Forensic Science International: Digital Investigation 32 (2020) 300915

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2020 EU – Proceedings of the Seventh Annual DFRWS Europe

Forensic Analysis of the Resilient File System (ReFS) Version 3.4

Paul Prade [a], Tobias Groß [a, *], Andreas Dewald [b, a, **]

[a] Friedrich-Alexander University Erlangen-Nürnberg, Germany
[b] ERNW Research GmbH, Heidelberg, Germany

ARTICLE INFO

Article history:

Keywords:
Digital forensics
Data recovery
File systems
ReFS

ABSTRACT

ReFS is a modern file system that is developed by Microsoft and its internal structures and behavior is not officially documented. Even so there exist some analysis efforts in deciphering its data structures, some of these findings have yet become deprecated and cannot be applied to current ReFS versions anymore. In this work, general concepts and internal structures found in ReFS are examined and documented. Based on the structures and the processes by which they are modified, approaches to recover (deleted) files from ReFS formatted file systems are shown. We also evaluated our implementation and the allocation strategy of ReFS with respect to accuracy, runtime and the ability to recover older file states.

1. Introduction

Storage media analysis is a common task in the field of digital forensic, when PCs or mobile devices get analyzed. Investigators have to rely on the proper functioning of their tools to provide them with correct interpretation of traces. File systems have to be interpreted and presented when analyzing storage media, doing this manually is unfeasible.

From this situation emerges the need for digital forensic tools to ideally support all of the file systems that are currently in use and may be encountered in a forensic analysis. Limitations of classical file systems such as low performance, limited capacity or unsuitability for SSD drives led to the development of new filesystems like APFS or ReFS. These new filesystems have to be supported in open source forensic tools and documentation. Transparency in forensic processes and tools is important when digital evidence is used in severe cases. That is even more important when filesystems are proprietary as the above-mentioned ones.

With this work, we want to provide the forensic community with tools and information to properly analyze ReFS partitions.

1.1. Related work

There exists related work that we show in the following which analyzed ReFS before, but they looked into versions 1.1 and 1.2. Some core concepts still exist in the latest version 3.4 but also major changes were applied between these versions, which draws older work incompatible to newer versions of ReFS.

Sinofsky (2012) explains key goals as well as features like checksums for meta- and content-data as well as the Copy-On-Write (COW) mechanism that are provided by ReFS. He also mentions a generic key-value interface which is used by ReFS to manage on-disk structures.

In a working report, Green (2013) analyzed the internal structures of ReFS and created an overview of identified structures. This report skipped how the internal key-value structure of ReFS works. He also looked into the recovery of deleted items, but only describes the recycle bin mechanism of the windows file explorer which is not specific to ReFS.

Another unofficial draft of structures found under ReFS is provided by Metz (2013). This work strongly focuses on the low-level presentation of how data structures in ReFS are composed and was the first to vaguely describe the key-value store. The author implemented a library which is based on his findings.

Georges (2018) aimed to develop a tool which outputs results comparable to `EnCase`. He also described how the allocation status of clusters is managed. With the developed tool, file extraction is possible for ReFS v1.2.

Nordvik et al. (2019) also examined data structures of ReFS. They mainly focused on v1.2 and v3.2 of ReFS. At the end they tested their findings on v3.4, too. For v3.2 they came to the conclusion that the versions did not differ much from v1.2 and that “the structures are almost identical”. In our work we come to the conclusion that

* Corresponding author.
** Corresponding author. Friedrich-Alexander University Erlangen-Nürnberg, Germany.
E-mail addresses: tobias.gross@cs.fau.de (T. Groß), adewald@ernw.de (A. Dewald).

https://doi.org/10.1016/j.fsidi.2020.300915

between v3.4 and v1.2 many data structures were added, deprecated or changed. We found also new functionality like virtual addresses which have to be taken into account when extracting data from the disk. In contrast to their work, we discovered more details and also investigate how the Copy-On-Write mechanism of ReFS is implemented and propose strategies to recover old versions of files.

1.2. Our contribution

In this work, we analyzed the proprietary Microsoft Windows Resilient File System (ReFS) version 3.4 and contribute the following:

- We analyzed the internal structures and mechanics of ReFS v3.4 and documented them in a technical report (Prade et al. (2019)).
- We extended The Sleuth Kit from Carrier to support ReFS.
- We propose strategies for recovering deleted files.
- We implemented a page carver which allows the reconstruction of deleted files and older file states.
- We evaluated the correctness of our implementation with different ReFS partitions.
- We analyzed the allocation strategy of ReFS v3.4 which affects the recoverability of older file states and compared the findings with our carver.

2. Background

The Sleuth Kit. The Sleuth Kit (TSK) is an open-source filesystem forensic tool. It was developed with high portability and extensibility in mind (Altheide and Carvey, 2011, p. 42). The highly extensible design of TSK allows developers to extend its functionality as well as the types of file systems it supports. TSK is structured into multiple layers of abstraction that map how data is stored on storage media. The File System Layer is of most importance within this work, as this is the abstraction where file systems such as FAT, NTFS and also ReFS are settled.

Copy-On-Write. Copy-On-Write (COW) is an update policy that may be used to alter data on a storage medium. A COW update policy makes sure that data is never updated in place. Whenever the content of a block should be altered, the block is read into memory, modified and its content is written to an alternate location on the storage medium. When the system crashes while a block is updated through a COW policy, the old state of the data remains untouched and still persists. COW offers a simple strategy to enforce atomicity in write operations and to ensure the integrity of data structures (Rodeh et al., 2013, p. 15).

3. ReFS internal structure

We analyzed the structure of ReFS and classified identfied structures into the categories *File System*, *Content*, *File Name*, *Metadata* and *Application* introduced by Carrier (2005). Detailed structure description of ReFS which are important for forensic analyses can be found in our technical report (Prade et al. (2019)). In this work we describe only the structures important for enumerating and recovering files and folders. The ordering corresponds to the occurrence when starting interpreting an ReFS partition at the boot sector.

