Università
Ca'Foscari
Venezia

Master's Degree programme — Second Cycle
(D.M. 270/2004)
in Informatica — Computer Science

Final Thesis

—

Ca' Foscari
Dorsoduro 3246
30123 Venezia

RecuperaBit: Forensic File System Reconstruction Given Partially Corrupted Metadata

SUPERVISOR
Ch. Prof. Riccardo Focardi

CANDIDATE
Andrea Lazzarotto
Matriculation number 833897

ACADEMIC YEAR
2014/2015

RECUPERABIT: FORENSIC FILE SYSTEM RECONSTRUCTION GIVEN PARTIALLY CORRUPTED METADATA

ANDREA LAZZAROTTO

Dept. of Environmental Sciences, Informatics and Statistics
Ca' Foscari University
Venice, Italy
February 2016

Ik maak steeds wat ik nog niet kan
om het te leeren kunnen.

— Vincent van Gogh

*I am always doing what I can't do yet
in order to learn how to do it.*

ABSTRACT

File system analysis is an important process in the fields of data recovery and computer forensics. The file system is formed by metadata describing how files and directories are organized in a hard drive. File system corruption, either accidental or intentional, may compromise the ability to access and recover the contents of files. Existing techniques, such as file carving, allow for the recovery of file contents partially, without considering the file system structure. However, the loss of metadata may prevent the attribution of meaning to extracted contents, given by file names or timestamps.

We present a signature recognition process that matches and parses known records belonging to files on a drive, followed by a bottom-up reconstruction algorithm which is able to recover the structure of the file system by rebuilding the entire tree, or multiple subtrees if the upper nodes are missing. Partition geometry is determined even if partition boundaries are unknown by applying the Baeza-Yates–Perleberg approximate string matching algorithm. We focus on NTFS, a file system commonly found in personal computers and high-capacity external hard drives. We compare the results of our algorithm with existing open source and commercial tools for data recovery.

ACKNOWLEDGMENTS

The years I spent as a graduate student have been sometimes challenging but very interesting and satisfying. During the time of writing this thesis I received help and support from several people. I would like to thank my supervisor, Prof. Riccardo Focardi, for his helpful guidance, his prompt answers to my questions and the advice he provided.

My gratitude goes to Nanni Bassetti, Davide Gabrini and Luigi Nicotera, the digital forensics professionals with whom I was lucky enough to exchange personal communications. Their suggestions and tests helped me to improve the software implementation.

This experience would not have been the same without the great people sharing it with me. I want to thank my classmates, principally Ismail, Fabio, Cecilia and Massimo.

I will never forget the time spent in the Netherlands. Thank you to Jumana, Myrthe, Hanrong and Hugo for being valuable groupmates and friends. A particular mention goes to Daan and Jules: not only for being very talented students and a pleasure to work with, but also for making me feel at home, enjoy the country and eat vlaai. Moreover, I was very glad to share my life in Eindhoven with Laura and Saara, i.e. the best housemates ever.

I would also like to acknowledge all my friends. In particular, huge thanks to Paola, Tana, Nicolò and Valentina for their encouragement, their sincere friendship and the superb laughs. Every time I seem to understand what friendship is, they bring it to the next level.

Finally, I am very grateful to my parents for their constant support and endorsement throughout all these years.

Andrea

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

ACL	Access Control List
ADS	Alternate Data Stream
APM	Apple Partition Map
CB	Cluster Base
CLI	Command Line Interface
CSV	Comma Separated Values
FAT	File Allocation Table
FLOSS	Free Libre Open Source Software
GUI	Graphical User Interface
JPEG	Joint Photographic Experts Group
LSN	`$LogFile` Sequence Number
MAC	Modification, Access and Creation
MBR	Master Boot Record
MFT	Master File Table
NTFS	New Technology File System
OEM	Original Equipment Manufacturer
OS	Operating System
RAM	Random Access Memory
SPC	Sectors per Cluster
UTC	Coordinated Universal Time
UTF-16	Unicode Transformation Format, 16-bit
VCN	Virtual Cluster Number

1 INTRODUCTION

Digital forensics is a branch of forensic science related to investigations using digital evidence. A digital forensic investigation is focused on some type of digital device involved in an incident or crime. It is a process that "develops and tests theories, which can be entered into a court of law, to answer questions about events that occurred." [3]

Classic examples of crimes involving digital forensic investigations are computer intrusions, Internet gambling, child pornography and homicide [5]. However, the massive amount of information routinely stored in computer systems has made the presence of digital evidence more and more frequent. This has broadened the usefulness of digital forensics in both civil and criminal cases.

Information in computer systems and portable devices is organized in file systems, hence it is important to understand their structure to access the contents of files. However, hardware might become damaged leading to metadata corruption, which makes the file system unaccessible by normal tools.

We tackle the problem of forensic file system reconstruction when parts of the metadata are corrupted or missing, allowing for the recovery of intact portions of the directory tree and file contents.

1.1 CONTRIBUTIONS

The contributions of this thesis can be summarized as follows:

- We propose our formalization of the forensic file system reconstruction problem, taking into account metadata corruption. We provide a generic model of nodes organized in the *Root* and *Lost Files* directories, where each file has a set of essential attributes.

- We apply a disk scanning approach to the collection of metadata from a hard drive. We introduce a file system independent, bottom-up directory tree reconstruction algorithm that uses the collected metadata to solve the problem successfully.

- Focusing on New Technology File System (NTFS), we introduce a file clustering technique to separate files from multiple file systems. We provide an algorithm that uses detected files to infer the geometry of a partition when its parameters are unknown.

