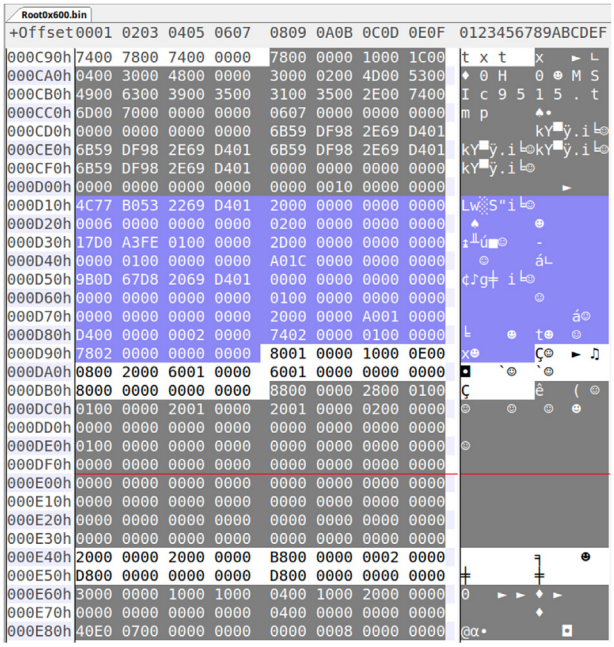

Fig. 18. Traces of a deleted file part II.

compared this with the previous parsing of the FNA file attribute, and found a data run at entry block offset 0xE60, which can be interpreted using Table 17.

Here we see that the data run consisted of 4 entry blocks (1 cluster) and it started at entry block 0x7E040 (LE). We recovered this file by extracting it. When we opened it, we recognized the content from the file we deleted. To extract the file we used the same command syntax as shown in Listing 2, but we changed the entry block and the name of the output file.

Allocation tables

In section 4.6 we have shown how to interpret the structures for files, including how we can find the file content. However, it is important to know if the file was allocated (active) or unallocated (not active). We show this using the data run we found in Fig. 18. It should include the entry blocks 0x7E040, 0x7E041, 0x7E042 and 0x7E043, where each entry block is 16 KiB in size. Since this is an ordinary file (FNA File) we will use the $Allocator_Med system file, where each bit represents 64 KiB.

When interpreting $Tree_Control in section 4.5 we found $Allocator_Med at entry block 0x20. Therefore, we just show the hex dump from the correct bitmap record in Fig. 19 from this entry block. We can interpret this by using Table 18.

Table 17
Structure of the data run.

E Offset	R offset	Length	Description
0xE60	0x00	0x04	Size of data run record
0xE64	0x04	0x02	Size of header
0xE66	0x06	0x02	Offset to value
0xE68	0x08	0x02	Unknown
0xE6A	0x0A	0x02	Size of data run
0xE6C	0x0C	0x04	Offset to data run
0xE70	0x10	0x08	Unknown (rel entry block pos)
0xE78	0x18	0x08	Number of entry blocks
0xE80	0x20	0x08	Entry block start
0xE88	0x28	0x08	Unknown

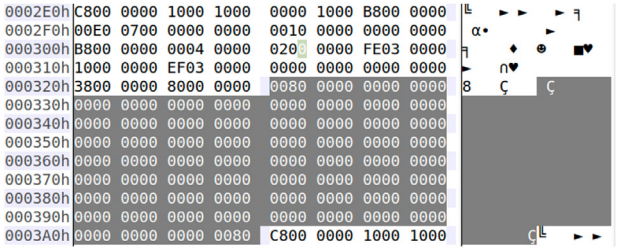

Fig. 19. Hex dump of a bitmap record.

Table 18
Structure of a bitmap record.

E Offset	R offset	Length	Description
0x2E0	0x00	0x04	Size of bitmap record
0x2E4	0x04	0x02	Record header length
0x2E6	0x06	0x02	Unknown
0x2E8	0x08	0x04	Unknown
0x2EC	0x0C	0x04	Size of Record value
0x2F0	0x10	0x08	Entry Block start
0x2F8	0x18	0x08	Number of Entry blocks
0x300	0x20	0x04	Size of Record value
0x304	0x24	0x04	Total number of allocation blocks
0x308	0x28	0x04	Unknown (observed 0x02)
0x30C	0x2C	0x04	Free blocks (bit = 0)
0x310	0x30	0x04	Bit offset to next free block (bit = 0)
0x314	0x34	0x04	Number of free blocks after first free (bit = 0)
0x318	0x38	0x08	Unknown
0x320	0x40	0x04	Offset to bitmap start relative to value start
0x324	0x44	0x04	Bytes in bitmap
0x328	0x48	0x80	The bitmap

It is the correct bitmap record because it starts from entry block 0x7E000 and it contains 0x1000 entry blocks. Our entry blocks must fit in here since the first starts at 0x7E040 and is 4 entry blocks in size. To find the correct bit we use the formula:

$$\text{AB bit offset} = \frac{\text{EB number} - \text{start EB}}{4} \tag{4}$$

$$\text{AB bit offset} = \frac{0x7E040 - 0x7E000}{4} = 0x10 = 16 \tag{5}$$

If we count from the top of the bitmap area (dark background) in Fig. 19 we will see that the 16^{th} bit has the value 0. Remember the first bit is the 0^{th} bit. This bit, the 16^{th} bit, represents the allocation block (AB). Each allocation block contains 4 entry blocks (EB). Therefore, this allocation block represents the entry blocks 0x7E040, 0x7E041, 0x7E042 and 0x7E043. Since the allocation bit is 0 these entry blocks are unallocated, which means the file is deleted.