3.1. Checkpoint

ReFS partitions have 2 checkpoint structures which are written alternately, to have at least one valid checkpoint in case of a crash. The checkpoint of a file system is the first structure that holds the current clock values, which indicates the latest checkpoint structure. Additionally, the checkpoint also contains the current log sequence number (LSN) which identifies the last entry that was written into the redo log. Since ReFS stores changes to the file system in the memory and writes them at a later point as a batch, the system may crash in this time window. A system crash reverts the system to its last valid state and would discard all changes executed in the meantime. However, every transaction that was successfully performed and is part of the batch is also written to a sequential log file that is discussed in our report. To know from where to start with the redo operations, it is essential to save an identifier for the last transaction that was part of the current checkpoint.

The most important structures found in the checkpoint are the table references. For the sake of thoroughness, Table 1 presents an overview of all tables referenced by the checkpoint along with their table identifiers.

3.2. Container table

Internally a ReFS volume is separated into multiple equally sized *bands* or *containers*. The size of a single band is relatively large at around 64 MiB or 256 MiB. The Container Table provides detailed information on that managed memory regions. It tracks how many clusters within a band are used and additionally collects statistics about the usage of data and access times within that band. Tipton (2015) argues that the introduction of Container Tables was important to treat multi-tiered storage systems more efficiently. In a multi-tiered storage system, it is common to have different types of storage media that offer different reading and writing characteristics. Some storage media such as flash memory allow performing random access faster than traditional hard disks, which are more suitable to perform sequential operations.

In a multi-tiered storage system, data needs to be reorganized according to its characteristics. To perform this reorganization, it is possible to swap bands, and thus their contents between different storage tiers. The additional tracking of metadata information within a band allows to monitor its heat, so to say how often data in it is accessed. This performance metric may be used to decide when to shuffle two bands.

This concept as it is described by Tipton (2015) and Das (2017) can be found in current ReFS versions. It is important to note that shuffling two different bands also changes the respective physical addresses of the data found in the bands. Before such a shuffle operation it would be necessary to adjust all pointers that reference data in the affected bands. To prevent the necessity of updating any pointers, ReFS v2 started implementing virtual addresses as a mechanism to leverage shuffle operations within containers. Nearly

Table 1
Tables referenced by the checkpoint structure.

Table Identifier	Table Name
0x2	Object ID Table
0x21	Medium Allocator Table
0x20	Container Allocator Table
0x1	Schema Table
0x3	Parent Child Table
0x4	Object ID Table, duplicate
0x5	Block Reference Count Table
0xb	Container Table
0xc	Container Table, duplicate
0x6	Schema Table, duplicate
0xe	Container Index Table
0xf	Integrity State Table
0x22	Small Allocator Table

all addresses used under ReFS have to be translated into real addresses before they may be used. After two containers were shuffled, it is only necessary to update their respective address mappings.

The Superblock, the Checkpoint, the Container Table, and the Container Allocator Table all use real addresses that do not need to be translated first, because they are needed to bootstrap address translation. We did not examine the inner structure of the rows of the Container Table in more detail. As of now, only two fields in these rows are known to us. One of these fields is required to perform the address translation process.

Fig. 1 portrays the practical usage of the Container Table in the address translation process. Every virtual address used under ReFS may be separated into a container- and an offset-component. The container component determines which row of the Container Table needs to be queried to perform an address translation. The offset component provides an offset that must be added to the translated address.

3.3. Object ID table

When conducting a forensic analysis of a ReFS formatted file system, the table of most importance is the so-called Object ID Table. This table references the root nodes of a variety of other tables and associates an identifier to them. Alongside a table that contains general information about the volume, a table storing an Upcase Table and a table containing an entry point to the redo log, the Object ID Table also references the root nodes of all Directory Tables. When searching for any directory, the Object ID Table must be queried. After a directory has been deleted, it is not referenced by the Object ID Table anymore. The Object ID Table is the only place where the actual addresses of Directory Tables are stored. Because of these circumstances, recovery techniques that attempt to restore directories under ReFS should focus on recovering rows found in the Object ID Table.

The Object ID Table additionally stores meta-information about the tables that it references. It stores the addresses and the checksums of the tables it references, as well as their last persisted log sequence number in the file system journal. Additionally, entries in the Object ID Table may also store a buffer with variable data. For links to Directory Tables, this buffer is filled with the next file identifier/metadata address to be used in the directory.

The root Directory Table is stored with the ID 0x600. Any other regular Directory Table get an ID assigned which is greater then 0x700.

As the Object ID Table takes a superior role to the tables which it references, more importance is attached to guarantee its integrity. A so-called *Duplicate Object ID Table* exists which contains the same entries as well as the same slack space as the regular *Object ID Table*. While Copy-On-Write is used to leverage atomic write operations, duplicate table structures seem to be used to battle bit rot. If one variant of the Object ID Table becomes corrupted, the ReFS driver may fall back to using the other variant of it.

3.4. Directory tables

Directory Tables implement the logic of directories. Every Directory Table contains a single row that stores metadata of the directory that the table represents. We refer to this row as the *directory descriptor*. The other rows found in a Directory Table mostly represent directory entries that are contained in the directory. There exist two different types of directory entries: file entry and directory link entry. A directory link mainly maps a name to a directory ID, which can be lookup in the Object ID Table. For all files in a Directory Table, an additional entry type exists, called ID2 that provides a mapping between the metadata address of a file and its current handle.

Metadata addresses are used to uniquely address files and directories. They offer a shorter notation than the path of a file. A metadata address in ReFS consists of two components, the directory ID and the file ID which get concatenated to a 128-Bit metadata address. The directory ID part identifies the Directory Table and the file ID the entry in this table. The directory identifier of a directory is equal to its table identifier. A table with the id `0x800` represents the directory with the metadata address `0x800|0`. The files inside this directory are referred to as `0x800|i`, where `i` >0 and `i` $<$ max_file_id$_{0x800}$. This choice of addressing files induces a tight coupling between metadata addresses and the actual paths in which files reside.