- We implement our file system reconstruction algorithm in RecuperaBit, a modular program for the reconstruction of damaged file systems. The tool analyzes a hard disk image, detecting and reconstructing possibly damaged NTFS instances.

- We report on a set of experiments performed to compare the quality of output produced by our software against several open source and commercial data recovery programs.

1.2 THESIS STRUCTURE

The thesis is organized as follows:

- In Chapter 2 we summarize the basic concepts related to digital forensics and we provide the problem specification.
- In Chapter 3 we describe the features of NTFS, which is the file system type we consider in this thesis.
- In Chapter 4 we go into detail about different kinds of scenarios in which metadata might become corrupted.
- In Chapter 5 we discuss our algorithm for the bottom-up reconstruction of directory trees.
- In Chapter 6 we describe a method to infer the geometry of a partition formatted in NTFS.
- In Chapter 7 we provide an overview of the architecture used in our software implementation.
- In Chapter 8 we analyze our empirical results.
- In Chapter 9 we end with some concluding remarks and suggestions for future work.

1.3 PREVIOUS WORK

The scientific literature contains several published works related to file systems and the recovery of file contents, with a special interest in deleted files. Given the low-level nature of this topic, most of these works are oriented to one specific type of file system. Different kinds of file systems exist, however NTFS and File Allocation Table (FAT) are more popular and extensively studied.

Nevertheless, most works focus on data rather than metadata, assuming that the latter is intact or only slightly corrupted. There is very little published about recovery procedures when metadata is missing [3]. When computer generated data is introduced as evidence in a court trial, knowledge about the process that produced the data is useful for proving its reliability [2]. With this thesis we aim at describing algorithms that can be independently checked and verified.

In the following we present the main published techniques related to file system data extraction and file recovery. They are divided by recovery approach rather than target file system.

Top-Down File System Access

Kai et al. [10] describe a common approach that assumes a working file system. The contents of the directories are parsed in a top-down fashion to recursively list the files in each directory and infer the directory tree. Their work is focused on deleted files and their recoverability. One interesting aspect of this approach is the *object oriented parsing* for NTFS, which associates one object to each feature: project, disk, partition, directory and document.

Another useful feature of the proposed software is the introduction of a Graphical User Interface (GUI) which allows to explore the file system interactively. Finally, a *recoverability assessment* is performed to estabilish if the contents of deleted files can be extracted.

File System Boundary Recovery

In many cases, file system corruption occurs near the beginning of a partition [9]. For this reason, it is common to encounter a slightly corrupted file system in which it is possible to gain access after restoring a backup copy of its BOOT SECTOR.

In addition to describing the features of many popular file systems in great details, Carrier [3] also provides examples of this technique for both FAT and NTFS. The process consists in *scanning* the entire disk for known *signatures*, in this case bytes `55 AA` at the end of a sector. These bytes may denote a boot sector for both of the aforementioned file system types.

When a valid backup boot sector is found, a copy of the image file is created with that sector restored at the beginning of its file system. Standard tools are then used to access the contents of the partition. This operation works if the file system is only slightly corrupted and file entries are still intact.

Corrupted File System Reconstruction

Not many published works discuss recovery when metadata is damaged. Naiqi et al. [13] describe an approach for NTFS reconstruction based on deleted files, which are stored in what is called a *well regulated binary tree*. They also suggest *scanning* the disk for interesting artifacts. Nevertheless, their approach to directory tree reconstruction is not very clear. The usage of the binary tree is not thoroughly discussed and no technical details are provided. Their output is briefly compared to a commercial product.

A similar technique was proposed by Jenevein [9] in his patent application for a method for recovering data from damaged media. He discusses an approach based on *scanning* the drive on a sector-by-sector basis to recognize useful data structures for file system recovery. While this method is generic and is described to work for both FAT and NTFS, most of the focus is on the former.

Moreover, he tackles the problem of recovering partition geometry information such as Sectors per Cluster (SPC) and Cluster Base (CB), i.e. the address of the first sector of the file system. However, in this case only FAT is considered. Little details are given with respect to NTFS, which has a different structure.

Finally, it should be noted that he introduces the concept of a *Lost and Found* directory, but it is used to contain traces of files left from other file systems after a reformatting, too. This is conceptually different from the *Lost Files* directory we present in Section 2.3.

Part I

FORENSIC FILE SYSTEM ANALYSIS

2 PROBLEM DESCRIPTION

In this chapter we discuss the basic concepts behind digital forensics and the specification of the forensic file system reconstruction problem. In Section 2.1 we introduce definitions related to file systems and the recovery of their contents. In Section 2.2 we explain why the reconstruction of a damaged file system might be necessary and desirable. Finally, in Section 2.3 we provide a formal problem specification.

2.1 BASIC CONCEPTS

From a physical point of view, computer data is stored in bytes. From a logical point of view, instead, the basic resource for information is a file. To control how data is stored, a special kind of structure called FILE SYSTEM is needed:

> *Definition (File System).* A mechanism to store data in a hierarchy of FILES and DIRECTORIES [3]. Each PARTITION on a storage device has a separate file system.

In particular, a file system relates the contents of a file with its additional metadata, usually including:

NAME Used by a computer system to address the file and access its content. The file system correlates the name with the content.

SIZE The length of the content, expressed in bytes.

PARENT DIRECTORY Identifier that determines what directory contains the file.

TIMESTAMPS Files usually include temporal information, referred as Modification, Access and Creation (MAC) times.