In another ReFS volume, refs-v1_2–3.E01 available at Mendeley (Nordvik, 2019), we found the complete FNA file from a deleted file, the datarun part of the FNA file attribute was zeroed, and the corresponding bit in the allocator was zeroed. This can be observed in the screen shot in Fig. 20. All 0x30 bytes from offset 0x11D0 that should have been there, are zeroed out. Before we deleted this file the data run content was starting in entry block 0x17C and it consisted of 4 entry blocks. In the screen dump in Fig. 21 we see the bitmap where entry block 0x17C (380 in decimal) belongs.

$$\text{AB bit offset} = 380 * \frac{16384}{65536} = 380 * \frac{1}{4} = 95 \tag{6}$$

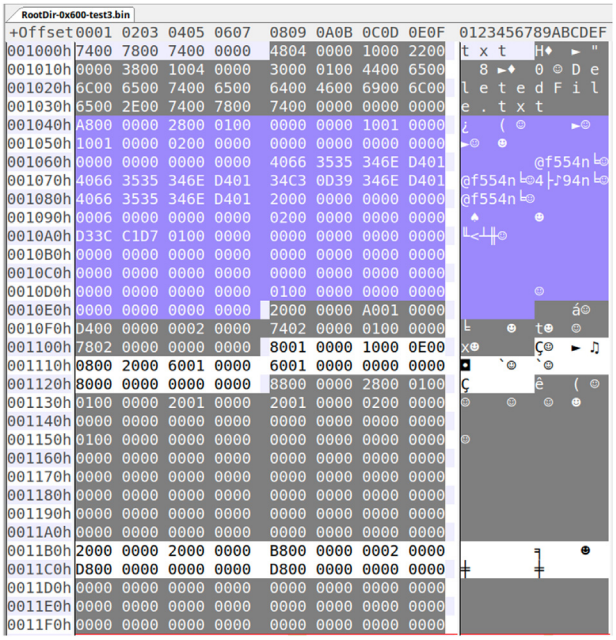

Fig. 20. Traces of a deleted file (FNA file complete).

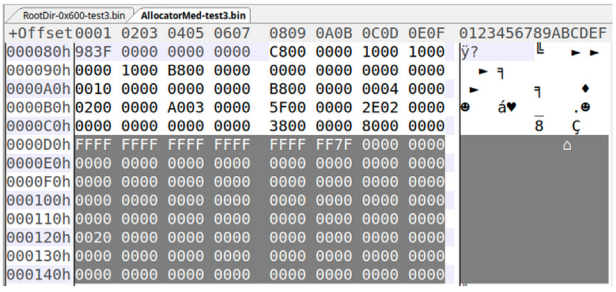

Fig. 21. Traces of a deleted file (Allocator Med (test3)).

$$\text{AB byte offset} = \frac{95}{8} = 11 \tag{7}$$

$$\text{Remainder bits} = 95 \mod 8 = 7 \tag{8}$$

If we count 11 bytes from byte zero at 0xD0 we will find the value 0x7F. Converted to bits this is 0111 1111. We count from the right to left, where the least significant bit is 0 and the most significant bit is 7. In this case we are looking for the most significant bit, and its value is 0. This means that this allocation block is not allocated. We show the content of the entry block 0x17c in Fig. 22.

We assess if our entry block is in the range of start entry block and the number of entry blocks in the allocator record. If it is not, we need to locate one allocator record that has the correct range.

$$\text{AB bit offset} = (\text{EB number} - \text{start EB}) * \frac{\text{EB block size}}{\text{AB block size}} \tag{9}$$

The first byte in the bitmap is byte number 0. And we can find the byte offset by dividing by 8.

$$\text{AB byte offset} = \frac{\text{AB bit offset}}{8} \tag{10}$$

We use modulus to find the remainder bits in order to find the bit we are looking for. A value of 1 means the corresponding allocation block is allocated while 0 means it is not allocated.

```
osf@osf-VirtualBox:~$ dd if=mount3/ewf1 bs=512 skip=$(( 0x10800 + (0x17c *0x20)
)) count=1 | xxd
00000000: 5468 6973 2066 696c 6520 6861 7320 6265  This file has be
00000010: 656e 2064 656c 6574 6564 0000 0000 0000  en deleted......
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Fig. 22. Content of a deleted file (Allocator Med (test3)).

$$\text{AB byte bit pos} = \text{AB bit offset} \mod 8 \tag{11}$$

We found three different allocation tables, one for small metadata structures, one medium one which we have shown for regular files, and one large one used for large files. As previously described they are named $Allocator_Lrg (for large files), $Allocator_Med (for files) and $Allocator_Sml (system files and metadata).

Interpretation of the other structures is performed in a similar manner, but it is important to use the correct values for allocation block size.

The $Object entry block

Entry block 0x240 was used by the $Object system file. It contains records showing the parent child relationship. We show this system file in Fig. 23.

In this entry block we had 6 records, and we show the record pointed to by the last pointer in the pointers area. This record started from offset 0x1D0, and was 0x30 in size. We saw that the child node id 703, found in the entry block offset 0x1F8, had the parent node id 702, found in the entry block 0x1E8. This record did not have any attribute type. The other records showed that the node ids 520, 701, 702, 704, 705, and 706 had the root directory (0x600) as parent.

Results - ReFS v3.2

We have described most of the structures for ReFS v1.2. In ReFS v3.2 the structures are almost identical. The forensic container file refs-v3_2.E01, available at Mendeley (Nordvik, 2019), allows the readers to follow our examples. ReFS v3.2 uses a cluster size of 8 sectors (4096 bytes) as standard. The size of the metadata entry blocks are still 16 KiB, which means ReFS v3.2 uses 4 clusters to define one entry block as standard. In records pointing to an entry

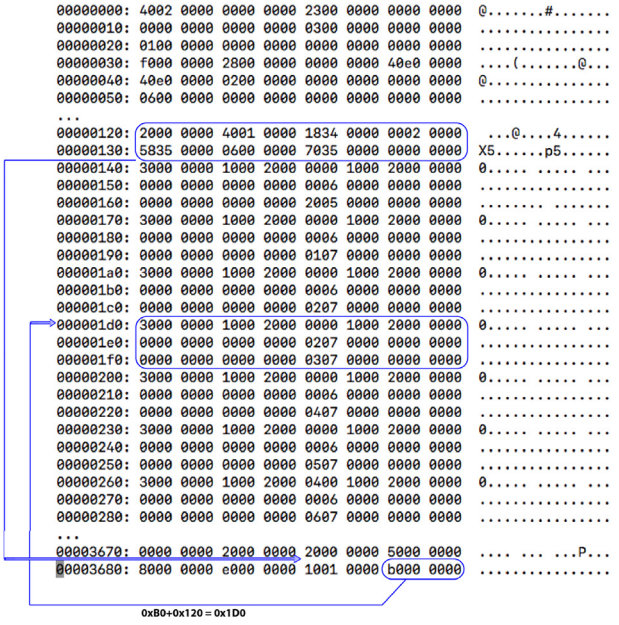

Fig. 23. $Object system file.

block, we now have 4 cluster pointers. The first points to the start of the entry block. This entry block consists of all the four clusters. The main difference in the volume boot record is the major and minor version fields.

Superblock

Finding the superblock can be done by using the command shown in Listing 3, assuming a cluster size of 8 sectors (4 KiB). The disk volume we use to explain the structures was using a GPT partition system, and the ReFS partition started at sector 0x10800.

$$\text{EB sector} = \text{VBR sector} + \frac{\text{EB number*Cluster size}}{\text{Sector size}} \quad (12)$$