If a file is moved from one directory to another directory its metadata address is altered as a new directory identifier, and a new file identifier are assigned to it. However, it is still possible to refer to the file by its original file- and directory-identifier. The original identifiers are still saved in a field of the metadata entry of the file. Additionally, an ID2 row in its original Directory Table gets created. This row is used to offer a mapping between the original- and the current-metadata address of the file. This measure allows metadata addresses of files to be stable even if a file is moved into a new directory.

There exist two special Directory Tables. The root directory (ID `0x600`) and the `File System Metadata` directory (ID `0x520`) which fulfills the similar purpose as the `$Extend` directory known from NTFS.

To locate an entry by its metadata address, one has to search the Directory Table in the Object ID Table with the directory ID part. After that, one has to locate the ID2 entry with the file ID, which links to the file entry.

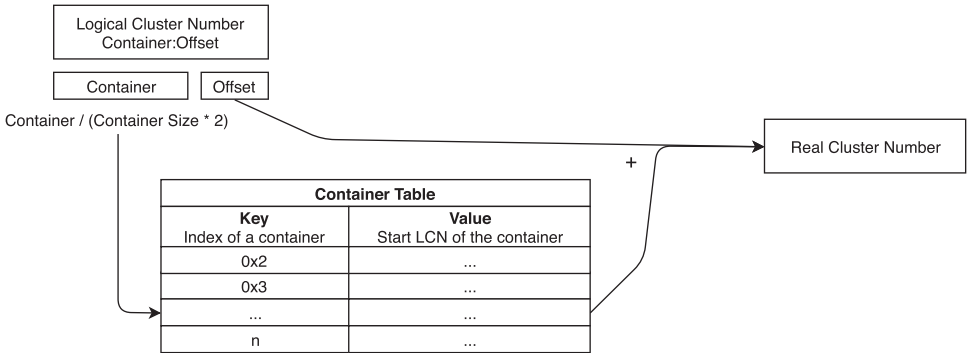

Fig. 1. Exemplary address translation process.

3.5. File table

Root nodes of File Tables are embedded within Directory Tables. Like all other tables, they start with a fixed-sized chunk of data that is used to describe table-specific information. The entries stored in a File Table are the properties that a file possesses. This concept strongly reminds of attributes that are used in NTFS. In this manner, MFT entries that are known from NTFS expose a similar behavior as file-descriptor-tables in ReFS. Both consist of a small fixed component as well as a variable-sized list of attributes.

We found the following attributes to be practically used by the ReFS driver: `$DIR_LINK`, `$INDEX_ROOT`, `$DATA`, `$NAMED_DATA`, and `$REPARSE_POINT`. Most of the information that was previously stored in the `$STANDARD_INFORMATION` attribute of NTFS has now become a part of the fixed data found in file descriptors. Since the form in which contents of a directory are represented was utterly changed and shifted into the responsibility of rows in Directory Tables, both the `$INDEX_ROOT` and the `$INDEX_ALLOCATION` attribute known from NTFS seem to have become obsolete. Still, we found all directories to use equally filled dummy `$INDEX_ROOT` attributes.

Aside from the `$DATA` attribute, the contents of all attributes under ReFS seem to be stored resident, meaning in place. In NTFS it was possible for the `$DATA` attribute to either be stored resident or non-resident. In ReFS, the `$DATA` attribute now seems always to be stored non-resident. Even if a file is only a few bytes large, ReFS allocates an own cluster for it and saves its data non-resident. Furthermore, the `$DATA` attribute seems to be the only attribute that spans an embedded tree, which stores rows of cluster runs. This approach also makes it easy to search for an offset relative in the file as the tree used to store data runs is collated by the relative start of a data run.

4. ReFS data recovery

In ReFS, most data are organized in key-value stores, so called tables. Internally, these tables are implemented as a B^+-tree. Microsoft calls their implementation Minstore B+.

When data gets written in ReFS, the updates are not performed in-place, instead a COW strategy is used. In the B^+-tree, the payload is always held in the leaf nodes. When data gets altered in this table structures, a new leaf node is created which takes over the old data and applies the modifications. Afterwards the pointer in the parent node has to be adjusted, which is also done with the COW strategy. This process bubbles up to the root node. This principle is shown in Fig. 2 where pointer adjustments bubbles up and create a new root node, by inserting 19 into the leaf node.

When an entry is removed from a node of the B^+-tree, the entry is not wiped. Instead, only a link to a data chunk is removed and the data chunk is released. Fig. 3 shows a node with deleted entries e_1, e_2 and e_4. They stay untouched, only the key indexes are made invalid. On the right-hand side the entry e_5 gets inserted which overwrites big parts of e_2. Every time a entry is inserted or deleted, the key index gets reordered.

We experienced that this non-referenced data is also copied by the COW strategy. We call this slack space propagation and verified this in an experiment where we filled slack space of a leaf node with markers. During usage of the filesystem (FS), these markers get propagated to new copies of this node, when data is written in a COW manner.

We can use the concept of non-referenced data in nodes as well as old non-referenced nodes to recover data in ReFS partitions. For recovering entries from nodes, we have to scan the entire data area for indicators. An entry starts with a header which we can check for plausibility.

As a second method, we can search the entire partition for tree node pages. Every node and even every page in ReFS start with a specific header. This header offers a great opportunity for identifying and locating all nodes in ReFS.

5. Implementation

5.1. TSK extension

With the knowledge we gained through analyzing ReFS we implemented an extension for TSK to support interpreting ReFS. ReFS shares the same DOS partition type as exFAT and NTFS, so we modified the mmls tool which analyses partition tables to output "ReFS" additional to exFat and NTFS as the output of filesystem names.