Different types of file systems may associate additional information to files, for example an Access Control List (ACL) or ownership information. The purpose of a forensic file system analysis is to access the file system structure (in the form of a directory tree) and read both metadata and file contents. In general, there are two kinds of analysis based on the status of the system [3, 5]:

LIVE SYSTEM This refers to a computer system which is turned on at the time of the analysis. This implies that the original Operating System (OS) and software are running on the target machine. This kind of analysis is necessary to access volatile information, for example the data stored in Random Access Memory (RAM) or the list of running processes. However, this process should be done carefully to minimize the amount of modifications made to the system.

DEAD SYSTEM The analysis is performed in a trusted environment, with the original system powered off. The drives are accessed in read-only mode to ensure that no modification is made. Each drive is physically copied to an image file, i.e. a bitstream copy which is an exact replica of every byte of the disk.

This thesis focuses on dead system analysis of bitstream images acquired from computer memories such as hard drives. As the main purpose of digital forensics is evidence finding, different search techniques can be applied to look for interesting information [3] such as:

FILE NAMES The first interesting information about a file is its name. This is chosen either by the user who creates a file or set by the software that generates it.

KEYWORDS Files are usually scanned to search for specific keywords of interest in their contents. These keywords depend on both the kind of crime and the needs of the investigator.

TEMPORAL DATA File timestamps can be used to filter the elements of interest inside a specific time frame. Additionally, it is possible to use MAC times to develop a timeline of activity on a system [4]. This allows to reconstruct the order of events.

SIGNATURES The format of a file can be determined by looking at its content and matching bytes with predefined signatures also known as MAGIC NUMBERS. For example, a Joint Photographic Experts Group (JPEG) picture always starts with bytes `FF D8`. This is useful to detect files of types that do not match their names. A user might try to hide a suspicious photograph by changing its extension from `.jpg` to `.dll`, but the signature would remain the same.

File format signatures can be used to recover file contents with the DATA CARVING technique:

Definition (Data Carving). The process of recovering the contents of files by extracting chunks of data, based on known signatures.

This approach does not require knowledge of the file system in which the files reside and it can also be used when the metadata is missing, but only files of known type can be recovered. Moreover, if a file is fragmented the carved data will likely not match the exact contents of the original file. Additional information, including the file name and MAC times, is not recovered.

2.2 MOTIVATION FOR RECONSTRUCTION

Two out of four search techniques described in the previous section explicitly refer to metadata. It is not possible to analyze names or timestamps related to files without accessing the file system.

Additionally, searching for keywords without knowledge of the file system may not yield all files containing the keywords. If a file is fragmented, a *logical volume search* will not find a keyword spanning across two non-consecutive data units. A *logical file system search* is necessary to reorder the pieces of a fragmented file [3].

Evidence found in metadata can be extremely valuable in different cases. The following are some examples of investigations in which metadata extracted from a file system can be important.

Example 1 (Data Theft). The theft of confidential and proprietary information is usually conducted by a competitor to gain an unfair commercial advantage over the victim company. This may happen with the help of an internal attacker, i.e. a worker at the victim company smuggling confidential information to the competitor. Alternatively, criminals might steal laptops owned by employees [6]. In both cases, an analysis of computer systems involved in the incident can be performed to reconstruct the events. This is done by building a timeline of stolen files in a specified time frame. If timestamps are combined with an analysis of ownership information or the containing directory, it may be possible to restrict the suspect pool.

Example 2 (Irregular Accounting). A possible scenario is that of a company that is suspected of having conducted fraud. It is likely that a hidden irregular accounting has been kept in the form of electronic documents, separated from official financial statements. The employees might have deleted the documents with specialized *data destruction* software.

This means that the data units have been overwritten with random data and the file contents are no longer recoverable. However, an analysis of metadata might still allow to read the records associated with directories and their structure. If they still contain a copy of the file names, a claim of the existence of irregular accounting documents can be supported.

Example 3 (Child Pornography). In this kind of investigation, file names and timestamps are used to determine if the material was only possessed or also distributed by the suspect. File names and dates can be compared to those of attachments in email messages found on the target computer [5]. Moreover, it is important to observe that the mere presence of illegal pictures does not necessarily constitute supporting evidence. A picture obtained by data carving does not provide context about its existence.

However, the analysis of its file name and location can help to decide which hypothesis is more likely. For example, a compromising picture might be found at the following path:

```
C:\Users\User\AppData\Local\Mozilla\Firefox\Profiles\
    i8j6n0ud.default\cache2\entries\
    ABFE136EF7ADA46E1E73D10E72F9FB0A1E4A5A42
```

This supports the hypothesis that the file was unintentionally encountered by the suspect while browsing the Web. Instead, a totally different scenario is revealed if the same file is found at the following location:

```
C:\Users\User\Pictures\fav\children\boys\03.jpg
```

In this case the file path clearly supports the claim that the user had knowledge of its existence.

Widespread applications such as file browsers and photo viewers rely on the OS to provide support for different file systems. The OS, in turn, can only read a file system if it is not corrupted. The steps to access data on a file system [11] are:

1. Find the beginning of the partition
2. Navigate to the file entry listing the root directory
3. Interpret the entry and locate the child nodes
4. Repeat the process to further explore the directory tree

This process assumes that the partition is correctly listed in the partition table and its file system is intact. If some metadata is damaged or missing, the file system may become corrupt and standard tools are not able to read the contents of the drive.

