NTNU
Norwegian University of
Science and Technology

Resilient Filesystem

Henry Georges

Information Security
Submission date: May 2018
Supervisor: Stefan Axelsson, IIK
Co-supervisor: Rune Nordvik, Politihøgskolen

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

ReFS

Henry Georges

22-05-2018

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Norwegian University of Science and Technology, 2018

Supervisor 1: Professor Stefan Axelsson, NTNU
Supervisor 2: Assistant Professor Rune Nordvik, PHS

Acknowledgement

I would like to thank my supervisor, Assistant Professor Rune Nordvik, for his patience and engagement.

H.G.

Abstract

MICROSOFT developed a new file system, REFS. This Resilient FileSystem is intended to replace NTFS, hence the importance and usage of REFS should increase over the next few years. Although we have been able to use REFS since the presence of WINDOWS SERVER 2012 and WINDOWS 8, there is almost no documentation on this file system available, especially not from MICROSOFT.

The aim of this master thesis is: to examine the structure of REFS; to enhance the knowledge about this file system and how it works; to examine how MICROSOFT designed the file system to improve data-integrity.

Because of the missing documentation and the lack of support of REFS by the main forensic tools, reverse engineering was the method of choice and utilized for the examination of prepared REFS partitions.

The results of this master thesis will help other forensicators to verify the results provided by other, usually commercial, tools, or, in the case that the tools do not provide the needed functions, to do these jobs manually.

This master thesis will describe the experimental setup and the way of reverse engineering. Furthermore it will provide the results in a detailed way, supplemented by tables for a quick access to the results. To round of the master thesis, it will also provide a self coded python tool, which can be used for analyzing a REFS a partition, at least with REFS version number v1.2. This code was used within the analysis as a proof of concept for the results.

Thereby my python tool, described in pull-outs within this master thesis, is the only open source tool, which is able to handle REFS. Because of comparing my results with the output of ENCASE, the only commercial tool I found, which was able to handle REFS, I was able to find faulty outputs from ENCASE, what is also a proof, that this work is necessary.

Contents

List of Figures

List of Tables

Listings

1 Introduction

At the time of writing this thesis, nearly everyone in the industrialized world owns at least one electronic device such as a personal computer, laptop or smartphone [3] [4]. These devices run one of the three major operating-system platforms: MICROSOFT, MACOS, LINUX. Each system has to be divided into a variety of versions, including the mobile derivates of the operating systems [5].

For forensic purposes there is a need to be familiar with these operating systems, at least with the most commonly used, to be able to examine the seized devices / data in a forensically sound manner. On the one hand, this examination should include finding electronic evidence in the used space of the file system, the allocated area. On the other hand it should find deleted and manipulated files, restore the original data, verify which users have touched the file, and when the file was touched.

Even if this does not sound as difficult when using forensic software like X-WAYS, ENCASE, FTK etc, if you have to prove the findings of the named tools, you will need to have more than a basic knowledge of the file systems you will examine. This knowledge becomes more important in the case of a file system, which does not belong to the mainly used file systems and which is not fully supported by the forensic tools. In this previously named case, there is only a few open source knowledge available, results and findings can not be cross checked using other tools.

Concluding from the facts written before, proving the results of the forensic tools become more difficult, the less the documentation of the file system and its handling by the forensic tool is available.

Undisputedly, MICROSOFT is the most used operating system on personal computers and laptops and there are also derivates with identical file systems for mobile devices like the SURFACE-Tablets [5]. NTFS has been in use since 1992 and since WINXP it is the standard file system for MICROSOFTS operating systems, and therefore NTFS should be the most used file system, but it is becoming more and more outdated [5] . Newer file systems with enhanced journaling and data integrity are called for [6].

REFS is intended to be MICROSOFTS new file system: useable and ready for the next decade [7] . The target group should not only be the private user. Professional users and server-systems like "StorageSpaces" [8] are also targeted [8]. Hence, for forensic purposes it is essential to become familiar with this new file system. This is challenging because of a lack of documentation on REFS, due to the fact, that REFS is still in a testing period. Nevertheless, using WINDOWS-SERVER 2012 or WIN 8 / 10 you can already format drives with REFS. Using REFS as bootable partitions is not possible at this time [9] [7] [10] [11].

1.1 Target Group

This master thesis will provide detailed information about the file system layout of REFS. These information presuppose a basic understanding of file systems and some crafts in reading hexdumps. It is addressed to experts in computer sciences and engineering, as well as forensic examiners. Because of my professional background this master thesis is also supposed to support law enforcement authorities investigating crimes.

1.2 The Significance of Open Source Knowledge in Forensics

REFS is intended to be MICROSOFTS new file system. At the time of writing this master thesis, Network Attached Storages (NAS) running MICROSOFT SERVER are purchasable. If this NAS is seized from a law enforcement authority, for example because of a search warrant for child abuse material, this device, in a first step, has to be acquired. If this acquisition is done on a physical layer, all bytes from the NAS are copied, the data of the copy should be identical to the original data. But what is the benefit, if the NAS uses a proprietary or unknown file system, so that the analyzing software can not interpret the data, or interpret the data in a wrong way, simulating to the analyzing officer, that there are no relevant data?

Even if there is a trustworthy forensic software, is it trustworthy on all aspects of forensic work?

A common technique used to verify that a tool is not introducing data is to validate the results with a second tool. [12]

There is a reason for the principle of DUAL TOOL VERIFICATION in computer forensics, because there is no software able to handle everything in the forensics everyday life in a perfect manner. But how can you know, which tool is working sufficient, without knowing how to do the work manually? Of course, information technology (hardware and software) is intended to make the work easier and more efficient, but can we relinquish the knowledge doing the work manually?

The long-term solution is to have a comprehensive test methodology to decrease the total number of flaws so that the chances of a malicious person exploiting them are decreased. Having access to a tool's source code will improve the quality of the testing process because bugs can be identified through a code review and by designing tests based on the design and flow of the software. Experienced and unbiased experts *should conduct these tests and all details should be published.* [12]

At the end, the analyzing officers (no matter if they are educated in computer forensics or not) have to present the results to the court, and if needed, have to explain, how they gained these results. It seems obvious, that only referring to the output of a software is an insufficient way.

The problem appearing at this point is, that commercial software does not explain its way of processing the data, because that belongs to the business secrets of the developer. The other possibility might be studying the documentation of the file system's developer team. But in case of REFS, MICROSOFT has not published all documentation.

At a minimum, closed source tools should publish design specifications so that third parties, such as National Institute of Standards and Technology (NIST) Computer Forensic Tool Testing (CFTT), can more effectively test the tool's procedures.[12]

The last theoretical possibility for the forensicator is doing a reverse engineering of the file system on his own. Understandably, that is not always practicable, considering the time approach of some hundred hours. And, at the end, the findings might not be validated, hence they will be assailable.

According to this, everyone is invited to check my results, and, in the best case, validate them and provide these results as open source knowledge. This will easier the work of all forensicators, because there will grow a referable knowledge base and the results of each become reliable.

1.3 ReFS - Reverse Engineering of a File System

Because of missing documentation and open source knowledge regarding REFS, I decided to reverse engineer (see section 2.6) MICROSOFTs new file system (see section 2.2). The only fact I knew, when starting this research, was that REFS uses B-TREES (see section 2.3). I found some help using ENCASE, because ENCASE named some structures of the file system. Because ENCASE does not document how it gains its results I decided to start from the scratch, not considering the findings from ENCASE.

A first step, analyzing a file system should always be looking at the Volume Boot Record (VBR). This was already a part of my specialization project [13], and, as I have described in the cited work, there was some basic work done on the VBR by Andrew Head [10].

The next step was to find repeating structures. Hence I started searching for 64 kiB structures, because this was the cluster size found in the VBR (see figure 21), also provided by the $fsutil - command in the WINDOWS command line. So I have evaluated a few hundred 64 kiB Clusters to find and to understand the structure. In this context evaluating means, that I compared the structures I have found, trying to interpret the values as offsets, following these offsets, where they lead me. At this very starting point of reverse engineering, the process might be described as a search for traces on binary (and / or hexadecimal) level, the method of choice, at least for me, was try and error.

The first structure I have found, which I have already described in my specialization project [13], was the ENTRYBLOCK. At this point another problem raised: the structure of the ENTRYBLOCK with 16 kiB deviates from the cluster size, provided by the VBR and $fsutil.

The next step was looking for the root node, because this should be the first structure in a B-TREE, which has to be analyzed, because that is the the starting point, containing information to the whole structure of the B-TREE. I found such a structure and compared it with the system files, provided by ENCASE, here the root node was named $TREE_CONTROL, but I could not find an explanation, why it was named this way.

The $TREE_CONTROL contains the cluster offsets to several other system files, some handling information to the objects, other containing information to allocation and to attributes. I compared the system files I had found following the cluster offsets with the system files provided by ENCASE. And again, because of the missing documentation from ENCASE, the naming from these system files was not explained.

A practice I used during the whole examination process was searching the different structures also in other partitions formatted with REFS and comparing them. This was especially useful while analyzing the system files. This way I was able to assign the directories to the system file $OBJECT and the directory structure to the system file $OBJECT_TREE. Another finding based on this method was the handling of extents by REFS, which I would not have been confronted with, if I have used only my first research setup, containing two files.

Analyzing the way of allocation, I compared REFS with other file systems, like EXFAT and NTFS. I determined three different allocation structures, that coincides with a statement from MICROSOFTS development team [7] (three allocation structures deviating in the granularity of mapped areas are used) and the allocation files found in ENCASE.

The last system file, named $ATTRIBUTES by ENCASE, I tried to assign some findings to, by changing some of the flags using the WINDOWS context menu. While I changed the flags for "read-only", "hidden" and "archive", the content of the $ATTRIBUTES remains unchanged, the meaning of this special file has to be revealed in a further work.

Finally I could locate the changes, based on the previously mentioned manipulations of the flags, in the metadata within the FILENAMEATTRIBUTE, a well-known structure from NTFS. Besides the flags I could also determine the timestamps ("created", "modified", "metadata modified" and "last access"), the data-runs and other interesting structures within this attribute.

Being successful finding the FILENAMEATTRIBUTE I tried to discover also other attributes, known from NTFS (like STANDARD INFORMATION ATTRIBUTE, ATTRIBUTE LIST, VOLUME INFORMATION etc), but this was not as successful as expected.

To analyze, how REFS handle data, which is a main function of a file system, I used partitions with a different amount of files copied to. That becomes more relevant, as I made some efforts to the ENTRYBLOCK and asking myself, how the file system handles directories with a big amount of files copied to it. This question was triggered as I discovered, that a record for a file in an ENTRYBLOCK has an average size of approximately 1000 bytes. Remembering the size of an ENTRYBLOCK (16 kiB), the capacity of an ENTRYBLOCK is limited. So I prepared a partition with nearly 30 pictures and analyzed it. The result was discovering the extents. The ENTRYBLOCK is filled up with file records, if it is filled, the file records are copied to another ENTRYBLOCK. The latter gets related in the original ENTRYBLOCK by creating a record, containing the ENTRYBLOCK number of the newly created extent.

Another object, forensicators are always interested in, is the $RECYCLE.BIN. The $RECYCLE.BIN is named here as a synonym for the handling of files and data, marked as deleted by the file system. Accordingly I prepared a partition with some deleted files in it. In the root directory I found a record for the $RECYCLE.BIN, and in a subdirectory of the user, who was deleting these files, I discovered the records for the deleted files. These records get named different to the original files, some records started with a "$I" and some records started with a "$R" (one CHILDATTRIBUTE and one FILENAMEATTRIBUTE per type). Similar files, named "$I" and "$R" could be observed by the NTFS file system. REFS uses the different types of files for different purposes. The "$I" file provides the metadata for the created file-record within the $Recycle.bin. It also contains a data-run; this data-run contains the original filename and parent folder. The original record in the parents node was, in the case of my examined device, still existent, but the pointer to the record had been deleted. The $R record seems to be a copy of the original records, hence this record contains the offset to the data-run.

An important step while reverse engineering is cross checking the results. A first cross checking was done, while I compared the results won on one testing partition against the results, won on another partition. This checking was done manually, and, in my eyes, this checking might always be influenced by the examiner. Hence the decision for an automated analysis was made. I coded a python tool, containing all the findings to REFS. This tool was used to analyze several partitions, comparing the results with the predetermined circumstances of creating these partitions.

Even if this is mentioned several times within this master thesis, it is mentioned again at this point. A reverse engineering depends on the knowledge and the skills of the examiner. It is also limited in its reliability, because there is a vast amount of theoretical possibilities for interpreting the found data in another way. I tried to reduce this amount of theoretical possibilities while checking my findings on several partitions, created only for this certain purpose. However, I am conscious about, that the results of my research have to been proven by others to gain reliability.

1.4 Overall Problem

The overall problem I was confronted with within my research is named and described several times in this master thesis, at this very special point it is named again, detailed and explained.

- **Documentation :** Even if some of the features are explained and commented by MICROSOFT's tech team and / or development team, this is not sufficient for forensic issues. The functionality of the file system is not explained. It is not comprehensible, how files and data are treated by the file system while storing, accessing or deleting them. REFS is like a black box, there is an input of data, and there is an output of data, but what happens in the meantime? Are there metadata stored to the data, providing useful information for forensic issues? Are these information as reliable, a court can sentence an accused person only based on these information?

- **Support :** While doing my research, I tried to analyze the prepared devices with several tools, commercial and non-commercial ones. In the absence of a license of FTK, I opened a REFS - image in FTK-Imager. The hypothesis here was, if FTK-Imager is able to parse a REFS -image, FTK could be utilized also. I downloaded the latest version of FTK-Imager (version 4.2.0) and mounted a REFS image. Even if FTK-Imager does not provided an error message, it was not able to parse the image, no information to the contained data were provided. In order to be able to exclude that FTK can handle a REFS -image, I contacted the support team of FTK, asking for a test license or the needed information. Until writing this master thesis, neither the license nor the requested information were provided by AccessData, the manufacturer of FTK. Reading the user guide for the actual version FTK 6.1 [14], REFS will be found as a supported file system.

  X-WAYS does not support REFS, and, as I was told by an employee of X-WAYS, it is not intended, that X-WAYS will support REFS in the near future.

  In contrast to the both tools tool named before, ENCASE does support REFS, this tool is able to find the system files and ENCASE has an own nomenclature for these files.

  Beside the named commercial tools, there are some non-commercial tools, first to be named the SLEUTHKIT with its graphical user interface AUTOPSY. The REFS images could not be interpreted by SLEUTHKIT. I also tried other non-commercial tools, like hex editors, for example IBORED. Of course, these tools were able to display the hex code, but the included templates were not applicable to REFS.

- **Verification :** It is obvious, that, with the given requirements of missing documentation and missing support, the results achieved by me during this research can not be verified easily. For a first verification I used ENCASE, in consciousness, that I already discovered weaknesses of the tool in handling REFS. In a second step I have analyzed the hex code and have compared the findings from different images. As the last step for verification in the scope of my master thesis I have utilized a self-coded tool. This tool was designed as a proof of concept, to test, if my findings led me to the right conclusions. Furthermore, this tool is an easy way to share my results with the open source community.

Beside the named problems, I was also confronted with several other problems. Vicariously for the several problems I will name, that I found no literature dealing with the procedure of reverse engineering a complete file system. So I had to find my own way by testing different methodical approaches, like experimenting, excluding and of course the scientific approach, defining hypothesis, testing, refusing or strenghten them.

Last but not least, there is a common problem in forensics. Publishing open source, sensitive or private data might be leaked. One solution might be publishing only technical information, which only can be avoided, while reconstructing the original electronic traces and substitute the sensitive information by artificial ones. Though this might result in a unrealistic research environment.

Certainly the artificial conditions can be utilized as an advantage: the research environment can be adapted to special research questions. Explained in a different way, I have created different testing environments the same way I have splitted the research questions. This way several research results could be discovered more easily, or explaining it by using a well known metaphor: I reduced the haystack to find my needles.

But the results have to be evaluated considering the artificial nature of the research environment and, in a last step, have to be proven on realistic cases. These differences between the artificial and the realistic research environment are also made as a central theme while assessing the reliability of a forensic tool [12]. Summarized, the named work problematizes an almost infinite number of necessary tests with artificial testing environments to prove the reliability of a forensic tool for covering all possible constellations of a realistic case.

1.5 Research Questions

Before doing a file system reverse engineering, several questions have to be answered, especially to the target direction. Although it is possible to use reverse engineering for a single forensic investigation, answering only a limited amount of questions, this will lead to at least three problems.

First is the cost vs. usage calculation. Assuming, that answering only some specific questions might require approximately two hundred or three hundred hours, a single forensic investigator is busy for months. The second reason is, that no one is able to evaluate or to understand, how reliable the results are, won in such a procedure. The third reason is, that the results are not useable for another investigation with a different initial situation.

Generalized formulation of the questions will exhaust significant more time, but the results might be used several times. So the cost vs. usage calculation will be pushed to growing efficiency, even a second usage of the results might result in a positive cost vs. usage calculation.

Generalized results are suitable for presenting them to a larger public, the results might be proven by others and might become reliable.

Hence the target direction of my master thesis was obvious to me. The research has to answer general questions, providing a basis for further research of other researchers and the daily work of forensic examiners.

Finding adequate and generalized questions, two main topics have to be considered, the technical and the legal part. Topics of the legal part might question, if a single research is reliable enough to withstand in a legal process at court, or if there are other possible interpretations of the facts found within this research. Topics for the technical part might question, if it is possible to extract data from a ReFS partition and if the results are exactly enough to use them in an automated process.

Both topics are worth having a complete research on their own, but this will be beyond the scope of one single master thesis. On the other hand side there is a possibility to combine both topics in a single research, which led me to a question:

***Is it possible to verify the results from the commercial tools, which support* ReFS?**

During my research it became obvious, that this question could be specified, hence, as I described in the Overall Problem (see 1.4), only one of the major commercial tools supports ReFS. At the end, the main research question for me was:

Main Question:) ***Is it possible to verify the results for a* ReFS *-image, provided by* Encase*?***

In my eyes, this questions the technical topic as well as the legal one. The technical topic will be a prerequisite for answering the legal one. If the results of my research are congruent to the results provided by Encase, this will be an indicator for the reliability of my results, and the results provided by Encase.

While my research follows a practical approach, due to the reverse engineering, my first aim was to get behind the structures of ReFS. According to this, the first research question was:

Q_1:) ***Does* ReFS *has a* Vbr *like other file systems, eg.* Ntfs *and* ExFat*?***

The Vbr might be a place, where an offset to the root node can be provided. Hence two possible conceivable hypothesis' related to the question Q_1 are:

The Vbr *provides the offset to the main structure of the file system.*
or
The main structure has a fixed position within the file system.

Following the practical approach of my research, the next research question should be related to main structure, the root node:

Q_2.) ***Which kind of information does the root node contain?***

Thinkable hypothesis' for the research question Q_2 are:

The root node contains offsets to nodes,
matching the system files provided by Encase*.*
and
All the system files provided by Encase *have their own child node.*

The next structure, needed to be analyzed, is the EntryBlock. The structure itself was already found and described in a previous work [13], but in this master thesis I planned a deeper analysis of the data within the EntryBlock.

Q_3.) ***Which kind of information are stored in the* EntryBlock*?***
How are the information organized within a single* EntryBlock*?

For this research question I prepared the following hypothesis':

The structures within the EntryBlock *are comparable to the attributes,*

known from NTFS.
and
Files and folders have the same records, they are treated the same way.

Parallel to answering the research questions above, I coded a tool. The main question here was:

Q4.) ***Does the self coded tool prove the findings? Are the results only by accident or does the tool provide results repetitive, on different partitions with different content?***

My results are presented in this master thesis, an additional report is provided in the appendix. This should enable other forensic investigators to prove my results, which make them reliable.

1.6 Limitations

In my eyes, there exist different types of limitations. Some of them will be explained in the following:

personal limitations: My masters thesis is the work of only one student. This will cause some limitations in the scope of the research. First of all, the time used for the research is limited to 900 hours. Even if I was well supported by my first supervisor Assistant Professor Rune Nordvik, himself very used to file system analysis and hex dump evaluations, I had a limited view on reverse engineering, but I never missed the focus on using the results as a forensic investigator. Other assumable focuses might be e.g. data integrity and encryption of data, these topics were neglected by this master thesis.

This forensic focus is also manifested in my python tool, this tool can be used for extracting files from a REFS partition, including the path and the related timestamps; different versions of the same file or retracting information to encryption of a file were not considered.

In the same manner I was limited in focussing, I was also limited in my solutions. I had to compare the results found in the research with my knowledge gained within this master program. I have used my knowledge about attributes from NTFS, about B-TREES from HFS+ and the basic skills to analyze a file system. But doing this research as a part of a bigger team, other file systems might also have be compared with, e.g EXT4 or BTRFS.

technical limitations: Even if I tried to prove my results several times, these proofs were done with nearly equal initial situations. I used only four different partitions, created on the same hard disk, using the same machine. So I have to ask myself, if there might be different results starting at different initial situations, e.g. different setups of the creating machine, different types of devices containing the partitions (HDD, SSD).

theoretical limitations: A main topic in this master thesis is the missing documentation from MICROSOFT. No matters, how much research is done on REFS, there will always be a high probability, that there are undiscovered features and properties hidden in the code.This lack of transparency will start with the source code and is continued in the version management, while new functions are only mentioned superficially, if they are mentioned at all.

Without wanting to anticipate, the latter can be underlined with an example, occurred during my research. A modified setup of the preparation machine was pushing me into some trouble. It took me several hours to recognize that my preparation machine installed a WINDOWS system update, also modifying the drivers for REFS. This resulted in a REFS version upgrade from version 1.2 to 3.2, which made my tool failing in analyzing REFS partitions. This upgrade was not documented, neither were the differences in the functions and properties of ReFS.

A conclusion from the incident described before and all the limitations named in this section is, that my research is limited on the file system version 1.2, other versions has to be reverse engineered as well to generalize the results of my research. Also my results might be evaluated from the point of view, that these results were drawn from the basis of my knowledge and my way approximating a problem.

1.7 Master Thesis Outline

- **Chapter 1:** The introduction provides general information to this master thesis, my intention of reverse engineering a file system, the problems which have occurred and the limitations I was confronted with.
- **Chapter 2:** Some background information, necessary to understand the master thesis, are provided. I have implemented a literature research, collecting the knowledge from external sources to different topics treated in this master thesis. The terminology of file system, B-TREE and reverse engineering is explained.
- **Chapter 3:** The research design is introduced and the research setup is described. I have also explained my proof of concept.
- **Chapter 4:** The results are presented in a textual way, accompanied by some visualizations. I have relinquished to provide breakdown templates at this point for reasons of clarity. The templates are provided in the appendix instead, together with detailed results. This chapter should create a general understanding for the findings.
- **Chapter 5:** Different results are discussed in this chapter. Have the expectations, resulting from the research questions, been fulfilled?
- **Chapter 6:** In this chapter the results are compared against the research questions and necessary conclusions are drawn. As one part of this chapter some future work, which seems to be recommended, is described.
- **Appendix :**
  - Technical and detailed setup is described.
  - Detailed results for the following findings are provided.
    - VBR
    - $TREE_CONTROL
    - ENTRYBLOCK
    - FNA
    - SecondNode
    - $RECYCLE.BIN
  - The most important and most complete templates for a file system analysis of a REFS partition are provided.

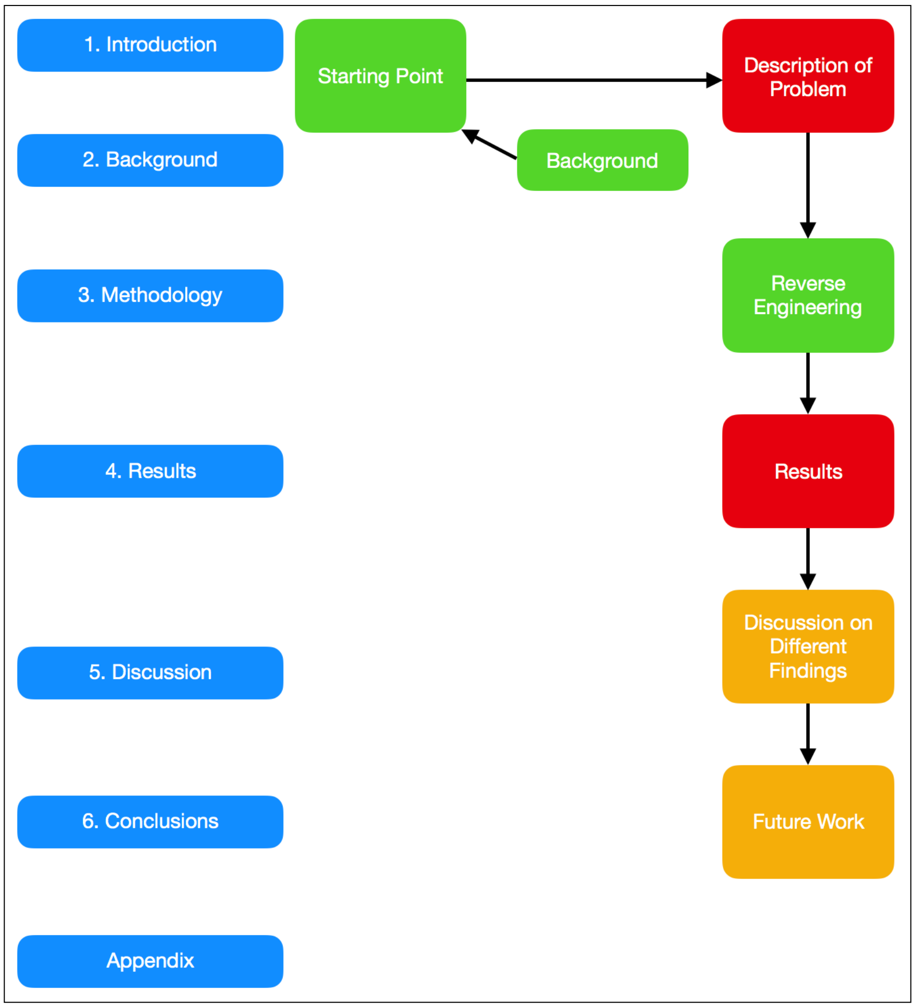


Figure 1: Structure of the Master Thesis

2 Background

2.1 Literature Research

Preparing this master thesis I did some further literature research, utilizing the resources of the world wide web. Below I listed some findings concerning the system files and the structures of REFS.

2.1.1 Building the next generation file system for Windows: ReFS [7]

Structure of ReFS

Steven Sinofsky, the previous head of MICROSOFTs development department published an article in 2012 dealing with some information relating to the structure of REFS. He claimed, that they had not invented REFS from scratch but had built it on parts selected from NTFS. For example they reused the code for implementing WINDOWS file systems semantics, which: implements the file system interface (read, write, open, close, change notification, etc.); maintains in-memory file and volume state; enforces security and maintains memory caching and synchronization for file data. The intention behind this was to make REFS highly compatible with NTFS. MICROSOFTs development department invented a newly architected engine that implements on-disk structures, such as the Master File Table (MFT: ongoing data stream of 1024 byte records, containing the information to a file or record, 2.2), to represent files and directories. [7] In his article Sinofsky visualized the structure with the graphic shown below:

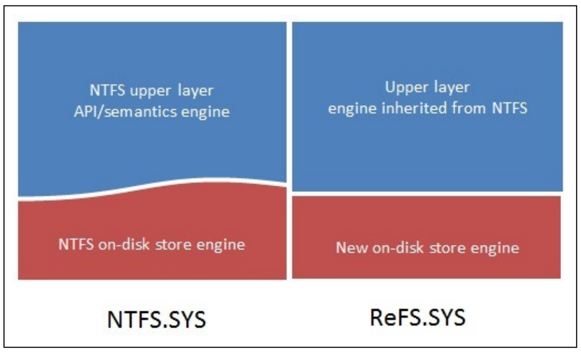


Figure 2: schematic visualization of the the components used in REFS [7]

Usage of B-Trees

To realize reliability and scalability they (development team of MICROSOFT) used B-TREES exclusively. Sinofsky claimed, that they used B-TREES as the single common on-disk structure to represent all information on the disk. Because the trees on the disk can be extremely large and multi-level, the file system ensures extreme scalability. The advantage of using only B-TREES is, as he claimed in his article, that code is reduced [7]. The information is stored in enumerable key-value sets in tables; most of the tables have unique id's, called object-id, for reference. These tables are indexed in a special object table. Below is a graphic, showing how the file system abstractions are constructed using B-TREES; the graphic is provided in the article itself.

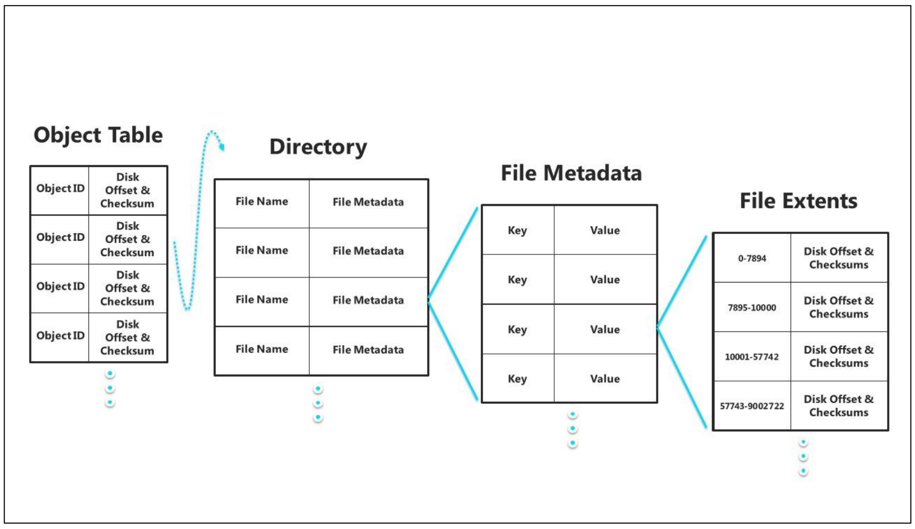


Figure 3: abstract visualization of the B-TREE-structure of the file system

As it is shown in the diagram above, directories are represented as tables, using B-TREE directories that are efficiently scalable. Files are also implemented as tables, embedded within a row of the parent directory; itself a table. This is visualized as File-Metadata in the graphic above. The row within the File-Metadata table represents the various file attributes. The file data extent locations are represented by an embedded stream table, which is a table of offset mappings.
Other global structures within the file system, such as ACLs (Access Control Lists), are represented as tables rooted within the object table.