The other implementations focus on the file system layer. TSK uses a list of filesystem openers which can be used to try opening filesystems without knowing the type beforehand. The opener returns a context object which stores basic information about an filesystem such as block size, first/last block of the FS and first/last metadata addresses. Additionally, it provides the TSK API as function pointers which allows to interact with the FS in a generic way.

Our opener implementation first parses and checks (with included signatures and checksums) the boot sector by reading the first sector of the FS. Additionally, we get the cluster and sector size from the boot sector.

After identifying a valid ReFS boot sector, we parse the superblock which resides in cluster 30 as well as the superblock backups at the end of the FS and pick the first valid one. We check the superblock with its self-checksum. From the superblock we get the volume signature and the cluster addresses of the checkpoint structures.

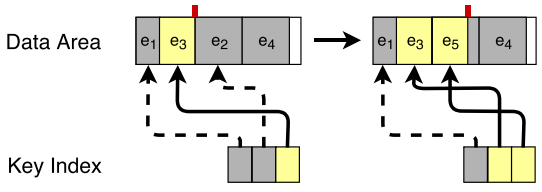

Fig. 3. Data organization in leaf nodes.

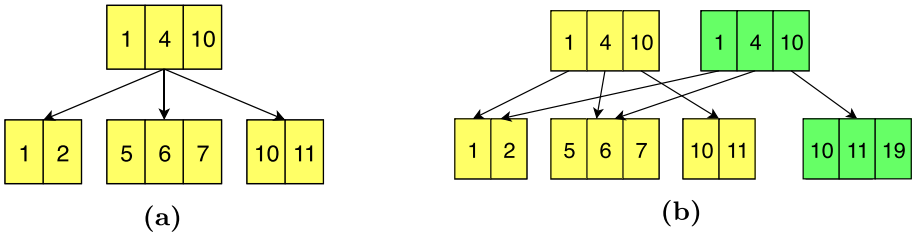

Fig. 2. (a) A basic b-tree (b) Inserting key 19, and creating a path of modified pages (based on Rodeh et al. (2013)).

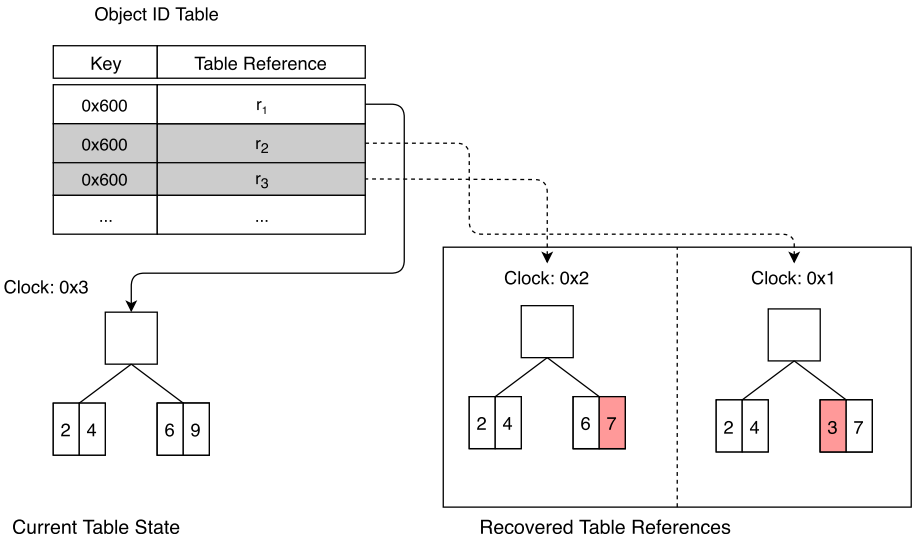

Fig. 4. Combining entries of existing and deleted tables that are referenced by the Object ID Table.

Our implementation picks the most recent valid checkpoint after parsing both checkpoints. Hitherto we refrain from combining both checkpoints because it is difficult to combine these tree structures, although one checkpoint references older root nodes left-behind through the COW process. 13 root node references get extracted from the chosen checkpoint including e.g. Container, Object ID and Volume Information Tables.

Next the Container Table gets enumerated and address translation data is extracted which is later used to translate page references to physical page addresses.

One of the main parts is the enumeration of the Object ID Table as it references directories among others. It should not matter whether the regular Object ID Table or its identical copy, the duplicate Object ID Table is processed since both should store the same contents and presumably also contain the same slack space. Many tables referenced by the Object ID Table represent directories. Because of that, it is not only important to read all regular entries found in the Object ID Table but also to recover deleted ones, which can be identified by signatures in their key.

When recovering links to Directory Tables, it is possible to find multiple links with the same table identifier. This behavior mainly occurs as a result of the COW process, that continually writes the root nodes of tables to new pages. Duplicate page references that store the same table identifier as others may reference different root nodes. As a consequence of that, it is possible for a directory to be formed by multiple tree structures which store older states of that folder. An example of this state of affairs is shown in Fig. 4. The grey marked rows in the figure represent rows that were recovered in the Object ID Table. When combining the keys found in the different restored tree structures, it is important that all keys must only be interpreted once. The current state of the table stores the keys `2`, `4`, `6`, and `9`. These are interpreted as regular entries. The recovered trees must be sorted in descending order of their recency represented with the virtual allocator clock. After a tree has been enumerated completely, all new keys are added to the set of existent keys. Keys that are not found in the regular referenced tree structure (`3`, `7`) but are located in different tables that were recovered, are considered to be deleted entries. It is only attempted to recover deleted entries which reference Directory Tables. All other table structures referenced by the Object ID Table are regularly processed by querying its index.