For this reason, it is necessary to develop a forensically sound methodology to reconstruct file systems from traces which are still found intact. The purpose is to rebuild a structure as close as possible to the original one. This allows to extract and interpret the parts of metadata that are not damaged.

2.3 PROBLEM SPECIFICATION

With respect to the motivations provided in the previous section, we formalize the FORENSIC FILE SYSTEM RECONSTRUCTION problem in the following way:

Problem (Forensic File System Reconstruction). Given as input:

1. A bitstream copy of a drive (image file)
2. One or more file system types to search

Develop an algorithm that scans the image file and reconstructs the directory structure of any instance of file system found, or parts thereof—if it belongs to one of the required types. The following requirements must be satisfied:

1. The reconstructed structure is a tree and includes any file or directory found. The first node has exactly two children:
   - the actual *Root* of the file system
   - a new directory for *Lost Files*, i.e. files or subdirectories whose parent cannot be determined
2. Information about files—if still available—must include:
   - unique identifier
   - parent node
   - file name
   - type (file or directory)
   - content size and all necessary references to recover it (not applicable to directories)
   - MAC times

The purpose of this algorithm is to give the examiner a precise and complete view of the status of file systems present on the drive or with some traces left. The main goal is to reconstruct a file system which may be damaged.

However, extracted information should also include deleted file systems—i.e. those partitions removed from the partition table—and overwritten file systems, if there are still traces allowing their reconstruction. In any case all relevant data must be accounted for, hence the reconstructed structure shall not be limited to current contents of a file system. It should also include information about DELETED ELEMENTS and GHOST FILES, which we define as follows:

Definition (Deleted Elements). All files and directories whose information is still present on disk, even though they are reported to be unallocated. These elements are not guaranteed to be recoverable because their contents could have been overwritten by other files, but their metadata is still accessible.

Definition (Ghost Files). Files which are definitely not recoverable but their past existence can be indirectly deduced from some other artifacts on disk. They might appear because the drive is heavily damaged, thus metadata corruption occurs.

It is also possible that they have been deleted for a long time and their information was lost, leaving only some traces. For this reason, they *should not* be considered deleted without further investigating why their content is not recoverable.

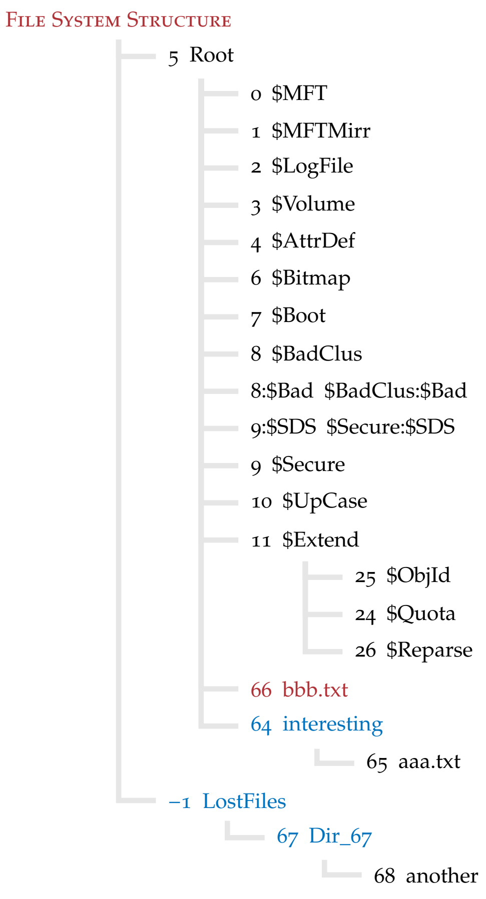


Figure 2.1: Example of directory structure

Figure 2.1 shows a reconstructed file system structure, conforming to our requirements. This example shows a NTFS instance, as can be seen from several metadata files whose name starts with a dollar sign. File `bbb.txt` with identifier 66 is *deleted*, hence it is shown in red.

Both directories `interesting` (id 64) and `Dir_67` (id 67) are *ghost*, so they are shown in blue. For the latter, recovery of its file name is not possible, so we assign a conventional name based on the identifier.

It is important to note that we should also consider `LostFiles` (with dummy id −1) to be *ghost*, because it does not exist in the original file system but it is introduced by our requirements.

3 NTFS BASICS FOR RECONSTRUCTION

New Technology File System (NTFS) was introduced by Microsoft as the default file system for Windows NT, to become a modern and more advanced replacement for FAT [6]. It is currently the default file system for all later versions of Windows [3]—including *2000*, *XP*, *Server 2003*, *Vista*, *7*, *8* and *10*.

Moreover, NTFS has become widespread in high capacity portable hard drives, since files saved in FAT partitions can be at most 4 GB in size. This file system is natively supported in read-only mode on OS X systems and many Linux distributions include the NTFS-3G driver, enabling read-write access to NTFS. This means that, despite being developed for Windows, NTFS can be used by other operating systems as well and it is very likely to be encountered during a digital forensics investigation.

In Sections 3.1, 3.2 and 3.3 we describe the main features of NTFS with respect to its forensic analysis. Sections 3.4 and 3.5 are dedicated to the main concerns regarding its effective reconstruction and data extraction. There is no official NTFS documentation released by Microsoft. Most of the knowledge about this file system derives from the technical documentation released by the Linux-NTFS Project [15]. As a main reference for this chapter, we use the book *File System Forensic Analysis* by Carrier [3], which discusses the aforementioned documentation with further explanations and examples.