Disk Allocation

All disk space allocation is managed by a hierarchical allocator, which represents free space by tables of free space ranges. For scalability, there are three such tables - the large, medium and small allocators. These differ in the granularity of space they manage: for example, a medium allocator manages medium-sized chunks allocated from the large allocator.

Checksums & Integrity Streams

To realize the goal of detecting and correcting data corruption, all REFS metadata are checksummed at the level of B-tree page; the checksums are stored independently from the page itself.
Additionally, an option was implemented to also checksum the file content, called "integrity stream". When it is enabled, REFS writes the file changes to a location different from the original one. The checksum write is automatically done with the data write ("allocate on write" - technique). The checksums mentioned above are 64-bit checksums.

2.1.2 The Microsoft ReFS On-Disk Layout [11]

On his website, Ballenthin, a reverse engineer, blogged some findings to REFS. Besides some basic knowledge about GPT and some findings, already stated and explained in my previous work, [13] he gives some findings about the structure of REFS.

VBR

Ballenthin's findings relating to the Volume Boot Record, VBR, are rudimentary and outdated, according to my findings, as explained in my previous work.

File System Metadata

In his blog, Ballenthin confirms my previous findings, that metadata are found spread over the whole partition on the volume. These metadata-blocks were called ENTRYBLOCKS in my previous work [13]. He also confirms the size of these blocks with 0x4000 bytes, dec 16384, which differs from details given by MICROSOFT[15] itself or by Head[10].

FileContents

Ballenthin gives a small amount of information about REFS' handling of small files. He stated that small files are not stored resident within the metadata block, as is the case in NTFS. This was proven in my previous work, using a small file (105 bytes). Ballenthin further stated, that small files might be stored at offset mod 0x10000, dec 65536. The last value is the corresponding value to 64 KiB, which is the total amount of bytes per cluster claimed by MICROSOFT. If this finding were to be proven, the forensic investigator possibly has to make differences in offset-calculation for meta-data and file content.

2.1.3 Verifying properties of ReFS [2]

This work is intended to verify the announced properties of REFS. In the introduction it is claimed that REFS is based on the most used file system, NTFS. REFS adapted a large, simplified part of NTFS; some lesser used functions were eliminated. Although it is only available on Windows Server, it is also planned for end user operating systems.

Basic parameters - Integrity

The file system, itself, recognizes a data integrity problem and manages to repair it without any user interference. Metadata integrity is ensured by checksums; updated metadata are stored on different locations to prevent data loss while writing data. Configuring Storage Spaces in mirroring mode, an optional function for data integrity can be chosen This function will automatically restore data from a backup copy, provided by the Storage Spaces. The optional function can also be configured manually, using the command line.

REFS also offers the option of an intelligent scanner, called scrubber, which scans the volume, the allocated data and metadata in regular intervals to verify the correctness of control calculations.

Storage Spaces

Horalek et al describe how easily Storage Spaces can be configured, expanded and repaired. Storage Spaces are created on a software layer, with different integrity options: without control options, mirroring and parity.

Basic parameters - Availability

To improve data availability, REFS, in a case of data corruption, which can not be repaired automatically, isolates the faulty locations, and the corresponding file record is removed, so that the faulty file vanishes. This action prevents effects on other files, while offering the option of repairing to the administrator.

Complications during implementation

REFS can not be used for formatting removable or bootable devices. The cluster size can not be changed during formatting; the only available size is 64 KiB.

SUPPORTED FUNCTION	UNSUPPORTED FUNCTION
Bitlocker encryption	Secondary stress
Access control list (ACL)	Object identifier
Journaling	File name shortening
Change reporting	File compression
Symbolic link to files and directories	Hardlinks
Volume connection to a directory	Sparse files
Additional information about the directory	File level encryption
Volume copy	File details
Straightforward file identifier	Disk quotas
File locking	

Table 1: REFS capacitive parameters [2]

REFS disk structure

Horalek et al provide some rudimentary information about the disk structure. Some basic offsets for the VOLUME BOOT RECORD are provided; just like the offset to the backup VBR, which is located in the last sector. Furthermore, Horalek et al provide the sector offset 4425 for the VOLUME NAME and the sector offset 4352 for the UPCASE TABLE.

Summary

Horalek et al refer to general information provided by MICROSOFT. Information, concerning the structure of REFS, seems not to be solid, hence there is no information concerning their methodological approach for retrieving such data. Furthermore, the provided data seems to be invalid, according to my own findings.

The findings of Horalek et al concerning the reading and writing speed of REFS compared with NTFS might be a first approach, but is not relevant for this work.

2.1.4 Open Source Digital Forensic Tools [12]

This paper of Brian Carrier focuses on the problem, how reliable a closed source forensic tool can be. He splitted his considerations into the three major topics acquisition, analysis and presentation.

Acquisition

The results of acquisitions can easily be compared between open source and closed source tools, and can be tested with a prepared testing environment. Because of this, the acquisition phase is not the main problem, being discussed by Brian Carrier.

Analysis
Brian Carrier describes in his paper, that the acquired data are searched for the three different types of evidence:

- Inculpatory Evidence
- Exculpatory Evidence
- Evidence of Tampering

The analysis phase is characterized by the examination of the files and contents and recovering of deleted data. It is fundamental important, that analyzing tools show all data that exists in an image. Because of the enormous bandwidth of possible data, it is nearly impossible to design comparable tests for all possibilities. In contrast to the analysis, the presentation is entirely based on policy and law, which differs according to the investigation setting (corporate, federal, military).

Admissibility
Evidence must be relevant and reliable, to be admissible in an US court. Therefore, the scientific evidence, where also the output from forensic tools is subsumed to, has to be checked before in a pre-trial "Daubert Hearing". This "Daubert Hearing" identifies four general categories:

- Can and has the procedure been tested?
- Is there a known error rate of the procedure?
- Has the procedure been published and subject to peer review?
- Is the procedure generally accepted in the relevant scientific community?

For each of this categories a specific guideline is used, which is not described in detail here. In this summary only one fact will be mentioned: According to the paper of Brian Carrier, the proper way to test forensic tools is by using an open method. Requirements must be created for each tool type and corresponding tests must be designed that enforce the requirements.

Brian Carrier names the both two arguments used as pro and contra in the discussion, if forensic tools should be open source.

- **Contra :** People with malicious intent can find flaws in the source code and exploit them without publishing the details.
- **Pro :** Having access to a tool's source code will improve the quality of the testing process because bugs can be identified through a code review and by designing tests based on the design and flow of the software.

Even if there is a basic understanding for the companies need for protecting their code, they should, as a compromise, publish their design specifications, so that third parties can more effecitvely test the tool's procedures. He underlines, that, if the extraction tool is open source, the results from the presentation tool can be proven The presentation tool, many new features in file system digital forensic analysis tools are based on the presentation part, remains closed source, so there is the possibility to differ from the other vendors.

Summary
The example of the US court shows explicitly, how important the reliability of a tool is. Brian Carrier argues, that this reliability might be easier to achieve, if open source tools are used, that the usage of an open source is not a vulnerability, the open source character is, according to experience, used for improving the tool. Open source tools can be code reviewed, this helps to fix bugs in general, without the need for testing every functionality.

2.1.5 Specialization Project [13]

During my studies I undertook a specialization project on the same topic. At this time, it was hard to find some documentation about REFS. Some of the sources I have found have already been cited in the specialization project. At this point, I only want to refer to Andrew Head [10], because his work was the most comprehensive one that I found. His work was the starting point of the named specialization project, hence I tried to prove his research in an initial step. In a following step, I did my own research and could show some basic structures. Furthermore, I provided some basic break-down tables for the basic structures that I had found. In chapter seven of the specialization project, I named topics on which further work should be done. These named topics also include themes like: handling of deleted files; B-TREES and Storage Spaces, which are treated within this master thesis.

2.2 File System

In computing, a file system or filesystem is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system". [16].

The file system is the link between the operating system and the data stored on the designated devices, e.g. HDD, SSD, CD. While naming the stored files with their plain text names in the application layer, the file system assign these plain text names the physical location on the disc.

A simplified example might be a book. You will find every chapter by searching the book page by page, but the more pages, the more ineffective this method is. Searching the chapter by looking it up in the table of content will work much faster.

The main task of the file system is to provide the information needed to store data or access the data stored on the storage devices. Beside the information to the location of data on the storage devices, other information like timestamps and permissions are stored as metadata by the file system. The kind of related data information stored by a file system are dependent from the file system, e.g. Mac based file systems (OS X) store a lot of information for manipulating pictures and documents as metadata, so these changes can be retracted [17].

Different file systems use different methods for storing the data and organizing the process of storage. One method of organizing these data is the MASTER FILE TABLE (MFT), used in MICROSOFTS NEW TECHNOLOGY FILE SYSTEM (NTFS). The MFT is a file, which stores the information to the stored data in 1024 byte blocks, these are lined up in a row. Because this is a linear setup, the search process might consists of a multitude of steps. An example for a non linear setup is the B-TREE, explained in the next section, which utilizes a logarithmic approach of searching (example: 2.3). These methods causes different access times, differences in reliability and data protection. A good example for the differences in data protection is, that protective file systems, while modifying data, write the new data to a new space. If the writing process is erroneous, an undamaged copy is existent.

2.3 B-Tree

In computer science, a B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two child nodes (Comer 1979, p. 123). Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in databases and file systems. [18]

The increasing need of memory, based on growing datasets [19], requires adapted methods of storing and organizing these data. Modern file systems, like APFS [20] and BTRFS [21], take account of these requirements. Both utilize structures used in databases, the B-TREES.

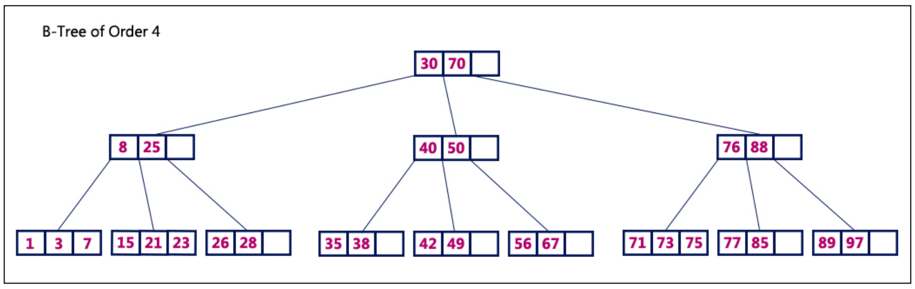


Figure 4: Schematic representation of a B-TREE[1]

Following, some of the advantageous characteristics of a B-TREE are explained.

- A B-TREE consists of the root node at the top and the leaf nodes at the bottom. Between these, there might be branch nodes.
- The amount of steps from the root node to a leaf node is defined and consistent to every leaf node. This is defined as the height of the B-TREE, because of this unified height some translates B-TREE as balanced tree.
- Every node contains a defined amount of keys (k), between k and $2 \times k$, except the root node, this contains one up to $2 \times k$ keys.
- Comparing a special key with the other keys at the same level, keys with a lower value have to be on one side, higher values on the other side, that is achieved by an increasing sort order.
- Example [18] : A fully filled node with the height of four ($h = 4$) and the maximum amount of child nodes of 1024 ($t = 1024$) has the maximum amount of $1024^4 - 1 = (2^{10})^4 - 1 = 2^{40} - 1$ keys stored. Because a search operation needs a maximum of $h + 1$ steps, a B-TREE with the parameters given above needs a maximum of five steps to find a key. In contrast to the

linear search in a MFT, this might be described as a logarithmic search.

The last example shows clearly the advantages of a B-TREE, compared with older file systems, handling large amount of data. While a file system based on a MFT needs a maximum of $2^{40} - 1$ steps, staying at the explained example, the B-TREE based file system needs only a maximum of five steps. This results in a better performance. Even if the MFT based file system stores the MFT in cache to guarantee shorter access times, it results in a lower performance, because of the higher use of storage, hence the B-TREE based file system loads only the needed branches in cache.

2.4 Encase

ENCASE is a commercial forensic tool from "Opentext" (formerly "Guidance Software"), which belongs to most accepted ones [22]. It unites functionalities for acquisition and examination. While acquisition is not in the focus here, ENCASE provides the functions for viewing the data in a tree-structure, in a binary view and in a disc view. The latter is an option, not provided by the german competitor X-WAYS (the standard tool at my local police department), which offers an easy and comfortable way to jump between the clusters. I used this view for a first superficial analysis of clusters, furthermore I dumped clusters for deeper analysis displayed by this view. This disk view itself discovered a failure from ENCASE. It revealed, that structures, named as several single system files by ENCASE, were contained in the same single cluster, which makes them a single file. Beside this failure in nomenclature, ENCASE was the only tool, available to me, which was able to reveal a first structure, the system files, in REFS.

2.5 File System Recognition Structure

The goal of file system recognition is to allow the Windows operating system to have an additional option for a valid but unrecognized file system other than "RAW". [23], [24]

The FILESYSTEM RECOGNITION was introduced on WINDOWS 7 [23] and should enable a WINDOWS operating system to recognize an unknown file system as a valid file system, instead of an unpartitioned device. This is realized by using the FILESYSTEM RECOGNITION STRUCTURE, a checksum, calculated across the whole VBR.

2.6 Reverse Engineering

Reverse engineering, also called back engineering, is the processes of extracting knowledge or design information from a product and reproducing it or reproducing anything based on the extracted information. The process often involves disassembling something (a mechanical device, electronic component, computer program, or biological, chemical, or organic matter) and analyzing its components and workings in detail. [25]

Reverse Engineering is a process, which can be used, e.g. for researching a product, like software or malware. There might be different reasons why a research on a product has to be done. In case of a malware, the reverse engineering might give some traces to the offenders, in case of a software the proper work might be checked 2.1.4.

Both, the malware and the commercial tool have in common, that there is usually no documentation. Of course, there might be other possibilities than a reverse engineering, like comparing the output of the tool or the communication of a malware. But this would not explain, why the observed things have happened. The best understanding of the malware or the tool will be reached by reviewing the source code. Sometimes it is not possible, because the programmer has obfuscated parts of the code, or it is simply not allowed by the copyright law. Hence another approach have to be chosen.

An appropriate way for analyzing a file system might be modifying single parts and comparing the differences in the changed bits and bytes on the storage devices. Starting with central issues, specific questions will arise. Examining these, the experimental layout has to be modified several times. Solving these single questions is like fitting in a puzzle piece, the whole becomes clearer.

While reverse engineering, there is the present danger of drawing premature conclusions. To prevent this, the single results, no matter how unimportant they might be, have to be cross checked. Every result has to be questioned, otherwise this will lead to a concatenation of errors, inherently conclusive, but, because it is based on an error at the beginning, the result will not be reliable.

To achieve reliability, a gapless documentation is elementary. Even if there are logical mistakes, the documentation will make these errors comprehensible for the reader, which will act like an external review.

Summarizing my experiences from my research on ReFS, a reverse engineering is a very interesting, but also a difficult and sometimes frustrating process. In the end, if it is successful, it is worthwhile and will lead to a plenty of new information.

3 Methodology

3.1 Research Design

Following the classical methodological design, I have to summarize, that I have used a qualitative design, uniting also elements of the naive inductivist [26] and exploratory approach. But is it the right way, pressing a very technical research in these theoretic schemes? Might there be a more practical scheme, defining the design of my research? Does the reader get a feeling of the work, done in this research, by reading the phrases of qualitative design or exploratory approach?

Perhaps, the better way is to reduce all the thoughts of what I have done in this master thesis to only two words: REVERSE ENGINEERING. These two words describe the technical work, which had to be done, and the reader of this master thesis is able to imagine the procedures and the work flow within this research, especially after reading the introduction 1.3 and the background 2.6.

But what does REVERSE ENGINEERING mean, at least in my research? Using simple words, I compared two different states of a REFS - partition. As it is described in the TESTING ENVIRONMENT 3.2, I prepared an empty REFS - partition and a REFS -partition with two files. Even if this setup might not give important impulses for the analysis of the VBR, but I expected this setup to give significant hints for analyzing the system files and the ENTRYBLOCKS. First findings can be verified by preparing other REFS partitions, adapted to the special needs. This would seem advisable analyzing flags and extents. In the first case a special partition could be prepared, containing several files, in the first attempt with unchanged, normal flags. In a second attempt, different flags can be changed for varying files, both dumps can be compared and should deliver some results. In the latter case a dump of a partition with a folder, filled with only one or two files can be compared against a dump of the same folder, filled with a big amount of files.

Another important part of my research is comparing the findings provided by ENCASE with the corresponding hex-dumps. This should grant findings, e.g. if every system file, provided by ENCASE, has its own record, which will imply, it is an independently file. The corresponding record should also contain the name of the system file, otherwise the naming of ENCASE will not be verified.

Another important point in my research is flexibility. Doing a REVERSE ENGINEERING of a nearly undocumented file system will cause the examiner to react to the findings in the hex-dump, adapting the testing environment to the current circumstances.

3.2 Testing Environment

In this section the testing environment is described, with the focus on comprehensibility. The detailed description, for everybody who is interested in, for example to reproduce and check my results, is provided in the appendix.

During the whole research I used two different machines, one for creating, manipulating and dumping the REFS partitions, the other for analyzing the dumps. This setup was choosen, because of several reasons.

1. The requirements for the different machines are quite different. While the preparation machine should have a fast data throughput and some possibilities to connect external drives, the examination machine needs to have more calculating power and a big display.
2. To avoid contamination of the created partitions, the preparation machine has to have a easy way to setup a write-blocker, also to avoid mounting, if this is not intended. As an example I will mention the research at the topic "timestamps", here it is elementary, that there is no unauthorized access to the drive. Ideally this machine should not have been used before and should not be used for other tasks during the research.
3. It turned out to be useful to block the internet connection of the preparation machine, because the WINDOWS system updates have modified the REFS driver. Because of that, the used REFS version changed at the end of the research.
4. The examination machine needs to have an internet connection due to the simple fact, that I only have access to an ENCASE network dongle, provided by the NORWEGIAN POLICE UNIVERSITY COLLEGE via a virtual private network (VPN) within their intranet.
5. The need of calculating power is based on the compute-intensive tools, which were used during the research, mostly several of these tools were used at the same time, for comparison reasons.

I decided to use a Lenovo Ideapad with an INTEL i5 and 8 GB RAM, running a Linuxmint, as the preparation machine. I virtualized a WIN 10 for creating and modifying the REFS partitions. Furthermore the Ideapad was set up as a forensic machine, so auto-mount etc was disabled. In connection with tools like "ewfacquire" [27] this seemed to be a good preparation machine to me.

As the examination machine I used a MacBook Air, connected to an external 24" display, with an Intel i7 and 8 GB RAM, also virtualizing a WIN 10. On this machine several tools were used, for example iBored and WxHexEditor (both on Mac OS) for displaying the hexcode; x-Ways Forensics and Encase as forensic tools (both on Windows). If there was a need to acquire disks at the examination machine, I used a software write-blocker (DiskArbitrator) and a hardware write-blocker simultaneously, to avoid contamination. Some examinations were done on a Linux host on the MacBook Air, a tool used here was sleuthkit.

After deciding about the machines, I had to decide about the disk setup. I used one disk for the basic research, this disk was partitioned into 4 partitions of different size, from a very small size of 4.9 GiB up to the big size of 97.7 GiB. Two partitions were equally sized with 73.2 GiB. The reason for the different sizes was to get information about the disk layout and if the disk size influences the disk layout.

Last thing to decide about were the tools to be used. I decided to use X-WAYS FORENSICS and ENCASE as the forensic tools. X-WAYS because I am used to it as part of my daily work. X-WAYS is based on a hex editor, so it is ideally suited for analyzing hex dumps. ENCASE got in my focus at the end of my specialization project [13], finding the system files. Another advantage of ENCASE is, in my eyes, the disk view modus. Using this, the navigation on the partition was easier for me. Another tool, heavily used, is wxHexEditor. This tool was used to dye the hexdumps for visualization purposes.

3.3 Proof of Concept

A problem, occurring while preparing the master thesis, was defining how the results might be proven. Because of a lack of comparable work I was convinced, that an appropriate way might be coding a tool. This tool could be used for analyzing the devices, prepared before. Hence the results provided by the tool can be compared against the target state, they are definable, because the devices got prepared by myself, and the results can be proven by a manual examination, analyzing the hex dump.

During the MISEB-study, I programmed tools using: BASH on Linux and MacOs -systems; the powershell on Windows-based systems and the python programming language. The advantage of the latter is that it is platform independent; it can also use a broad base of forensic tools. Hence the decision was easily made in favor of python as the programming language.

After finishing the code, with all the functions that I thought they were necessary, the code contained approximately 1500 lines. Because there are a lot of lines concerning output formatting or, within the scope of this master thesis, uninteresting code, I decided to include only the code, which was important to handle a REFS partition. These code snippets can be found in the appendix. The complete code can be requested from the supervisors of this master thesis and, of course, from me, at least until completion this master thesis. Afterwards it is planned to publish the source code, for example via git-hub.

As it was foreseeable, coding the tool led me to a plenty of new findings, details I have not worried about before. In the end, my tool was able to find all the files and details, I considered while preparing the devices. It is working on all the devices, I have prepared in the scope of this master thesis, but I can not state, that it is working on every ReFS partition. Again, everybody is invited to continue working on the code to improve the performance and the results.

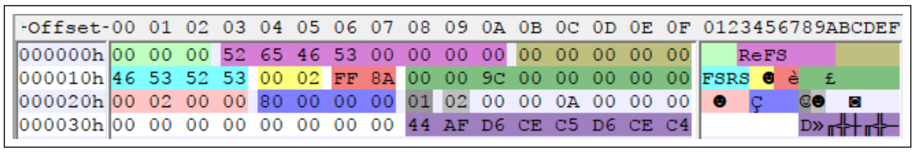
4 Results

In this chapter the results of my research are presented. The main emphasis lies on presenting a general overview. Detailed results, like templates for the breakdown of hex dumps, are provided in the appendix in sections, equally named like the sections in the results chapter.

4.1 VBR

As it was mentioned in the B-TREES section 2.3, the starting point for analyzing a B-TREE - based file system should be the root node, containing information to the structure within the B-TREE.

There are at least two ways for finding the root node, the first is using the data from the VBR. A hex-dump from a typical VBR of a REFS partition is provided below.



```
-Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 00 00 00 52 65 46 53 00 00 00 00 00 00 00 00 00    ReFS
000010h 46 53 52 53 00 02 FF 8A 00 00 9C 00 00 00 00 00 FSRS ☻ è  £
000020h 00 02 00 00 80 00 00 00 01 02 00 00 0A 00 00 00  ☻  Ç   ☺☻  ◙
000030h 00 00 00 00 00 00 00 00 44 AF D6 CE C5 D6 CE C4         D»╓╬┼╓╬─
```

Figure 5: Hex-dump from a VBR

In this section only the dark green colored value (0x00009C0000000000) is needed, for the complete breakdown see A.2.1. This value has to be read as little Endian (0x9C0000) and provides the offset to the BACKUP VBR, which is a safety copy of the VBR. This value deviates from the size of the partition, so there is some unaddressed space at the end of the partition. The size of the unaddressed space varies and seems to depend on the size of the partition [13]. The BACKUP VBR is located in the last sector of the addressed space. Furthermore, in the third last cluster of the addressed space the offsets to the $TREE_CONTROL (which seems to be the root node, providing the offsets to the system files) are provided.

An interesting and new structure can be seen at relative offset 0x10, colored in light blue, the structure identifier. Together with the checksum, found at relative offset 0x16, colored in red, it constitutes the FILESYSTEM RECOGNITION STRUCTURE (2.5).

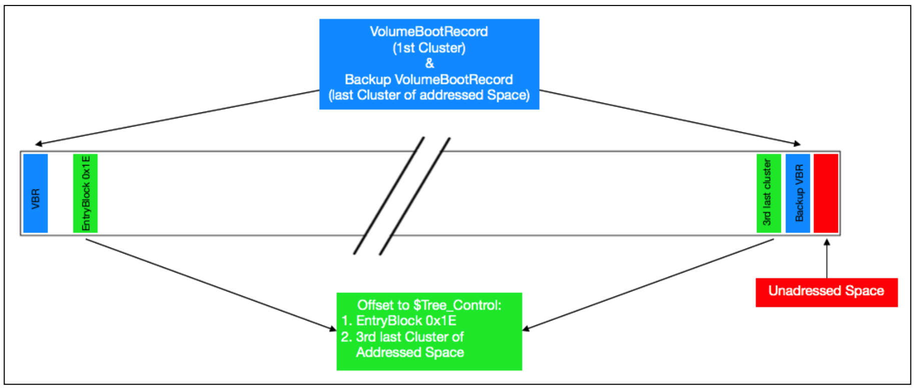
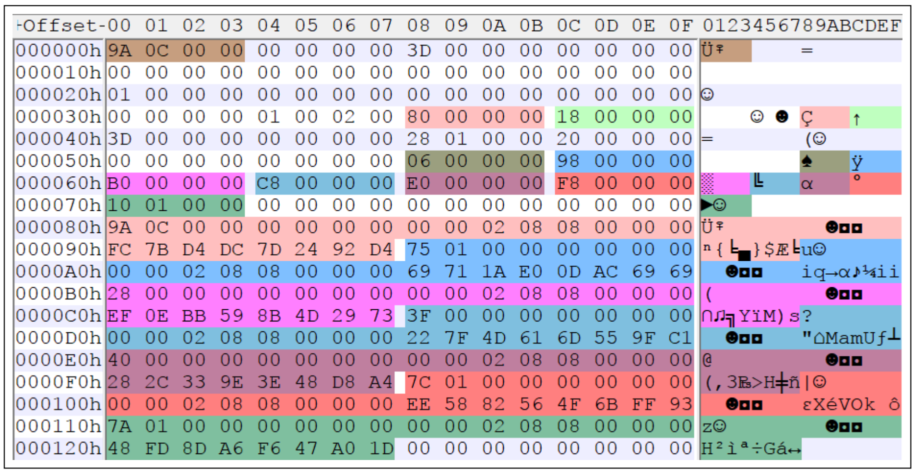


Figure 6: Schematic layout of a REFS partition with focus on the VBR and the location of the offsets to the $TREE_CONTROL