The TSK tool fsstat outputs general filesystem information. It depends on a fsstat function provided from the context object. We implemented an `refs_fsstat` function which outputs information from the Volume Information Table. A brief overview of the other context functions implementations is given next.

inode lookup. This function is used to get general metadata for a file or directory specified by the provided metadata address. Every metadata address under ReFS is formed by a file identifier and a directory identifier. To find the location of a file, one must query the directory table which is identified by the directory identifier. With the file identifier as a key, the directory table gets queried and the found directory descriptor- or file table can then be parsed. The data attribute is non-resident as the data of files in ReFS resides in external clusters. The implementation enumerates the data table to receive all data runs. An enumeration strategy function attempts to recover data runs that have been removed from the index. Afterwards, the data is transformed to the TSK metadata representation.

dir open meta. This function associates the names of files and folders to their metadata addresses. This function needs the metadata address of a folder to process. It populates a directory data structure with mappings of file names and associated metadata addresses. For this the whole directory table gets enumerated and the file and directory links are transformed and inserted into the directory data structure.

file get sidstr. For our implementation we reused large parts of the NTFS implementation. For a given metadata entry it extracts its security identifier (SID) string.

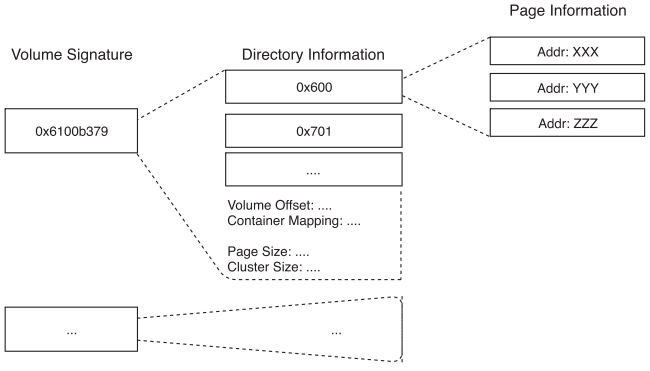

Fig. 5. Classification process in the collection phase of the carver.

Usnjls. ReFS allows for creating an Update Sequence Number (USN) journal similar to NTFS. The implementation of this tool can display the content of an USN journal which is stored as a common file. We extended the NTFS implementation with a new USN record format present in ReFS USN journals.

One major drawback in our implementation is, that TSK supports only 64-bit large metadata addresses to reference files and directories. ReFS however uses two 64-bit identifier to refer to a metadata entry. One encodes the directory table in which the metadata entry resides. The other is used to address the entry within the directory table. Therefore, we changed the definition of the `TSK_INUM_T` from `uint64_t` to a struct that holds a low and a high 64-bit value. This change however makes all application using TSK incompatible to our implementation and need to get revised in the future.

Another drawback is, that the COW creates multiple copies of files which represent their state at different times. TSK does not cover the idea of multiple copies of a file with a shared metadata address. Major changes have to be made in TSK to support this concept in the future.

5.2. Page carver

Even though many deleted entries may be restored by analyzing remnants in tree structures, still many nodes of previously allocated pages remain unseen. The reason for this is that some nodes are not referenced by any accessible page anymore and there exists no way to locate them. It is possible that at an arbitrary position on the disk, important data is located that belongs to the metadata of a deleted file or a directory structure. If we, however, have no way to find this structure, it remains unseen and potentially substantial evidence is suppressed. The implementations for the file systems in TSK are purely based on parsing existing structures and thus react highly responsive. It would, however, be necessary to scan an entire volume if all deleted pages have to be recovered. Therefore, a carver independent of the actual ReFS implementation was developed.

The carver is based on the `tsk_img_open` function provided by TSK. A user may provide an arbitrary image and an offset to the carver which then tries to recover deleted files and directories from the image. The carver works in two steps.

Collection phase. This is the phase in which the carver starts. The carver scans the entire provided image for blocks that look like pages of tree nodes. The step size of the carver is 4 KiB. In every step, it verifies the page header within the first 80 bytes of a read block. If these bytes fulfill the characteristic properties of a page header, the page is kept for further analysis. If the table identifier of the page is equal to the table identifier of a container table or any directory table the page is retained, else it is discarded.

The carver maintains a structure that maps volume signatures to an object that represents the state of a ReFS file system. This structure is used to distinguish different instances of ReFS file systems that may exist if the underlying disk contains multiple ReFS partitions or was formatted multiple times with the ReFS file system. Within the state of each volume, a map of directory identifiers is maintained. This map assigns table identifiers to a list of pages. When looking up the key to a directory table in this map the caller may obtain a list of all pages that store information relevant to this table, so to say all nodes that once belonged and nodes that still belong to a table with a given identifier.

The carver also derives further information from pages that were read. The number of addresses stored in the page header is used to decide whether the page belongs to a volume with a cluster size of 4 KiB or 64 KiB. It is also important for the carver to know at which disk offset the volume started. At a later time, the application must interpret cluster runs. For this, it is necessary to know to which starting address they refer. Fig. 6 illustrates this issue. The red marked file system starts at the beginning of the disk image. Address references used in this file system may be interpreted without adding any offsets. The blue marked file system starts somewhere in the middle of the disk image. Therefore, its starting offset relative to the start of the disk image needs to be added to all cluster references that are interpreted in it. Luckily, the cluster numbers found in the page header of the Container Table are equal to their physical addresses. Thus, the first found page that belongs to a container table may be used to determine the starting location of a ReFS file system.

The collection process may also be halted at any time. Whenever the carver has read and classified 256 MiB of data, it writes its progress state as well as the list of found pages into a file. When restarting the application at a later time, the collection process may continue at the last written state. The idea of this feature as well as lots of other concepts, were extracted from the `refsutil.exe` utility. `refsutil.exe` was introduced in early 2019 by Microsoft. It allows to verify or recreate a boot sector, to reclaim leaked clusters, to remove corrupt files, or to recover files from a ReFS file system. The recovery operation is also referred to as `salvage`. We analyzed the implementation of this operation to get valuable for recovering files.