3.1 STRUCTURE OF NTFS

The main concept of NTFS is that *everything is a file*. This means that every metadata structure which is part of the file system is listed as a file inside it. As a consequence, NTFS does not have a fixed layout, because the entire file system is considered a data area. The files containing metadata can be anywhere in the file system [3].

The Master File Table (MFT) is the most important data structure in NTFS. It is essential because it contains information about all files and directories [3], including the MFT itself and a backup copy of its first entries. This structure, as any other file, can be located anywhere. Its file name is `$MFT` and its identifier is 0, because it is the first file and its used to access the directory structure. The information for determining the location of the MFT is contained in the BOOT SECTOR.

The boot sector corresponds to the content of the `$Boot` metadata file, which is the only file with a fixed location [3]. This is the first sector (512 bytes) of the file system and it contains information about

BYTE RANGE	DESCRIPTION
0–2	Assembly instruction to jump to boot code
3–10	Original Equipment Manufacturer (OEM) name
11–12	Bytes per sector
13–13	Sectors per Cluster (SPC)
14–15	Reserved sectors[1]
16–20	Unused[1]
21–21	Media descriptor
22–23	Unused[1]
24–31	Unused[2]
32–35	Unused[1]
36–39	Unused[2]
40–47	Total sectors in file system
48–55	Starting cluster address of MFT
56–63	Starting cluster address of MFT mirror
64–64	Size of file record (MFT entry)
65–67	Unused
68–68	Size of index record
69–71	Unused
72–79	Serial number
80–83	Unused
84–509	Boot code
510–511	Signature (`0xAA55`)

1 Microsoft says it must be 0
2 Microsoft says it is not checked

Table 3.1: NTFS boot sector structure [3]

the partition geometry, including the size of clusters. Table 3.1 shows its structure. With respect to file system access, only the following fields are essential:

SECTORS PER CLUSTER The positions of the MFT and its mirror are expressed in clusters, but so are the positions of file contents. This value is needed to convert a cluster address in a sector address and it is discussed more thoroughly in Section 3.4.

STARTING ADDRESS OF MFT This value allows to read the MFT and thus to access the directory structure of the file system.

STARTING ADDRESS OF MFT MIRROR If the MFT is damaged, then the first few entries of the table can be read from its mirror. This permits to access the file system even in the case of slight damages to the first entries in the table.

Although important in describing the structure of the file system, these fields usually have standard values and can be safely guessed:

BYTES PER SECTOR This is always 512 bytes. The only exceptions are recent disks with 4 KB native block size, but these are still very rare. Moreover, many of them include firmware emulation that allows addressing of 512 bytes sectors.

SIZE OF FILE RECORD This is 1024 bytes, i.e. 2 sectors.

SIZE OF INDEX RECORD This is 4096 bytes, i.e. 8 sectors.

The other fields can be safely ignored. To determine if a sector is really a NTFS boot sector, it might be useful to know that the OEM name *usually* contains the string `NTFS`, however this is not mandatory. The last sector of the file system contains a BACKUP BOOT SECTOR, which can be used if the main sector is damaged because it constitutes an exact copy. In this case, the value of TOTAL SECTORS allows to compute the position of the first sector in the file system.

When considering these values, care must be taken as all numerical values in NTFS are *saved in little-endian format*.

3.2 MFT ENTRIES AND ATTRIBUTES

Table 3.2 shows the data structure of a MFT entry, i.e. a record that contains information about a file. The first 42 bytes constitute the header, which can be recognized by the standard signature `FILE`. It is important to note that operating systems might flag corrupted entries with the signature `BAAD`. In an investigation, these records should also be analyzed because they might contain useful data.

Before reading the contents of a NTFS entry which is greater than 1 sector in length, FIXUP VALUES must be applied. These are a mechanism used by the file system to verify the integrity of data [3, 15]. The

BYTE RANGE	DESCRIPTION
0–3	Signature (`FILE` or `BAAD`)
4–5	Offset to fixup array
6–7	Number of entries in fixup array
8–15	`$LogFile` Sequence Number (LSN)
16–17	Sequence value
18–19	Link count
20–21	Offset to first attribute
22–23	Flags (in-use and directory)
24–27	Used size of MFT entry
28–31	Allocated size of MFT entry
32–39	File reference to base record
40–41	Next attribute id
42–1023	Attributes and fixup values

Table 3.2: File record (MFT entry) structure [3]

last two bytes of every sector are replaced with a constant *signature* while the original values are saved in the *fixup array*.

Example (Fixup Values). Let us consider a file record (MFT entry) consisting of the following two sectors:

$$\begin{array}{cccccccc} a_0 & a_1 & \dots & a_{508} & a_{509} & \texttt{68} & \texttt{F1} \\ b_0 & b_1 & \dots & b_{508} & b_{509} & \texttt{EE} & \texttt{F4} \end{array}$$

Assuming a fixup value of `0x0001`, the sectors are written as:

$$\begin{array}{cccccccc} a_0 & a_1 & \dots & a_{508} & a_{509} & \texttt{01} & \texttt{00} \\ b_0 & b_1 & \dots & b_{508} & b_{509} & \texttt{01} & \texttt{00} \end{array}$$

The fixup array is made of values `0x0001`, `0xF168` and `0xF4EE` (the first being the signature). The OS checks the value of the signature against the last bytes of each sector of the file record. If they do not match, the MFT entry is marked as corrupted.

After the signatures are checked and the original values restored, the entry can be analyzed to recover information about its file. In particular, we are interested in the following fields:

FLAGS This field contains information about the kind of file and its status. If flag `0x02` is set, then the entry refers to a directory. If flag `0x01` is *not* set then the entry is in-use, otherwise it refers to a deleted file.