4.2 $Tree_Control and System Files

Another possibility to find the $TREE_CONTROL is jumping to the ENTRYBLOCK (see 4.3) 0x1E. On all partitions I have analyzed, there were data in this special structure, containing the offsets to the $TREE_CONTROL. Following these offsets, the $TREE_CONTROL will be found, a screenshot is provided below.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 9A 0C 00 00 00 00 00 00 3D 00 00 00 00 00 00 00 Ü‡      =
000010h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000020h 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
000030h 00 00 00 00 01 00 02 00 80 00 00 00 18 00 00 00     ☺ ☻ Ç   ↑
000040h 3D 00 00 00 00 00 00 00 28 01 00 00 20 00 00 00 =       (☺
000050h 00 00 00 00 00 00 00 00 06 00 00 00 98 00 00 00         ♠   ÿ
000060h B0 00 00 00 C8 00 00 00 E0 00 00 00 F8 00 00 00 ░   ╚   α   °
000070h 10 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ►☺
000080h 9A 0C 00 00 00 00 00 00 00 00 02 08 08 00 00 00 Ü‡        ☻◘◘
000090h FC 7B D4 DC 7D 24 92 D4 75 01 00 00 00 00 00 00 ⁿ{╘▄}$Æ╘u☺
0000A0h 00 00 02 08 08 00 00 00 69 71 1A E0 0D AC 69 69   ☻◘◘   iq→α♪¼ii
0000B0h 28 00 00 00 00 00 00 00 00 00 02 08 08 00 00 00 (         ☻◘◘
0000C0h EF 0E BB 59 8B 4D 29 73 3F 00 00 00 00 00 00 00 ∩♫╗YïM)s?
0000D0h 00 00 02 08 08 00 00 00 22 7F 4D 61 6D 55 9F C1   ☻◘◘   "⌂MamUƒ┴
0000E0h 40 00 00 00 00 00 00 00 00 00 02 08 08 00 00 00 @         ☻◘◘
0000F0h 28 2C 33 9E 3E 48 D8 A4 7C 01 00 00 00 00 00 00 (,3₧>H╪ñ|☺
000100h 00 00 02 08 08 00 00 00 EE 58 82 56 4F 6B FF 93   ☻◘◘   εXéVOk ô
000110h 7A 01 00 00 00 00 00 00 00 00 02 08 08 00 00 00 z☺        ☻◘◘
000120h 48 FD 8D A6 F6 47 A0 1D 00 00 00 00 00 00 00 00 H²ìª÷Gá↔
```

Figure 7: $TREE_CONTROL

The detailed breakdown is provided in A.2.2. At offset 0x58 the amount of records is given, offset 0x5C provides the offset to the first record, in the example above there are 0x06 records starting at offset 0x98.

The $TREE_CONTROL points to system files (nodes) that can be categorised into:

- Objects

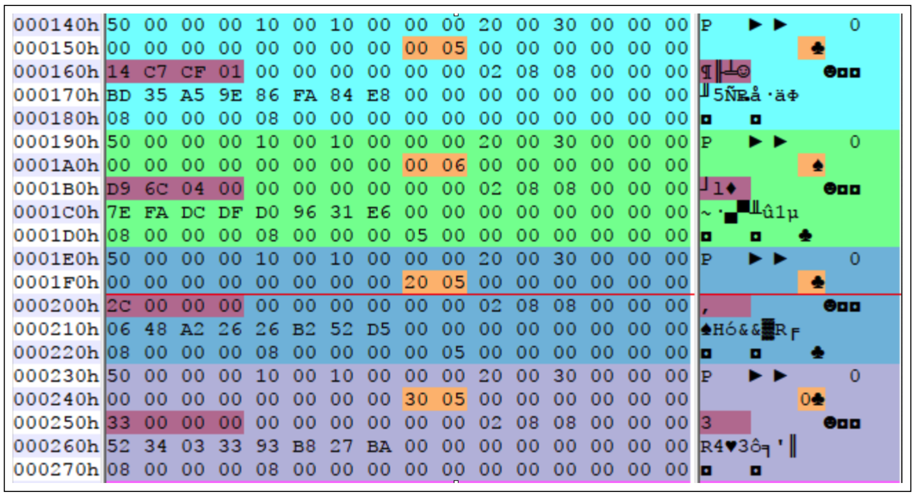
- Allocations (large, medium, small)
- Attributes

4.2.1 Objects

$Object_Tree

The EntryBlock offset to the $Object_Tree is found at offset 0x98 of the $Tree_Control (see figure 7), 0x175 (LE). The record area of the $Object_Tree is provided in figure 8.

The first record in figure 7, starting at offset 0x98 and colored in blue, provides the EntryBlock offsets (number of 16 kiB structures starting from the beginning of the partition) for the system file, called $Object_Tree by Encase.



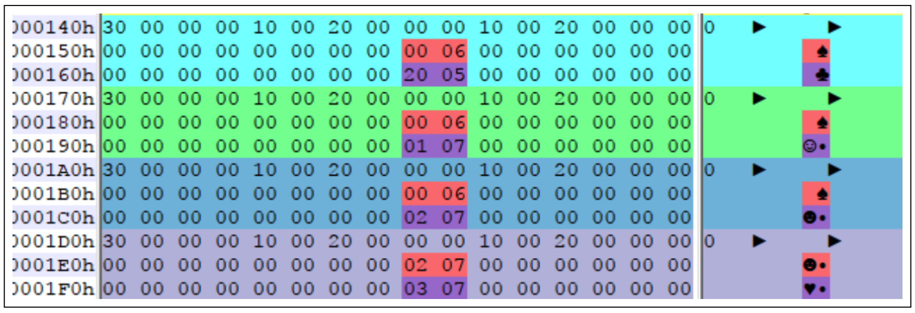
```
000140h 50 00 00 00 10 00 10 00 00 00 20 00 30 00 00 00  P   ►►        0
000150h 00 00 00 00 00 00 00 00 00 05 00 00 00 00 00 00            ♣
000160h 14 C7 CF 01 00 00 00 00 00 00 02 08 08 00 00 00  ¶╟╧☺      ☻◘◘
000170h BD 35 A5 9E 86 FA 84 E8 00 00 00 00 00 00 00 00  ╜5Ñ₧å·äΦ
000180h 08 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00  ◘   ◘
000190h 50 00 00 00 10 00 10 00 00 00 20 00 30 00 00 00  P   ►►        0
0001A0h 00 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00            ♠
0001B0h D9 6C 04 00 00 00 00 00 00 00 02 08 08 00 00 00  ┘l♦       ☻◘◘
0001C0h 7E FA DC DF D0 96 31 E6 00 00 00 00 00 00 00 00  ~·▄▀╨û1µ
0001D0h 08 00 00 00 08 00 00 00 05 00 00 00 00 00 00 00  ◘   ◘   ♣
0001E0h 50 00 00 00 10 00 10 00 00 00 20 00 30 00 00 00  P   ►►        0
0001F0h 00 00 00 00 00 00 00 00 20 05 00 00 00 00 00 00            ♣
000200h 2C 00 00 00 00 00 00 00 00 00 02 08 08 00 00 00  ,         ☻◘◘
000210h 06 48 A2 26 26 B2 52 D5 00 00 00 00 00 00 00 00  ♠Hó&&▓R╒
000220h 08 00 00 00 08 00 00 00 00 05 00 00 00 00 00 00  ◘   ◘    ♣
000230h 50 00 00 00 10 00 10 00 00 00 20 00 30 00 00 00  P   ►►        0
000240h 00 00 00 00 00 00 00 00 30 05 00 00 00 00 00 00           0♣
000250h 33 00 00 00 00 00 00 00 00 00 02 08 08 00 00 00  3         ☻◘◘
000260h 52 34 03 33 93 B8 27 BA 00 00 00 00 00 00 00 00  R4♥3ô╕'║
000270h 08 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00  ◘   ◘
```

Figure 8: Dump from record area of the $Object_Tree

In the dump four records are displayed, every record provides the node ID (colored in ocher) and, colored in purple, the EntryBlock - offset (see 4.3), they are located in. In contrast to the branched structure of a tree, this is a very plain structure, listing only the objects. In context of the B-Tree-structure, the $Object_Tree is the top node, containing all the offsets to the directories as its child nodes. The $Object_Tree does not provide any information to the relations between the single objects. Hence, in my eyes this file should be better named as $Object.

$Object

The last record in figure 7, starting at offset 0x110 and colored in green, provides the offset to a file, named $Object by Encase, a screenshot is provided below.



```
00140h 30 00 00 00 10 00 20 00 00 00 10 00 20 00 00 00 0    ►      ►
00150h 00 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00          ♠
00160h 00 00 00 00 00 00 00 00 20 05 00 00 00 00 00 00          ♣
00170h 30 00 00 00 10 00 20 00 00 00 10 00 20 00 00 00 0    ►      ►
00180h 00 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00          ♠
00190h 00 00 00 00 00 00 00 00 01 07 00 00 00 00 00 00          ☺•
001A0h 30 00 00 00 10 00 20 00 00 00 10 00 20 00 00 00 0    ►      ►
001B0h 00 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00          ♠
001C0h 00 00 00 00 00 00 00 00 02 07 00 00 00 00 00 00          ☻•
001D0h 30 00 00 00 10 00 20 00 00 00 10 00 20 00 00 00 0    ►      ►
001E0h 00 00 00 00 00 00 00 00 02 07 00 00 00 00 00 00          ☻•
001F0h 00 00 00 00 00 00 00 00 03 07 00 00 00 00 00 00          ♥•
```

Figure 9: Dump from the record area of the $OBJECT

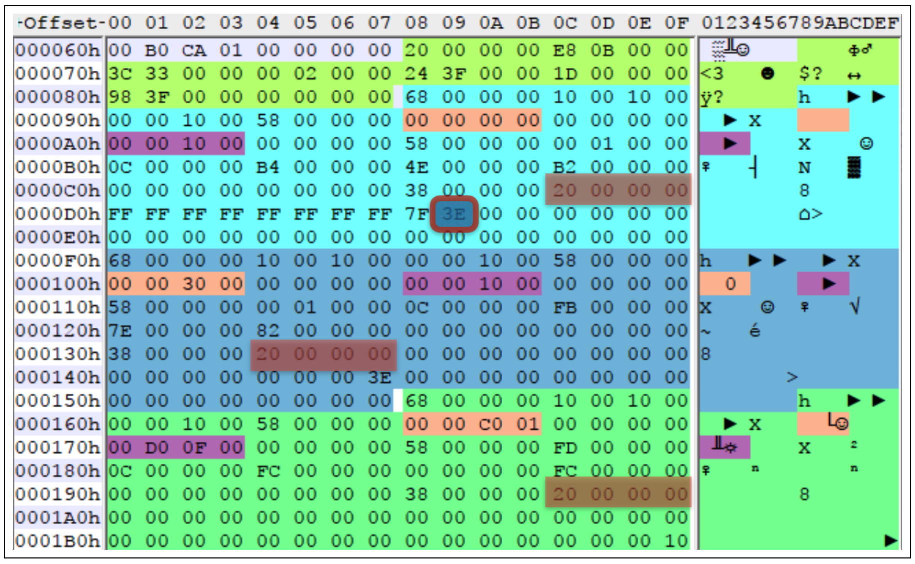
This dump also displays four records. Every record provides a parent ID and a child ID, this way it shows the dependencies between objects. Analyzing partitions with a plenty of objects, this file provides the information necessary to rebuild the directory structure. Visualizing the directory structure, a tree structure will arise. In my eyes this file should be named $OBJECT_TREE.

4.2.2 Allocation

MICROSOFT claimed, that they have used three different allocator files, deviating in the granularity of mapped areas [7], but they have not named these files. The names used below are provided by ENCASE.

$Allocator_Lrg

The $ALLOCATOR_LRG allocates 64 MiB clusters, so it is used for storing large amounts of data with less fragmentation. Below a screenshot from a $ALLOCATOR_LRG is provided.



```
-Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000060h 00 B0 CA 01 00 00 00 00 20 00 00 00 E8 0B 00 00  ░╨☺         Φ♂
000070h 3C 33 00 00 00 02 00 00 24 3F 00 00 1D 00 00 00 <3   ☻   $?  ↔
000080h 98 3F 00 00 00 00 00 00 68 00 00 00 10 00 10 00 ÿ?      h   ► ►
000090h 00 00 10 00 58 00 00 00 00 00 00 00 00 00 00 00   ► X
0000A0h 00 00 10 00 00 00 00 00 58 00 00 00 00 01 00 00   ►     X    ☺
0000B0h 0C 00 00 00 B4 00 00 00 4E 00 00 00 B2 00 00 00 ♀   ┤   N   ▓
0000C0h 00 00 00 00 00 00 00 00 38 00 00 00 20 00 00 00         8
0000D0h FF FF FF FF FF FF FF FF 7F 3E 00 00 00 00 00 00         ⌂>
0000E0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000F0h 68 00 00 00 10 00 10 00 00 00 10 00 58 00 00 00 h   ► ►   ► X
000100h 00 00 30 00 00 00 00 00 00 00 10 00 00 00 00 00   0       ►
000110h 58 00 00 00 00 01 00 00 0C 00 00 00 FB 00 00 00 X    ☺  ♀   √
000120h 7E 00 00 00 82 00 00 00 00 00 00 00 00 00 00 00 ~   é
000130h 38 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 8
000140h 00 00 00 00 00 00 00 3E 00 00 00 00 00 00 00 00        >
000150h 00 00 00 00 00 00 00 00 68 00 00 00 10 00 10 00         h   ► ►
000160h 00 00 10 00 58 00 00 00 00 00 C0 01 00 00 00 00   ► X     └☺
000170h 00 D0 0F 00 00 00 00 00 58 00 00 00 FD 00 00 00  ╨☼     X   ²
000180h 0C 00 00 00 FC 00 00 00 00 00 00 00 FC 00 00 00 ♀   ⁿ       ⁿ
000190h 00 00 00 00 00 00 00 00 38 00 00 00 20 00 00 00         8
0001A0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001B0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10                ►
```

Figure 10: Dump from the record area of the $ALLOCATOR_LRG

PART	OFFSET	rel. Offset	LENGTH	DESCRIPTION
Node Header	0x68	0x00	0x20	length of Header
	0x6C	0x04	0x04	offset to next free record
	0x70	0x08	0x04	free space in the node
	0x74	0x0C	0x04	
	0x78	0x10	0x04	offset to first pointer
	0x7C	0x14	0x04	amount of pointers in this node
	0x80	0x18	0x08	offset to end of node
Record (rel. offset)		0x00	0x10	Record Header
		0x00	0x04	Length of the structure
		0x10	0x08	Starting ENTRYBLOCK described in this record
		0x18	0x08	Number of ENTRYBLOCKS described in this record
		0x20	0x08	????
		0x28	0x08	????
		0x30	0x08	????
		0x38	0x08	????
		0x40	0x04	Offset to bitmap, starting after the Header
		0x44	0x04	Length of the Allocation Table
		0x48	0x20	Allocation Table

Table 2: Breakdown of the $ALLOCATOR_LRG

The dump shows the node header (see 4.3) and the first three records of the $ALLOCATOR_LRG. In the node header, at offset 0x7C, the amount of records within the allocator file is provided, here the value is 0x1D, decimal 29. Colored in dark yellow, the starting offset for the allocation map is provided. So the first record, colored in light blue, describes the first part of the partition, because the starting offset is zero, offset 0x98. Colored in purple, the amount of ENTRYBLOCKS is provided, in case of the first record this value is 0x100000 (LE), decimal 1048576. Colored in brown the amount of bytes, used for the allocation table, is provided. Using this knowledge, the second record, colored in dark blue and starting at offset 0xF0, has to describe another part of the partition, containing some data. The last record, colored in green and starting at offset 0x158, describes the end of the partition. According to the purple colored value at offset 0x170 this record does not describes the full amount of ENTRYBLOCKS. In the allocation table of the last record, the value 0x10 is displayed, this might be explained because of the allocation of the BACKUP VBR and some system files at the end of the partition. Using the values, provided by the $ALLOCATOR_LRG, some calculations to the allocated data can be done. The value at record offset 0x18 (colored purple) provides the amount of allocated ENTRYBLOCKS, in the first two records this value is 0x100000 (read as little Endian), decimal 1048576. The size of an ENTRYBLOCK is 16 kiB (see 4.3). Multiplying the amount of ENTRYBLOCKs with its size, one record allocates 16 GiB of data.

$$\begin{aligned}\textsc{AllocatedData / Record} &= amount\ \textsc{EntryBlocks}\ x\ size\ \textsc{EntryBlock}\\ \textsc{AllocatedData / Record} &= 1.048.576\ x\ 16\ kiB = 1.048.576\ x\ 16.384\ B\\ \textsc{AllocatedData / Record} &= 17.179.869.184\ B = \underline{\underline{16\ GiB}}\end{aligned}$$

If a record allocates 16 GiB of data, and therefor using 0x20, decimal 32, allocation bytes (see brown colored value at record offset 0x44, read as little Endian), a single allocation byte maps 512 MiB, a single allocation bit maps 64 MiB.

$$\begin{aligned}\textsc{MappedData / Byte} &= \textsc{AllocatedData / Record}\ \div size\ \textsc{AllocationTable}\\ \textsc{MappedData / Byte} &= 16\ Gib\ \div\ 32 = 0,5\ GiB = \underline{\underline{512\ MiB}}\\ \textsc{MappedData / bit} &= \textsc{MappedData / Byte}\ \div\ 8\\ \textsc{MappedData / bit} &= 512\ MiB\ \div\ 8 = \underline{\underline{64\ MiB}}\end{aligned}$$

Also the maximum size of the partition can be calculated by multiplying the ALLOCATEDDATA / RECORD by the amounts of records. The latter value was already mentioned at the beginning of this section, here it is 0x1D, decimal 29.

$$\begin{aligned}\textsc{Max. Partition Size} &= \textsc{AllocatedData / Record}\ x\ amount\ records\\ \textsc{Max. Partition Size} &= 16\ GiB\ x\ 29 = \underline{\underline{464\ GiB}}\end{aligned}$$

The calculation above only provides the maximum partition size, because at least the last record does not allocates the full amount of 1.048.576 ENTRYBLOCKs, instead it allocates only 0xFD000, decimal 1.036.288 ENTRYBLOCKS (record offset 0x18 of the green record, colored in purple).

But how does the allocation work? To explain it on an example, I will use the red surrounded value in figure 10 (first record, record offset 0x51), here it is 0x3E. Because it is the tenth allocation byte, the first ENTRYBLOCK allocated by this allocation byte is ENTRYBLOCK 294.912.

$$\textsc{Starting EntryBlock} = \frac{amount\ \textsc{EntryBlocks}}{size\ \textsc{Allocation Table}}\ x\ (\textsc{Allocation / Byte} - 1)$$

$$\textsc{Starting EntryBlock} = \frac{1048576}{32}\ x\ (10 - 1)$$

$$\textsc{Starting EntryBlock} = \underline{\underline{294.912}}$$

To make it more decriptive, there are 9 allocation bytes before, each allocating 512 MiB, so there are 4,5 GiB of allocated data before the data, I want to show my example on.

The allocation byte 0x3E (which is provided in figure 10 in the light blue record) has to be broken into its nibbles, and they have to be written in their binary notation. To get the continuous flow, the bits have to be reordered (the notation is given in Little Endian, so it has to be read starting with the smallest value, which is located on the right side).

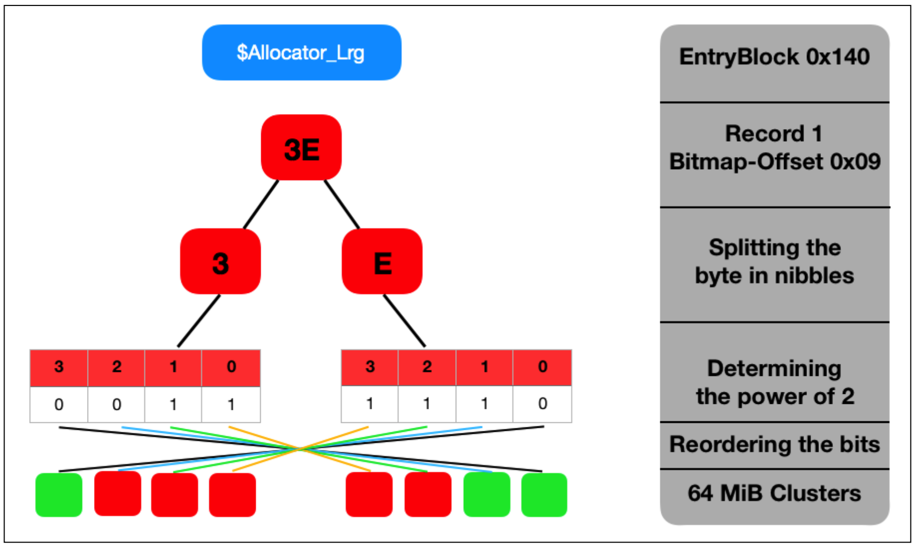


Figure 11: How does allocation work

According to the figure above, the first 64 MiB cluster (on the left side after reordering the bytes) is marked unallocated (colored in green), the five next 64 MiB clusters are marked as allocated (colored in red), the both last 64 MiB clusters (on the right side after reordering the bytes) are marked unallocated (colored in green). There is to mention, that allocated means, there have to be some data in the cluster, not necessarily the full amount of 64 MiB. In contrast to that, unallocated means that there is absolutely no data in the cluster, so it can be used for allocation by the $ALLOCATOR_LRG, while free space in the allocated clusters can be allocated using the other allocation files.

$Allocator_Med

The $ALLOCATOR_MED system file is used for allocation of 64 kiB clusters, which is the cluster size for data streams in REFS (see section 4.3). So it is used to avoid cluster slack because of the big cluster size of the $ALLOCATOR_LRG. Because the structure and the functionality equals the $ALLOCATOR_LRG 4.2.2, it will not be explained here in a detailed way.

$Allocator_Sml

The last allocation file is the $Allocator_Sml, which is used for allocating 16 kiB clusters, which is the cluster size for the system files in ReFS (see section 4.3). It minimizes the cluster slack because of the bigger sizes of the allocated areas of the $Allocator_Lrg and $Allocator_Med. Again, for a detailed explanation see section 4.2.2.

Interaction between the Allocator Files

If there is need to find free space for data on a partition formatted with ReFS, the first view should be in the $Allocator_Lrg. This is the most efficient way, especially if a great amount of data has to be allocated. This is intended to prevent fragmentation of data, which should increases the performance of the file system.

If the data which should be allocated use less than 64 MiB, the $Allocator_Med has to be used. Even if a special area of the partition is marked as allocated in the $Allocatr_Lrg, there might be space for smaller data.

Because data streams were stored in 64 kiB clusters and metadata were stored in 16 kiB clusters, the allocation of the latter utilizes the $Allocator_Sml. Like it was described before, even if a special area on the disk is marked as allocated within the $Allocator_Lrg and $Allocator_Med, there might be enough place for the metadata.

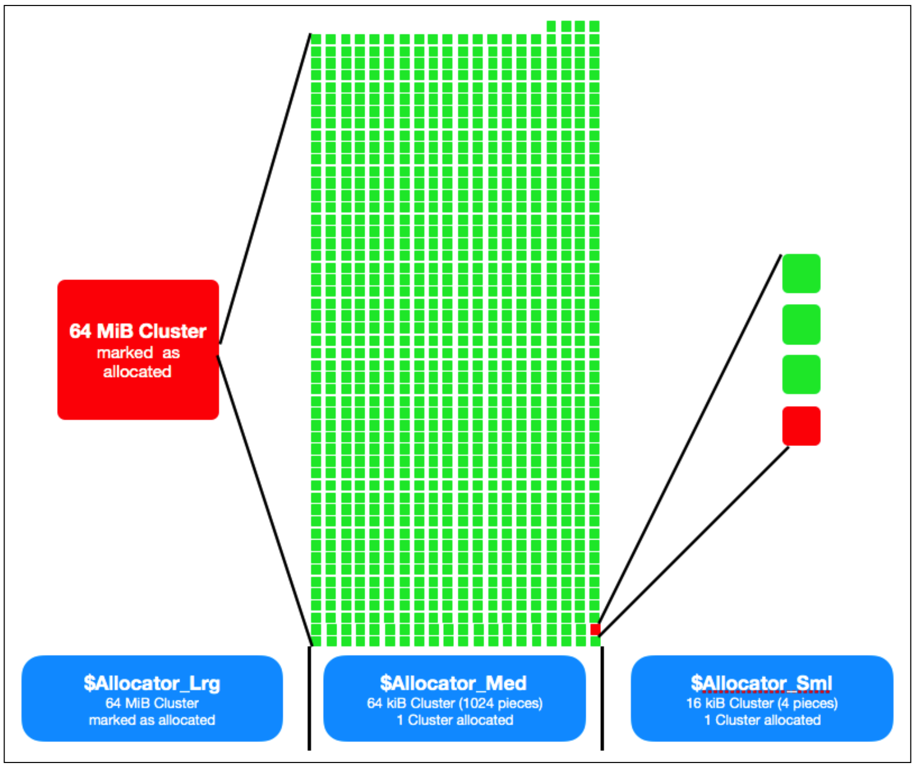


Figure 12: Interaction between the $ALLOCATOR files

The link between the $ALLOCATOR files is illustrated in figure 12. It should visualize, that a cluster can be marked as allocated (colored as red) in the $ALLOCATOR_LRG even if there is only a 16 kiB cluster used for allocating metadata (colored in red on the right side of the illustration). With other words, there are 1023 clusters marked unallocated (colored in green) in the $ALLOCATOR_MED and can be used for storing data. This was already observed during my research on the empty file system, because there were several clusters marked as allocated in the $ALLOCATOR_LRG.

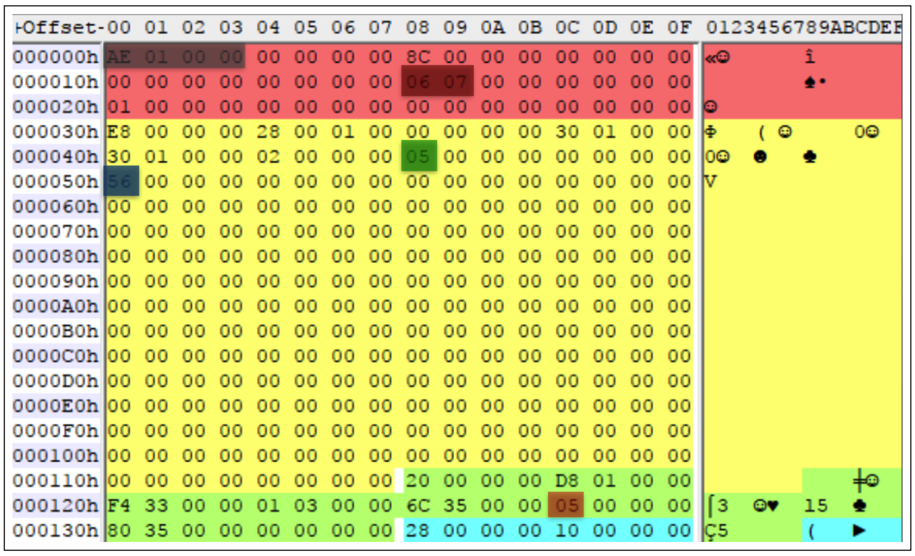
4.2.3 Attributes

The $ATTRIBUTE_LIST was not part of my master thesis, so there have to be done some future work on it.

4.3 EntryBlock

4.3.1 Starting Area

MICROSOFT claimed, that they have used 64 kiB structures, which equals the value, provided and calculated by the VBR A.2.1. But that is, as I experienced during my research, not the whole truth. MICROSOFT uses 64 kiB structures for storing data streams, while using 16 kiB structures for storing system files and metadata. This 16 kiB structures were named by me as ENTRYBLOCKS, to avoid confusion. Below, a typical structure of an ENTRYBLOCK is provided.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h AE 01 00 00 00 00 00 00 8C 00 00 00 00 00 00 00 «☺      î       
000010h 00 00 00 00 00 00 00 00 06 07 00 00 00 00 00 00         ♠•      
000020h 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺               
000030h E8 00 00 00 28 00 01 00 00 00 00 00 30 01 00 00 Φ   ( ☺     0☺  
000040h 30 01 00 00 02 00 00 00 05 00 00 00 00 00 00 00 0☺  ☻   ♣       
000050h 56 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 V               
000060h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000070h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000080h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000090h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000A0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000B0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000C0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000D0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000E0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000F0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000100h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000110h 00 00 00 00 00 00 00 00 20 00 00 00 D8 01 00 00             ╪☺  
000120h F4 33 00 00 01 03 00 00 6C 35 00 00 05 00 00 00 ⌠3  ☺♥  l5  ♣   
000130h 80 35 00 00 00 00 00 00 28 00 00 00 10 00 00 00 Ç5      (   ►   
```

Figure 13: Starting of an ENTRYBLOCK

The first 0x30 bytes, colored in red, describe the ENTRYBLOCK, so I named this structure ENTRYBLOCK DESCRIPTOR, starting with the ENTRYBLOCK number (colored in brown, offset 0x00). The ENTRYBLOCK number is an increasing number across the whole partition, even if an ENTRYBLOCK is unused or used for storing data streams. The next value (colored in red, offset 0x18) provides the node ID.

Colored in yellow, this structure is named by me as NODE DESCRIPTOR, because it describes the node structure. The size of this structure can vary, I have recognized sizes between 8 bytes to 232 bytes, while the first value (offset 0x30, 4 bytes, read as little Endian), here 0xE8, provides the length of the structure. The value colored in green (offset 0x48), here 0x05, provides the amount of the extents of this special node.

Extents (1.3) are used by the file system, if an ENTRYBLOCK runs out of space. This is illustrated exemplary in figure 14. The amount of records within an ENTRYBLOCK is limited, because of the size of 16 kiB of an ENTRYBLOCK. In the case, the ENTRYBLOCK contains the metadata for a folder and there are added a plenty of files (e.g. pictures from the last holiday in a dedicated folder, named like the location of the holidays), the capacity of an ENTRYBLOCK easily can be exceeded. On the left side of figure 14 an ENTRYBLOCK with six existing records is illustrated. Another seven records have to been added, which exceeds the capacity of the illustrated ENTRYBLOCK

Hence, the original ENTRYBLOCK will no longer stores any records, it will only stores the offsets to the new ENTRYBLOCKs, which store all the records. With other words, the original ENTRYBLOCK becomes a parent node, the metadata are stored in its new child nodes.

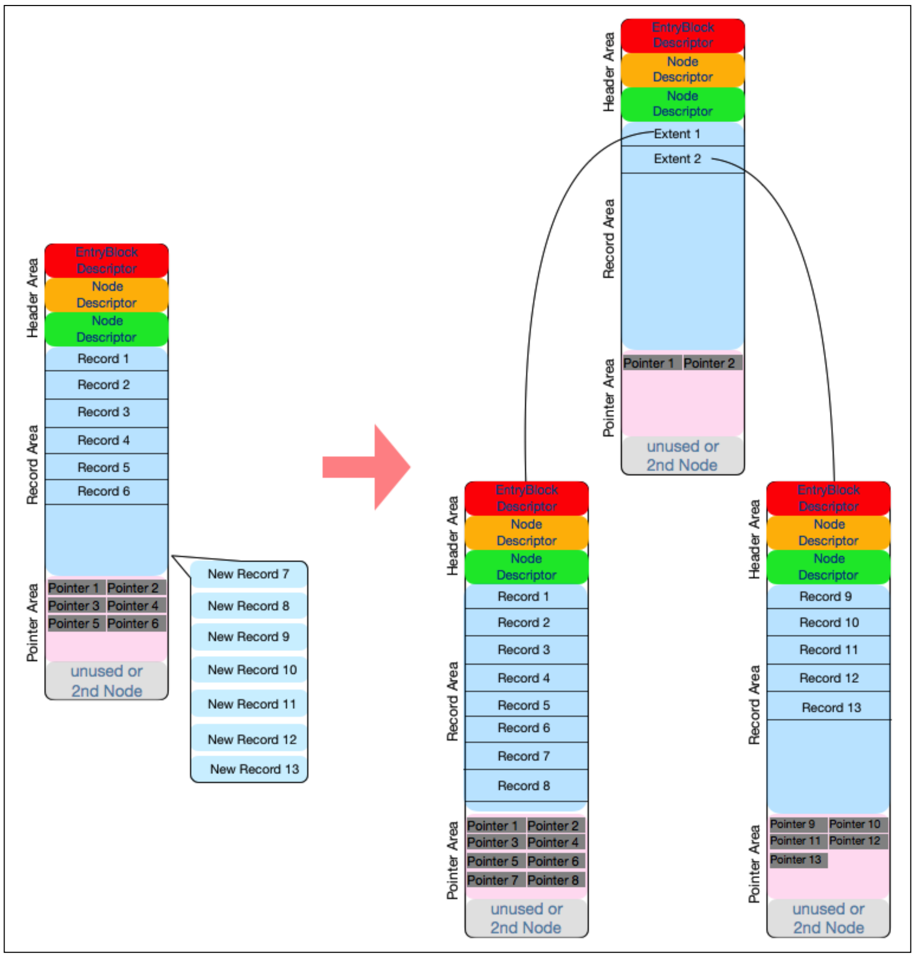


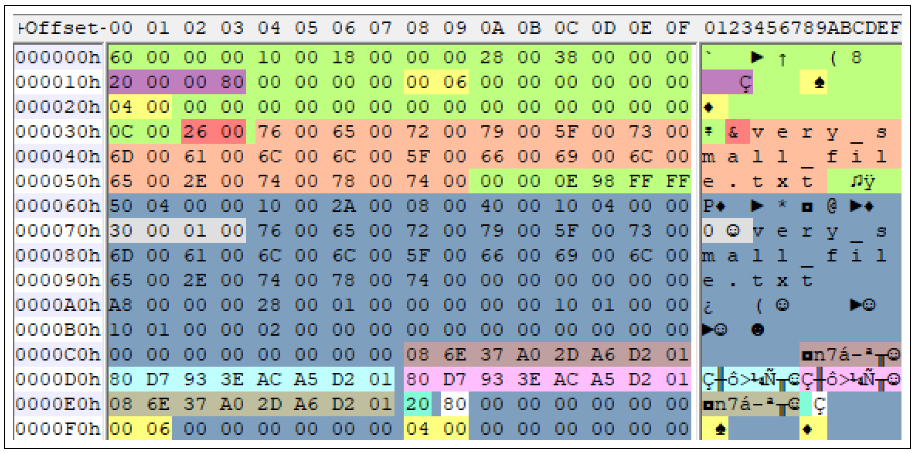
Figure 14: Visualization for the usage of extents

Coming back to the breakdown of the ENTRYBLOCK and figure 13, this node only contains the offset to the extents. Colored in blue, offset 0x50, the amount of records in this node (with its extents) is provided. Here the value is 0x56, decimal 86, which is simultaneously the reason for the extents, because an ENTRYBLOCK is running out of space containing 86 records with an approximately record size of 1000 bytes.