```
dd if=mount/ewf1 bs=512 skip=$(( 0x10800 + ((0x1E * 4096)
    ↪ /512) )) count=1 | xxd
```

Listing 3: Command to show the superblock at 0x1E (cluster size 4KiB).

We can use Table 3 for the ReFS v3.2 superblock also, but we must add 0x20 to each offset. The 0x20 bytes are at the beginning of the entry block, which makes each entry block header 0x50 in size. The entry block descriptor starts with the magic signature 0x53555042 (BE) or SUPB in ASCII. This is why we have called the 0x1E block the superblock. We found the entry block pointer to the $Tree_Control checkpoint at offset 0xC0, which was 0x2874.

Checkpoint

In the previous section we were able to find the pointer to the $Tree_Control entry block, which is a checkpoint. We have 0x20 additional bytes at the top of this entry block. The magic signature is 0x43484B50 (BE) or CHKP in ASCII.

Therefore, we had to add 0x20 to the previously presented Table 4. We also had to add an additional 0x18 bytes to find the amount of additional records. In Table 19 we have shown the new offsets.

The main difference in the checkpoint entry block was that the record size has increased, and there are more records. In ReFS v1.2 there were 6 additional records, but in ReFS v3.2 we observed 13 additional records.

In Fig. 24 we see the first additional record, which included a pointer to $Object_Tree.

In Table 20 we show how the checkpoint record should be interpreted. When it comes to the entry blocks the second WORD has a special meaning. For example in the hex dump in Fig. 24 we saw the value 0xB83D**01** (BE), which is 0x**01**3DB8 (LE). We interpreted this as $0x013DB8(LE) - 0x10000(LE) = 0x3DB8(LE)$, because the second word starts with 01. We have observed when working with ReFS volumes that the second WORD could start with 01, 04, 06 or 0C. There could be other values. This value has an impact on where to find the physical location of the entry block. For example if the Entry block number value is 0x580104 (BE), which has the value 04 in the first byte of the second WORD, then the entry block location will be found in entry block $0x040158(LE) - 0x34000(LE) = 0xC158(LE)$. Another example is the entry block value 0x128006 (BE), where the physical location entry block was in $0x068012(LE) - 0x44000(LE) = 0x24012(LE)$. We have created a structure table in Table 21, which shows the values to substract from the Entry Block cluster pointers based on the value of the MSB (not counting zero bytes).

Equation (13) shows a formula that can be used to find the subtract value if the MSB is larger than 1.

$$\text{Subtract value} = 0x10000 + (\text{MSB}*0x8000) + (((\text{MSB}+1)\bmod 2)*0x4000) \quad (13)$$

In this specific record we had cluster pointers to 0x3DB8, 0x3DB9, 0x3DBA and 0x3DBB. Since a record includes pointers to four clusters, this mean that we need four clusters to get one metadata entry block of 16 KiB, or 32 sectors. The first cluster pointed to the start of the $Object_Tree entry block.

Table 21 is not complete, but computed using Equation (13).

Table 19
Structure of the $Tree_Control checkpoint entry block.

E Offset	R offset	Length	Description
0x50	0x00	0x8	Unknown
0x58	0x08	0x04	Offset to first record
0x5C	0x0C	0x04	Size of a record
0x60	0x10	0x10	Unknown
0x70	0x20	0x20	Unknown
0x90	0x40	0x04	Amount of additional records
0x94	0x44	Var	4 byte offset to each records

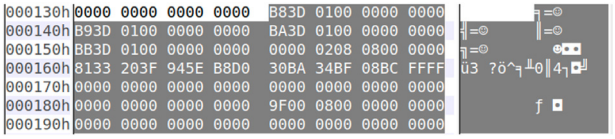

Fig. 24. ReFS v3.2 Checkpoint Record ($Tree_Control).

Table 20
Structure of the ReFS v3.2 checkpoint record.

E Offset	R offset	Length	Description
0x138	0x00	0x8	First cluster of entry block
0x140	0x08	0x8	Second cluster of entry block
0x148	0x10	0x8	Third cluster of entry block
0x150	0x18	0x8	Fourth cluster of entry block
0x158	0x20	0x8	Unknown
0x160	0x28	0x8	Checksum of entry block
0x168	0x30	0x8	Unknown
0x170	0x38	0x30	Unknown

Object tree

The $Object_Tree is a B+ tree, and the difference from ReFS v1.2 is that an additional 0x20 is added to the top of the entry block descriptor. This block starts with the magic signature 0x4D53422B or MSB+ in ASCII. All entry blocks that include a B+ tree will have this signature.

Table 21
Conversion table for cluster pointer to Entry Block.

MSB	Subtract value
1	0x10000
2	0x24000
3	0x28000
4	0x34000
5	0x38000
6	0x44000
7	0x48000
8	0x54000
9	0x58000
A	0x64000
B	0x68000
C	0x74000
D	0x78000
E	0x84000
F	0x88000
E2	0x724000
E3	0x728000

As for other B+ trees we have the node descriptor, the node header, records and a pointer area. The new part in the $Object_-Tree is that the records are larger.

In the hex dump in Fig. 25 we show the record that points to the 0x600 root directory. We used Table 22 to interpret this record. The node id is 0x600 (LE), and the first entry block was found in entry block offset 0xA50 is 0x3D6C (LE). Notice that we subtracted 0x10000 because of the number 01 in the second WORD.

Root directory

In the root directory we will find the ordinary entry block descriptor, node descriptor, node header, records (attributes) and the pointer area. We interpret most of the entry blocks just as we would do for ReFS v1.2. We show a FNA directory attribute in Fig. 26.

This folder has the name TestFolder and the node id 0x702. It was created on 19th of October 2018 at 05:46:45 UTC. The rest of the timestamps had the same time, 19th of October at 05:47:30 UTC.

FNA file

We have observed that the FNA file attribute might not always include the data run as an internal node in the attribute value part. In Fig. 27 we show the FNA file attribute of a small text file from the entry block 0x3D6C.

The data run attribute was missing, and a record that pointed to four other entry block clusters were found. We saw that they were in sequence and therefore formed one entry block. In order to find more information regarding the data runs we needed to go to this entry block, in this case 0x3D70.