Reconstruction phase. After all pages have been read and classified according to Fig. 5, the carver goes into the reconstruction phase. The reconstruction phase loops over all found volume states and performs identical operations on them. First, the latest state of the Container Table is restored so that it is possible to translate virtual addresses in a volume to real addresses. Next, the carver loops over all directory tables found in a volume.

Metadata reconstruction. Directory tables are stored as a flat sequence of pages. The carver searches these pages for signatures that may be used to identify files, directory links and directory descriptors. When a corresponding signature is found, the carver executes various sanity checks on the found data and tries to interpret it as the structure it believes it to be. The carver must also restore the subtree formed by directory descriptor tables and file tables to access their attributes, and their data runs. If the carver was successful in restoring an entry, the entry is saved in the class format file_information or dirref_information. Every directory stores these entries as a set.

If a `file_information` entry or a `dirref_information` entry is equal to an already existing entry, it is discarded. To determine whether two entries are identical, multiple properties of them are compared. It would not be sufficient to merely check whether the file identifier already exists in the set of found entries as this would discard potential older copies of files. Instead, the file identifier, the last modification time, and the file size are used in conjunction to check whether two file entries are equal. If they are not equal, both may be contained in the same result set.

Extraction phase. After all directory pages of a volume have been analyzed and their entries have been transformed into the internal representation of files and directories, the extraction phase begins. First all directory tables found in the volume are created as folders of the form `<vol_sig>/<dir_id>`. Next, a volume report with the file name `<vol_sig>/report.txt` is created. The report describes the properties of the corresponding volume such as its

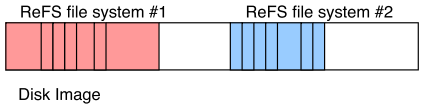

Fig. 6. Multiple file systems at different locations within a disk.

offset in the disk image, its cluster size and the number of directories that it contains. The report also describes the address translation mapping that was extracted from the container table of the volume.

Next, the application creates a file called `<vol_sig>/structures.txt` that is filled with the reconstructed directory hierarchy formed by the file system. Since the application stores the extracted directories in a flat form and only refers to them through their directory identifiers, this file may be used to re-establish directory structures. Finally, the application iterates over all directory structures and dumps the files found in them. Into every directory a file called `<vol_sig>/<dir_id>/directory_report.txt` is written which contains a list of all files found in the directory. The list also contains the metadata of duplicate files that may have been created through the Copy-On-Write mechanism. The contents of all files are finally dumped into the corresponding directory. To prevent naming conflicts among restored files that utilize identical file names files are written as `<vol_sig>/<dir_id>/<file_id>_<copy_id>`. Their names however may easily be looked up by examining the `<vol_sig>/structures.txt` or the `<vol_sig>/<dir_id>/directory_report.txt` file.

As of now, the carver is only able to extract files and directories of a file system, and contrary to the ReFS implementation in the file system layer of TSK does not show additional information about a volume. It might be best to use both of those tools in conjunction. The file system implementation of TSK provides valuable information such as attributes of files as well as file system metadata that has not yet been added to the analysis process of the carver. A crucial advantage of the implementation in the file system layer of TSK is that its implementation can decide whether a file is allocated or has been removed. The carver is not aware of whether the files that are extracted were allocated or not. On the other hand, the carver provides a valuable tool for an investigator that intends to examine a corrupted ReFS file system or an investigator who wants to obtain potential older copies of files.

6. Evaluation

As of now, there exists no work that looks at recovering files and folders from ReFS formatted file systems. Thus also, no work has yet stated test cases that could be used to evaluate tools that serve such a purpose. We designed multiple test scenarios which portrayed a fictive usage of the file system. The evaluation process of the tools is strongly based on the description of the evaluation dataset used by Plum and Dewald (2018). We generated 8 ReFS images with different configurations of overall size (2GiB, 5GiB, 40GiB, 100GiB) and cluster size (4KiB, 64KiB). On each we performed 1000 actions from the following list with different likelihood of occurrence:

- **add file** $P(X) = 0.35$: Add a random file from the EDRM File Formats Data Set 1.0[1] to the volume
- **delete file** $P(X) = 0.2$: Delete a random file
- **move file** $P(X) = 0.05$: Moves a random file
- **change file** $P(X) = 0.1$: Insert a random amount of `A`'s into a random file
- **copy file** $P(X) = 0.1$: Copies a random file
- **add folder** $P(X) = 0.15$: Creates a randomly named folder
- **delete folder** $P(X) = 0.05$: Removes a randomly picked folder

If an action could not be executed in a step (e.g. *removing a directory, if no directory exists at all*), a different action was picked randomly. This choice of the distribution was picked deliberately to generate a more "realistic" reflection of the usage of a filesystem. The distribution of the actions, however, was an arbitrary guess of the authors of how often these actions occur.

After every action we documented the outcome in a modified TSK body file.[2] Both, the TSK implementation as well as the carver were adjusted so that they were able to output the modified body file format to compare results. The following characteristics were stored in the modified body file for every file or directory present on the filesystem: An MD5 checksum of the file content, which is set to 0 for directories. The name of a file or a directory without its complete path. The current directory identifier of a file, for directories their parent identifier is stored. The initial metadata address of the file or folder, which stays constant, even if the file gets moved. The MAC timestamps which track the last modification- access- and the creation-time of a file. The size of a file.

The python applications that are used to acquire the ground truth of the state of the file system use the function `os.stat`, which was only able to obtain a 64-Bit metadata address. For our evaluation this was no limitation because all created metadata addresses were small enough.

The capture of the final state of the file system includes all existent files and directories, whereas the outcomes of the single executed operations only log changes made to files and the creation of directories. Every action that creates, alters, moves or copies a file generates a log entry for the modified state of the file. We refrained from logging the changes in the metadata of directories resulting from file actions as a deliberate decision in weighting the results. In practice, it might be of more importance restoring metadata entries of files than of directories. We did not want to weight the incapability to restore an old timestamp of a directory equal to the incapability of restoring the contents of a file.