RECORD NUMBER If we read a single MFT entry without knowing the context, we need this value to know the identifier of the current file. This information was not available in earlier versions of NTFS and the value was deduced from the position inside the MFT [14]. However, starting from version 3.1 included with Windows XP, the value is stored in bytes 44–51.

OFFSET TO FIRST ATTRIBUTE This value determines where to start parsing the attributes. Moreover, it allows to understand if the MFT entry contains the record number.

ATTRIBUTES Some of the attributes listed in a file record are essential to recover the file name and its contents. Moreover, some are useful to retrieve information about ghost files.

Attributes are data structures associated to a MFT entry that include all the metadata for its file. They are stored starting from the offset written in the file record, one after the other. Each attribute has a different structure, however they share a common header. Table 3.3 shows their general structure, with an important categorization.

Resident attributes are totally contained inside the MFT entry. This is possible only for small attributes, since the whole entry must fit in 1024 bytes. Bigger attributes are *non-resident*, i.e. their content is somewhere else in the file system. The location of a non-resident attribute is described in a RUNLIST.

The runlist is a very compact data structure representing the position of one or more runs, which constitute the contents of a *non-resident* attribute and it is composed by one field for each run. The first byte of each field determines the size of its elements. The lower 4 bits contain the size of the *run length*, while the upper 4 bits contain the size of the *run offset*.

The values written in each run are expressed in clusters and the first bit determines the sign. The first value is relative to the start of the file system, while each other value is relative to the previous one.

Example (Runlist). Consider the following runlist:

```
31 02 50 BA 00 31 01 6C 31 02 21 05 24 60 00
```

The first byte is `0b00110001` (`0x31`), indicating that the following 1 byte contains the run length and the next 3 bytes contain the run offset [3]. Thus, byte `0x02` denotes a length of 2 and bytes `50 BA 00` must be interpreted as little-endian. The other runs are decoded in the same way, leading to the following:

- length 2, offset $o_1 = 47696$ (`0x00BA50`)
- length 1, offset $o_2 = o_1 + 143724$ (`0x02316C`)
- length 5, offset $o_3 = o_2 + 24612$ (`0x6024`)

BYTE RANGE	DESCRIPTION
0–3	Attribute type identifier
4–7	Length of attribute
8–8	Non-resident flag
9–9	Length of name
10–11	Offset to name
12–13	Flags
14–15	Attribute identifier

(a) Common header

BYTE RANGE	DESCRIPTION
16–19	Size of content
20–21	Offset to content

(b) Resident attribute

BYTE RANGE	DESCRIPTION
16–23	Starting VCN of the runlist
24–31	Ending VCN of the runlist
32–33	Offset to the runlist
34–35	Compression unit size
36–39	Unused
40–47	Allocated size of attribute content
48–55	Actual size of attribute content
56–63	Initialized size of attribute content

(c) Non-resident attribute

Table 3.3: MFT attribute structure [3]

There are several different types of attributes and custom ones can also be defined. However, the most important ones to reconstruct the directory tree are:

$ATTRIBUTE_LIST (ID 32) If a file record cannot hold all attributes, additional *non-base* MFT entries are reserved for storing them. The attribute list contains one entry per attribute. Bytes 0–3 of each entry contain the attribute type, while bytes 16–23 hold the file reference to the MFT entry. Bytes 8–15 contain the starting Virtual Cluster Number (VCN), used to order multiple chunks of an attribute which may be split across several MFT entries. The length of each attribute entry is stored in bytes 4–5.

$STANDARD_INFORMATION (ID 16) This attribute has the lowest id and it is the first to be encountered in a MFT entry. It stores MAC times at bytes 16–23, 24–31 and 0–7 respectively. It also includes an additional timestamp at bytes 8–15, called MFT modification time. This is the last time metadata was updated, but it is not displayed by Windows. Timestamps are saved in a weird format: the *number of one-hundred nanoseconds* since January 1, 1601 Coordinated Universal Time (UTC) [3]. The attribute also includes flags and ownership information.

$FILE_NAME (ID 48) Each *base* MFT entry contains one or several attributes for file names. This is usually the case as DOS requires names in the *8.3 format* while Windows supports longer names, hence both versions are stored. Byte 64 contains the length of the name, which is stored starting from byte 66. Bytes 0–7 contain a reference to the parent directory. We need this information to rebuild the directory structure of the file system, using the algorithm described in Chapter 5.

$INDEX_ROOT (ID 144) In NTFS, directory contents are stored inside B-trees. This is the root node, listing *some* of the files and subdirectories. If a directory contains many children, additional nodes are required to store the remaining contents. Bytes 0–3 contain the type of attribute in the index (usually 48). The node header for index entries can be found starting from byte 16.

$INDEX_ALLOCATION (ID 160) This attribute stores nodes needed to describe the contents of directories with many files or subdirectories. There can be multiple `$INDEX_ALLOCATION` attributes in the same MFT entry and they are always non-resident. They should not exist without a corresponding `$INDEX_ROOT` attribute. The runlist of this attribute points to locations continaing INDEX RECORDS, which are described in Section 3.3.