Browsing through 0x3D70 can give the impression that this is an entry block that includes files and folders, but it is not. There is only one main record/attribute in use. The rest are remnants from previous states of the B + Tree. The reason for these remnants are that ReFS uses Copy-On-Write (COW), which means it reads the allocated content of the entry block into memory, make changes, and writes the new entry block to a new location on disk (Wikipedia. Copy-on-write, 2019), leaving the previous one unchanged and unallocated. New ReFS operations can allocate previously unallocated entries and overwrite parts or all of the previous information.

This main record/attribute had a node structure inside it, as shown in Fig. 28. From entry block offset 0x20 we can see this entry block contains information about the entry block 0x3D70.

From the data run header we saw in entry block offset 0x98 that this was a data attribute (value 0x80 (LE)). We do not know much about the data run node descriptor, except the locations for physical and logical sizes as shown in Table 23. The data run node header starting at entry block offset 0x128 gave us the offset to the data run pointer area from entry block offset 0x138. The pointer offsets were relative to the start of the data run node header.

Using the pointers from the pointer area we found that record 1 started at entry block offset 0x150, and that record 2 started at offset 0x168. These two records must be interpreted together, each by using Table 24. Even though there were two records, this does not necessarily mean the file is fragmented. The first record had the start cluster value 0x52232(*LE*), and by using the subtraction Table 21 we saw that this data run record started at cluster 0x52232 − 0x38000 = 0x1A232. The second data run record in this example had the start cluster value 0x18000 (LE), and started at cluster 0x18000 − 0x10000 = 0x8000. In this case, there is a gap between the two data run records. This could mean the file is

Table 22
Structure of the ReFS v3.2 $Object_Tree record.

E Offset	R offset	Length	Description
0xA10	0x00	0x4	Record size
0xA14	0x04	0x06	Unknown
0xA1A	0x0A	0x2	Record header size
0xA1C	0x0C	0x2	Record value size
0xA1E	0x0E	0x2	Unknown
0xA20	0x10	0x8	Unknown
0xA28	0x18	0x4	Node ID
0xA2C	0x1C	0x4	Unknown
0xA30	0x20	0x8	Unknown
0xA38	0x28	0x4	Relative offset to first entry block
0xA3C	0x2C	0x4	Size of entry block section
0xA40	0x30	0x10	Unknown
0xA50	0x40	0x8	1st Entry block cluster number
0xA58	0x48	0x8	2nd Entry block cluster number
0xA60	0x50	0x8	3rd Entry block cluster number
0xA68	0x58	0x8	4th Entry block cluster number
0xA70	0x60	0x8	Unknown
0x178	0x68	0x8	Checksum of entry block
0x80	0x70	var	Unknown

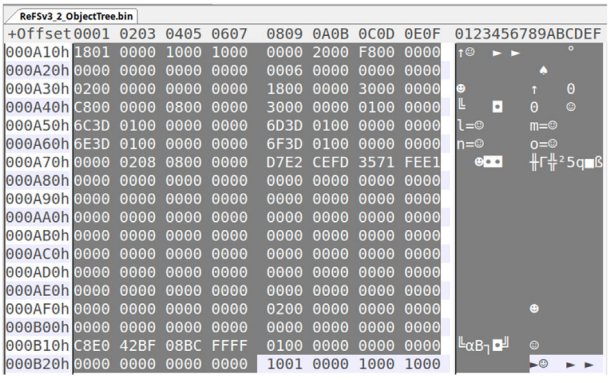

Fig. 25. ReFS v3.2 Object tree record.

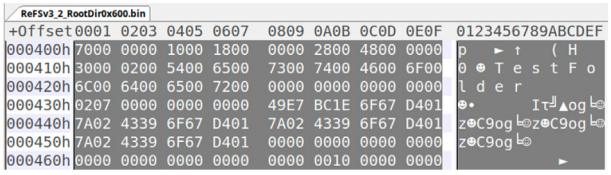

Fig. 26. ReFS v3.2 FNA directory attribute.

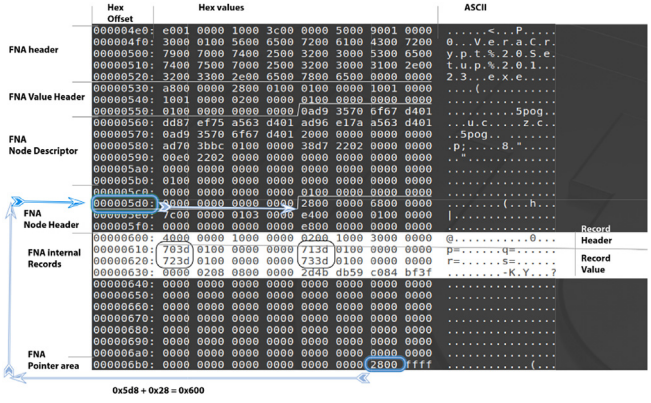

Fig. 27. ReFS v3.2 FNA file attribute.

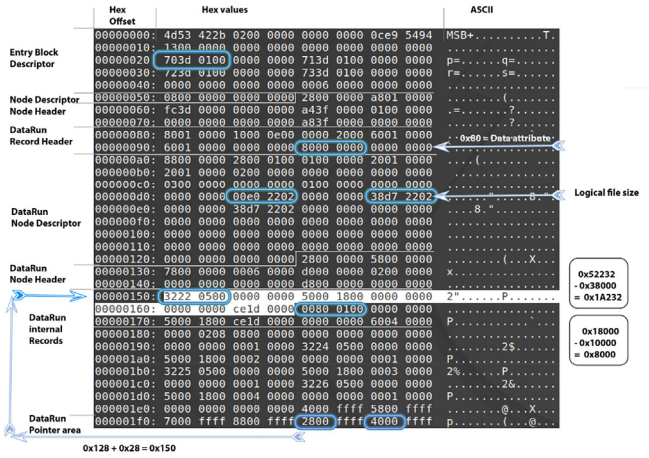

Fig. 28. ReFS v3.2 FNA file DataRun attribute.

Table 23
Structure of ReFS v3.2 DataRun node descriptor.

E Offset	R offset	Length	Description
0xA0	0x00	0x4	Record size
0xA4	0x04	0x30	Unknown
0xD4	0x34	0x8	Physical size of file
0xDC	0x3C	0x8	Logical Size
0xE4	0x44	0x8	Logical Size?

Table 24
Structure of ReFS v3.2 DataRun internal record.

E Offset	R offset	Length	Description
0x150	0x00	0x8	Start Cluster
0x158	0x08	0x2	Unknown (seen 0x50)
0x15A	0x0A	0x2	Size of record? (seen 0x18)
0x15C	0x0C	0x4	Virtual start Cluster Number
0x160	0x10	0x4	Unknown (zeros)
0x164	0x14	0x4	Number of clusters in data run

fragmented. It is important to only include the number of allocated data run records at the offsets given by the offset pointers in the data run pointers area. Our hypothesis was that the data run fragments could be located in any order on disk, however in this example this can not be true. When we extracted the file using the two fragments we got another hash value than the hash of the original file. Therefore, we falsified this hypothesis. A visualization of the falsified hypothesis is shown in Fig. 29.

The allocated clusters are represented by grey boxes in the figure. The first two grey boxes, in this example, represents 0x460 clusters. The last 3 grey boxes represents 0x1DCE clusters. In this case, the first data run started after the second data run.

A new hypothesis was constructed: The complete first data run fragment must be located before the next, etc, but not necessarily contiguous. If the next fragment is located before the start of the previous, then this next fragment must be a contiguous fragment, continuing directly after the previous fragment. However, if the next fragment is after the end of the previous fragment, then the cluster start of the next fragment should be interpreted normally. This is illustrated in Fig. 30.

Extracting the data

Using the information from the data run records, and the logical size from the data run node descriptor, we can extract the file using

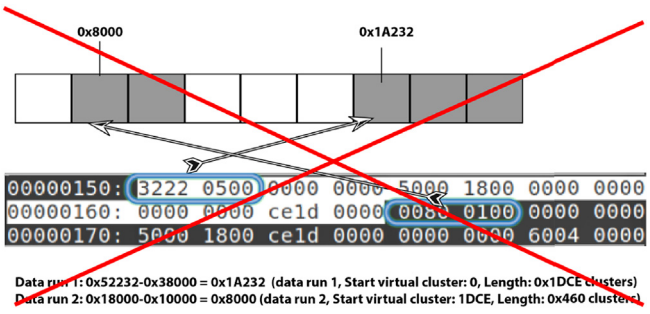

Fig. 29. ReFS v3.2 FNA file with a fragmented DataRun.

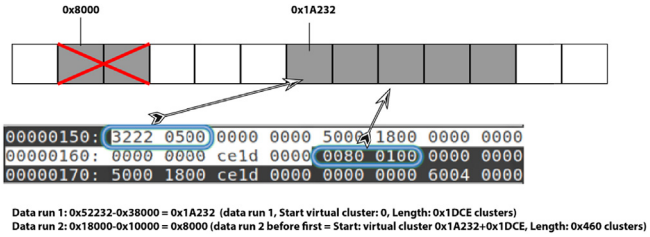

Fig. 30. ReFS v3.2 FNA file with a not fragmented DataRun.

simple **dd** commands. If we have hashed the file while the volume was mounted in Windows, then we only have the hash of the logical size of the file. Therefore, if we want to verify that we have the same file, we need to extract the logical size of the file. In Fig. 4 we show how the extraction of the file can be done using simple **dd** commands. Here we show how we extracted using our first falsified hypothesis.