Colored in green, this structure is named by me as NODE HEADER, based on the nomenclature of nodes in B-TREES. Important values are the size of the structure (relative offset 0x00), the offset to the next free record (relative offset 0x04), the amount of free space in this node (relative offset 0x08), the offset to the pointers (relative offset 0x10), the amount of pointers (relative offset 0x14) and the offset to the end of the node (relative offset 0x18). As it was mentioned in connection with the extents, this node contains five records, the NODE HEADER provides the offset to the pointers, pointing to these five records. It is requisite to use these pointers, because the records are not necessarily ordered continuous in the node, sometimes I could observe invalid records between the valid ones. The offsets provided by the NODE HEADER are offsets relative to the start of the NODE HEADER.

4.3.2 Record Area

Below a screenshot of a part of the record area of the node 0x600 is provided. In all partitions I have analyzed, the node 0x600 was the root directory.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
000000h 60 00 00 00 10 00 18 00 00 00 28 00 38 00 00 00  `   ► ↑   ( 8
000010h 20 00 00 80 00 00 00 00 00 06 00 00 00 00 00 00     Ç     ♠
000020h 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ♦
000030h 0C 00 26 00 76 00 65 00 72 00 79 00 5F 00 73 00  ♀ & v e r y _ s
000040h 6D 00 61 00 6C 00 6C 00 5F 00 66 00 69 00 6C 00  m a l l _ f i l
000050h 65 00 2E 00 74 00 78 00 74 00 00 00 0E 98 FF FF  e . t x t   ♫ÿ
000060h 50 04 00 00 10 00 2A 00 08 00 40 00 10 04 00 00  P♦  ► * ◘ @ ►♦
000070h 30 00 01 00 76 00 65 00 72 00 79 00 5F 00 73 00  0 ☺ v e r y _ s
000080h 6D 00 61 00 6C 00 6C 00 5F 00 66 00 69 00 6C 00  m a l l _ f i l
000090h 65 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00  e . t x t
0000A0h A8 00 00 00 28 00 01 00 00 00 00 00 10 01 00 00  ¿   ( ☺     ►☺
0000B0h 10 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00  ►☺  ☻
0000C0h 00 00 00 00 00 00 00 00 08 6E 37 A0 2D A6 D2 01          ◘n7á-ª╥☺
0000D0h 80 D7 93 3E AC A5 D2 01 80 D7 93 3E AC A5 D2 01  Ç╫ô>¼Ñ╥☺Ç╫ô>¼Ñ╥☺
0000E0h 08 6E 37 A0 2D A6 D2 01 20 80 00 00 00 00 00 00  ◘n7á-ª╥☺ Ç
0000F0h 00 06 00 00 00 00 00 00 04 00 00 00 00 00 00 00   ♠       ♦
```

Figure 15: Record Area of an ENTRYBLOCK

Light green is the child attribute, 0x60 bytes in size, the size is provided by the value at (relative) offset 0x00. Colored purple is the attribute identifier for the child attribute (relative offset 0x10), colored yellow are the parent ID (relative offset 0x18) and the child ID (relative offset 0x20). Another value provided by the child attribute is the length of the filename, colored in red (2 bytes, read as little Endian, 0x0026), the file name starts right behind this value and is given in unicode (colored in orange).

Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. [28]

The filename has a length of 0x26 bytes. Every character is defined using 2 bytes. Hence the filename consists of 0x13, decimal 19 characters. This means it starts with 0x7600 (not reversed) at offset 0x34.

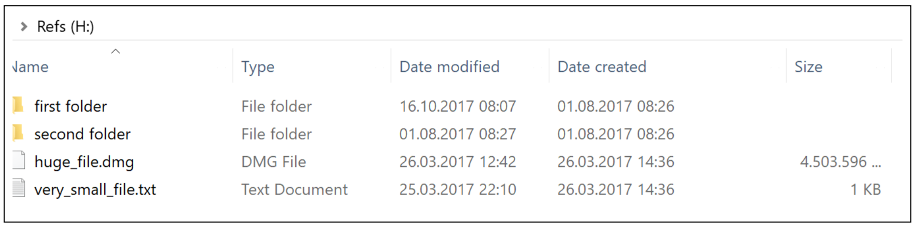
The blue area is the filename attribute, with a size of 0x450 (value provided at relative offset 0x00), with the attribute identifier colored in grey (relative offset 0x10).

Because I have started analyzing the records for the HUGE_FILE.DMG and the VERY_SMALL_FILE.TXT I found the attribute identifier 0x030000100 first. In the further analyzes I also found the attribute identifier 0x30000200. Examining the differences in the records with the differing attribute identifier it gets obvious, that files owned the attibute identifier 0x30000100 and folders the identifier 0x30000200. According to this finding I set up a new working hypothesis:

Files own the attribute identifier 0x30000100 and folders own the attribute identifier 0x30000200.

In order to check, if this hypothesis is true, I prepared a new partition and copied several files in different folders and subfolders to it. According to the analysis of this partition I can state, that the hypothesis could be strengthened. I also checked other partitions, created at a later time, I could not find any deviations from this hypothesis. This finding was also implemented into my code. Analyzing several partitions with this code, this hypothesis could be strengthened. At the end I can state, that every examination of one of the several REFS partitions has strengthened this hypothesis.

Other values provided are the timestamps for "created" (colored in brown), "modified" (colored in light blue), "metadata modified" (colored in pink) and "last accessed" (colored in olive-green), starting at relative offset 0x68. The timestamps are given in WINDOWS filetime, counting hundred nanosecond intervals, starting at 1st of January 1601 [29]. Converting the timestamps into human readable format, the created and the last accessed time is 2017-03-26 12:36:43 Sun UTC and the modified and metadata modified time is 2017-03-25 21:10:35 Sat UTC. Below a screenshot from the file explorer is provided.



> Refs (H:)

Name	Type	Date modified	Date created	Size
first folder	File folder	16.10.2017 08:07	01.08.2017 08:26	
second folder	File folder	01.08.2017 08:27	01.08.2017 08:26	
huge_file.dmg	DMG File	26.03.2017 12:42	26.03.2017 14:36	4.503.596 ...
very_small_file.txt	Text Document	25.03.2017 22:10	26.03.2017 14:36	1 KB

Figure 16: Screenshot from a file explorer, providing the timestamps for the very_small_file.txt (timestamps are provided in CET - 25.03.2017 - and CEST - 26.03.2017)

Furthermore, the FNA contains the flags. 0x20 (colored in turquois, relative offset

0x88) is the archive flag. Examining this value at 0xE8 (relative offset 0x88 of the record) from different files on a binary level, the scheme behind is obvious.

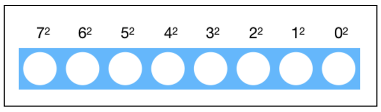


Figure 17: FLAGS-scheme, no flags, value=0

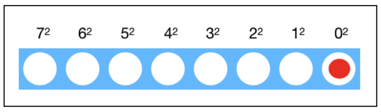


Figure 19: FLAGS-scheme, read-only flag, value=1

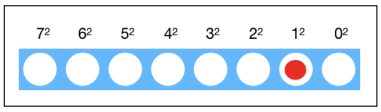


Figure 18: FLAGS-scheme, hidden flag, value=2

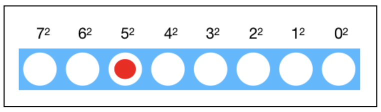


Figure 20: FLAGS-scheme, archive flag, value=20

VALUE	DESCRIPTION
0x00	no flags
0x01	read only
0x02	hidden
0x20	archive

Table 3: Flags

Of course, a file can own various flags, so the values for the different flags can be cumulated. These found flags correspond to the flags already known from NTFS [30].

At offset 0xE9 (relative offset 0x89), colored in white, the value 0x80 is provided, which is a value I discovered at the end of my master thesis, handling storage spaces and resiliency. It is not proven up to now, but it is probably a flag for checksums. Colored in yellow, the parent ID 0x600 (relative offset 0x90) and the child ID 0x04 (relative offset 0x98) are displayed. Hence this file is a child of the root directory. Not displayed here are the data runs of the files, which are included in the FNA.

5 Discussion

In this chapter I will discuss the hypothesis' named in the research question section 1.5. The hypothesis' will be discussed in the same order, as they are named.

Hypothesis$_1$: *The* VBR *provides the offset to the main structure of the file system.*

Examining NTFS and EXFAT volumes, the VBR provides the offset to the main structure directly by naming the offset to the MASTERFILETABLE respectively to the FILEALLOCATIONTABLE[30]. Accordingly I tested the opposite hypothesis to H$_1$, which is as follows:

The VBR *does not provide the offset to the main structure of the file system*

As I described in the VBR section 4.1, the VBR provides the offset to the end of the addressed space of the partition. The last sectors of the addressed space contain the BACKUP VBR and the offset to the $TREE_CONTROL. Interpreted strictly, the VBR does not contain the offset to the main structure, it only contains the offset to the pointers to the main structure. The VBR only can be used as a detour for finding the root node. Hence the opposite hypothesis has been strengthened, the H$_1$ has been rejected.

Hypothesis$_2$: *The main structure has a fixed position within the file system.*
opposite H$_2$: *The main structure does not have a fixed position within the file system.*

In every partition I have examined I have found the offset to the $TREE_CONTROL in the ENTRYBLOCK 0x1E, decimal 30. This structure contains two pointers to the $TREE_CONTROL, the actual and the previous one, which is also caused by the copy-on-write feature of REFS (If some of the content has to be changed, a new copy with the modified content will be created. Hence the last working version will be stored and can be used in case of a writing error of the new version.) Accordingly it is not practicable to assign the $TREE_CONTROL a fixed position on the partition, REFS solved this by providing the offsets to the main structure,

the $TREE_CONTROL, on a fixed position.
For forensic purposes this fixed position of the pointers has some advantages. Even if the VBR is deleted, probably as an attempt of a suspect to prevent law enforcement authorities from finding evidences, the content of the file system can be restored using the pointers to the root node. And because of the second location of the pointers at the end of the partition, there is another protection in case of one of the locations of the pointers is also deleted.
Considering the copy-on-write feature, I reject the opposite hypothesis, the hypothesis H_2 has been strengthened, even if only the offsets to the $TREE_CONTROL are provided on a fixed position, not the $TREE_CONTROL itself.

Hypothesis$_3$: *The root node contains offsets to nodes, matching the system files provided by* ENCASE.
opposite H_3: *The root node does not contain offsets to nodes, matching the system files provided by* ENCASE.

The root node, named $TREE_CONTROL by ENCASE, contains the ENTRYBLOCK offsets to six system files, at least on the partitions I have examined. All these six system files ($OBJECT, $OBJECT_TREE, $ALLOCATOR_LRG, $ALLOCATOR_MED, $ALLOCATOR_SML, $ATTRIBUTE_LIST) are also provided by ENCASE. Hence the opposite H_3 has to be rejected, the hypothesis H_3 has been strengthened.

Hypothesis$_4$: *All the system files provided by* ENCASE *have their own child node.*
opposite H_4: *Not every system file provided by* ENCASE *has its own child node.*

Beside the six system files named above (Hypothesis$_3$), ENCASE provides seven further system files ($BOOT, $SDS DATA, $SECURITY DESCRIPTOR STREAM, $UPCASE, $VOLUME DIRECT IO FILE, $VOLUMEINFO, $VOLUME LABEL), see [13]. These system files are contained in single ENTRYBLOCKS, all the offsets to the ENTRYBLOCKS could not be discovered until now. At this point of my research I have to reject the opposite H_4, the hypothesis H_4 has been strengthened.

Hypothesis$_5$: *The structures within the* ENTRYBLOCK*s are comparable to the attributes, known from* NTFS.
opposite H_5: *The structures within the* ENTRYBLOCK*s are not comparable to the attributes, known from* NTFS.

As it is described in the results chapter 4.3, I discovered two attributes, the CHILD ATTRIBUTE and the FILENAME ATTRIBUTE. While the CHILD ATTRIBUTE provides the file name of the child object, the FILENAME ATTRIBUTE provides timestamps, the data runs and the flags. Compared to the attributes used in NTFS, REFS seems to concatenate several attributes to the attribute, I named FILENAME ATTRIBUTE. At the time of writing this master thesis not all of the attributes known from NTFS could be identified, some further work has to be done here.

Summing up, REFS also uses attributes, which can be identified by their attribute identifier. Some of the attributes known from NTFS could be identified as a part of the FNA, used by REFS, hence the opposite H_5 has to be rejected, the hypothesis H_5 has been strengthened.

Hypothesis$_6$: *Files and folders have the same records, they are treated the same way.*
opposite H_6: *Files and folders do not have the same records, they are not treated the same way.*

In the volumes I have analyzed, a file owned two records, the CHILD ATTRIBUTE and the FILENAME ATTRIBUTE, while folders only owned the FILENAME ATTRIBUTE. The FNA of folders also missed the data run, the parent ID and the child ID, it only provides the timestamps and the offset to the metadata for the folder. Hence the opposite H_6 has been strengthened, the hypothesis H_6 have to be rejected, because files and folders own different records and are treated in a different way.

While examining the records for files, I found the attribute identifier 0x30000100. Accordingly the working hypothesis here was:

Hypothesis H_7: *Files and folders have the same attribute identifier 0x30000100.*
opposite H_7: *Files and folders have different attribute identifiers.*

As it is described in the results chapter 4.3.2 I prepared a special partition, containing a plenty of files and folders. Analyzing this prepared partition, I discovered that every file in this partition had the attribute identifier 0x30000100, but folders had the attribute identifier 0x30000200. This finding has strengthened the opposite H_7, the working hypothesis H_7 was rejected. Because of this result, the value 0x30000100 has to be the attribute identifier for files, while the value 0x30000200 has to be the attribute identifier for folders.

6 Conclusions

In the introduction chapter a lot of problems were made a subject of discussion, first of all the missing documentation for the file system REFS and the related requirement of reverse engineering the file system for forensic needs. I named different objectives and formulated the research questions. After providing the results of my master thesis, this chapter should be used for summing up, if I reached these objectives.

In the results chapter I have provided my results to the principles of data treating and data storing by the file system REFS. Beyond the scope of the master thesis, I also provided several templates for a detailed analysis of REFS in the appendix. These results contain information to such important structures like the VBR, the system files and the structure I named ENTRYBLOCK. On the other hand side, a lot of functions (e.g. checksums and data integrity) of the file system could not be examined, mostly because of a lack of time and the missing documentation for the file system.

Nevertheless, the provided information should enable the reader to comprehend the very basic structure of MICROSOFT's new file system. My research is a necessary first step to compensate the missing documentation, and while this first step was done, other steps has to follow.

6.1 Research Questions

research question Q_1: Does REFS has a VBR like other file systems, eg. NTFS?

As it is shown in the results chapter, REFS utilizes a VOLUMEBOOTRECORD, some details, e.g. to the sector and cluster size, are provided. The VBR also contains the FILESYSTEM RECOGNITION STRUCTURE (FSRS), which enables an operating system to accept also an unknown file system, and the checksum of the VBR. Compared to other file systems, REFS is currently not intended to be bootable, hence the VBR does not contain boot code and utilizes only 64 bytes.

research question Q_2: Which kind of information does the root node contain?

The root node provides the offsets to six system files, these system files contain valuable information to the objects contained in the file system and the used space. Evaluating the contained information, the basic structure of the REFS partition is disclosed and a starting point for a manual analysis of the file system is gained.

research question Q$_3$: Which kind of information are stored in the ENTRYBLOCK? How are the information organized within a single ENTRYBLOCK?

The ENTRYBLOCK contains the metadata. At the beginning, information to the ENTRYBLOCK itself are provided, e.g. to the parent node, possible extents, the offset to the pointer area and the amount of pointers. The record area contains records, these provide information to the files and folders stored on the partition. While a record for a folder only contains reduced data, a record for a file contains data, organized similar to the attributes known from NTFS.

research question Q$_4$: Does the self coded tool prove the findings? Are the results only by accident or does the tool provide results repetitive, on different partitions with different content?

In the context of the limited results of my research (I only have analyzed a limited number of specially prepared partitions), my tool fulfills exactly, what it is expected to. It has analyzed all the prepared REFS partitions automatically, among them partitions with a variety of files, modified flags, providing a variety of metadata information to files and folders. This tool can also be used to extract data from a REFS partition, even if they are marked as deleted.

Evaluating the results of my master thesis, there is one fact to be mentioned. MICROSOFT released a new file system version for REFS. This new version attracted my attention, because my self coded tool, working properly up to this point, failed. A short analysis made it obviously, that a change in the structure of REFS causes the failure. At this point, I already run out of time, so my tool could not be adapted to the new properties. A conclusion of this discovery is, that my tool only can be used on REFS version 1.2. How time consuming the necessary modification will be, can not be predicted precisely.

main research question: Is it possible to verify the results for a REFS -image, provided by ENCASE?

I have started my reverse engineering of REFS already in my specialization project [13], here I was able to gain some information to the ENTRYBLOCK. After starting to use ENCASE I was able to get some structure into the discovered information. My starting point here were the system files, provided and named by ENCASE.

Verifying the provided results has to be parted into two parts, according to my point of view. The first part, the pure data, the hex-code, could be verified easily. I compared the hex-code provided by ENCASE several times with hex-code, provided by X-WAYS, IBORED ([31]) and the dd-command from the terminal. I could not determine any discrepancy all the time. In my eyes, this is a very important finding, because it means, that ENCASE does not modify the data.

The second part, the analysis and interpretation of the data, has to be looked at more detailed. On the one hand side several disagreements between the sparse information from MICROSOFT and the implementation of REFS by MICROSOFT itself and ENCASE were revealed, partly comprehensible, partly not.

- ENCASE provides more system files than correlating ENTRYBLOCKS can be found in a REFS partition, see also [13].
- ENCASE's nomenclature of the system files could not be retraced within the file system.
- ENCASE provides a cluster size of 16 kiB (the clusters are visualized as 16kiB clusters in disk view), although a cluster size of 64 kiB is the default setting for REFS. Furthermore, also the calculated value for the cluster size, based on the values of the VBR, is 64 kiB.

On the other hand side, the information to the stored data, compared to the information provided by my own tool, were correct.

Summing up, I verified the results of the analysis' of REFS images, provided by ENCASE. According to my results, the data were not tampered and the interpretation was correct. Only the visualization of the data, this contains as well the nomenclature as displaying the data in 16 kiB clusters, mismatches my results. At the time of writing, it is not clear, if these disagreements have an influence on the reliability of the file system analysis done by ENCASE, even if an update of the used tool might fix these problems.

6.2 Future Work

To stay within the scope of this master thesis, some further examination of several topics got postponed. Below are some points, further work seems to be important to be done on:

- The nomenclature of the found structures has to be revised to achieve a meaningful and easily understandable nomenclature.
- Do my results fit to the new versions of REFS? What has been changed? In a further work the new file system versions of REFS has to be compared with my results to prove the changes within the newer versions.
- In my research, the data integrity feature of REFS was not examined. Hence the system files $SDS DATA and $SECURITY DESCRIPTOR STREAM were not analyzed up to now. Another system file, the $ATTRIBUTE_LIST was not analyzed. A further work should complete the analysis of the system files.

- Besides the flags for archive, hidden and read-only, I also expect flags for the user rights and the permissions. These were not discovered up to now, so there have to be done further work on this.
- As it is described several times, my research does not claim to be complete. I cross checked my results using partitions with different sizes and with different content. Theoretically, there might be constellations, which do not match my results. Further works should use other partitions, to cover a wider range of possible constellations, which enlarges the reliability of the results.

Bibliography

[1] B-trees. URL: http://btechsmartclass.com/DS/U5_T3.html (Visited 04.03.2018).

[2] Horalek, Sobeslav, & Cimler. 2015. Verifying properties of resilient file system.

[3] Limited, D. T. T. 10 2017. Global mobile consumer survey. URL: https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Technology-Media-Telecommunications/gx-global-mobile-consumer-survey-2nd-edition.pdf (Visited 2017-10-09).

[4] Kemp, S. 10 2017. Welcome to digital in 2017. URL: https://www.slideshare.net/wearesocialsg/digital-in-2017-global-overview?ref=https://wearesocial.com/uk/blog/2017/01/digital-in-2017-global-overview (Visited 2017-10-09).

[5] gs.statcounter.com. 08 2017. Usage operating systems. URL: https://de.statista.com/statistik/daten/studie/157902/umfrage/marktanteil-der-genutzten-betriebssysteme-weltweit-seit-2009/ (Visited 2017-10-09).

[6] Prabhakaran, V., Arpaci-Dusseau, A., & Arpaci-Dusseau, R. Analysis and evolution of journaling file systems. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/general/full_papers/prabhakaran/prabhakaran_html/main.html (Visited 06.04.2018).

[7] Sinofsky, S. Building the next generation file system for windows: Refs (online). URL: https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/ (Visited 23.07.2017).

[8] Microsoft. Understanding microsoft storage spaces. URL: http://www.serversdirect.com/file%20library/libraries/_8_understanding_FINAL.pdf.

[9] Resilient file system (refs) overview. URL: https://docs.microsoft.com/en-us/windows-server/storage/refs/refs-overview.

[10] Head, A. Forensic investigation of microsoft's resilient file system (refs) (online). 04 2015. URL: http://www.resilientfilesystem.co.uk.

[11] Ballenthin, W. The microsoft refs on-disk layout (online). URL: http://www.williballenthin.com/forensics/refs/disk/index.html (Visited 24.07.2017).

[12] Carrier, B. Open source digital forensics tools. URL: http://www.digital-evidence.org/papers/opensrc_legal.pdf (Visited 10.02.2018).

[13] Georges, H. Refs - a re-research,specialization course (imt4883 report). 06 2017.

[14] Userguide ftk 6.1. URL: https://support.accessdata.com/hc/en-us/articles/204056525-FTK-User-Guide.

[15] Microsoft. 11 2013. Resilient filesystem overview. URL: https://technet.microsoft.com/en-us/library/hh831724(v=ws.11).aspx (Visited 2017-03-18).

[16] File system. URL: https://en.wikipedia.org/wiki/File_system (Visited 08.02.2018).

[17] McConnell, M. A beginner's guide to mac file versioning. URL: https://www.makeuseof.com/tag/guide-mac-file-versioning/ (Visited 06.04.2018).

[18] Wikipedia. B-tree. URL: https://en.wikipedia.org/wiki/B-tree (Visited 03.03.2018).

[19] Council, I. D. Data age 2025. URL: https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf (Visited 04.03.2018).

[20] Apple file system. URL: https://en.wikipedia.org/wiki/Apple_File_System (Visited 06.04.2018).

[21] Btrfs. URL: https://en.wikipedia.org/wiki/Btrfs (Visited 06.04.2018).

[22] Hamm, J. Extended fat file system. URL: https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf (Visited 06.03.2018).

[23] Microsoft. Filesystem recognition. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd442652(v=vs.85).aspx.

[24] Microsoft. 2017. Computing a filesystem recognition checksum. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd442649(v=vs.85).aspx.

[25] URL: https://en.wikipedia.org/wiki/Reverse_engineering (Visited 07.02.2018).

[26] Nola, R. 2005. Philosophy, science, education and culture.

[27] (online)URL: https://linux.die.net/man/1/ewfacquire.

[28] Wikipedia - unicode. URL: https://en.wikipedia.org/wiki/Unicode.

[29] Windows filetime. URL: https://en.wikipedia.org/wiki/System_time (Visited 14.02.2018).

[30] Elrick, D. 2014. *Forensic Examination of Windows Supported File Systems*. First edition.

[31] Tempelmann, T. ibored-app. URL: http://apps.tempel.org/iBored/index.php.

[32] (online)10 2017. URL: https://www.sleuthkit.org/sleuthkit/docs.php (Visited 2017-10-09).

[33] (online)URL: https://github.com/aburgh/Disk-Arbitrator (Visited 2017-10-09).

[34] Microsoft. Computing a filesystem recognition checksum (online). 2017. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd442649(v=vs.85).aspx.

[35] Security identifier. URL: https://en.wikipedia.org/wiki/Security_Identifier.

[36] Carvey, H. A. 2014. *Windows Forensic Toolkit: Advanced Analysis Techniques for Windows 8*. Syngress.

[37] Python argparse-package. URL: https://docs.python.org/2/library/argparse.html.

[38] Python os-package. URL: https://docs.python.org/2/library/os.html.

[39] Python datetime-package. URL: https://docs.python.org/2/library/datetime.html.

[40] Python time-package. URL: https://docs.python.org/2/library/time.html.

[41] Python struct-package. URL: https://docs.python.org/2/library/struct.html.

A Appendix

A.1 Preparation

In this section the preparation of the hard disk drive (HDD) and the examination machine is described.

A.1.1 HDD

physical

- HDD WD 320 GB ; Model: WD3200AAKS-00UU3A0
- HDD Seagate 750 GB, Model: Barracuda 7200.12

logical

- 4 partitions were created on the HDD WD 320 GB, using the preparation machine
- In WINDOWS the command line (with escalated privileges) with the command "diskpart" was used for creating the images. This tool was also used to set the second partition as bootable.
- First partition: 204800000 sectors with 104.857.600.000 bytes, 97,7 GiB
- Second partition: 10240000 sectors with 5.242.880.000 bytes, 4,9 GiB
- Third partition: 153600000 sectors with 78.643.200.000 bytes, 73,2 GiB
- Fourth partition: equals the third partition
- Unallocated space: 49,1 GiB
- For control, if formatting succeeded, the fsutil command was used
-

  ```
  $ fsutil fsinfo refsinfo < drivename >
  ```

  According to the output, the REFS version 1.2 was used.
- In another attempt, I used another HDD (Seagate Barracuda) and listed the possible options for formatting REFS, using the $ filesystems -option of diskpart. The purpose was to change the cluster-size. According to the output, the default cluster-size for REFS is 64KiB, and this is also the only possible option. Trying to change the cluster-size to 32kiB or 128kiB resulted in an error message.

A.1.2 Preparation Machine

Host

- Lenovo IdeaPad Z500, 8GB RAM, 320 GB HDD, i5 3230M (4x2.6 GHz), Linuxmint 18.1, Mate 64-bit
- Virtualization tool: VirtualBox 5.1.22 (r115126)

Guest

- Win10 educational (OS-Version: 10.0.14393), 2 cores, 4 GB Ram

A.1.3 Examination Machine

Host

- MacBook Air, 13-inch, mid 2013, 8 GB Ram, 512 GB SSD, i7, MacOsX 10.12.3
- Imaging tool FTK-Imager 2.9 for Mac (GUI version 1.0)
- Virtualization tool: Parallels (v12.1.3)
- Hex-Viewer iBored (v1.1.19)
- FileMerge (Mac-On-Board)
- WxHexEditor Mac (v.023-beta)

Guest

1. - Win10 educational, 2 cores, 4 GB Ram, installed on a 128 GB Jetdrive Light SD
   - Examining tool: x-ways forensics 18.7 - 7
   - Encase, v. 7.11.01.05
   - WxHexEditor (NCFI-version)
2. - Linuxmint 18.1, Mate 64-bit, 1 core, 4GB Ram,
   - Imaging tool - ewfacquire [27]
   - Forensic tool - sleuthkit [32]

A.1.4 Other Hardware / Software

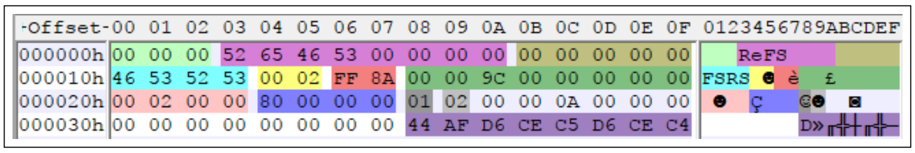
- Sharkoon Drivelink Combo USB-Sata adapter with integrated write-adapter. This was tested for correct function prior to use.
- DiskArbitrator: software write-blocker, was also tested for correct function prior to use)

The partitions were created without mounting the drive. After creating and formatting, partition 2 (4,9 GB) was defined as the active partition.
After creating the 4 partitions, I connected the disk via the Sharkoon Drivelink Combo (activated write-blocker) and activated DiskArbitrator (read-only modus) [33]. Using FTK-Imager I made an image of the whole disk, titled "emptyrefs.E01". Because of enabled compression (level 9) the image size is 586,7 MB.

A.2 Detailed Results

A.2.1 VBR

In the scope of this master thesis I have not analyzed the VOLUMEBOOTRECORD (VBR), this was already done in my specialization project ([13]). But for the sake of completeness, my findings for the VBR are provided below.