$DATA (ID 128) This attribute contains the raw file contents. For files over 700 bytes it is usually non-resident and it does not

BYTE RANGE	DESCRIPTION
0–3	Signature (`INDX`)
4–5	Offset to fixup array
6–7	Number of entries in fixup array
8–15	`$LogFile` Sequence Number (LSN)
16–23	VCN of this record in the index stream
24+	Node header (see Table 3.5)

Table 3.4: Index record structure [3]

have specific size requirements [3]. During an analysis, we must be careful because directories may contain a `$DATA` attribute and this is suspicious. Although files usually contain only one data attribute, they might also have an Alternate Data Stream (ADS) or even more.

An ADS is a named `$DATA` attribute associated to a file. Windows does not provide GUI tools to access these kind of streams, however they might be created by referencing a file name followed by a colon and the name of the stream. For example, an executable file might be hidden inside an ADS of a text file with the following command [4]:

```
type notepad.exe > myfile.txt:np.exe
```

For this reason, it is important to check for hidden information in all `$DATA` attributes in files *and* folders. A reconstruction algorithm should take every ADS into account and consider it a component of the enclosing file—or a separate node whose name includes also the main one. Figure 2.1 is an example of the latter choice, because streams `$BadClus` and `$BadClus:$Bad` are shown as different nodes in the directory tree.

3.3 INDEX RECORDS

Index records form the contents of non-resident `$INDEX_ALLOCATION` attributes. Table 3.4 shows the structure of the first 24 bytes of a record, i.e. its header. This kind of record shares some similarities with a MFT entry. First of all, the first sector of an index record is recognizable from the signature `INDX`. Moreover, fixup values must be applied before decoding the contents of the record.

An index record contains some nodes of the B-tree. Each node starts with a standard header, which is shown in Table 3.5. The header contains information about the starting and ending offsets of the *index entry list*, i.e. a list of the contents in the node. The root node contained inside a `$INDEX_ROOT` attribute has exactly the same structure.

BYTE RANGE	DESCRIPTION
0–3	Offset to start of index entry list[1]
4–7	Offset to end of used portion of index entry list[1]
8–11	Offset to end of allocated index entry list buffer[1]
12–15	Flags

1 Relative to start of the node header

Table 3.5: Index node header structure [3]

BYTE RANGE	DESCRIPTION
0–7	MFT file reference for file name
8–9	Length of this entry
10–11	Length of `$FILE_NAME` attribute
12–15	Flags
16+	`$FILE_NAME` attribute
Last 8 bytes	VCN of child index node (if flag is set)

Table 3.6: Index entry structure [3]

The structure of an index entry is shown in Table 3.6. For the purpose of directory tree reconstruction, we are interested in:

MFT FILE REFERENCE This is the identifier for the MFT entry of the file corresponding to this index entry. This value is normally read by the OS to access a file from its path.

$FILE_NAME ATTRIBUTE This is an exact replica of the same attribute stored in the MFT for the corresponding file. This means that we can infer the name and MAC times of files which appear in index records but which do not have a MFT entry anymore. In other words, this duplication allows to find *ghost* files [5].

It is important to note that old index entries might still be written in the SLACK SPACE, i.e. the bytes between the end of the *used* portion of the index entry list and the end of its *allocated* space. An entry may be in the slack space because the file was deleted or because the nodes in the B-tree were reallocated [3].

Finally, another useful information is the MFT file reference to the directory which owns the index record. From Table 3.4, we can see that this information is *not explicitly stored* inside the record. However, we can use bytes 0–7 of each `$FILE_NAME` attribute to read the identifier for the parent MFT entry. In all practical cases these values are all equal and thus determine the corresponding directory. Additional details are provided in Section 6.2.

3.4 PARTITION GEOMETRY

As seen in Section 3.2, non-resident MFT attributes address their contents using runlists, which contain values expressed in clusters. These values affect two crucial attributes:

$INDEX_ALLOCATION The list of children in a directory.

$DATA The actual file contents.

To be useful, runlists depend on:

CLUSTER BASE The CB of a file system is the position of its first cluster and also its first sector. For NTFS, this is also the position of the boot sector.

SECTORS PER CLUSTER Offsets need to be converted in sectors by using the value of SPC stored in the boot record. This can be different for each NTFS partition.

These two values determine the *partition geometry* and are easily discovered from a NTFS boot sector. If for some reason such sector is not available, we must try to recover them. The lack of the SPC and CB values precludes access to non-resident `$INDEX_ALLOCATION` and `$DATA` attributes. This would imply that most *ghost* files and the contents of any moderately large file for a NTFS instance *would not be recoverable*.

Jenevein [9] presents formulas for computing the values, but they work only for FAT. Moreover, he explains that "if information on the number of Sectors per Cluster (SPC) is lost, any file system information storing location of files and directories using clusters becomes useless." [9] For this reason, effective file system reconstruction includes determining the aforementioned values.

We underline the fact that this has to do with the *recoverability* of a partition. While actual file contents might not be recoverable and information about *ghost* files may not be complete, it is still possible to reconstruct the directory tree of a NTFS instance by using information found in MFT entries, with the algorithm presented in Chapter 5.

3.5 LIMITATIONS OF RECONSTRUCTION TECHNIQUES

As discussed in Section 1.3, the most common technique to access file systems is to follow a TOP-DOWN approach. With respect to NTFS, this means reading the boot sector to determine the position of the MFT and then starting to read directories. A more refined approach could include detecting backup copies of boot sectors and their recovery to gain access to deleted partitions.

A top-down approach cannot be considered a valid reconstruction technique, because the procedure relies on the fact that the file system is still intact. Information about *deleted* or *ghost* files might be deduced from index records, but nothing more. If the start of the MFT is corrupted, the file system would be completely unreadable and useful portions of the directory tree would not be reconstructed.