```
dd if=mount/ewf1 bs=512 skip=$(( 0x10800 + ( 0x1A232 *
    ↪ (4096/512)) )) count=$(( 0x1DCE * (4096/512) )) of=
    ↪ part1.bin
dd if=mount/ewf1 bs=512 skip=$(( 0x10800 + ( 0x8000 *
    ↪ (4096/512)) )) count=$(( 0x460 * (4096/512) )) of=
    ↪ part2.bin
cat part1.bin part2.bin > file-physical.bin
dd if=file-physical.bin bs=1 skip=0 count=$(( 0x222D738 ))
    ↪ of=VeraCryptSetup1.23.exe
```

Listing 4: Extracting the file from the image using first hypothesis.

```
dd if=mount/ewf1 bs=512 skip=$(( 0x10800 + ( 0x1A232 *
    ↪ (4096/512)) )) count=$(( (1DCE+460) * (4096/512) ))
    ↪ of=file-physical.bin
dd if=file-physical.bin bs=1 skip=0 count=$(( 0x222D738 ))
    ↪ of=VeraCryptSetup1.23.exe
```

Listing 5: Extracting the file from the image using the second hypothesis.

In Listing 5 we show the commands we used to correctly extract the file used in this example. The hash of the output file was the same as the original file. However, when a file is fragmented, and the next fragment is after the previous, we will need a similar approach as in Listing 4.

Checksums

When we parsed the superblock and checkpoint, and the directory records within $Object_Tree, a MSB + node, we found fields that we assumed were 64 bit checksums. We also found 64 bit fields that could be checksums within some of the attributes within MSB + nodes. One of our hypotheses was that the checksum should be saved within some of the attribute in the MSB+, especially the FNA File attribute or the FNA Directory attribute, but we did not find any checksums in the FNA Directory attribute. In the

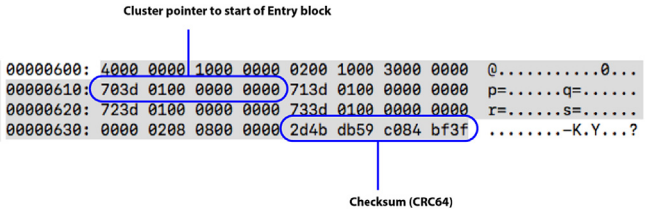

Fig. 31. ReFS v3.2 FNA file data attribute pointer.