```
-Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 00 00 00 52 65 46 53 00 00 00 00 00 00 00 00 00    ReFS
000010h 46 53 52 53 00 02 FF 8A 00 00 9C 00 00 00 00 00 FSRS ☻ è  £
000020h 00 02 00 00 80 00 00 00 01 02 00 00 0A 00 00 00  ☻  Ç   ☺☻  ◙
000030h 00 00 00 00 00 00 00 00 44 AF D6 CE C5 D6 CE C4         D»╓╬┼╓╬─
```

Figure 21: VBR from the second partition, colors named in the table below, Table 4

Offset	Length	Description	Colour
0x00	3	Jump Instruction	light green
0x03	8	FileSystemName	light purple
0x07	5	Reserved space, containing all zeros	olive
0x10	4	Structure Identifier	light blue
0x14	2	Number of bytes in the VBR	yellow
0x16	2	Checksum for the FSRS	red
0x18	8	Offset to BACKUPVBR	dark green
0x20	4	Bytes per Sector	light red
0x24	4	Sectors per Cluster	blue
0x28	1	Filesystem Major Version	darker grey
0x29	1	Filesystem Minor Version	lighter grey
0x2A	2	unknown	
0x2C	4	unknown	
0x30	8	unknown	
0x38	8	Volume Serial Number (LE)	purple

Table 4: BreakDown of the VBR(see Figure **??**)

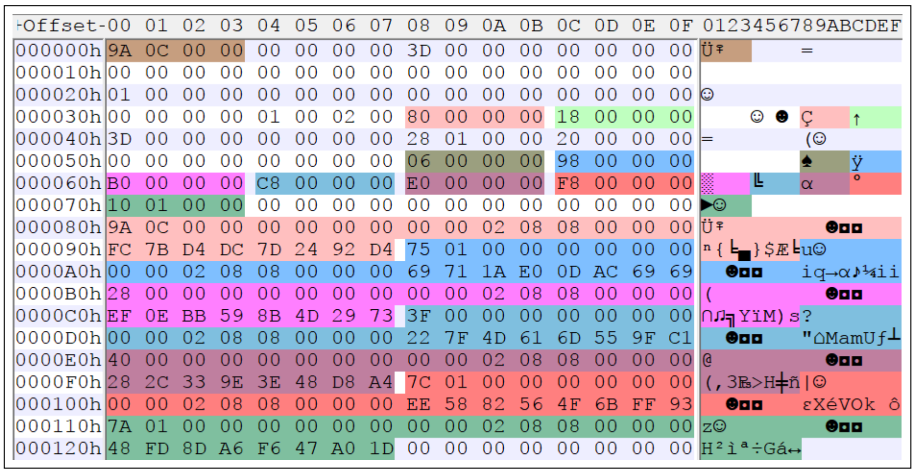
0x00: This is the jump instructor for the bootcode, here all zero (see also [10]).
0x03: The next eight bytes give the name of the filesystem (see also [10]).
0x07: In the analyzed partitions these five bytes remained always zero (see also [10]).
0x10: The next four bytes are a structure, named FILESYSTEMRECOGNITION-STRUCTURE by Microsoft. This structure should enable operating systems to recognize the structure, thus being able to determine that the file system in use (REFS) is a valid file system - [10].

0x14: This two-byte value gives the number of bytes in the VBR.
0x16: At this point, the checksum calculator (for Code see Figure **??**) from Microsofts Development Center has to be mentioned [34], because on one hand side it verifies, that this value is a checksum, at the other hand side it provides the possibility to integrate the FSRS in your own code. Analyzing the code, it shows, that the checksum is computed for the length of the structure, value is given in offset 0x14, except the checksum itself.
0x18: Read as little endian, this value gives the offset in sectors to the BACKUPVBR. Remembering the partition size, given by the $ mmls, it is obvious, that the BACKUPVBR isn't the last sector of the partition, so using this value, you can also calculate the size of the area behind the BACKUPVBR, which seems to be not addressable for the filesystem.

$$\text{SIZE NOTADDRESSABLE AREA} = \text{TOTALSECTORSOFPARTITION} - \text{VALUEFROMOFFSET 0X18}$$

0x20: Four bytes, read as little endian, provide the value for the bytes per sectors. In the hexdump-examples in the "Appendix-Section" this value is 0x0002, read as little endian 0x200, decimal 512.
0x24: Four bytes, read as little endian, provide the value for the sectors per cluster. In the hexdump-examples in the "Appendix-Section" this value is 0x80000000, read as little endian 0x80, decimal 128. This is the standard cluster size for REFS (see also [10])
0x28 & 0x29: These two bytes provide the filesystem version number, splitted into a major and a minor value.
0x38: This is the serial number of the volume, read as little endian.

A.2.2 $Tree_Control



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 9A 0C 00 00 00 00 00 00 3D 00 00 00 00 00 00 00 Ü‡      =
000010h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000020h 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
000030h 00 00 00 00 01 00 02 00 80 00 00 00 18 00 00 00     ☺ ☻ Ç   ↑
000040h 3D 00 00 00 00 00 00 00 28 01 00 00 20 00 00 00 =       (☺
000050h 00 00 00 00 00 00 00 00 06 00 00 00 98 00 00 00         ♠   ÿ
000060h B0 00 00 00 C8 00 00 00 E0 00 00 00 F8 00 00 00 ░   ╚   α   °
000070h 10 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ►☺
000080h 9A 0C 00 00 00 00 00 00 00 00 02 08 08 00 00 00 Ü‡        ☻◘◘
000090h FC 7B D4 DC 7D 24 92 D4 75 01 00 00 00 00 00 00 ⁿ{╘▄}$Æ╘u☺
0000A0h 00 00 02 08 08 00 00 00 69 71 1A E0 0D AC 69 69   ☻◘◘   iq→α♪¼ii
0000B0h 28 00 00 00 00 00 00 00 00 00 02 08 08 00 00 00 (         ☻◘◘
0000C0h EF 0E BB 59 8B 4D 29 73 3F 00 00 00 00 00 00 00 ∩♫╗YïM)s?
0000D0h 00 00 02 08 08 00 00 00 22 7F 4D 61 6D 55 9F C1   ☻◘◘   "⌂MamUƒ┴
0000E0h 40 00 00 00 00 00 00 00 00 00 02 08 08 00 00 00 @         ☻◘◘
0000F0h 28 2C 33 9E 3E 48 D8 A4 7C 01 00 00 00 00 00 00 (,3₧>H╪ñ|☺
000100h 00 00 02 08 08 00 00 00 EE 58 82 56 4F 6B FF 93   ☻◘◘   εXéVOk ô
000110h 7A 01 00 00 00 00 00 00 00 00 02 08 08 00 00 00 z☺        ☻◘◘
000120h 48 FD 8D A6 F6 47 A0 1D 00 00 00 00 00 00 00 00 H²ìª÷Gá↔
```

Figure 22: Third ENTRYBLOCK of $TREE_CONTROL of the second partition

OFFSET	LENGTH	VALUE (LE)	DESCRIPTION
0x00	0x08	0x0C9A	ENTRYBLOCK-number
0x38	0x04	0x80	Offset to own reference
0x3C	0x04	0x18	Length of own reference
0x58	0x04	0x06	Amount of pointers in the extent
0x5C	0x18		6 pointers, 4 bytes each
0x5C	0x04	0x98	1st pointer
0x98	0x18		1st record
0x98	0x04	0x0175	ENTRYBLOCK-number, this record refers to
0xB0	0x18		2nd record
0xB0	0x04	0x28	ENTRYBLOCK-number, this record refers to
0xC8	0x18		3rd record
0xC8	0x04	0x3F	ENTRYBLOCK-number, this record refers to
0xE0	0x18		4th record
0xE0	0x04	0x40	ENTRYBLOCK-number, this record refers to
0xF8	0x18		5th record
0xF8	0x04	0x017C	ENTRYBLOCK-number, this record refers to
0x110	0x18		6th record
0x110	0x04	0x017A	ENTRYBLOCK-number, this record refers to

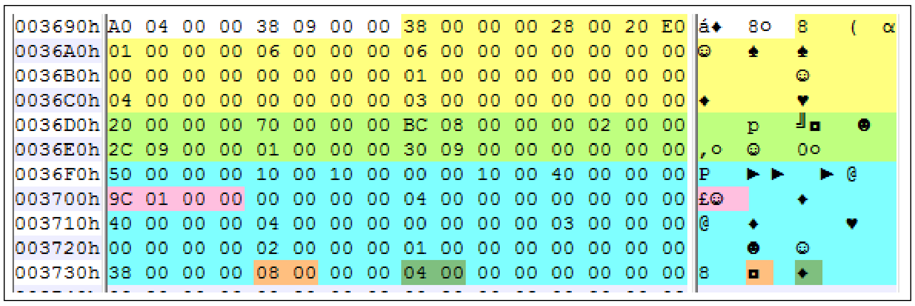
Table 5: possible Breakdown of the second extent of the TREE_CONTROL

0x00: The ENTRYBLOCK-number, this value was found in the $TREE_CONTROL at offset 0xA0.
0x38: This value gives the offset to a record, which is a reference of the record to itself, hence the provided ENTRYBLOCK-number equals the ENTRYBLOCK, which is examined.
0x3C: This value 0x18, read as little Endian, corresponds to the length of the record, which is starting at offset 0x80. Furthermore, it is the length also of the other records, within this extent.
0x58: This value 0x06, read as little Endian, corresponds to the amount of records found in this extent.
0x5C: This value 0x98, read as little Endian, corresponds to the offset to the first record in this extent, counted from the beginning of the extent. According to the amount of pointers, given at offset 0x58, 6 pointers can be found here.
0x98: This value 0x175, read as little Endian, gives the ENTRYBLOCK-number, the record refers to. The first four bytes from the other records have to be read in the same way.

The third extent contains partly deviating values in the first 0x98 bytes. The values at offset 0x3C, 0x58 and 0x5C are the same as in the second extent; the breakdown for these values seems to be suitable. The records, starting at offset 0x98, are identical to these in the second extent (see Figure **??**).

A.2.3 Second Node

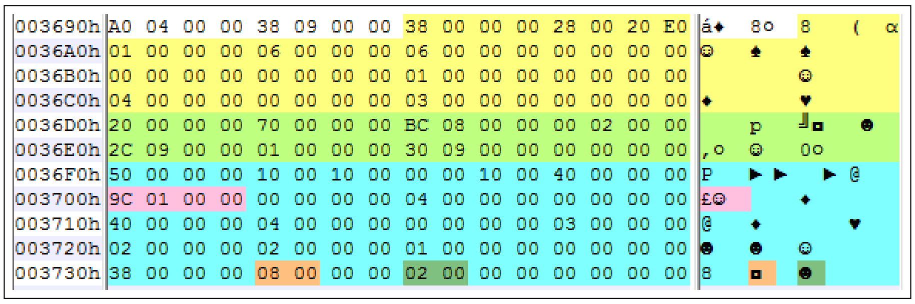
In the case of the EntryBlock 0x19E, I also examined a second node in the EntryBlock, located at offset 0x3698.



```
003690h A0 04 00 00 38 09 00 00 38 00 00 00 28 00 20 E0  á♦  8○  8   (  α
0036A0h 01 00 00 00 06 00 00 00 06 00 00 00 00 00 00 00  ☺   ♠   ♠
0036B0h 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00          ☺
0036C0h 04 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00  ♦       ♥
0036D0h 20 00 00 00 70 00 00 00 BC 08 00 00 00 02 00 00      p   ╝◘   ☻
0036E0h 2C 09 00 00 01 00 00 00 30 09 00 00 00 00 00 00  ,○  ☺   0○
0036F0h 50 00 00 00 10 00 10 00 00 00 10 00 40 00 00 00  P   ► ►   ► @
003700h 9C 01 00 00 00 00 00 00 04 00 00 00 00 00 00 00  £☺      ♦
003710h 40 00 00 00 04 00 00 00 00 00 00 00 03 00 00 00  @   ♦       ♥
003720h 00 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00      ☻   ☺
003730h 38 00 00 00 08 00 00 00 04 00 00 00 00 00 00 00  8   ◘   ♦
```

Figure 23: EntryBlock 0x19E, second node, partition 2

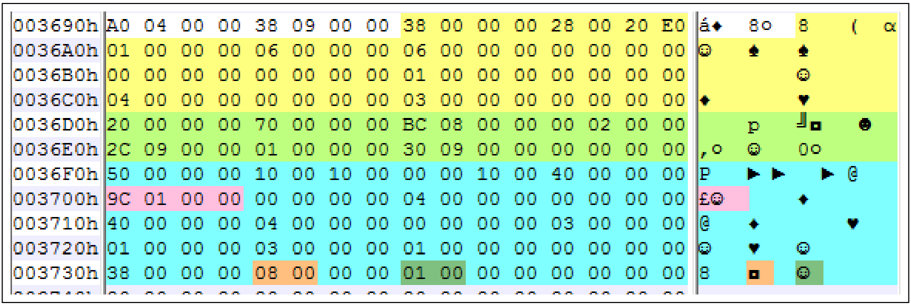
At offset 0x3700, colored pink, I found a value, that reminded me of the EntryBlock, I had just examined, so or So I decided to use EnCase to get to the EntryBlock 0x19C but on the way I also had a look at EntryBlock 0x19D (screenshot below).



```
003690h A0 04 00 00 38 09 00 00 38 00 00 00 28 00 20 E0  á♦  8○  8   (  α
0036A0h 01 00 00 00 06 00 00 00 06 00 00 00 00 00 00 00  ☺   ♠   ♠
0036B0h 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00          ☺
0036C0h 04 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00  ♦       ♥
0036D0h 20 00 00 00 70 00 00 00 BC 08 00 00 00 02 00 00      p   ╝◘   ☻
0036E0h 2C 09 00 00 01 00 00 00 30 09 00 00 00 00 00 00  ,○  ☺   0○
0036F0h 50 00 00 00 10 00 10 00 00 00 10 00 40 00 00 00  P   ► ►   ► @
003700h 9C 01 00 00 00 00 00 00 04 00 00 00 00 00 00 00  £☺      ♦
003710h 40 00 00 00 04 00 00 00 00 00 00 00 03 00 00 00  @   ♦       ♥
003720h 02 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00  ☻   ☻   ☺
003730h 38 00 00 00 08 00 00 00 02 00 00 00 00 00 00 00  8   ◘   ☻
```

Figure 24: EntryBlock 0x19D, second node, partition 2

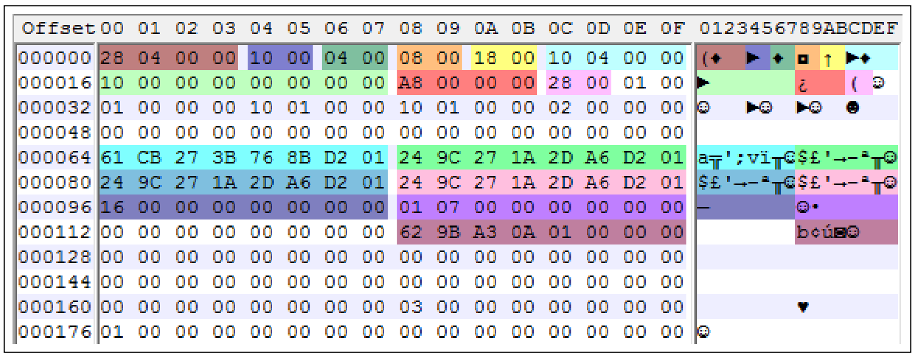
The value at offset 0x3700, the assumed value for the related node, as well as the value at offset 0x3734 remains the same, while the value at offset 0x3738 has decreased. Next, I examined the EntryBlock 0x19C (screenshot below).



```
003690h A0 04 00 00 38 09 00 00 38 00 00 00 28 00 20 E0
0036A0h 01 00 00 00 06 00 00 00 06 00 00 00 00 00 00 00
0036B0h 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
0036C0h 04 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00
0036D0h 20 00 00 00 70 00 00 00 BC 08 00 00 00 02 00 00
0036E0h 2C 09 00 00 01 00 00 00 30 09 00 00 00 00 00 00
0036F0h 50 00 00 00 10 00 10 00 00 00 10 00 40 00 00 00
003700h 9C 01 00 00 00 00 00 00 04 00 00 00 00 00 00 00
003710h 40 00 00 00 04 00 00 00 00 00 00 00 03 00 00 00
003720h 01 00 00 00 03 00 00 00 01 00 00 00 00 00 00 00
003730h 38 00 00 00 08 00 00 00 01 00 00 00 00 00 00 00
```

Figure 25: ENTRYBLOCK 0x19C, second node, partition 2

As observed in the previous example, the value for the related node, as well as the value at offset 0x3734 stays the same, the value at 0x3738 has decreased. Looking for increasing values, I had a look at the ENTRYBLOCK 0x19F and 0x1A0. While the value at offset 0x3738 increased to 0x08 in the ENTRYBLOCK 0x19F, the ENTRYBLOCK 0x1A0 contained the INDEXERVOLUMEGUID, 76 bytes, and nothing else. To summarize, I found that the ENTRYBLOCKs 0x19C to 0x19F all relate to the ENTRYBLOCK 0x19C at offset 0x3700; a constant value 0x08 at offset 0x3734 and a value increasing from 0x01 to 0x08 as a power off 2 at offset 0x3738. This constant value 0x08 and the increasing value are also given in the METADATAATTRIBUTE, (see figure below), colored in orange and dark green (offset 0x08 and 0x06).

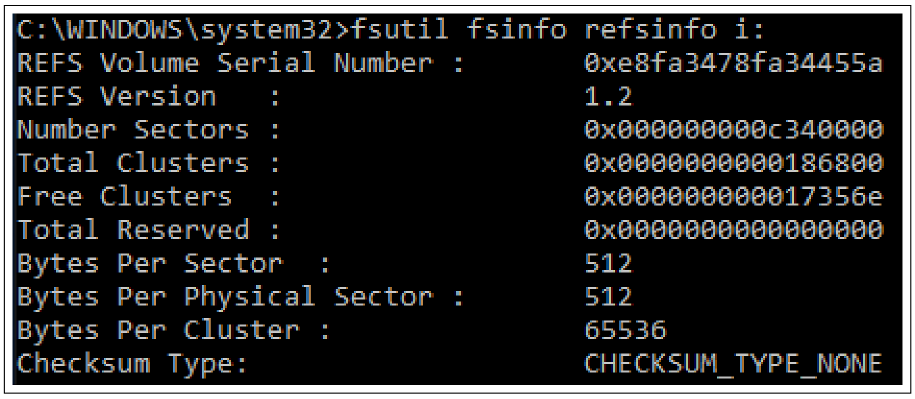


```
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000 28 04 00 00 10 00 04 00 08 00 18 00 10 04 00 00
000016 10 00 00 00 00 00 00 00 A8 00 00 00 28 00 01 00
000032 01 00 00 00 10 01 00 00 10 01 00 00 02 00 00 00
000048 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000064 61 CB 27 3B 76 8B D2 01 24 9C 27 1A 2D A6 D2 01
000080 24 9C 27 1A 2D A6 D2 01 24 9C 27 1A 2D A6 D2 01
000096 16 00 00 00 00 00 00 00 01 07 00 00 00 00 00 00
000112 00 00 00 00 00 00 00 00 62 9B A3 0A 01 00 00 00
000128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000144 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000160 00 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00
000176 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 26: Starting bytes FNA of Sys.info record

So at this point, I propose the hypothesis that, through the second node, the ENTRYBLOCKs become related to the ENTRYBLOCK with the most actual data; an internal counter gives the amount of maximum copies and the number of this special copy. This hypothesis will be checked at a later point in this section.

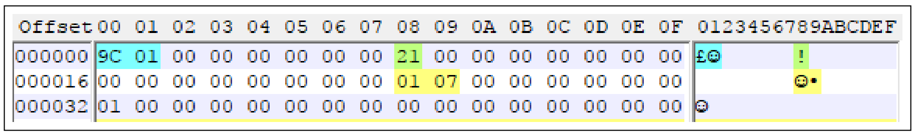
When comparing the content of the four named ENTRYBLOCKS, focussing on the timestamps in the METADATAATTRIBUTE of the folder SYSTEMVOLUMEINFORMATION, all ENTRYBLOCKS have the sameCREATED-timestamp. While the ENTRYBLOCKS 0x19C, 0x19D and 0x19F have complete identical timestamps for MODIFIED, METADATA MODIFIED and LASTACCESSED, deviating less than a second from the CREATED, the ENTRYBLOCK 0x19E gives a completely different timestamp for MODIFIED, METADATA MODIFIED and LASTACCESSED (as can be seen in the figure below), so it can be assumed that this ENTRYBLOCK contains the most actual data.



```
C:\WINDOWS\system32>fsutil fsinfo refsinfo i:
REFS Volume Serial Number :       0xe8fa3478fa34455a
REFS Version   :                  1.2
Number Sectors :                  0x000000000c340000
Total Clusters :                  0x0000000000186800
Free Clusters  :                  0x000000000017356e
Total Reserved :                  0x0000000000000000
Bytes Per Sector  :               512
Bytes Per Physical Sector :       512
Bytes Per Cluster :               65536
Checksum Type:                    CHECKSUM_TYPE_NONE
```

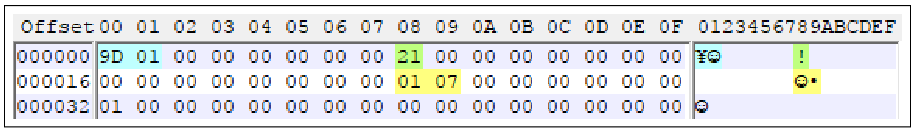
Figure 27: Output from the $fsutil command from the WINDOWS CLI

I searched, therefore, for a value, that pointed to the most actual data. Such a value has to be located in the ENTRYBLOCK-Descriptor, together with a reference to the related node. Below are the screenshots from the four ENTRYBLOCKS.



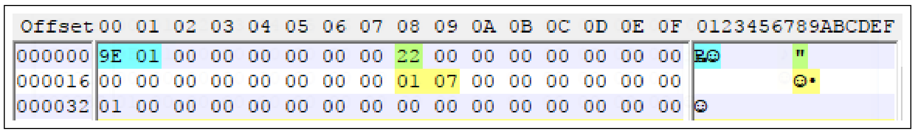
```
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000 9C 01 00 00 00 00 00 00 21 00 00 00 00 00 00 00 £☺      !
000016 00 00 00 00 00 00 00 00 01 07 00 00 00 00 00 00         ☺•
000032 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
```

Figure 28: ENTRYBLOCK-DESCRIPTOR 0x19C, partition 2



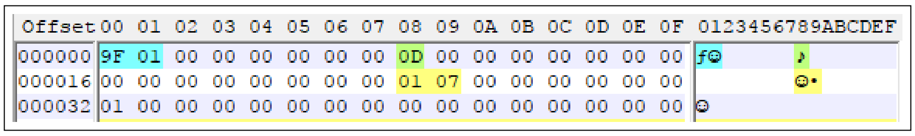
```
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000 9D 01 00 00 00 00 00 00 21 00 00 00 00 00 00 00 ¥☺      !
000016 00 00 00 00 00 00 00 00 01 07 00 00 00 00 00 00         ☺•
000032 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
```

Figure 29: ENTRYBLOCK-DESCRIPTOR 0x19D, partition 2



```
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000 9E 01 00 00 00 00 00 00 22 00 00 00 00 00 00 00 ž☺      "
000016 00 00 00 00 00 00 00 00 01 07 00 00 00 00 00 00         ☺•
000032 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
```

Figure 30: ENTRYBLOCK-DESCRIPTOR 0x19E, partition 2



```
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000 9F 01 00 00 00 00 00 00 0D 00 00 00 00 00 00 00 ƒ☺      ♪
000016 00 00 00 00 00 00 00 00 01 07 00 00 00 00 00 00         ☺•
000032 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
```

Figure 31: ENTRYBLOCK-DESCRIPTOR 0x19F, partition 2

OFFSET	LENGTH	DESCRIPTION
0x00	0x08	ENTRYBLOCK-Number
0x08	0x08	Counter
0x18	0x08	Node ID

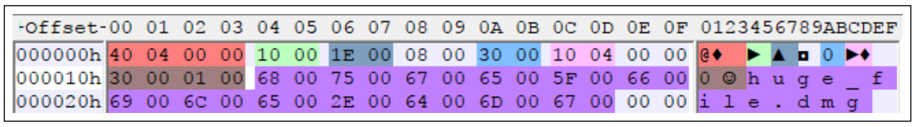
Table 6: Breakdown of the ENTRYBLOCK-Descriptor

Comparing the screenshots, the value at offset 0x08 attracts attention. It is obvious that the ENTRYBLOCK 0x19E has the highest value at this offset, so the hypothesis at this time is that the ENTRYBLOCK with the highest value is the most actual one. This might also lead to the conclusion that checking the ENTRYBLOCKS with decreasing values at offset 0x08 might gives an overview to the changes on the system. Up to now, to hypothesis, proposed in the middle of this section, is approved.

The last thing to be found was the connection between the SYSTEMVOLUMEINFORMATION-record in ENTRYBLOCK 0x190 (Figure 36), and the ENTRYBLOCKS 0x19C to 0x19F. The value is given in the FNA, right behind the filename, at offset 0x48 and also at offset 0x0A as well as the NODE ID from the corresponding node.

A.2.4 FNA

FNA for Files



```
-Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 40 04 00 00 10 00 1E 00 08 00 30 00 10 04 00 00 @♦  ► ▲ ◘ 0 ►♦
000010h 30 00 01 00 68 00 75 00 67 00 65 00 5F 00 66 00 0 ☺ h u g e _ f
000020h 69 00 6C 00 65 00 2E 00 64 00 6D 00 67 00 00 00 i l e . d m g
```

Figure 32: $FILENAME -attribute part 1, ENTRYBLOCK 0x190, record for HUGE-FILE, partition 2

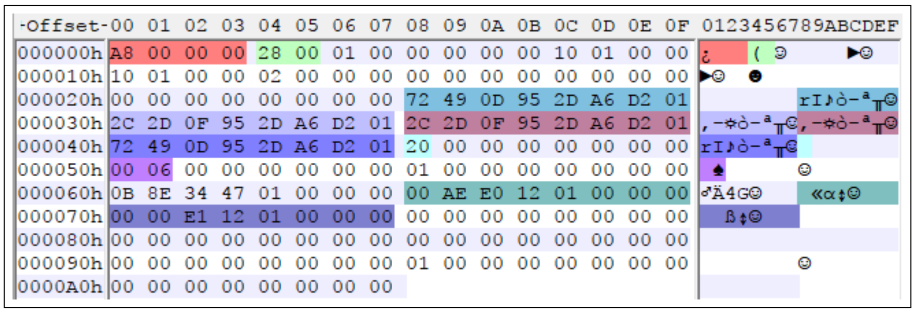
The screenshot above displays the first 0x30 bytes of a record. The record contains several structures. Obviously the filename is given at the beginning, headed by a value 0x30000100, which is similar to the FNA from NTFS (a breakdown is shown below).

OFFSET	LENGTH	VALUE (LE)	DESCRIPTION
0x00	0x04	0x440	Length of the whole record
0x04	0x02	0x10	Length of header
0x06	0x02	0x1E	Length of filename, including attribute identifier
0x0A	0x02	0x30	Offset to next structure
0x0C	0x04	0x410	Remaining data in the record
0x10	0x04	0x30000100	Attribute-Type identifier
0x14	various		Filename

Table 7: Breakdown for FILENAME-attribute

0x00 : This value, read as little endian, gives the length of the whole record. In the example above the value is 0x440, dec 1088.
0x04: This value gives the length of the header, or in other words, the offset to the attribute identifier.
0x06 : This value gives the length of the filename, including the attribute identifier.
0x0A : This value gives the offset to the next structure; in this example the value is 0x30. During the examination, values of up to 0x48 were found.
0x0C : This value, read as little endian, gives the length of the remaining data in this record, starting right after this attribute.
0x10 : This value identifies whether if it is a file or a directory; the latter was found to have the value 0x30000200. Except for the included value for files or directories, this value reminds me of the ATTRIBUTETYPE-Identifier, known from NTFS.
0x14 : Here the filename is given; the length of the attribute is obviously correlated to the length of the filename.

The next structure starts after the filename, according to the value at offset 0x0A. The first value is the length of the structure.