6.1. TSK extension

It is expected that the TSK extension can produce a report equal to the actual state of the file system. We tested the TSK extension on the 8 generated images and called the tool *fls* with the option to only display allocated files. The resulting output should be equal to the current state of the file system that Windows reports.

As shown in Table 2, for all randomly generated testing scenarios, the output of the TSK extension matches the actual state of the file system, except the folder "System Volume Information" (Table ID: 0x701) and its content "WPSettings.dat" and "IndexerVolumeGuid". The numbers x/y count the number of detected files/folders (x) and the number of files/folders present on the final state (y). These entries were not included in the extracted final state of the file system because Windows hides this folder from regular users. The same applies to the "File System Metadata" (Table ID: 0x520) folder, which we omitted proactively in this output of the TSK, since it is not a regular directory. The results show that we are able to output the exact actual ReFS state with our TSK extension.

6.2. File recovery capabilities

With this evaluation we want to compare the capability of recovering deleted files and reconstructing previous states of files and folders. We compare the outputs of the developed tools with the action trace log which was logged during test image creation.

In the tables referenced in the following, the numbers x/y have to be interpreted as the number of exact outputted files/folders (x) and the number of all different file folder states once present on the

[1] https://www.edrm.net/resources/data-sets/edrm-file-format-data-set/.

[2] https://wiki.sleuthkit.org/index.php?title=Body_file.

filesystem (y). Table 3 compares the output of the TSK extension only considering allocated files and folders to all states. Noteworthy, less directories match than in the evaluation seen in Table 2. That is because we use the action log as comparison instead of the final filesystem state. In the action trace we only log directory creation and no timestamp modifications due to changes of child files and folders. Therefore, only few folders outputted by the implemented applications which got never an update in timestamps match exactly the logged folders. As expected, the applications output at least all files present in the final image state.

Table 4 includes the file and directory recovery capabilities of the TSK extension. Compared to table 3 more files and folders are outputted correctly. As the ReFS extension can only restore entries from referenced pages, it is likely that there remain pages that the TSK extension cannot locate. Additionally, the TSK extension only retrieves the most recent state of a file and is unable to address and thus to retrieve past states of a modified file.

Table 5 shows all files that could be recovered by using the carver. In all scenarios the carver is able to recover more files and directories than the TSK extension. The major drawback of the carver is, that it is unable to differentiate between existing and removed files.

Table 6 conclusively give an overview of the runtimes of the various test scenarios. To conduct all experiments, we used a 4 TiB large Western Digital hard drive (Product ID: WDBHDW0040BBK). With the application `dd` we estimated a sequentially read speed of 95–110 MiB/s. Depending on the size of a volume, the carver is most of the time busy with reading and collecting pages from the volume. The runtime of the TSK extension also varies strongly based on the number of pages that are read.

6.3. Recovering old file states

In this evaluation we look at how the COW process impacts the recovery of old file states. For this experiment we developed a small application that writes text into a file in a ReFS file system. The text was artificially written at the speed at which a person types (175 characters per minute). Every 2 min the text file was saved, and a checksum of its current intermediate state was logged together with its metadata.

Because of the COW policy that ReFS uses, it is likely that metadata, that describes the file and the location of its clusters, is dispensed into multiple places of the volume. We used the developed carver to find as many existent old copies of the file as possible. Additionally, we instrumented the ReFS driver to log cluster allocation and COW information, e.g. when data is written from one page to another. The data generated in this process allowed us to reconstruct which intermediate states of the file were generated.

The continuous modification of a single page in the COW process can be viewed as a chain of page states. Every page state has a physical address associated to it. If this physical address is reused at a later time for an allocation, this state cannot be recovered anymore. All older pages in such a page chain that get not reallocated may still yield valuable contents.

The experiment focused on writing data into a single file. Additionally, no other modifications on the file system were made while the file was altered. While this is a lab setup, it gives an upper boundary for the recovery capability of old file states on ReFS partitions.

The experiment was conducted with five different data sets. Each data set was a 5 GiB large ReFS volume with a cluster size of 4

Table 2
Output of the TSK extension, compared to the final state of the file system.

Configuration	Interpreted entries	
	Directories	Files
2GiB 4KiB	47/46 (102.17%)	112/110 (101.82%)
2GiB 64KiB	44/43 (102.33%)	83/81 (102.47%)
5GiB 4KiB	45/44 (102.27%)	123/121 (101.65%)
5GiB 64KiB	20/19 (105.26%)	33/31 (106.45%)
40GiB 4KiB	25/24 (104.17%)	78/76 (102.63%)
40GiB 64KiB	56/55 (101.82%)	143/141 (101.42%)
100GiB 4KiB	35/34 (102.94%)	64/62 (103.23%)
100GiB 64KiB	53/52 (101.92%)	110/108 (101.85%)

Table 3
State of all allocated files (TSK extension), compared to action log.

Configuration	Interpreted entries	
	Directories	Files
2GiB 4KiB	14/154 (9.09%)	110/613 (17.94%)
2GiB 64KiB	16/159 (10.06%)	81/590 (13.73%)
5GiB 4KiB	11/144 (7.64%)	121/606 (19.97%)
5GiB 64KiB	9/159 (5.66%)	31/585 (5.30%)
40GiB 4KiB	9/129 (6.98%)	76/615 (12.36%)
40GiB 64KiB	7/159 (4.40%)	141/616 (22.89%)
100GiB 4KiB	6/147 (4.08%)	62/592 (10.47%)
100GiB 64KiB	12/152 (7.89%)	108/593 (18.21%)

Table 4
State of all allocated and recovered files (TSK extension), compared to action log.