```
Runes-iMac:ReFS runenordvik$ ./verifychecksum /User
s/runenordvik/Downloads/ReFS/mount/ewf1 193408 32
Calling function verfifyChecksum
Sector to parse: 193408
Starting on sector: 193408
Sectors in entry block: 32
Checksum: 3fbf84c059db4b2d
```

Fig. 32. ReFS v3.2 Verify Crc64 checksum.

FNA File attribute, we found a 64 bit checksum within its internal data attribute pointer at relative byte 0x38. This attribute points to a new entry block, and it is the checksum of this entry block that is saved in the checksum field. In Fig. 27 highlighted in white it is the last 8 bytes that is the checksum. In Fig. 31 we only show the part that we previously had highlighted.

We found that attribute records do not have checksums for their attribute content, but a record that points to another entry block does include a 64 bit checksum. In order to compute the checksum we need the sector to where the block starts, and the size of the block. For metadata the entry block is 16 KiB (32 sectors). In Fig. 32 we verify the checksum found in Fig. 31. The checksum shown in the hexdump in Fig. 31 must be read as little endian in order to give the same output as the program verifychecksum. This program use the same CRC64 algorithm as found in the NTFS-3g driver project (Tuxera, 2017).

Another hypothesis was that 64 bit checksums were implemented in the MSB + node $Object_Tree records. We found 64 bit fields in this MSB + node records that are 64 bit checksums of the entry blocks they point to.

Experiment - manipulation of metadata

We performed experiments to see if it was possible to use existing metadata tools in order to manipulate timestamps in ReFS. We tested using the SetMACE v1.0.0.16 (Schicht, 2014). However, the tool failed because it did not find any $MFT.

We performed another experiment where we manually manipulated a timestamp using a hex editor. We changed one byte of the file creation timestamp of the file Manipulated_Time.txt in the root directory. The result of this manipulation was that the file system could not be repaired or mounted.

We also tested by changing the metadata checksum value for the corresponding record in the $Object_Tree, and the checksum that corresponds in the $Tree_control for the record pointing to $Object_Tree. The changes of metadata were performed by using a hex editor on a physical hard disk with a ReFS v3.2 volume[2]. The result was the same, both for ReFS 3.2 and/or ReFS 3.4, the file system was not mountable.

These experiments show that manipulation of the metadata is hard without the user detecting it. When we manually changed back to original values, the file system was mountable again.

This does not mean it is impossible to change metadata. It means we have not documented all the integrity features involved in protecting metadata.

Discussion

Data attribute

The use of multiple data run records in ReFS v3.2, even for the same contiguous data stream, will allow very large files. A file can have a maximum size of 35 petabytes (Microsoft, 2018c), and we assume this limit is because of the maximum size of the ReFS volume. In the data attribute node record we find logical and physical file sizes, where the physical size uses the boundaries of clusters. Because of the maximum file size these fields most probably are 8 bytes, but only 7 of the bytes are needed. This is because $2^{55} = 32PiB < 35PiB < 2^{56} = 64PiB$.

Another reason for multiple data runs for contiguous files might be because of sharing blocks between files. If we copy a file it will use the same data runs. If we change one of the files, both files might need to split up in different data runs in order to share only parts of the blocks. This sharing of blocks between files might make carving more difficult without knowing the corresponding data runs. Two files could for instance share the first two clusters, but not the third cluster, then the fourth might be shared, etc.

Allocation unit

Using 128 sectors for small files is a waste of space, however it is fast because the system does not need to reallocate a lot of clusters frequently. According to Vogels (1999) most deleted files are very small (only 4 percent of deleted files are more than 40 KiB in Windows NT). The exact percentage of small files could have changed since Volgels (Vogels, 1999) paper from 1999, but we still assume that most deleted files are small. This is also important for digital forensics, because most of the files will only use one cluster. One of the problems with traditional carving techniques are fragmented data runs. However, any user file less than 64 KiB will only occupy 1 cluster in ReFS v1.2. Typically user documents (text documents and office documents) are often less that 64 KiB, and therefore carving of these will be very easy. We do not have this advantage in ReFS v3.2, because the standard cluster size is now 4 KiB. However, it is possible to format ReFS v3.2 with 64 KiB cluster size.

Finding traces of deleted files

When shift delete is used to delete a file it will not be saved in the $Recycle.bin (Microsoft, 2019). $Recycle.bin does not really include deleted files. We focused on traces of real deleted files. In one of our experiments we observed that a record with the FNA attribute was partly overwritten, and we saw the file child attribute structure from the deleted file. The child attribute, if found, will contain the file name and the parent node id. However since the entry block uses node structures organized as B+ trees we assume that the different records containing attributes would easily be overwritten when the B+ tree is being balanced. As we have observed, it is possible to find previous attributes containing data runs. The same observations of remnants after B+ tree balancing have been seen on database systems using B+ trees, as described by Stahlberg et al. (2007). It is possible to perform a signature search for data runs, for example searching ReFS v1.2 for the pattern 0x3000000010001000 and including 30 bytes maximum. When a complete FNA file attribute from a deleted file is present, then the

[2] During our multiple test, we observed that our mounted ReFS v.3.2 volume was automatically updated to ReFS v 3.4 in our final tests.

data run is normally zeroed. This means we can not easily find the datarun, but we will see all other metadata for this deleted file. However, we found complete FNA file attribute in previous used entry blocks. For ReFS v3.2 we can use the pattern 0x4000000010000000 to find the FNA File data attribute pointer, and from there find the Entry Block MSB + node that includes the data runs.

ReFS recoverability

We have observed that attributes no longer in use are still available within entry blocks. When a user deletes a file, metadata remnants might still be found in the same entry block or in a previously used entry block. This is because of the Copy-On-Write (COW) functionality of ReFS. Previous attributes in unallocated entry blocks make it possible to use this metadata to recover files, but also to connect previous metadata to previous versions of the file content. This is because it is not only the metadata entry blocks that use COW, but also the data entry blocks. This means if the user changes a file, the changes are saved to another location, leaving the previous version in unallocated space. We have only tested this for files that use a single data block, and we could find previous versions of the file in previous allocated area. Finding remnants on SSD disks is not applicable because of the use of garbage collection.

The allocation blocks used by the file are not wiped when a file is deleted, making metadata carving a possible technique to recover the file content. For instance we could search for 0x30000100 (BE) in order to search for FNA files, or we could search for 0x4000000010000000 (BE) in order to find data attribute pointers. We also observed reused metadata entry blocks containing most of the attributes from its previous use. This is good for criminal investigationand recoverability, but bad for privacy.

Recovery of ReFS volumes

If the main VBR gets corrupted we can find the backup VBR, as can be seen in Listing 6. The sectors in the volume can be found in the corrupted VBR, assuming the sectors in volume field is not corrupted or modified. If the main VBR is overwritten and no MBR or GPT partition entries are found, we can search for the backup VBR and compute where the main one must have been located. The VBR contains the string ReFS from byte offset 3. Therefore, using sigfind, a Sleuthkit tool, to search for the hex value 0x52654653 (hex value of ReFS) from offset 3 in each sector would find any VBR instances, we assume the FSName is not changed by the user. Then we can recover the bytes of this volume and copy the backup VBR to the position where the main VBR should have been located. Other alternatives, without changing the evidence, are to manually parse the volume in a hex viewer or develop a tool that can use the backup VBR as input.