```
-Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h A8 00 00 00 28 00 01 00 00 00 00 00 10 01 00 00 ¿   ( ☺     ▶☺
000010h 10 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ▶☺  ☻
000020h 00 00 00 00 00 00 00 00 72 49 0D 95 2D A6 D2 01         rI♪ò-ª╥☺
000030h 2C 2D 0F 95 2D A6 D2 01 2C 2D 0F 95 2D A6 D2 01 ,-☼ò-ª╥☺,-☼ò-ª╥☺
000040h 72 49 0D 95 2D A6 D2 01 20 00 00 00 00 00 00 00 rI♪ò-ª╥☺
000050h 00 06 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ♠      ☺
000060h 0B 8E 34 47 01 00 00 00 00 AE E0 12 01 00 00 00 ♂Ä4G☺    «α↕☺
000070h 00 00 E1 12 01 00 00 00 00 00 00 00 00 00 00 00   ß↕☺
000080h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000090h 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00         ☺
0000A0h 00 00 00 00 00 00 00 00
```

Figure 33: FILENAME-attribute, part 2, ENTRYBLOCK 0x190, record for HUGE-FILE, partition 2

This structure that I have referred to as "inside record", contains at least two important pieces of information which are the timestamps and the sizes of the related data.

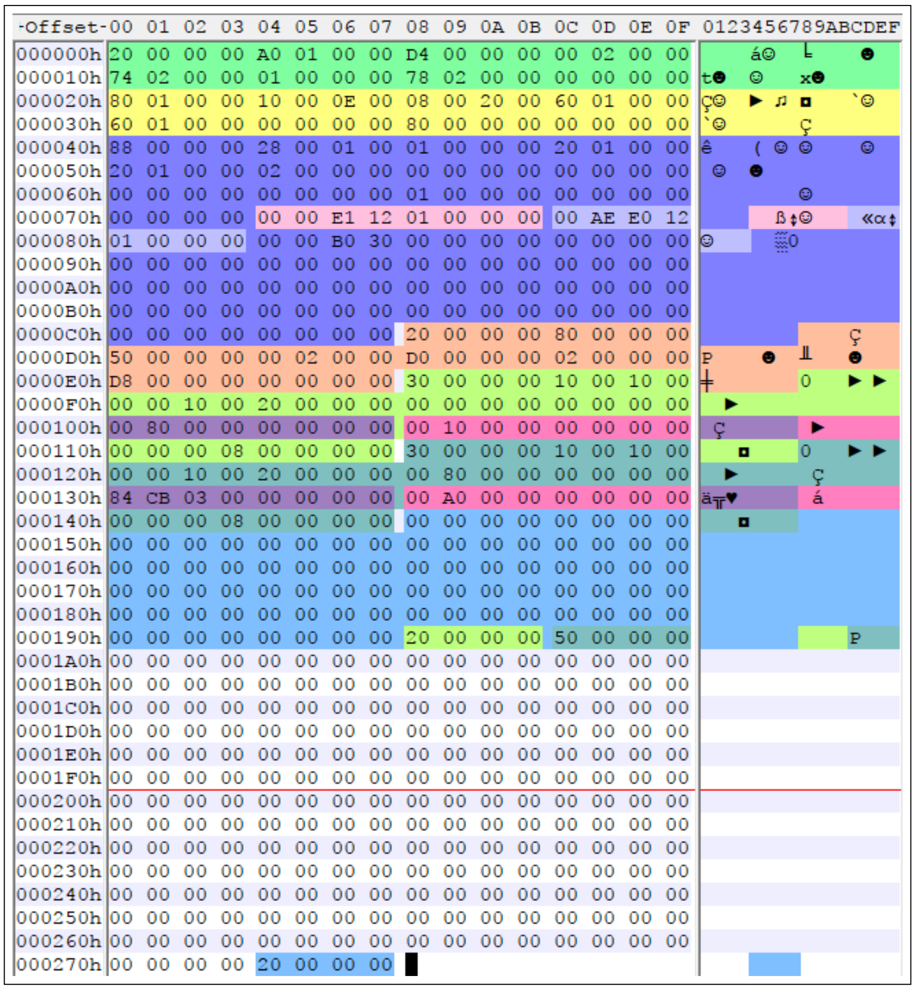
OFFSET	LENGTH	VALUE (LE)	DESCRIPTION
0x00	0x04	0x08	Length of the inside record
0x04	0x02	0x28	Possible offset to first timestamp
0x28	0x08	1D2A62D950D4972	1st timestamp
0x30	0x08	1D2A62D950F2D2C	2nd timestamp
0x38	0x08	1D2A62D950F2D2C	3rd timestamp
0x40	0x08	1D2A62D950D4972	4th timestamp
0x48	0x04	0x20	Archive flag
0x50	0x04	0x600	Node ID of parent directory
0x60	0x08		Unknown value
0x68	0x08	0x112E0AE00	Logical file-size
0x70	0x08	0x112E10000	Physical file-size

Table 8: Breakdown for METADATA-attribute

0x00 : This value, read as little endian, gives the length of the attribute. All examined File-Records had an equal sized length of this METADATA-attribute of 0xA8 bytes, dec 168.
0x04 : This value might give the offset to the first value within the attribute.
0x28 : These eight bytes, read as little endian, is a windows file-time timestamp, counting 100-nanoseconds intervals from January 1st, 1601 (UTC). Here the value is 0x1D2A62D950D4972, dec 131350053851580786, which is 2017-03-26 14:36:25 Sun CEST, which is the time the file was copied from my working machine to the REFS-volumes.
0x30 : Also eight byte value, to be read as little endian, gives another windows file-time timestamp; in this example the hex-value is 0x1D2A62D950F2D2C, dec 131350053851704620.
0x38 : Same as the second timestamp at offset 0x30.

0x40 : Same as the first timestamp at offset 0x30
0x48 : The next bytes are the flags for the file. I observed flags for "read only", "hidden" and "archive"; the latter has the value 0x20, which is shown in the screenshot above (for further details see **??**).
0x50 : This value gives the node ID of the parent folder.
0x68 : This value, 8 bytes of length and read as little endian, gives the logical size of the file. Here the value is 0x112E0AE00, dec. 4611681792, which is nearly 4.611 GB.
0x70 : This value, 8 bytes of length and read as little endian, gives the physical size of the file, which is the size the file occupies on the disk. The value here is 0x112E10000, dec 4611702784, which is 20.992 bytes bigger than the logical file-size.

After the first inside record, another header could be seen (see screenshot below).



```
-Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000h 20 00 00 00 A0 01 00 00 D4 00 00 00 00 02 00 00     á☺  ╘    ☻  
000010h 74 02 00 00 01 00 00 00 78 02 00 00 00 00 00 00 t☻  ☺   x☻      
000020h 80 01 00 00 10 00 0E 00 08 00 20 00 60 01 00 00 Ç☺  ► ♫ ◘   `☺  
000030h 60 01 00 00 00 00 00 00 80 00 00 00 00 00 00 00 `☺      Ç       
000040h 88 00 00 00 28 00 01 00 01 00 00 00 20 01 00 00 ê   ( ☺ ☺    ☺  
000050h 20 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00  ☺  ☻           
000060h 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00         ☺       
000070h 00 00 00 00 00 00 E1 12 01 00 00 00 00 AE E0 12       ß↕☺     «α↕
000080h 01 00 00 00 00 00 B0 30 00 00 00 00 00 00 00 00 ☺     ░0        
000090h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000A0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000B0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0000C0h 00 00 00 00 00 00 00 00 20 00 00 00 80 00 00 00             Ç   
0000D0h 50 00 00 00 00 02 00 00 D0 00 00 00 02 00 00 00 P    ☻  ╨   ☻   
0000E0h D8 00 00 00 00 00 00 00 30 00 00 00 10 00 10 00 ╪       0   ► ► 
0000F0h 00 00 10 00 20 00 00 00 00 00 00 00 00 00 00 00   ►             
000100h 00 80 00 00 00 00 00 00 00 10 00 00 00 00 00 00  Ç       ►      
000110h 00 00 00 08 00 00 00 00 30 00 00 00 10 00 10 00    ◘    0   ► ► 
000120h 00 00 10 00 20 00 00 00 00 80 00 00 00 00 00 00   ►      Ç      
000130h 84 CB 03 00 00 00 00 00 00 A0 00 00 00 00 00 00 ä╦♥      á      
000140h 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00    ◘            
000150h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000160h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000170h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000180h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000190h 00 00 00 00 00 00 00 00 20 00 00 00 50 00 00 00             P   
0001A0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0001B0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0001C0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0001D0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0001E0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
0001F0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000200h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000210h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000220h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000230h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000240h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000250h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000260h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00                 
000270h 00 00 00 00 20 00 00 00                                         
```

Figure 34: FILENAME-attribute, part 3, ENTRYBLOCK 0x190, record for HUGE-FILE, partition2

This attribute seems to be complex. There is a header, marked green in the screenshot above, that refers to the pointer at offset 0x274 and to the structure end at offset 0x278. The yellow marked structure also seems to be a header, that refers to the end of the structure at offset 0x180. After the following 0x88 bytes of data, marked in blue, another header appears, which refers to the pointer at offset 0xD0 and to the end of the structure at offset 0xD8, which is the same offset that the yellow marked header refers to. Below is a possible breakdown for the whole structure:

OFFSET	LENGTH	VALUE (LE)	DESCRIPTION
0x00	0x20		Header for the whole structure
0x00	0x04	0x20	Length of the header
0x04	0x04	0x1A0	Offset to next free record
0x08	0x04	0xD4	Free space in the node
0x10	0x04	0x274	Offset to the pointer
0x14	0x04	0x01	Amount of pointers
0x18	0x04	0x278	End of the structure
0x20	0x20		Next header structure
0x20	0x04	0x180	Length of the structure
0x30	0x04	0x160	Amount of data within the structure, except header
0x38	0x04		Unknown
0x40	0x88		Record
0x40	0x04	0x88	Length of the record
0x74	0x08	0x112E10000	Physical size of the file
0x7C	0x08	0x112E0AE00	Logical size of the file
0xC8	0x20		Next header structure
0xC8	0x04	0x20	Length of the header
0xCC	0x04	0x80	Offset to next free record
0xD0	0x04	0x50	Free space in the node
0xD8	0x04	0xD0	Offset to pointers
0xDC	0x04	0x02	Amount of pointers
0xE0	0x04	0xD8	Offset to end of the structure
0xE8	0x30		1st data-run
0xE8	0x04	0x30	Length of the data-run
0x100	0x08	0x8000	Amount of clusters in this data-run
0x108	0x08	0x1000	Starting cluster of this data-run
0x118	0x30		2nd data-run
0x118	0x04	0x30	Length of the data-run
0x130	0x08	0x3CB84	Amount of clusters in this data-run
0x138	0x08	0xA000	Starting cluster of this data-run
0x198	0x04	0x20	1st pointer of data-run structure
0x19C	0x04	0x50	2nd pointer of data-run structure
0x274	0x04	0x20	Pointer of the whole structure

Table 9: Breakdown of DATA-RUN- of the FNA

0x00 : A typical header structure with a length of 0x20 bytes; the length is usually given at the first four bytes.
0x04 : The value, 4 bytes, read as little endian, give the offset to the next free

record.
0x08 : This value, four bytes, read as little endian, gives the amount of free space in the node.
0x10 : This value, four bytes, read as little endian, gives the offset to the pointer, at the end of the whole structure.
0x14 : This value, four bytes, read as little endian, gives the amount of pointers of the whole structure.
0x18 : This value, four bytes, read as little endian, gives the offset to the end of the structure.
0x20 : Another header, but with a differing structure. The first four bytes give the length of the whole structure.
0x30 : This value, four bytes, read as little endian, gives the amount of data, excluding the data of the header.
0x38 : At this time, it is only a guess, but this might be an ATTRIBUTE-TYPE - identifier. In comparison with the known identifiers known from NTFS, the identifier 0x80000000 is for the DATA-RUN-attribute.
0x40 : The following structure that has a length of 0x88 bytes, taken from the first four bytes, contains a few values, which have already been identified.
0x74 : The next 8 bytes, read as little endian, match the physical size of the file.
0x7C : This value, read as little endian, matches the logical size of the file.
0xC8 : At this offset another header with the known structure begins. It starts with the length of the header, which is given at offset 0xC8.
0xCC : This value, four bytes, read as little endian, gives the offset to the next free record.
0xD0 : This value, four bytes, read as little endian, gives the amount of free bytes in the node.
0xD8 : This value, four bytes, read as little endian, gives the offset to the pointers of the structure.
0xDC : This value, four bytes, read as little endian, gives the amount of pointers in the value.
0xE0 : This value gives the offset to the end of the structure, is four bytes long and has to be read as little endian.
0xE8 : At this offset, the first data-run starts. The length of the data-run is given within the first four bytes.
0x100 : At this offset, the amount of clusters handled by this data-run, is given. It is an eight byte value, read as little endian.
0x108 : This offset names the start-cluster of the first data-run. This value has a length of eight bytes, read as little endian.
0x118 : According to the pointers, at this offset the second data-run begins. The length of this data-run is given within the first four bytes.
0x130 : Just as in the first data-run, at this offset the amount of clusters in this data-run is given. This value has eight bytes, read as little endian.

0x138 : This value, eight bytes of length, read as little endian, gives the starting cluster.
0x198 : These four bytes are the first pointer of the data-run.
0x19C : Second pointer of the data-run.
0x274 : Pointer of the whole structure.

Using the table above, the data-runs can be calculated.

1. Data-run:

$$\begin{aligned} \text{offset to file} &= \text{value offset 0x108} * \text{ClusterSize in bytes} \\ \text{offset to file} &= \text{0x1000} * \text{0x4000} = \text{0x4000000} = 67.108.864 \end{aligned}$$

$$\begin{aligned} \text{DataSize} &= \text{value offset 0x100} * \text{ClusterSize in bytes} \\ \text{DataSize} &= \text{0x8000} * \text{0x4000} = \text{0x20000000} = 536.870.912 \end{aligned}$$

2. Data-run

$$\begin{aligned} \text{offset to file} &= \text{value offset 0x138} * \text{ClusterSize in bytes} \\ \text{offset to file} &= \text{0xA000} * \text{0x4000} = \text{0x28000000} = 671.088.640 \end{aligned}$$

$$\begin{aligned} \text{DataSize} &= \text{value offset 0x130} * \text{ClusterSize in bytes} \\ \text{DataSize} &= \text{0x3CB84} * \text{0x4000} = \text{0xF2E10000} = 4.074.831.872 \end{aligned}$$

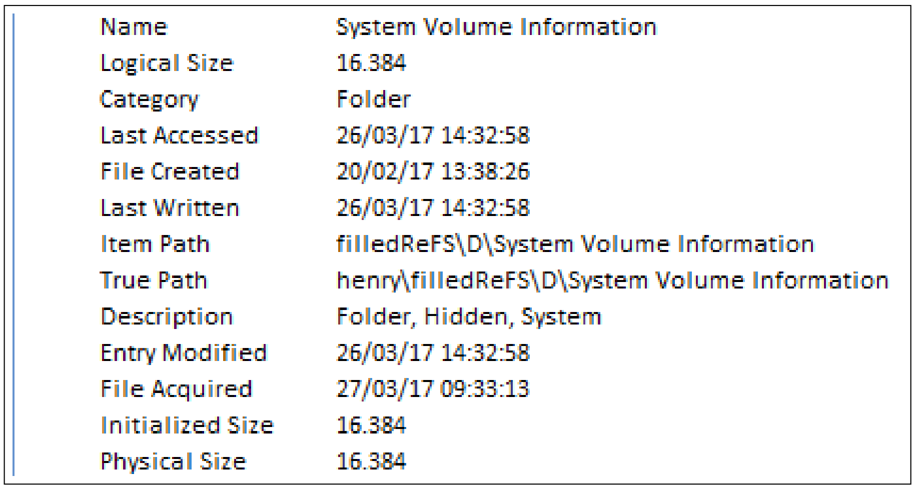
Filesize

$$\begin{aligned} \text{FileSize} &= \text{bytes first DataSize} + \text{bytes second DataSize} \\ \text{FileSize} &= \text{0x20000000} + \text{0xF2E10000} = \text{0x112E10000} \\ \text{FileSize} &= 536.870.912 + 4.074.831.872 = 4.611.702.784 \end{aligned}$$

The calculated file-size corresponds to the value at offset 0x74 of this record; just as the value at offset 0x70 of the first inside record of the FILENAME-attribute.

FNA for Folders

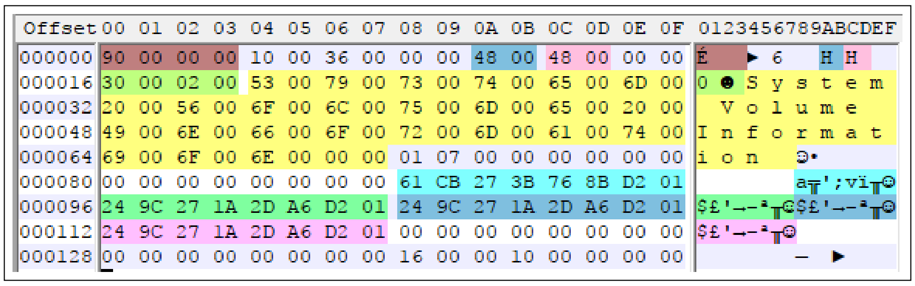
The dump of the ENTRYBLOCK, numbered 0x190 from the second partition, also contains a record concerning a structure, which ENCASE identifies as a folder.



Name	System Volume Information
Logical Size	16.384
Category	Folder
Last Accessed	26/03/17 14:32:58
File Created	20/02/17 13:38:26
Last Written	26/03/17 14:32:58
Item Path	filledReFS\D\System Volume Information
True Path	henry\filledReFS\D\System Volume Information
Description	Folder, Hidden, System
Entry Modified	26/03/17 14:32:58
File Acquired	27/03/17 09:33:13
Initialized Size	16.384
Physical Size	16.384

Figure 35: FileNameAttribute - Folder, screenshot ENCASE

The dump gives some interesting information.



```
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000000 90 00 00 00 10 00 36 00 00 00 48 00 48 00 00 00 É  ► 6   H H
000016 30 00 02 00 53 00 79 00 73 00 74 00 65 00 6D 00 0 ☻ S y s t e m
000032 20 00 56 00 6F 00 6C 00 75 00 6D 00 65 00 20 00   V o l u m e
000048 49 00 6E 00 66 00 6F 00 72 00 6D 00 61 00 74 00 I n f o r m a t
000064 69 00 6F 00 6E 00 00 00 01 07 00 00 00 00 00 00 i o n   ☺•
000080 00 00 00 00 00 00 00 00 61 CB 27 3B 76 8B D2 01         a╥';vï╥☺
000096 24 9C 27 1A 2D A6 D2 01 24 9C 27 1A 2D A6 D2 01 $£'→-ª╥☺$£'→-ª╥☺
000112 24 9C 27 1A 2D A6 D2 01 00 00 00 00 00 00 00 00 $£'→-ª╥☺
000128 00 00 00 00 00 00 00 00 16 00 00 10 00 00 00 00         ▬  ►
```

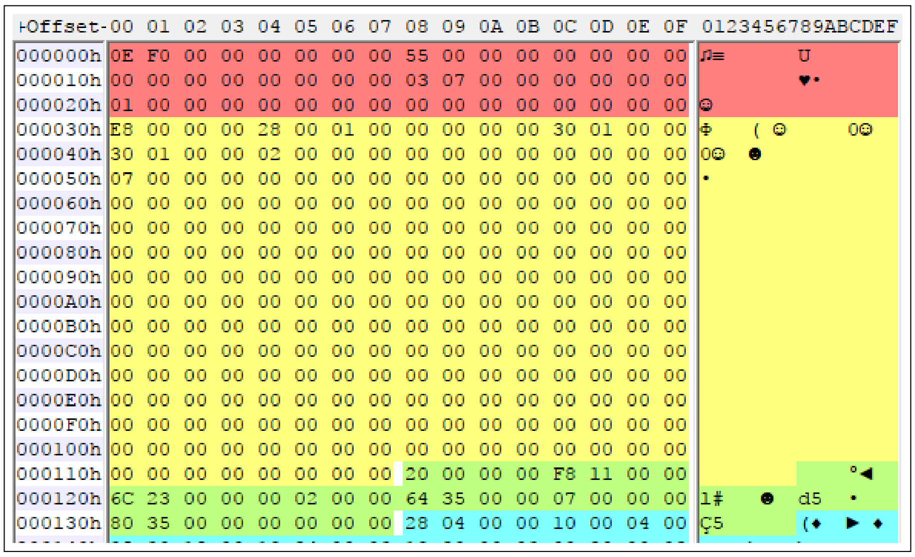
Figure 36: FileNameAttribute - Folder, dump from ENTRYBLOCK 0x190

At offset 0x10, decimal 16, the attribute identifier names 0x30000200, which is reminiscent of the attribute identifier for FNA in NTFS, 0x30000000. In NTFS the FNA contains the file (folder) name and a set of timestamps [30]. The screenshot, above, suggests that the properties from the FNA of NTFS might be adapted to REFS, at least for folders. Besides the folder name and a set of timestamps, the FNA for folders also contains the node ID of the folder. This has to be used for looking up the ENTRYBLOCK, which describes this special node in the $OBJECT_TABLE. The complete breakdown of the FNA for folders is in the appendix (see template 18).

A.2.5 $Recycle.bin

As it has already been shown in chapter **??**, there is a node for the $RECYCLE.BIN, but because of a nearly unused filesystem, it is nearly empty. It contains a folder

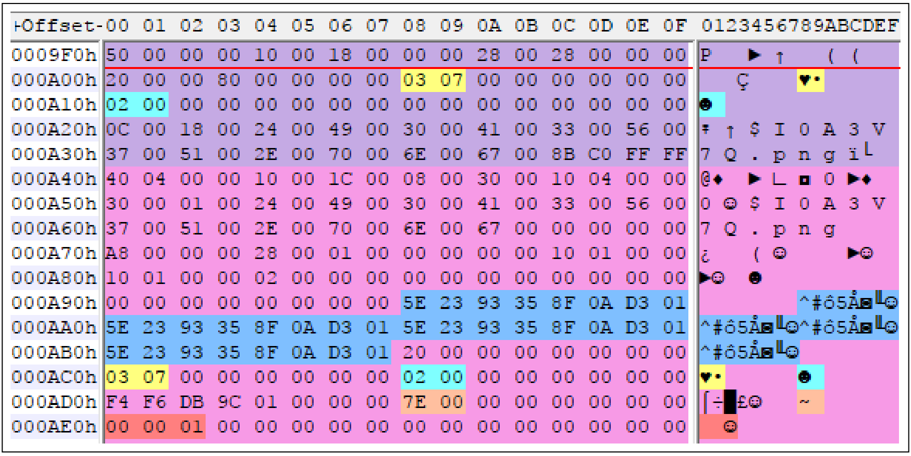
which has been named according to the User Security Identifier (SID, immutable and unique identifier for a user, group or other security principal [35]). This folder is empty, except for the DESKTOP.INI (see Figure **??**). To explore the $RECYCLE.BIN, I remounted the HDD with the four partitions and copied some files and folders to the partitions; one of the files I deleted. I made a new dump from the device and analyzed the nodes 0x702 and 0x703.
The Node 0x702 still contains only the folder for the User SID; the timestamps of the FNA are unchanged. The record for the User SID folder received a new METADATA MODIFIED-timestamp that contains the time when the file was deleted.
The Node 0x703 now contains seven records, as can be seen in the screenshot below, which is four records more than before the file was deleted.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
000000h 0E F0 00 00 00 00 00 00 55 00 00 00 00 00 00 00  ♫≡      U
000010h 00 00 00 00 00 00 00 00 03 07 00 00 00 00 00 00          ♥•
000020h 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ☺
000030h E8 00 00 00 28 00 01 00 00 00 00 00 30 01 00 00  Φ   ( ☺     0☺
000040h 30 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00  0☺  ☻
000050h 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  •
000060h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000070h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000080h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000090h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000A0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000B0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000C0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000D0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000E0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000F0h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100h 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000110h 00 00 00 00 00 00 00 00 20 00 00 00 F8 11 00 00              °◄
000120h 6C 23 00 00 00 02 00 00 64 35 00 00 07 00 00 00  l#   ☻  d5  •
000130h 80 35 00 00 00 00 00 00 28 04 00 00 10 00 04 00  Ç5      (♦  ► ♦
```

Figure 37: ENTRYBLOCK DESCRIPTOR of Node 0x703, after deleting a file

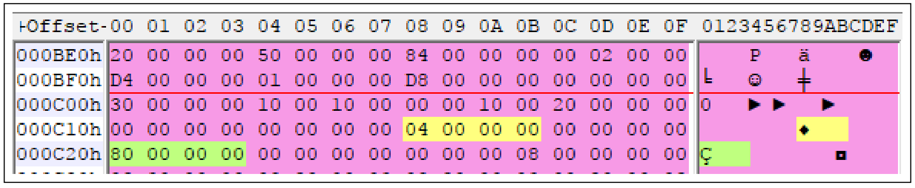
A screenshot from the first new entries; record four and five is shown below.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
0009F0h 50 00 00 00 10 00 18 00 00 00 28 00 28 00 00 00 P   ▶ ↑   ( (
000A00h 20 00 00 80 00 00 00 00 03 07 00 00 00 00 00 00    Ç    ♥•
000A10h 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☻
000A20h 0C 00 18 00 24 00 49 00 30 00 41 00 33 00 56 00 ♀ ↑ $ I 0 A 3 V
000A30h 37 00 51 00 2E 00 70 00 6E 00 67 00 8B C0 FF FF 7 Q . p n g ï└
000A40h 40 04 00 00 10 00 1C 00 08 00 30 00 10 04 00 00 @♦  ▶ ∟ ◘ 0 ▶♦
000A50h 30 00 01 00 24 00 49 00 30 00 41 00 33 00 56 00 0 ☺ $ I 0 A 3 V
000A60h 37 00 51 00 2E 00 70 00 6E 00 67 00 00 00 00 00 7 Q . p n g
000A70h A8 00 00 00 28 00 01 00 00 00 00 00 10 01 00 00 ¿   ( ☺     ▶☺
000A80h 10 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ▶☺  ☻
000A90h 00 00 00 00 00 00 00 00 5E 23 93 35 8F 0A D3 01         ^#ô5Å◙╙☺
000AA0h 5E 23 93 35 8F 0A D3 01 5E 23 93 35 8F 0A D3 01 ^#ô5Å◙╙☺^#ô5Å◙╙☺
000AB0h 5E 23 93 35 8F 0A D3 01 20 00 00 00 00 00 00 00 ^#ô5Å◙╙☺
000AC0h 03 07 00 00 00 00 00 00 02 00 00 00 00 00 00 00 ♥•      ☻
000AD0h F4 F6 DB 9C 01 00 00 00 7E 00 00 00 00 00 00 00 ⌠÷█£☺   ~
000AE0h 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00   ☺
```

Figure 38: Record four and five of the $RECYCLE.BIN

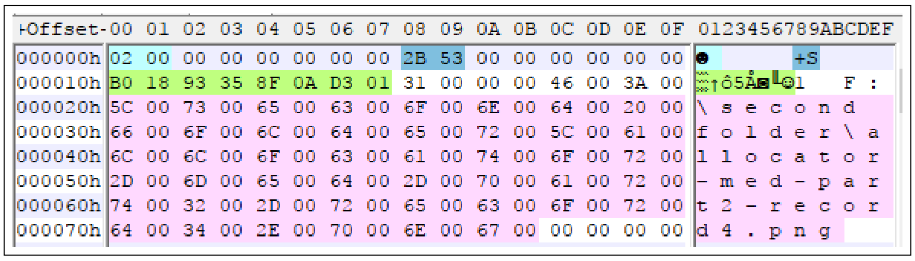
As can be seen, the original filename of the deleted file, (allocator-med-part2-record4.png), was replaced by $I0A3V7Q.png. Both records are pointing to the parent node (offset 0x18 of record 4 and offset 0x80 of record 5), node 0x703, which gives this record the child ID of two (offset 0x20 of record 4 and offset 0x88 of record 5). The timestamps in the METADATA-ATTRIBUTE at the beginning of record 5 give the 01.08.2017 08:27:11 UTC, which is the time when I deleted the file. The logical file-size is given as 0x7E, decimal 126; the physical with 64 KiB, 4 clusters. The DATARUN-ATTRIBUTE gives one DATARUN, 4 clusters in size, starting at cluster 0x80 (screenshot below).



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000BE0h 20 00 00 00 50 00 00 00 84 00 00 00 00 02 00 00     P   ä     ☻
000BF0h D4 00 00 00 01 00 00 00 D8 00 00 00 00 00 00 00 ╘   ☺   ╪
000C00h 30 00 00 00 10 00 10 00 00 00 10 00 20 00 00 00 0   ▶ ▶   ▶
000C10h 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00         ♦
000C20h 80 00 00 00 00 00 00 00 00 00 00 08 00 00 00 00 Ç          ◘
```

Figure 39: DATARUN-ATTRIBUTE of record 5 of the $RECYCLE.BIN

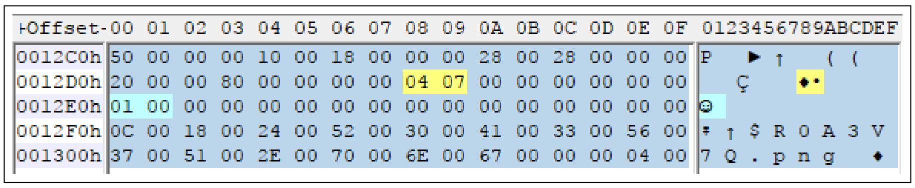
Moving to the cluster 0x80, decimal 128, byte-offset 2.097.152 of the partition, the data is given (shown in the screenshot below).



```
Offset   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
000000h  02 00 00 00 00 00 00 00 2B 53 00 00 00 00 00 00  ☻       +S
000010h  B0 18 93 35 8F 0A D3 01 31 00 00 00 46 00 3A 00  ░↑ô5Å◙╙☺1   F :
000020h  5C 00 73 00 65 00 63 00 6F 00 6E 00 64 00 20 00  \ s e c o n d
000030h  66 00 6F 00 6C 00 64 00 65 00 72 00 5C 00 61 00  f o l d e r \ a
000040h  6C 00 6C 00 6F 00 63 00 61 00 74 00 6F 00 72 00  l l o c a t o r
000050h  2D 00 6D 00 65 00 64 00 2D 00 70 00 61 00 72 00  - m e d - p a r
000060h  74 00 32 00 2D 00 72 00 65 00 63 00 6F 00 72 00  t 2 - r e c o r
000070h  64 00 34 00 2E 00 70 00 6E 00 67 00 00 00 00 00  d 4 . p n g
```

Figure 40: $RECYCLE.BIN, real filename of deleted file in cluster 0x80

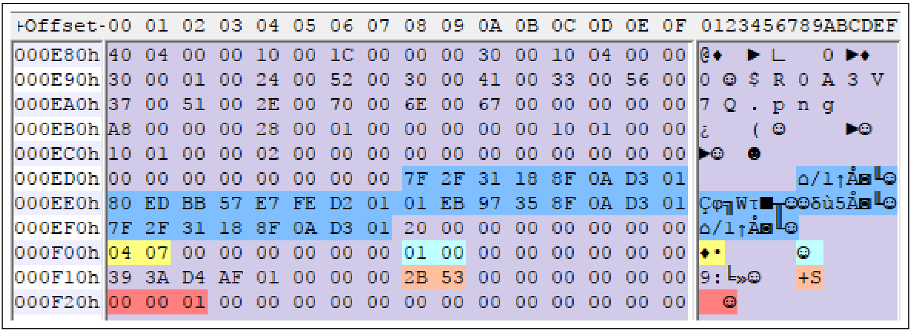
The rest of the four clusters (64 KiB) are empty. The value at offset 0x0 might be the child ID of the record in the $RECYCLE.BIN. The next value, offset 0x08 gives the size of the original file, 0x532B, decimal 21291 bytes. At offset 0x10 the time of deleting the file is given. Starting at offset 0x20, the filename and the name of the parent is given, as is usual in unicode. After the filename, there are two more zero bytes, not colored in the dump. The meaning of these two bytes is explained at a later point.
Up to this point, the $I-record gives some information about the original file, for example the filename and the timestamp for deleting the file, but the data for the original DATARUN is not given, so or so take a look at the following records. The first obvious thing is that the order has changed. Below is a screenshot of the last record.



```
Offset   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
0012C0h  50 00 00 00 10 00 18 00 00 00 28 00 28 00 00 00  P   ►  ↑   ( (
0012D0h  20 00 00 80 00 00 00 00 04 07 00 00 00 00 00 00     Ç    ♦•
0012E0h  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ☺
0012F0h  0C 00 18 00 24 00 52 00 30 00 41 00 33 00 56 00  ♀ ↑ $ R 0 A 3 V
001300h  37 00 51 00 2E 00 70 00 6E 00 67 00 00 00 04 00  7 Q . p n g   ♦
```