Configuration	Restored entries	
	Directories	Files
2GiB 4KiB	15/154 (9.74%)	132/613 (21.53%)
2GiB 64KiB	19/159 (11.95%)	125/590 (21.19%)
5GiB 4KiB	14/144 (9.72%)	159/606 (26.24%)
5GiB 64KiB	28/159 (17.61%)	75/585 (12.82%)
40GiB 4KiB	12/129 (9.30%)	97/615 (15.77%)
40GiB 64KiB	11/159 (6.92%)	175/616 (28.41%)
100GiB 4KiB	10/147 (6.80%)	113/592 (19.09%)
100GiB 64KiB	13/152 (8.55%)	142/593 (23.95%)

Table 5
State of all allocated and recovered files (carver), compared to action log.

Configuration	Restored entries	
	Directories	Files
2GiB 4KiB	15/154 (9.74%)	139/613 (22.68%)
2GiB 64KiB	21/159 (13.21%)	133/590 (22.54%)
5GiB 4KiB	16/144 (11.11%)	169/606 (27.89%)
5GiB 64KiB	34/159 (21.38%)	88/585 (15.04%)
40GiB 4KiB	16/129 (12.40%)	102/615 (16.59%)
40GiB 64KiB	11/159 (6.92%)	183/616 (29.71%)
100GiB 4KiB	10/147 (6.80%)	124/592 (20.95%)
100GiB 64KiB	13/152 (8.55%)	150/593 (25.30%)

Table 6
Runtimes of the applications.

Configuration	TSK (alloc. files)	TSK (all files)	Carver
2GiB 4KiB	1.233 s	1.379 s	20.411 s
2GiB 64KiB	1.21 s	1.947 s	20.519 s
5GiB 4KiB	0.613 s	0.694 s	50.1 s
5GiB 64KiB	0.917 s	1.206 s	50.764 s
40GiB 4KiB	0.972 s	0.94 s	383.027 s
40GiB 64KiB	2.256 s	2.409 s	402.892 s
100GiB 4KiB	1.441 s	1.415 s	997.922 s
100GiB 64KiB	2.037 s	2.344 s	1012.462 s

Table 7
Experiment to analyze the recoverability of COW copies.

Duration	Save Op.	Recovered	Recoverable
		States	Pages
10 m	5	3 (2 valid)	6/15
30 m	15	3 (3 valid)	5/31
60 m	30	3 (3 valid)	5/47
120 m	60	3 (3 valid)	5/86
240 m	120	3 (3 valid)	4/141

KiB. For every data set, the text file was written for a different time period, ranging from 10 min to 2 h.

As seen in Table 7, the number of recoverable file states does not differ much between the various experiments. In all scenarios, the valid files that could be recovered corresponded to the last 3 states of the file, with the exception of the shortest run. That outlier occurs because the carver found an empty version of the new created file. In the rightmost column of Table 7 you can see that even though various runtimes were used for the experiments, and for longer running experiments more page states were created, the number of recoverable pages stays nearly constant. The right number shows the total amount of allocated pages during the runtime. It seems like the allocator in the ReFS driver reused old pages relatively fast.

7. Summary

With this work we investigated the internal structures of the new Resiliant File System. We used the insights to extend The Sleuth Kit to be able to parse and interpret ReFS partitions. Open that we implemented a page carver to recover file system data. We evaluated both tools and come to the conclusion that our TSK extension works as intended and reports the current state of a ReFS partition equal to the ReFS driver of Windows. We also showed that with page carving we can recover more data than with only using deleted entries which are still present on the disk.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) as part of the FIDI project.

References

Altheide, C., Carvey, H., 2011. Digital Forensics with Open Source Tools. Elsevier.
Carrier, B.. The Sleuth Kit. URL. https://www.sleuthkit.org/sleuthkit/. last visited: 2019-10-04.
Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Professional.
Das, R., 2017. ReFS support for SMR drives. Presentation. SDC, 2017. https://www.snia.org/sites/default/files/SDC/2017/presentations/smr/Das_Rajsekhar_ReFS_Support_For_Shingled_Magnetic_Recording_Drives.pdf.
Georges, H., 2018. Resilient Filesystem. Master's thesis. NTNU. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2502565.
Green, P., 2013. Resilient File System (ReFS), Analysis of the File System Found on Windows Server 2012. Staffordshire University. Technical Report.
Metz, J., 2013. Resilient file system (ReFS). last visited: 2019-10-04. https://github.com/libyal/libfsrefs/blob/master/documentation/ResilientFileSystem(ReFS).pdf.
Nordvik, R., Georges, H., Toolan, F., Axelsson, S., 2019. Reverse engineering of ReFS. Digit. Invest. 30, 127–147. https://doi.org/10.1016/j.diin.2019.07.004. http://www.sciencedirect.com/science/article/pii/S1742287619301252.
Plum, J., Dewald, A., 2018. Forensic APFS file recovery. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. ACM, New York, NY, USA, p. 47. https://doi.org/10.1145/3230833.3232808, 1–47:10. URL. https://dl.acm.org/citation.cfm?id=3232808.
Prade, P., Groβ, T., Dewald, A., 2019. Forensic Analysis of the Resilient File System (ReFS) Version 3.4. Technical Report CS-2019-05. Department Informatik. URL. https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/12526, 10.2.32328085593/issn.2191-5008/CS-2019-05.
Rodeh, O., Bacik, J., Mason, C., 2013. BTRFS: the linux B-tree filesystem. ACM Trans. Storage 9, 9. https://doi.org/10.1145/2501620.2501623. URL. https://dl.acm.org/citation.cfm?id=2501623.
Sinofsky, S., 2012. Building the next generation file system for Windows: ReFS. URL. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/. last visited: 2019-10-04.
Tipton, J., 2015. ReFS v2, Cloning, projecting, and moving data. Presentation. SDC, 2015. https://www.snia.org/sites/default/files/SDC15_presentations/file_sys/JRTipton_ReFS_v2.pdf.