```
dd if=mount/ewf1 bs=512 skip=$((VBRSector +
    ↪ SectorsInVolume - 1)) count=1 | xxd
```

Listing 6: Command to show the backup VBR.

We described in section 4.4 that there were three superblocks, one found at 0x1E, one found in the third last entry block, and the third in the second last entry block. This means that if the main superblock gets corrupted, we can compute the location of the second backup superblock using the command shown in Listing 7. Using multiple backup superblocks is well known from Ext2, 3 or 4 [5, p.405], but the use of superblocks is new for Windows file systems. ReFS use both VBR and superblocks.

```
dd if=mount/ewf1 bs=512 skip=$((VBRSector +
    ↪ SectorsInVolume - ((0x3 * EntryBlockSize) /
    ↪ SectorSize))) count=1 | xxd
```

Listing 7: Command to show the superblock backup.

We tested this by formatting an existing ReFS v3.2 volume with FAT32, see the forensic container refs-v3_2-3-fat32.E01 available at Mendeley (Nordvik, 2019). The VBR and the first superblock were not available. We used the backup superblock to identify the entry block for the checkpoint. This enabled recovery of the data. If we have no idea where to find the superblock, we can just search for the magic string SUPB.

Knowing the structures of ReFS enables the investigator to recover the volume, and this might be successful even if the user has reformatted the volume using another file system as long as the structures and data content are not overwritten.

Data runs

We found that the next data run in a FNA file with multiple data runs could point to a cluster before the current data run. However, the real cluster start used was the next cluster after the clusters used by the previous data run. This indicates that clusters used by later data runs for a FNA file, will either be contiguous or start on a cluster later in the file system.

The finding of previous FNA file attributes, with their included data runs, will allow the digital forensic investigator to recover previously allocated files.

Carving and metadata

We have described that metadata checksums are on by default, and integrity streams for data can be enabled on files, directories and volumes. If integrity streams for data are enabled, it should be possible to compute the checksum of an integrity stream enabled file, using the known crc64 algorithm, and then use the output checksum to search for the associated metadata. Finding both the file and the metadata will increase the evidential value of the artifact.

We have also found signatures for some attributes, which could be used for metadata carving. Finding FNA file attributes could be used to locate the metadata and the data runs, and this way metadata carving can be used as a method for recovery of files, even for previous versions of the files (assuming previously used clusters are not overwritten). We found previous versions of the B+ tree nodes, which included previous versions of attributes.

Conclusion and further work

The first objective for our study was how digital forensic investigators can verify the reliability of ReFS support in existing Digital Forensic tools. The result of our reverse engineering allows digital forensic analysis of ReFS v1.2 and v3.2. Investigators now have the knowledge to verify digital forensic tools that claim to support ReFS. Microsoft is still implementing more features in ReFS, and ReFS v3.2 is the latest version we used for our study, which currently is only supported by one Digital forensic tool. We found that ReFS was updated to version 3.4 when we performed our final tests on ReFS. This version information was found in the corresponding VBR of volumes we mounted to Windows 10, but it did not change the structures we have described in this paper. A complete list of ReFS versions can be found on wikipedia (Wikipedia. Refs. 2019).

The second objective was to see how B+ tree balancing impacts the recoverability of files in ReFS. We have found metadata attributes in previously used B+ tree nodes, and this information can be used for recovery of previously deleted information. This is a security vulnerability and an opportunity for law enforcement. Therefore, metadata carving should be considered as a technique for recovery of files on ReFS.

The third research question was to see if digital forensic investigators can be confident in the accuracy of integrity protection mechanism of metadata structures. We found that manipulation of metadata could currently not be manipulated by known file system manipulation tools. We tested with one often used tool, **SetMace** (Schicht, 2014), and we performed one manual manipulation of a timestamp, and one manipulation where we updated checksums. These simple experiments show that the manipulation of metadata in ReFS was unsuccessful, and that a manual manipulation could leave the file system in an unrecoverable state. Our results increase the confidence of the metadata protection mechanism found in ReFS. Since the integrity protection algorithm is known, we may assume that others will be able to modify all elements of the protection mechanism. The latter will decrease our confidence.

The fourth research question was how ReFS will impact the recovery of files compared to NTFS. Because of remnants of FNA file attributes, the recovery of files could be possible. However, B+ tree balancing could also overwrite nodes and attributes making the recovery more difficult than in NTFS. Carving of data content could still be used, and because of the COW (Copy-On-Write) we can expect to find previous versions of file content.