Figure 41: Record 7 of the $RECYCLE.BIN

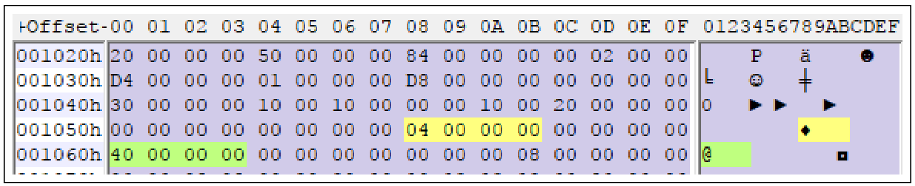
This dump gives some interesting information. Offset 0x18 gives the original parent node ID; offset 0x20 the original child ID. The filename is nearly the same as in record 4 and 5, $R0A3V7Q.png, but with the different start value $R. In record 6, two different timestamps are given; the CREATED,METADATA MODIFIED and LAST ACCESSED give the time when the file was deleted. The MODIFIED gives the 17.07.2017 12:27:51 UTC, which is the time when the file originally was created. The next two values are the original parent ID and child ID, already seen in record 7. Offset 0x98 gives the original file-size; offset 0xA0 the physical file-size.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000E80h 40 04 00 00 10 00 1C 00 00 00 30 00 10 04 00 00 @♦  ► ∟   0 ►♦
000E90h 30 00 01 00 24 00 52 00 30 00 41 00 33 00 56 00 0 ☺ $ R 0 A 3 V
000EA0h 37 00 51 00 2E 00 70 00 6E 00 67 00 00 00 00 00 7 Q . p n g
000EB0h A8 00 00 00 28 00 01 00 00 00 00 00 10 01 00 00 ¿   ( ☺     ►☺
000EC0h 10 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ►☺  ☻
000ED0h 00 00 00 00 00 00 00 00 7F 2F 31 18 8F 0A D3 01         ⌂/1↑Å◙╙☺
000EE0h 80 ED BB 57 E7 FE D2 01 01 EB 97 35 8F 0A D3 01 Çφ╗Wτ■╥☺☺δù5Å◙╙☺
000EF0h 7F 2F 31 18 8F 0A D3 01 20 00 00 00 00 00 00 00 ⌂/1↑Å◙╙☺
000F00h 04 07 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ♦•      ☺
000F10h 39 3A D4 AF 01 00 00 00 2B 53 00 00 00 00 00 00 9:╘»☺    +S
000F20h 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00   ☺
```

Figure 42: Record 6 of the RECYCLE.BIN

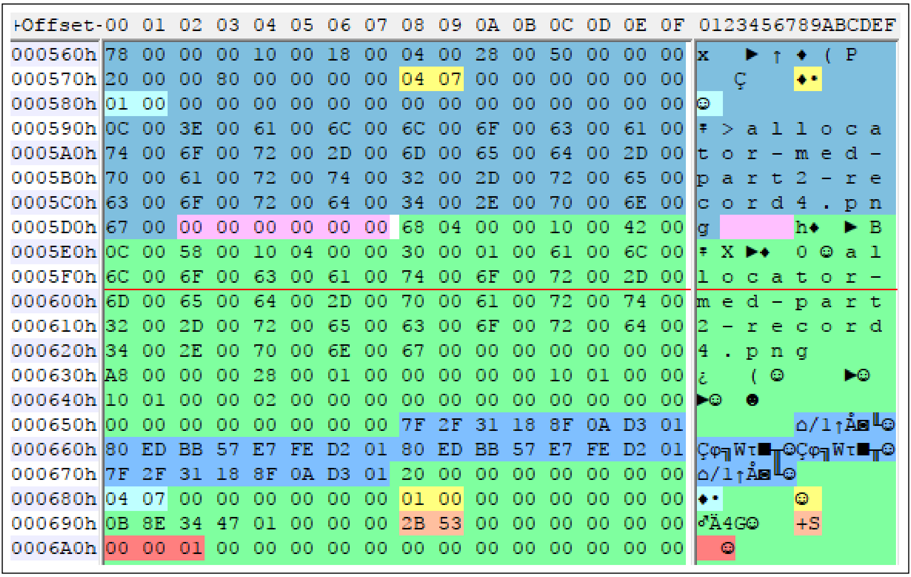
The DATARUN-ATTRIBUTE gives the starting cluster 0x40 and a cluster amount of 4.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
001020h 20 00 00 00 50 00 00 00 84 00 00 00 00 02 00 00     P   ä    ☻
001030h D4 00 00 00 01 00 00 00 D8 00 00 00 00 00 00 00 ╘   ☺   ╪
001040h 30 00 00 00 10 00 10 00 00 00 10 00 20 00 00 00 0   ► ►   ►
001050h 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00         ♦
001060h 40 00 00 00 00 00 00 00 00 00 00 08 00 00 00 00 @          ◘
```

Figure 43: DATARUN - ATTRIBUTE of record 6 in the $RECYCLE.BIN

The data of the file allocator-med-part2-record4.png is located at cluster offset 0x40, byte-offset 1.048.576 of the partition.
As a last step, a screenshot of the original record in node 0x704 is shown below.



```
Offset-00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000560h 78 00 00 00 10 00 18 00 04 00 28 00 50 00 00 00 x   ► ↑ ♦ ( P
000570h 20 00 00 80 00 00 00 00 04 07 00 00 00 00 00 00    Ç    ♦•
000580h 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ☺
000590h 0C 00 3E 00 61 00 6C 00 6C 00 6F 00 63 00 61 00 ♀ > a l l o c a
0005A0h 74 00 6F 00 72 00 2D 00 6D 00 65 00 64 00 2D 00 t o r - m e d -
0005B0h 70 00 61 00 72 00 74 00 32 00 2D 00 72 00 65 00 p a r t 2 - r e
0005C0h 63 00 6F 00 72 00 64 00 34 00 2E 00 70 00 6E 00 c o r d 4 . p n
0005D0h 67 00 00 00 00 00 00 00 68 04 00 00 10 00 42 00 g       h♦  ► B
0005E0h 0C 00 58 00 10 04 00 00 30 00 01 00 61 00 6C 00 ♀ X ►♦  0 ☺ a l
0005F0h 6C 00 6F 00 63 00 61 00 74 00 6F 00 72 00 2D 00 l o c a t o r -
000600h 6D 00 65 00 64 00 2D 00 70 00 61 00 72 00 74 00 m e d - p a r t
000610h 32 00 2D 00 72 00 65 00 63 00 6F 00 72 00 64 00 2 - r e c o r d
000620h 34 00 2E 00 70 00 6E 00 67 00 00 00 00 00 00 00 4 . p n g
000630h A8 00 00 00 28 00 01 00 00 00 00 00 10 01 00 00 ¿   ( ☺     ►☺
000640h 10 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ►☺  ☻
000650h 00 00 00 00 00 00 00 00 7F 2F 31 18 8F 0A D3 01         ⌂/1↑Å◙╙☺
000660h 80 ED BB 57 E7 FE D2 01 80 ED BB 57 E7 FE D2 01 Çφ╗Wτ■╥☺Çφ╗Wτ■╥☺
000670h 7F 2F 31 18 8F 0A D3 01 20 00 00 00 00 00 00 00 ⌂/1↑Å◙╙☺
000680h 04 07 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ♦•      ☺
000690h 0B 8E 34 47 01 00 00 00 2B 53 00 00 00 00 00 00 ♂Ä4G☺    +S
0006A0h 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00   ☺
```

Figure 44: original records of the deleted file in Node 0x704

This dump is identical to the records of the $R - record in the $RECYCLE.BIN.

A similar filesystem behavior was observable in NTFS. Deleting a file here will lead to the creation of two new files, starting with $I and $R, and a cryptical filename at the end [36].

A.3 Coding

A.3.1 Basics

Some of the code lines will break the lines of this master thesis, so line breaks were added. These lines begin with a "∼" in the following listings.

I am aware that there is potential for improving the code. I have not optimized the code, because this is not a master thesis about coding a python tool, but this code is a proof of concept; my findings can be used to analyze a REFS-partition. During coding, I adapted the code several times, because the requirements changed while creating new partitions. Of course, I could not simulate all possible scenarios of what can happen to files and folders in a filesystem, but I tried to depict the common ones. Because of that, there might be REFS - partitions this code does not work for.

First I would like to reference the modules I used:

```python
#Import needed modules/packages
import struct
import datetime
import time
import sys
import argparse
import os
import platform
from operator import itemgetter, attrgetter, methodcaller
```

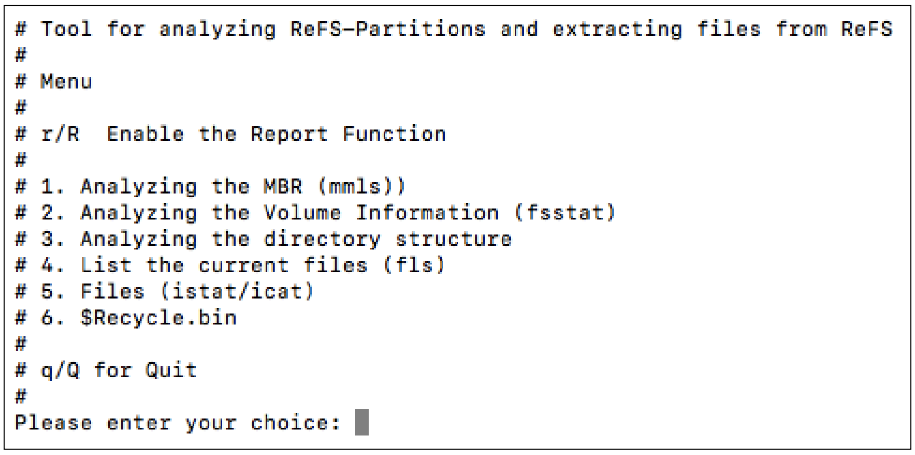
Listing A.1: Imported modules

Besides the modules, which are used regularly for coding, such as ARGPARSE (gives arguments to the code via the command line) [37], OS (enables functionality for operating system interfaces) [38], DATETIME [39] and TIME [40] (gives some functions for time calculation) I would like to comment on the module STRUCT [41]. This module contains a lot of functions that work with data streams; especially to extract values from little endian formatted streams (examples will be shown later).

The tool is started via commandline, using the command

$ python refs.py

,without any arguments, which gives the output



```
# Tool for analyzing ReFS-Partitions and extracting files from ReFS
#
# Menu
#
# r/R  Enable the Report Function
#
# 1. Analyzing the MBR (mmls))
# 2. Analyzing the Volume Information (fsstat)
# 3. Analyzing the directory structure
# 4. List the current files (fls)
# 5. Files (istat/icat)
# 6. $Recycle.bin
#
# q/Q for Quit
#
Please enter your choice: 
```

Figure 45: Screenshot from the Main Menu of my tool

A.3.2 Vbr

The code for analyzing the MBR is not implemented here, because it is not REFS - specific, in contrast to the code for analyzing the VBR. The VBR of my first partition is given in figure **??**, so the following code can be checked against it. Based on the following code, I shall explain the used code explicitly; later listings will be shortened.

```
def analyze_vbr(_filename,offset_partition):
        """ Analyzes the VolumeBootRecord """
        vbr={}
        list=()
        f=open(_filename, "rh")
#applying the breakdown-table for the VBR
        f.seek(int(offset_partition)+3)
        FileSystemName=struct.unpack_from("Q", f.read(8))[0]
#checking, if Fs is a ReFS, otherwise end
        if str(FileSystemName) == "1397122386":
                FileSystemName="ReFS"
        else:
                print_log("FileSystem is not ReFS")
                exit(0)
        f.seek(offset_partition+20)
        bytes_vbr=struct.unpack_from("H", f.read(2))[0]
#reading the checksum from the VBR
        check=struct.unpack_from("H", f.read(2))[0]
#calculating the checksum, using the code delivered by Microsoft, adapted to python
        f.seek(offset_partition)
        calc_check=f.read(64)
        list=struct.unpack_from("B"*64, calc_check)
        count=0
        checksum=0
        for i in list:
                if count == 22 or count == 23:
                        count+=1
                        continue
                if (checksum & 1) == 1:
                        checksum=0x8000+int(checksum>>1)+i
                        count+=1
                else:
                        checksum=0+int(checksum>>1)+i
                        count+=1
#comparing the checksums
        if check == checksum:
                verified=True
        else:
                verified=False
        f.seek(offset_partition+24)
```

```
41         backupvbr=struct.unpack_from("Q", f.read(8))[0]
42         f.seek(offset_partition+32)
43         bytespersector=struct.unpack_from("I", f.read(4))[0]
44         sectorpercluster=struct.unpack_from("I", f.read(4))[0]
45         MajorVersion=struct.unpack_from("B", f.read(1))[0]
46         MinorVersion=struct.unpack_from("B", f.read(1))[0]
47         f.seek(offset_partition+56)
48         volume_id=struct.unpack_from("Q", f.read(8))[0]
49         vbr={'Backup VBR':backupvbr,'Volume ID':hex(volume_id),'Verified':verified,'Checksum in VBR'
50         ~ :hex(check),'Calculated Checksum':hex(checksum),'Bytes per Sector':bytespersector,'Bytes per Vbr'
51         ~ :bytes_vbr,'File System':FileSystemName,'Major Version':MajorVersion,'Minor Version':MinorVersion,
52         ~ 'Sectors per Cluster':sectorpercluster,}
53         f.close()
54         return vbr
```

Listing A.2: Function for analyzing the VBR

As can be seen within the first line, I utilized the code as a function, which I tried to maintain throughout the whole program as far as possible. This function receives two values, delivered through the MBR - analysis, by calling it, the filename of the dump and the partition-offset. To store, the extracted data I initialized the VBR - variable as a dictionary. Furthermore, the LIST - variable is initialized as a tuple; this will be needed for the checksum calculation. Line 5 shows the opening of the data stream as read only and hexadecimal, using the variable F. The first three bytes in a REFS partition are not used at this time, so they are skipped (line 7). The following 8 bytes contain the FILESYSTEMNAME, which is extracted using the STRUCT - package [41]. Using the character "Q", an unsigned 8 byte integer is defined. Because of a missing ">" it is defined as LITTLE ENDIAN. Of course, in the dump the FILESYSTEMNAME is given in BIG ENDIAN. Nevertheless I decided to extract this value as LITTLE ENDIAN because of the following two reasons:

- All values are stored in LITTLE ENDIAN, so treating strings as LITTLE ENDIAN as well might simplify coding.
- After extracting this string, the hex-values will be transformed into a decimal and will be compared against a control-value. Reading it as LITTLE ENDIAN will shorten the calculated value; in the case of strings with less than 8 characters.

After extracting the string for the FILESYSTEMNAME, its decimal correspondence is compared against the decimal correspondence for the string REFS. In the case of accordance, the tool will continue, otherwise it will end with a corresponding message (lines 10 - 14). Lines 15 and 16 will jump to the offset for the byte-size of the VBR and will extract the value; the next line will extract the checksum, deposited in the VBR. Lines 20 to 34 are the python adaption of the c+ -code, delivered by microsoft [24], which is compared against the deposited checksum in lines 36 to 39. Lines 40 to 48 will extract further information from the VBR (see breakdown table 10). Line 49 fills a dictionary called VBR with the extracted data; the dictionary is returned, after closing the data stream.

The content of the dictionary VBR is piped into another function (code is not displayed here), which will take over the reporting function. In my tool I decided to create an output on the commandline; another output is produced as a report file in HTML - format.

A.3.3 Tree-Control

The first data, needed to be parsed, is the ENTRYBLOCK containing the $TREE_CONTROL. As it has been described already multiple times within this master thesis, the $TREE_CONTROL is located in the third last ENTRYBLOCK of the partition **??**. One finding during coding the tool was, that there is another copy of the $TREE_CONTROL in ENTRYBLOCK 0x1E, decimal 30 (see section **??**).

```
 1 def tree_control(_offset_part,_number_tree):
 2         offset_object_tree=_offset_part + _number_tree*16384
 3         f.seek(offset_object_tree+88)
 4         amount_record=struct.unpack_from("I", f.read(4))[0]
 5         offset_record=struct.unpack_from("I", f.read(4))[0]
 6         f.seek(offset_object_tree+offset_record)
 7         i=0
 8         node=[]
 9         for i in range(amount_record):
10                 offset_entry_block=struct.unpack_from("H", f.read(2))[0]
11                 node.append(offset_entry_block)
12                 f.read(22)
13                 i+=1
14         return node
```

Listing A.3: Function for analyzing the $TREE_CONTROL

This function is called with the offset of the partition and the ENTRYBLOCK - number of the $TREE_CONTROL. These values are used to calculate the byte-offset of the $TREE_CONTROL (line 2). Line 4 extracts the amount of records in the $TREE_CONTROL, while line 5 extracts the offset to the first record of the $TREE_CONTROL. The ENTRYBLOCK - numbers for the single objects, contained in the $TREE_CONTROL are extracted and appended to a list-variable, called "node".

Before looking at the code for parsing the single records of the $TREE_CONTROL, another code snippet is needed. The following code is needed in this form and functionality many times within my tool, because it returns the pointer of an ENTRYBLOCK. The code is displayed and explained here and will be referenced next time it is needed.

```
def tree_control_nodes(_offset_part,_node):
        offset_node=_offset_part+_node*16384
        (length_node_descriptor,amount_records)=node_descriptor(offset_node)
        pointers=node_pointers(offset_node+48+length_node_descriptor,amount_records,_node)
        return pointers,length_node_descriptor
```

Listing A.4: Function for analyzing the basic information of an ENTRYBLOCK

The function above is called with the values for the partition offset and the ENTRYBLOCK - number. After calculating the byte offset for the ENTRYBLOCK, two other functions are called; the first for extracting the information of the NODE DESCRIPTOR; the latter for extracting the information, amount and offset for the pointers from the NODE HEADER.

```
def node_descriptor(_offset):
        f.seek(_offset+48)
        length=struct.unpack_from("H", f.read(2))[0]
        f.seek(_offset+80)
        amount=struct.unpack_from("H", f.read(2))[0]
        return length,amount
```

Listing A.5: Function for analyzing the NODE DESCRIPTOR

The function is called with the byte-offset to the ENTRYBLOCK. Because of the fix value of 0x30 bytes, decimal 48, for the size of the ENTRYBLOCK DESCRIPTOR, this function might not work on every ENTRYBLOCK, but otherwise there were only a few ENTRYBLOCKS with a deviating size (e.g. $UPCASE TABLE), so these deviating blocks are analyzed in a customized form. This function extracts the values for the amount of records in the node and the length of the descriptor as a tuple.

```
 1 def node_pointers(_offset,_amount_records,_node):
 2         f.seek(_offset +16)
 3         first_pointer=struct.unpack_from("I", f.read(4))[0]
 4         amount=struct.unpack_from("I", f.read(4))[0]
 5         if _amount_records != -1:
 6                 check_pointers_records(amount,_amount_records,_node)
 7         f.seek(_offset+first_pointer)
 8         j=0
 9         pointers=[]
10         for j in range(amount):
11                 offset_record=struct.unpack_from("I", f.read(4))[0]
12                 pointers.append(offset_record)
13                 j+=1
14         return pointers
```

Listing A.6: Function for analyzing the NODE HEADER

This function is called with the same values as the function above A.5. Additionally it needs the number of the records in the node. At this point, it should be noted that the value "_offset" in this function differs from the value in the function above, because calling this function, the offset and the length for the NODE DESCRIPTOR are cumulated already while calling the function (see listing A.4 line 4). The function extracts the amount of pointers and the offset to the first pointer, line 3 to 4. Line 5 to 6 checks whether the number of records equals the number of pointers. Afterwards, the pointers are extracted and appended to a list-variable called "pointers", which is returned to the calling code.

```
#Calling the single entries of the records_tree_control for further examination
        for i in range(len(records_tree_control)):
#Looking up the pointers and the length of the nodedescriptor
                (pointers,length_node_descriptor)=tree_control_nodes(offset, records_tree_control[i])
#$Object_Tree
                if i == 0:
#Parsing the $Object_Tree, returning the Node_ID, the Cluster-Offset and an unknown value
                        object_records=object_record_parser(offset+records_tree_control[i]*16384+48+
                        ~ length_node_descriptor,pointers)
                        for k in object_records:
                                f.seek(offset+k['Cluster Offset']*16384+24)
                                node_id=struct.unpack_from("I", f.read(4))[0]
#Parsing the Node_ID 0x500 - $System
                                if hex(node_id) == "0x500":
                                        parsed_nodes['Node '+str(hex(node_id))]=node_500_parser(offset+
                                        ~ k['Cluster Offset']*16384)
```

```
17 #Parsing the Node_ID 0x600 - Root-Directory
18                                 else:
19                                         f.seek(offset+k['Cluster Offset']*16384+72)
20                                         test_extent=struct.unpack_from("B",f.read(1))[0]
21                                         if test_extent == 0 :
22                                                 parsed_nodes['Node '+str(hex(node_id))]=node_parser(offset+
23                                                 ~ k['Cluster Offset']*16384)
24                                         else:
25                                                 parsed_nodes=node_extents(offset, offset+k['Cluster Offset']
26                                                 ~ *16384, node_id, parsed_nodes)
27                 if i == 1:
28                         free_space_lrg=allocator_parser(offset+records_tree_control[i]*16384+48+
29                         ~ length_node_descriptor,pointers,4096,32)
30                 if i == 2:
31                         free_space_med=allocator_parser(offset+records_tree_control[i]*16384+48+
32                         ~ length_node_descriptor,pointers,4,128)
33                 if i == 3:
34                         free_space_sml=allocator_parser(offset+records_tree_control[i]*16384+48+
35                         ~ length_node_descriptor,pointers,1,128)
36                 if i == 5:
37                         dir_structure=directory_structure(offset+records_tree_control[i]*16384+48+
38                         ~ length_node_descriptor,pointers)
39         f.close()
40         return partition_size,parsed_nodes,free_space_lrg,free_space_med,free_space_sml,dir_structure
```

Listing A.7: Function for parsing the $TREE_CONTROL

This is a part of the code that joins the functions named above. This code-snippet has the task to parse the single records of the $TREE_CONTROL. The $TREE_CONTROLS I observed, while writing this master thesis, all had the same order of $System-Files, so I used the observed order to simplify the code.

$Object_Tree

The first record always contained the offset to the $System-File, called $OBJECT_TREE by ENCASE. Iterating over the records of the $TREE_CONTROL, starting in line 2, special code is called for the single records. Starting in line 6, the $OBJECT_TREE IS PARSED. Therefore a snippet of code (not displayed here) parses the records of the $OBJECT_TREE, extracting the node IDs and the corresponding ENTRYBLOCKS. According to the node IDs, special code-snippets for the different $System-Files are called, starting with a parsing code for the node 0x500, containing metadata information to the filesystem, like the volume label, the offset to the UPCASE TABLE, timestamps for creating and last mounting of the filesystem. The extracted information is stored in a dictionary, named accordingly to the node ID. This dictionary is added to the dictionary "parsed_nodes", which was initialized to store all information to the parsed nodes.

```
1  def node_500_parser(_offset):
2          node_500={}
3          (length_node_descriptor,amount_records)=node_descriptor(_offset)
4          pointers=node_pointers(_offset+48+length_node_descriptor,-1,0)
5          for j in pointers:
6                  offset_record=_offset+48+length_node_descriptor+j
7                  f.seek(offset_record+16)
8                  counter_node_id=struct.unpack_from("I", f.read(4))[0]
9                  if hex(counter_node_id) == "0x510":
10                         f.seek(offset_record+10)
11                         start_name=struct.unpack_from("H", f.read(2))[0]
12                         length_name=struct.unpack_from("H", f.read(2))[0]
13                         f.seek(offset_record+start_name)
14                         label_volume=struct.unpack("B"*length_name, f.read(length_name))
15                         volume_label=""
16                         i=0
17                         for k in label_volume:
```

```
18                                  if not i%2:
19                                          volume_label+=chr(k)
20                                  i+=1
21                          node_500['Volume Label']=volume_label
22                  if hex(counter_node_id) == "0x520":
23                          f.seek(offset_record+168)
24                          node_500['Created']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
25                          f.seek(offset_record+184)
26                          node_500['Accessed']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
27                  if hex(counter_node_id) == "0x530":
28                          f.seek(offset_record+224)
29                          node_500['Upcase Table']=struct.unpack_from("I",f.read(4))[0]
30          return node_500
```

Listing A.8: Function for analyzing the node 0x500

In the partitions I analyzed, the node 0x500 contained different records with a differing structure, so those records received different code-snippets, according to a hex-value found within the records (0x510, 0x520 etc..). The volume label is given in unicode. In order to display it as a readable string, only every second character is stored to the variable volume_label, line 17 to 20. In the case, the second byte in unicode is used, my code has to be adapted.

The next records of the $OBJECT_TREE are constructed in the same way, so only one function is needed. This function contains approximately 200 lines of code, so it is only partially shown here.The code is divided into several blocks, according to the different attribute identifier.

```
if hex(attribute_identifier) == "0x10000000":
        f.seek(_offset+48+length_node_descriptor+pointers[j])
        record_length=struct.unpack_from("I",f.read(4))[0]
        f.read(2)
        record_inside_amount=struct.unpack_from("H",f.read(2))[0]
        f.read(2)
        record_header_length=struct.unpack_from("H",f.read(2))[0]
        inside_offset=_offset+48+length_node_descriptor+pointers[j]+24
#Parsing the records inside the first record
        for m in range(record_inside_amount):
                f.seek(inside_offset)
                record_inside_length=struct.unpack_from("I",f.read(4))[0]
#Parsing the timestamps
                if m == 0:
                        offset_to_timestamp=struct.unpack_from("H",f.read(2))[0]
                        f.seek(inside_offset+offset_to_timestamp)
                        nodes['Created']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
                        nodes['Modified']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
                        nodes['Metadata Modified']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
                        nodes['Last Accessed']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
                        f.read(8)
                        int_node_id=struct.unpack_from("I",f.read(4))[0]
```

Listing A.9: Function for analyzing the directory metadata, part 1

The first parsed record is the directory metadata record, having the attribute identifier 0x10000000. This record contains different records itself, so there is another iteration inside. The first inside-record contains the four known timestamps for the directory. The meaning of the second and third inside record could not be revealed up to the point of writing, so they have been skipped here. The fourth inside-record contains different values, according to the kind of node it belongs to. In the case it is contained in the node 0x600 or node 0x520, the same information is extracted; only the kind of node is named differently, either ROOT DIRECTORY, or FOLDER.

```
if hex(int_node_id) == "0x600":
        f.seek(inside_offset+16)
        pointer_to_name_end=struct.unpack_from("H",f.read(2))[0]
        f.seek(inside_offset+ 24)
        length_name=struct.unpack_from("H",f.read(2))[0]
        pointer_to_name_start=pointer_to_name_end - length_name
        f.seek(inside_offset + 16+pointer_to_name_start)
        name=struct.unpack_from("B"*length_name, f.read(length_name))
        dir_label=""
        i=0
        for k in name:
                if not i%2:
                        dir_label+=chr(k)
                i+=1
        nodes['Dir Label']=dir_label
        nodes['Node Typ: ']="Root Directory"
        nodes['Attribute']=attribute_identifier
        nodes['Node ID']=int_node_id
        node['Node: '+str(count)]=nodes
```

Listing A.10: Function for analyzing the directory metadata, part 2

If the inside-record is contained in another node, the node type is also FOLDER, but, additional to the named information above, four alternate timestamps are provided and extracted.

Other records are categorized according to the attribute identifier contained. For both folder and files, a filename attribute identifier was discovered. Both record-types start with the same construction the code-snippet is shown on the example of the FNA for files.

```
#Distinguishing Files from Folders
if hex(attribute_identifier) == "0x30000100":
        file_name=""
        i=0
        for k in name:
                if not i%2:
                        file_name+=chr(k)
                i+=1
        f.seek(_offset+48+length_node_descriptor+pointers[j]+second_structure)
        meta_data_record_length=struct.unpack_from("I",f.read(4))[0]
        meta_data_offset_timestamp=struct.unpack_from("H",f.read(2))[0]
        f.seek(_offset+48+length_node_descriptor+pointers[j]+second_structure+meta_data_offset_timestamp)
        nodes['Attribute']=attribute_identifier
        nodes['File Name']=file_name
        nodes['Created']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
        nodes['Modified']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
        nodes['Metadata Modified']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
        nodes['Last Accessed']=getFiletime(struct.unpack_from("Q",f.read(8))[0])
```

Listing A.11: Function for analyzing the FNA, part 1

The code snippet above is nearly the same for both type of FNA; differences are in the order of the contained information. The FNA for files contains additional information to the child id; the logical and physical file-size etc. The parsing code for this information is not significant, so it is not displayed here. Another piece of information, given in the FNA for files, is the data-run. The data-run, itself, contains two records; the second contains the offsets and cluster length for the single data-runs.

```
second_datarun_offset_pointers=node_pointers(offset_first_datarun+data_run_first_subrecord_length,-1,0)
z=1
datarun=[]
for x in second_datarun_offset_pointers:
        offset_data_run=offset_first_datarun+data_run_first_subrecord_length+x
        f.seek(offset_data_run)
        length_data_run=struct.unpack_from("I",f.read(4))[0]
        f.read(20)
        amount_clusters=struct.unpack_from("Q",f.read(8))[0]
        start_cluster=struct.unpack_from("Q",f.read(8))[0]
        run={}
        run['DataRun']=z
        run['Amount Clusters']=amount_clusters
        run['Size Run']=amount_clusters*16384
        run['Start Cluster']=start_cluster
        datarun.append(run)
        z+=1
nodes['DataRun']=datarun
```

Listing A.12: Function for analyzing the FNA, part 2

The single data-runs are numbered (appended to the list-variable data-run), which itself is added to the directory nodes. The last type of record, while parsing the ENTRYBLOCKS, is the CHILDRECORD, which has the attribute identifier 0x20000080. This record contains information to the parent node and the child ID; coding does not reveal specificities, so the code is not displayed here.