Our contribution allows investigators to recover files manually, and to manually connect metadata structures to the recovered files if possible. We have also described how investigators can recover a ReFS volume that has been reformatted with FAT32.

This is the first major reverse engineering attempt of ReFS, and there is a need for further verification of our findings. Our findings can be used when implementing new digital forensic tools.

References

Apple, 2018. Apple File System Reference visited 2018-10-02. https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf.

Apple, 2018. Hfs Plus Volume Format. https://developer.apple.com/library/archive/technotes/tn/tn1150.html, visited 2018-10-29.

Ballenthin, W., 2018. The Microsoft Refs In-Memory Layout visited 2018-10-29. http://www.williballenthin.com/forensics/refs/memory/.

Ballenthin, W., 2018. The Microsoft Refs On-Disk Layout. http://www.williballenthin.com/forensics/refs/disk/, visited 2018-11-04.

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

Computing, Microsoft, 2018. A File System Recognition Checksum visited 2018-10-21. https://docs.microsoft.com/en-us/windows/desktop/fileio/computing-a-file-system-recognition-checksum.

Fairbanks, K.D., 2012. An analysis of ext4 for digital forensics. Digital Investigation. In: The Proceedings of the Twelfth Annual DFRWS Conference, vol. 9, pp. S118–S130.

Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12.

Garfinkel, S.L., 2010. Digital forensics research: the next 10 years. Digital Investigation. In: The Proceedings of the Tenth Annual DFRWS Conference, vol. 7, pp. S64–S73.

Georges, H., 2018. Resilient Filesystem. http://hdl.handle.net/11250/2502565.

Gudadhe, S., Deoghare, R., Dhade, P., 2015. Window refs file system: a study. Int. J. Adv. Res. Comput. Commun. Eng. 4.

Hamm, J., 2009. Extended Fat File System 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.

Head, A., 2015. Forensic Investigation of Microsoft's Resilient File System (Refs) visited 2018-10-29. https://www.resilientfilesystem.co.uk/.

Horsman, G., 2018. Framework for reliable experimental design (fred): a research framework to ensure the dependable interpretation of digital data for digital forensics. Comput. Secur. 73, 294–306.

Marshall, A.M., Paige, R., 2018. Requirements in digital forensics method definition: observations from a UK study. Digit. Invest. 27, 23–29.

Metz, J., 2013. Resilient File System (Refs) visited 2018-10-1. https://github.com/libyal/libfsrefs/blob/master/documentation/Resilient%20File%20System%20(ReFS).pdf.

Microsoft, 2018. Building the Next Generation File System for Windows: Refs visited 2018-10-29. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/.

Microsoft, 2018. Refs Integrity Streams visited 2018-11-04. https://docs.microsoft.com/en-us/windows-server/storage/refs/integrity-streams.

Microsoft, 2018. Resilient File System (Refs) Overview visited 2018-11-04. https://docs.microsoft.com/en-us/windows-server/storage/refs/refs-overview.

Microsoft, 2019. How to Bypass the Recycle Bin when Deleting Files visited 2019-02-18. https://support.microsoft.com/en-us/help/171051/how-to-bypass-the-recycle-bin-when-deleting-files.

Nordvik, R., 2019. Resilient Filesystem Forensic Containers. http://dx.doi.org/10.17632/9d7p6mbhpz.1.

Paragon Software, 2019. Refs for Linux by Paragon Software. https://www.paragon-software.com/business/refs-linux/, visited 2019-06-02.

Paragon Software, 2019. Refs for Windows by Paragon Software. https://www.paragon-software.com/business/refs-for-windows/, visited 2019-06-02.

Plum, J., Dewald, A., 2018. Forensic apfs file recovery. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. ARES, New York, NY, USA, pp. 47:1–47:10, 2018. ACM.

Popper, K.R., 1953. Science: Conjectures and Refutations visited 2018-10-29. http://www.nemenmanlab.org/~ilya/images/0/07/Popper-1953.pdf.

Scanlon, M., Aug 2016. Battling the digital forensic backlog through data deduplication. In: 2016 Sixth International Conference on Innovative Computing Technology (INTECH), pp. 10–14.

Schicht, J., 2014. Setmace. https://github.com/jschicht/SetMace visited 2019-01-29.

Stahlberg, P., Miklau, G., Levine, B.N., 2007. Threats to privacy in the forensic analysis of database systems. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07. ACM, New York, NY, USA, pp. 91–102.

Tuxera, 2017. Open Source: Ntfs-3g. Online; accessed 2019-06-04. https://www.tuxera.com/community/open-source-ntfs-3g/.

Unknown, 2019. Windows Refs Versions. https://gist.github.com/0xbadfca11/da0598e47dd643d933dc, visited 2019-06-01.

US-Supreme-Court, 1993. Daubert V. Merrell Dow Pharmaceuticals, inc., 509 u.S. 579. Online; accessed 2018-10-11. https://supreme.justia.com/cases/federal/us/509/579/.

Vandermeer, Y., Le-Khac, N., Carthy, J., Kechadi, M.T., 2018. Forensic Analysis of the Exfat Artefacts. CoRR, abs/1804.08653.

Vogels, W., Dec. 1999. File system usage in windows nt 4.0. SIGOPS Oper. Syst. Rev. 33 (5), 93–109.

Wikipedia, 2019. Copy-on-write. https://en.wikipedia.org/wiki/Copy-on-write, visited 2019-06-01.

Wikipedia. Refs, 2019. Online; accessed 2019-06-04. https://en.wikipedia.org/wiki/ReFS.

Winaero, 2018. How to Format Any Drive in Windows 8.1 with Refs. https://winaero.com/blog/how-to-format-any-drive-in-windows-8-1-with-refs/, visited 2018-10-21.