Allocator-Files

The next three records are the the $Allocator-Files. The structure of the three types of $Allocator-files is the same, even if they allocate different block sizes, so the same code-snippet can be used for them.

```
 1 def allocator_parser(_offset,pointers,blocks_per_bit,max_bytes):
 2         k=0
 3         j=0
 4         free_cluster=0
 5         for k in pointers:
 6                 table=[]
 7                 start_record=_offset+pointers[j]
 8                 f.seek(start_record+16)
 9                 starting_block=struct.unpack_from("Q", f.read(8))[0]
10                 amount_blocks=struct.unpack_from("Q", f.read(8))[0]
11                 f.seek(start_record+64)
12                 offset_start_table=struct.unpack_from("I", f.read(4))[0]
13                 length_table=struct.unpack_from("I", f.read(4))[0]
14                 for m in range(length_table):
15                         var=struct.unpack_from("B", f.read(1))[0]
16                         table.append(var)
17                 real_length_table=amount_blocks/blocks_per_bit/8
18                 if real_length_table != max_bytes:
19                         free_cluster+=calc_free_space(real_length_table,8,table)
20                         free_cluster+=calc_free_space(1,amount_blocks/blocks_per_bit%8,table)
21                 else:
22                         free_cluster+=calc_free_space(len(table),8,table)
23                 allocator_record={'Record No':j,'Allocation Table':table,'Starting Block':starting_block,
24                 ~ 'Amount Blocks':amount_blocks,'Offset Start Table':offset_start_table,
25                 ~ 'Length Table':length_table}
26                 allocator_records.append(allocator_record)
27                 j+=1
28         free_space=free_cluster*blocks_per_bit*16384
29         return free_space
```

Listing A.13: Function for analyzing the $Allocator-Files

The function is called with the offset for: the $Allocator-file; the pointers; the number of blocks, which are allocated per bit and the maximum amount of bytes per allocation table (see also the calling code A.7). The starting block and the amount of blocks in each record are extracted in lines 9 and 10; the offset to the start of the table and the length of the table are extracted in lines 12 and 13. The bytes of the allocator table are extracted individually and added to the list variable "table". Because the size of a filesystem normally does not equal a multiple of 16 GiB (512 MiB allocated per byte in $Allocator_Lrg multiplied by 32, the maximum bytes in $Allocator_Lrg-table), the allocation tables are not fully filled. Firstly, the number of bytes used in the allocator table is calculated. Therefore the number of blocks is divided by the number of blocks per bit and by 8 (line 17). Because dividend and divisors are all integers, the result is also an integer. If the result equals the maximum number of bytes, the allocation table is filled completely; the else loop is called (line 22). Otherwise, the free _space_calc function is called for the integer value of included bytes; afterwards the same function is called for the remainder value.

```
1 def calc_free_space(length,last_byte,table):
2         free=0
3         for n in range(length):
4                         for o in range(last_byte):
5                                 vars=testBit(table[n],o)
6                                 if vars == 0:
7                                         free+=1
8         return free
```

Listing A.14: Function for adding unallocated bytes

The function is called with the integer value for the amount of bytes or the value one, in the case of calculating the remainder-byte; the value 8 for the integer calculation, because here every byte contains eight allocations. In the case of the remainder byte the remainder is transmitted as well as the allocation table. For every byte the test_bit function is called either eight times, in case of an integer byte, or as often as the remainder defines.

```
def testBit(integer,offset):
        mask=1 << offset
        return (integer & mask)
```

Listing A.15: Function for bitwise operation on every byte

This function takes the allocation byte and returns, according to the state of the calling loop, the corresponding bit, so the calling function calls every returned zero-bit because this identifies an unallocated byte (line 6 and 7 of listing A.14). The amount of zero bits is returned to listing A.13. According to the type of the allocation table (large, medium, small), the amount of free bits is multiplied by the amount of blocks_per_bit; the value this function is called with (line28 of listing A.13).

$Attribute_List

This record of the $TREE_CONTROL is not parsed in my tool, because I could not discover the meaning of this record, until the time of writing this master thesis.

$Object

Last record in the $TREE_CONTROL is the record called $Object by ENCASE; rightly it should be named $OBJECT_TREE (see section **??**).

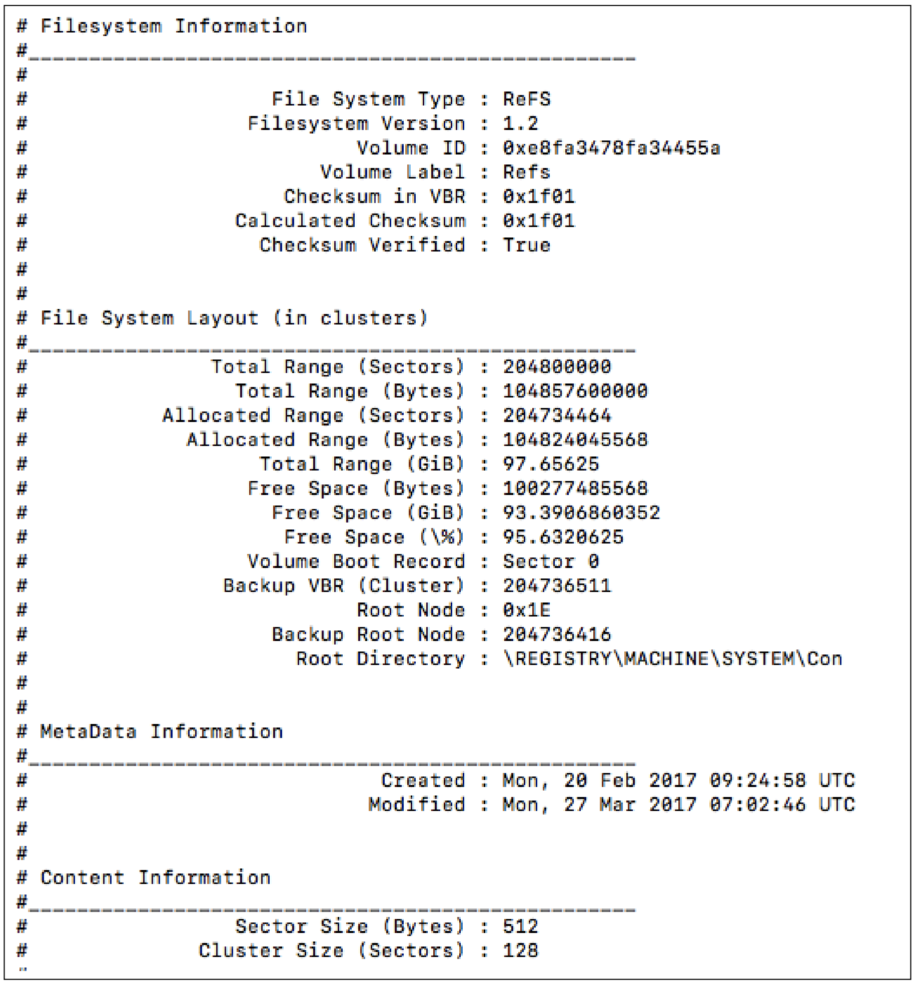
```
 1 def directory_structure(_offset,pointers):
 2                 dir_structure=[]
 3                 j=0
 4                 for k in pointers:
 5                         f.seek(_offset+k+24)
 6                         node_id=struct.unpack_from("I", f.read(4))[0]
 7                         f.seek(_offset+k+40)
 8                         child_id=struct.unpack_from("I", f.read(4))[0]
 9                         if len(dir_structure) == 0:
10                                 j+=1
11                                 node_name="Node"+str(j)
12                                 node={}
13                                 node[node_name]=node_id
14                                 node["Child"]=[]
15                                 if child_id != 0:
16                                         node["Child"].append(child_id)
17                                 dir_structure.append(node)
18                         else:
19                                 control=0
20                                 for i in range(len(dir_structure)):
21                                         node_name="Node"+str(i)
22                                         cache=dir_structure[i]
23                                         if node_id in cache.values():
24                                                 cache["Child"].append(child_id)
25                                                 control+=1
26                                 if control == 0:
27                                         j+=1
28                                         node_name="Node"+str(j)
29                                         node={}
30                                         node[node_name]=node_id
31                                         node["Child"]=[]
32                                         if child_id != 0:
33                                                 node["Child"].append(child_id)
34                                         dir_structure.append(node)
35                 return dir_structure
```

Listing A.16: Function for parsing the $Object

This function parses every node of the $OBJECT for the value of the node ID at offset 018, decimal 24, and the value for the child ID at offset 0x28, decimal 40, up to line 8. The directory structure should be stored in a corresponding variable, defined as a list. For the first parsed node, a check whether if the node already exists in the variable is not needed, so a dictionary variable node is initialized; the node ID is added; the child ID is appended to a list variable child, which is added to the node variable. The whole node dictionary is appended to the dir_structure variable (lines 9 to 17).
Beginning with the second record of the $OBJECT, it is neccessary to check whether if there is already an entry for the parent node within the dir_structure variable. If the node ID is already listed, the child ID is appended to the child list (lines 20 to 25), otherwise a new node is initialized and the child ID is appended to a new child-list. The directory structure is not visualized automatically; the user has to choose a special option in the main menu.

Summary $Tree_Control

At this point, all necessary information from the records of the $TREE_CONTROL are extracted and stored in the variable parsed_nodes. The rest of the tool will refer to this variable. Furthermore, the code will hand over the variable parsed_nodes to the output-function (already described in the last lines of subsection VBR (A.3.2)), which gives the user the filesystem information.

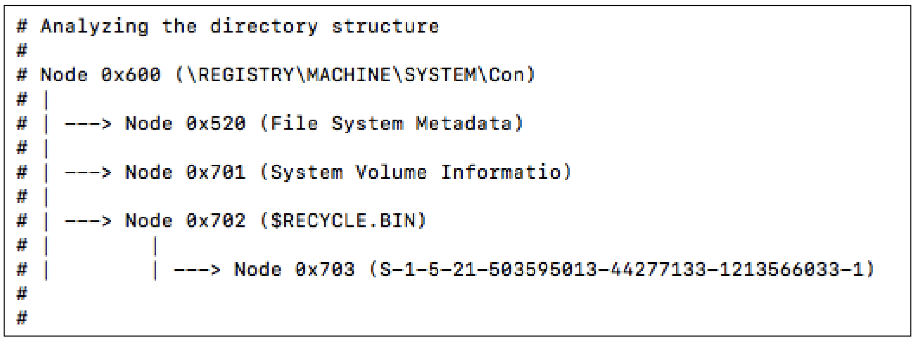


```
# Filesystem Information
#___________________________________________________
#
#                   File System Type : ReFS
#                 Filesystem Version : 1.2
#                          Volume ID : 0xe8fa3478fa34455a
#                       Volume Label : Refs
#                    Checksum in VBR : 0x1f01
#                Calculated Checksum : 0x1f01
#                  Checksum Verified : True
#
#
# File System Layout (in clusters)
#___________________________________________________
#              Total Range (Sectors) : 204800000
#                Total Range (Bytes) : 104857600000
#          Allocated Range (Sectors) : 204734464
#            Allocated Range (Bytes) : 104824045568
#                  Total Range (GiB) : 97.65625
#                 Free Space (Bytes) : 100277485568
#                   Free Space (GiB) : 93.3906860352
#                     Free Space (\%) : 95.6320625
#                 Volume Boot Record : Sector 0
#               Backup VBR (Cluster) : 204736511
#                          Root Node : 0x1E
#                   Backup Root Node : 204736416
#                     Root Directory : \REGISTRY\MACHINE\SYSTEM\Con
#
#
# MetaData Information
#___________________________________________________
#                            Created : Mon, 20 Feb 2017 09:24:58 UTC
#                           Modified : Mon, 27 Mar 2017 07:02:46 UTC
#
#
# Content Information
#___________________________________________________
#                Sector Size (Bytes) : 512
#             Cluster Size (Sectors) : 128
```

Figure 46: Screenshot from the Fsstat of my tool

A.3.4 Analysis of the extracted information

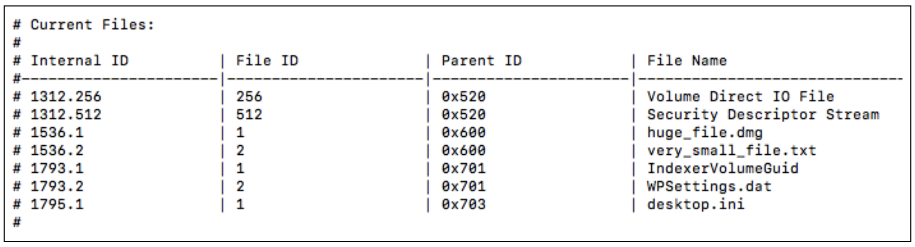
Below, some screenshots of the output from my tool are provided, starting with the visualized output from the $OBJECT, which is requested by choosing option 3 in the mainmenu.



```
# Analyzing the directory structure
#
# Node 0x600 (\REGISTRY\MACHINE\SYSTEM\Con)
# |
# | ---> Node 0x520 (File System Metadata)
# |
# | ---> Node 0x701 (System Volume Informatio)
# |
# | ---> Node 0x702 ($RECYCLE.BIN)
# |         |
# |         | ---> Node 0x703 (S-1-5-21-503595013-44277133-1213566033-1)
#
#
```

Figure 47: Screenshot from the visualized directory structure

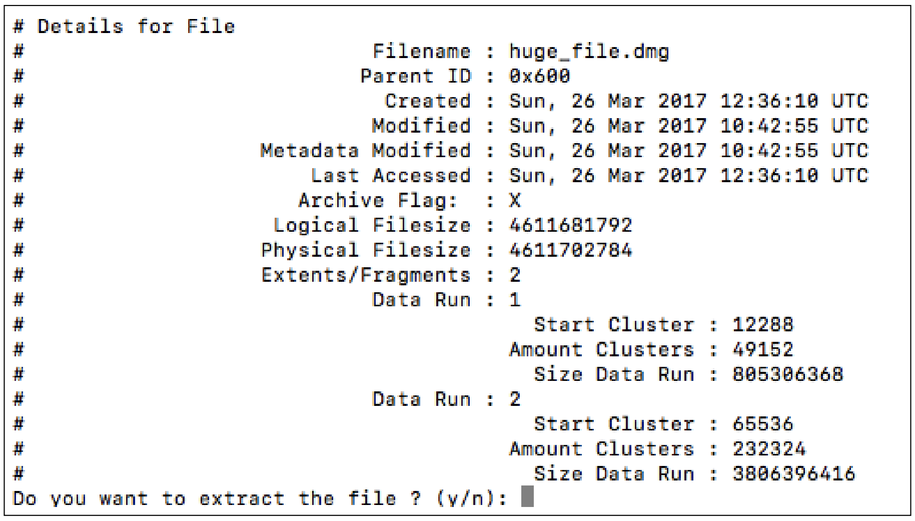
The next option in the mainmenu is the fls-option, known from sleuthkit, which provides a list of the current files in the filesystem.



```
# Current Files:
#
# Internal ID          | File ID              | Parent ID            | File Name
#----------------------|----------------------|----------------------|------------------------------
# 1312.256             | 256                  | 0x520                | Volume Direct IO File
# 1312.512             | 512                  | 0x520                | Security Descriptor Stream
# 1536.1               | 1                    | 0x600                | huge_file.dmg
# 1536.2               | 2                    | 0x600                | very_small_file.txt
# 1793.1               | 1                    | 0x701                | IndexerVolumeGuid
# 1793.2               | 2                    | 0x701                | WPSettings.dat
# 1795.1               | 1                    | 0x703                | desktop.ini
#
```

Figure 48: Screenshot from the list of current files

When creating this overview, I realized that a child ID could be assigned several times, at least in different nodes, so or so I had to create a more specific id, a combination of the parent ID and the child id, named internal id. This internal ID is needed for the file details, therefore the internal ID is split into its components; afterwards the information can be drawn out of the node-dictionary, created by parsing the single nodes.

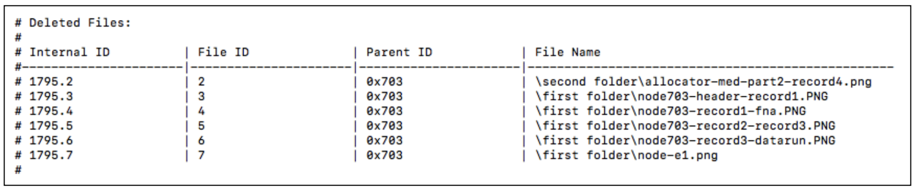


```
# Details for File
#                          Filename : huge_file.dmg
#                         Parent ID : 0x600
#                           Created : Sun, 26 Mar 2017 12:36:10 UTC
#                          Modified : Sun, 26 Mar 2017 10:42:55 UTC
#                 Metadata Modified : Sun, 26 Mar 2017 10:42:55 UTC
#                     Last Accessed : Sun, 26 Mar 2017 12:36:10 UTC
#                     Archive Flag:  : X
#                  Logical Filesize : 4611681792
#                 Physical Filesize : 4611702784
#                Extents/Fragments : 2
#                          Data Run : 1
#                                      Start Cluster : 12288
#                                    Amount Clusters : 49152
#                                      Size Data Run : 805306368
#                          Data Run : 2
#                                      Start Cluster : 65536
#                                    Amount Clusters : 232324
#                                      Size Data Run : 3806396416
Do you want to extract the file ? (y/n): █
```

Figure 49: Screenshot from file details

The last piece information shown in the screenshot is the DataRuns. This information is used for extracting the files to a path, which is to be specified after entering a "y" to the question at the bottom.

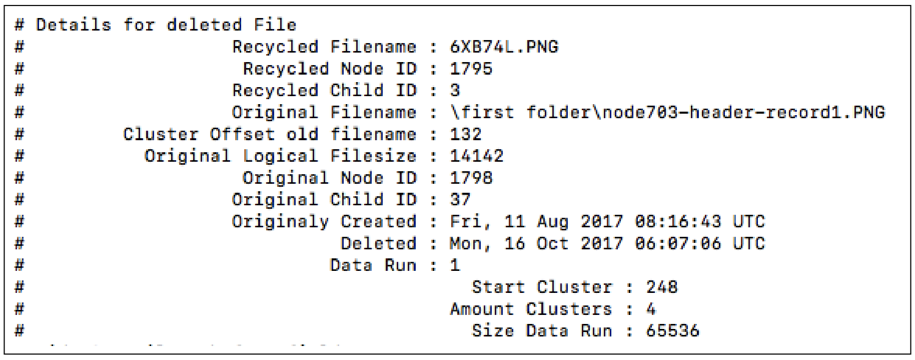
The last option in the mainmenu concerns the $Recycle.bin.



```
# Deleted Files:
#
# Internal ID            | File ID              | Parent ID            | File Name
#------------------------|----------------------|----------------------|-------------------------------------------------
# 1795.2                 | 2                    | 0x703                | \second folder\allocator-med-part2-record4.png
# 1795.3                 | 3                    | 0x703                | \first folder\node703-header-record1.PNG
# 1795.4                 | 4                    | 0x703                | \first folder\node703-record1-fna.PNG
# 1795.5                 | 5                    | 0x703                | \first folder\node703-record2-record3.PNG
# 1795.6                 | 6                    | 0x703                | \first folder\node703-record3-datarun.PNG
# 1795.7                 | 7                    | 0x703                | \first folder\node-e1.png
#
```

Figure 50: Screenshot from the $RECYCLE.BIN

According to the code for the current files, the files from the $RECYCLE.BIN also get individualized by an internal ID; here I used the node ID from the $RECYCLE.BIN-node, not from the original parent folder. Recalling the information for a single file from the $RECYCLE.BIN, the internal ID is split into its components, but, as described in the $RECYCLE.BIN-section **??**, for every deleted file a $R and a $I record exists; one containing the information for the record within the $RECYCLE.BIN-node; the other one for the record in the original node. On the one hand, the tool has to extract the information from all the named sources. On the other hand, this information has to be presented in a meaningful manner.



```
# Details for deleted File
#                     Recycled Filename : 6XB74L.PNG
#                      Recycled Node ID : 1795
#                     Recycled Child ID : 3
#                     Original Filename : \first folder\node703-header-record1.PNG
#          Cluster Offset old filename : 132
#            Original Logical Filesize : 14142
#                      Original Node ID : 1798
#                     Original Child ID : 37
#                     Originaly Created : Fri, 11 Aug 2017 08:16:43 UTC
#                               Deleted : Mon, 16 Oct 2017 06:07:06 UTC
#                              Data Run : 1
#                                           Start Cluster : 248
#                                         Amount Clusters : 4
#                                           Size Data Run : 65536
```

Figure 51: Screenshot from file details (deleted files)

After displaying the information for the deleted files, the user is asked to extract the file again, by which all the functions of my tool are described.

A.4 Templates

A.4.1 VBR

PART	OFFSET	LENGTH	DESCRIPTION
VBR	0x00	0x03	Jump Instruction
	0x03	0x08	FileSystemName
	0x07	0x05	Reserved space, containing all zeros
	0x10	0x04	Structure Identifier
	0x14	0x02	Number of bytes in the VBR
	0x16	0x02	Checksum for the FSRS
	0x18	0x08	Offset to BACKUPVBR
	0x20	0x04	Bytes per Sector
	0x24	0x04	Sectors per Cluster
	0x28	0x01	Filesystem Major Version
	0x29	0x01	Filesystem Minor Version
	0x2A	0x02	unknown
	0x2C	0x04	unknown
	0x30	0x08	unknown
	0x38	0x08	Volume Serial Number (LE)

Table 10: VBR

A.4.2 $Tree_Control

PART	OFFSET	LENGTH	DESCRIPTION
Tree Control Node	0x00	0x30	ENTRYBLOCKDescriptor
	0x00	0x08	ENTRYBLOCK-number
	0x20	0x04	unknown
	0x30	0x10	unknown
	0x50	0x04	Offset to first pointer to extents
	0x54	0x04	amount of extents
	0x58	0x04	Offset to record
	0x5C	0x04	Length of the record
Extent (rel. offset)	0x00	0x08	ENTRYBLOCK-number
	0x3C	0x04	Length of record
	0x58	0x04	Amount of records in the extent
			Followed by the pointers
	0x5C	0x04	Pointer to the first record
	0x98	0x18	1st record

Table 11: Breakdown-table for the $TREE_CONTROL

A.4.3 $Object_Tree

PART	OFFSET	LENGTH	DESCRIPTION
EntryBlock Descriptor	0x00	0x08	ENTRY-BLOCK-Number
	0x00	0x30	Entry-Block-Descriptor
Node Descriptor	0x30	0xF0	Node-Descriptor
	0x30	0x04	Length of NODE-DESCRIPTOR
	0x50	0x04	amount of records in the node
Node Header	0x120	0x20	length of Header
	0x124	0x04	offset to next free record
	0x128	0x04	free space in the node
	0x12C	0x04	
	0x130	0x04	offset to first pointer
	0x134	0x04	amount of pointers in this node
	0x138	0x08	offset to end of node
Record (rel. offset)	0x00	0x04	length of record
	0x18	0x04	Node ID
	0x20	0x04	EntryBlock - number (cluster offset)
	0x30	0x04 / 0x08?	unknown (Identifier??)

Table 12: $OBJECT_TREE

A.4.4 $Allocator_Lrg, $Allocator_Med, $Allocator_Sml

PART	OFFSET	LENGTH	DESCRIPTION
EntryBlock Descriptor	0x00	0x08	ENTRY-BLOCK-Number
	0x00	0x30	Entry-Block-Descriptor
Node Descriptor	0x30	0x38	Node-Descriptor
	0x30	0x04	Length of NODE-DESCRIPTOR
	0x50	0x04	amount of records in the node
Node Header	0x68	0x20	length of Header
	0x6C	0x04	offset to next free record
	0x70	0x04	free space in the node
	0x74	0x04	
	0x78	0x04	offset to first pointer
	0x7C	0x04	amount of pointers in this node
	0x80	0x08	offset to end of node
Record (rel. offset)	0x00	0x10	File-Signature????
	0x10	0x08	Starting ENTRYBLOCK described in this record
	0x18	0x08	Number of ENTRYBLOCKs described in this record
	0x20	0x08	????
	0x28	0x08	????
	0x30	0x08	????
	0x38	0x08	????
	0x40	0x04	Length of Header, beginning after the signature
	0x44	0x04	Length of the Allocation Table
	0x48	0x80	Allocation Table

Table 13: $ALLOCATOR_LRG,$ALLOCATOR_MED, $ALLOCATOR_SML

A.4.5 $Object

PART	OFFSET	LENGTH	DESCRIPTION
		0x30	ENTRYBLOCK - Descriptor
			NODE DESCRIPTOR
		0x20	NODE HEADER
$OBJECT (rel. offset)	0x00	0x02	length of record
	0x18	0x02	parent node id
	0x28	0x02	child node id

Table 14: $OBJECT

A.4.6 EntryBlock

Part	Offset	Length	Descritpion
EntryBlock Descriptor	0x00	0x08	Entry-Block-Number
	0x00	0x30	Entry-Block-Descriptor
	0x08	0x08	counter for copies of the EntryBlock
	0x18	0x08	Node ID
Node Descriptor (rel. offset)	0x00	0x38 / 0xE8	Node-Descriptor
	0x00	0x04	Length of Node-Descriptor
	0x18	0x02	amount of extents
	0x20	0x04	amount of records in the node
Node Header (rel. offset)	0x00	0x20	length of Header
	0x04	0x04	offset to next free record
	0x08	0x04	free space in the node
	0x0C	0x04	
	0x10	0x04	offset to first pointer
	0x14	0x04	amount of pointers in this node
	0x18	0x08	offset to end of node

Table 15: EntryBlock (Descriptor, Node-Descriptor and Node-Header)

A.4.7 Directory Metadata Record (0x10000000)

Part	Offset	Length	Description
Record Header	0x00	0x04	length of the record
	0x0A	0x02	offset to first record
	0x10	0x04	attribute identifier
Meta Data Attr.	0x00	0x04	Length of the attribute
	0x04	0x02	offset to first timestamp
	0x28	0x08	first timestamp
	0x30	0x08	second timestamp
	0x38	0x08	third timestamp
	0x40	0x08	fourth timestamp
	0x50	0x04	Node ID
Inside Record Header	0x04	0x02	length of header
	0x10	0x02	offset to first pointer
	0x12	0x02	amount of pointers
1. inside Record	0x20	0x04	length of unknown structure
2. inside Record (rel. Offset)	0x00	0x04	length of second record
	0x10	0x02	pointer to end of name
	0x18	0x02	length of name

Table 16: $Object

A.4.8 FileNameAttribute File (0x30000100)

Part	Offset	Length	Description
Fna	0x00	0x04	Length of the whole record
	0x0A	0x02	Length of the filename-attribute
	0x0C	0x04	remaining data in the record
	0x10	0x04	Attribute-Type identifier
	0x14	varies	filename

Table 17: FileNameAttribute -File

A.4.9 FileNameAttribute Folder (0x30000200)

PART	OFFSET	LENGTH	DESCRIPTION
FNA	0x00	0x04	Length of the whole record
	0x0A	0x02	offset to end of name
	0x0C	0x04	remaining data in the record
	0x10	0x04	Attribute-Type identifier
	0x14	varies	foldername
	value offset 0x0A	0x08	Node ID
	value offset 0x0A + 0x10	0x08	CREATED
	value offset 0x0A + 0x18	0x08	MODIFIED
	value offset 0x0A + 0x20	0x08	METADATA MODIFIED
	value offset 0x0A + 0x28	0x08	LAST ACCESSED
			Flags for Files, 0x00 - no flags, 0x01 - read only, 0x02 - hidden, 0x20 - archive

Table 18: FILENAMEATTRIBUTE -Folder

A.4.10 MetaDataAttribute

PART	OFFSET	LENGTH	DESCRIPTION
META DATA ATTR.	0x00	0x04	Length of the attribute
	0x04	0x02	possible offset to first value in the attribute??
	0x28	0x08	first timestamp
	0x30	0x08	second timestamp
	0x38	0x08	third timestamp
	0x40	0x08	fourth timestamp
	0x48	0x01	Flags for Files, 0x00 - no flags, 0x01 - read only, 0x02 - hidden, 0x20 - archive
	0x50	0x04	Parent Node ID
	0x58	0x04	Child ID
	0x60	0x08	unknown value
	0x68	0x08	logical filesize
	0x70	0x08	physical filesize

Table 19: METADATAATTRIBUTE

A.4.11 DataRunAttribute

PART	OFFSET	LENGTH	DESCRIPTION
Header	0x00	0x20	Header for the whole structure
	0x00	0x04	Length of the header
	0x04	0x04	offset to next free record
	0x08	0x04	free space in the node
	0x10	0x04	offset to the pointer
	0x14	0x04	amount of pointers
	0x18	0x04	end of the structure
Header	0x20	0x20	next header structure
	0x20	0x04	length of the structure
	0x30	0x04	amount of data within the structure, except header
	0x38	0x04	Attribute-Type identifier??
Record	0x40	0x88	record
	0x40	0x04	length of the record
	0x74	0x08	physical size of the file
	0x7C	0x08	logical size of the file
DATARUN ATTR.	0xC8	0x20	next header structure
	0xC8	0x04	length of the header
	0xCC	0x04	offset to next free record
	0xD0	0x04	free space in the node
	0xD8	0x04	offset to pointers
	0xDC	0x04	amount of pointers
	0xE0	0x04	offset to end of the structure
	0xE8	0x30	first datarun
	0xE8	0x04	length of the datarun
	0x100	0x08	amount of clusters in this datarun
	0x108	0x08	starting cluster of this datarun
	0x118	0x30	second datarun
	0x118	0x04	length of the datarun
	0x130	0x08	amount of clusters in this datarun
	0x138	0x08	starting cluster of this datarun
	0x198	0x04	first pointer of datarun structure
	0x19C	0x04	second pointer of datarun structure
	0x274	0x04	pointer of the whole structure

Table 20: DATARUNATTRIBUTE

A.4.12 Child Attribute (0x20000080)

Part	Offset	Length	Description
Child Attr.	0x00	0x04	Length of the whole record
	0x04	0x02	offset to attribute identifier
	0x06	0x02	remaining bytes in header
	0x0A	0x02	length of header
	0x0C	0x02	remaining bytes in record
	0x10	0x04	attribute identifier (0x20000080)
	0x18	0x04	Parent ID
	0x20	0x08	Child ID
	0x30	0x02	always 0x000C ?!
	0x32	0x02	length of name
	0x34		filename, written in unicode

Table 21: Child Attribute