Even worse, the first entries of a MFT might be intact but at a later point some file records related to subdirectories might be missing or corrupted. In that case, the output would be a tree in which the top part is correct, but some portions at the bottom are *completely pruned*, starting from the damaged directories. It might be difficult to deduce it from the result, i.e. relevant data might be silently lost.

More advanced techniques are based on the idea of *scanning* the disk for known records and building the tree in a BOTTOM-UP fashion [9]. In this case, parts that cannot be linked back to the root are reconstructed in a directory dedicated to lost files. This avoids the problems of a top-down approach. However, this method requires determining the partition geometry, which sometimes is not available.

These techniques might also suffer from limitations. If the software aims at data recovery rather than forensic examination, it might focus only on recoverable files. Hence, it would ignore the presence of *ghost* files in `$INDEX_ALLOCATION` attributes or that of *deleted* files whose `$DATA` attribute has been overwritten by other contents.

4 CORRUPTED METADATA

There are several reasons why both data and metadata may become corrupted in hard drives, impacting integrity of file systems. These can be accidental or caused by human action. According to the kind of corruption and its severity, different scenarios may arise that make file system reconstruction harder. In this chapter we discuss the main corruption scenarios, their causes and their potential effects.

4.1 BROKEN FILE RECORDS

The basic case of file system corruption is that of damaged file records. A directory may reference a child whose file record is either missing, unreadable or overwritten by something else. The impact of this situation is usually limited to the files and directories involved, but the file system as a whole is hardly affected.

In NTFS, a missing or broken MFT entry means that the contents of the file cannot be recovered, i.e. the data is lost. However, its existence can be inferred from index records, as discussed in Section 3.3. This means that despite the corruption, some of the essential metadata can be recovered, including the file name and MAC times.

If the MFT entry was that of a directory, then the impact could be higher. In fact, losing the `$INDEX_ROOT` attribute prevents access to some of the directory children, so it is not possible to check for *ghost* files. The same reasoning seems to apply also for `$INDEX_ALLOCATION` attributes, however these are non resident and index records can be found by using signatures as seen in Section 3.3.

The effects of this problem to file system reconstruction are limited, because missing file records do not impact the general structure of the directory tree. A software reading the file system top-down might be blocked at the first corrupted directory node, however a clever bottom-up approach does not suffer from this problem, e.g. using the algorithm presented in Chapter 5. Moreover, the partition geometry is not affected at all.

4.2 OVERWRITTEN PARTITIONS

If a hard drive has been partitioned more than once, there might be traces left from previous file systems. These artifacts are extremely interesting from a forensic point of view, as they could provide additional information about files previously stored on disk. This may include both metadata and actual file contents.

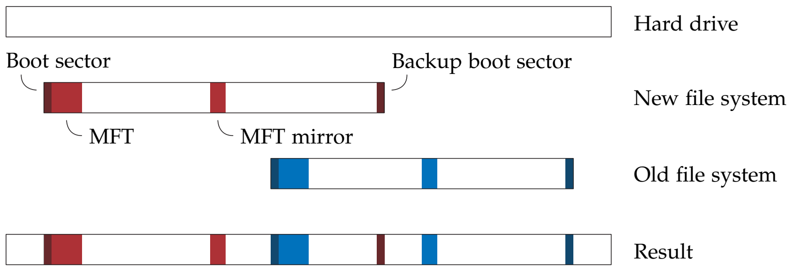


Figure 4.1: Non-overlapping MFT

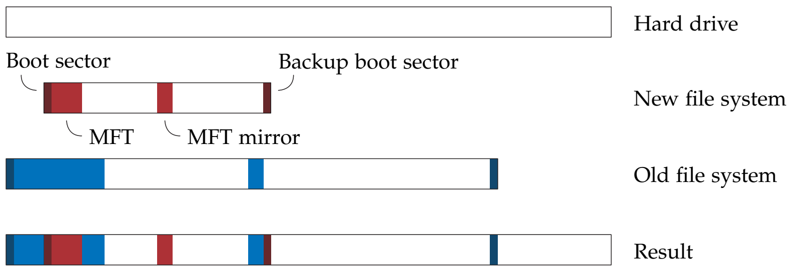


Figure 4.2: Overlapping MFT

A partition whose entry has been deleted from the partition table does not pose a great challenge with respect to reconstruction, because its content is not affected from this operation. However, when new partitions are created and initialized with a file system, the situation becomes much worse. The amount of damage to the old file system and the effects on the recoverability may vary. They depend on the positions, sizes and amount of overlap of partitions.

In this section we consider different scenarios with two NTFS file systems, *right after* the creation of the new one. The *old* file system is marked in blue and the *new* one is marked in red.

Figure 4.1 shows what happens when the new file system overlaps with the old one, but its different position causes the old MFT to be preserved. Thus, every MFT entry can be read and both directory trees can be fully reconstructed. The red backup boot sector overwrites a portion of data in the old file system, hence some files may not be recoverable. The old MFT and file contents might be overwritten after the new partition starts to be filled with data.

Figure 4.2 depicts a situation with partitions starting from nearby positions. In this case, the damage is much worse because the red MFT is overwriting parts of the blue one. This means that any overwritten MFT entry in the old file system cannot be restored. The existence of affected files might still be recoverable but they would show up as *ghost* files, thus their contents are lost.

The MFT is allocated sequentially and it is located in the first 12.5% of the file system [3]. Therefore, as soon as some files are created in the new partition, additional file records are lost in the old one. The effects of a broken file record are multiplied by the number of

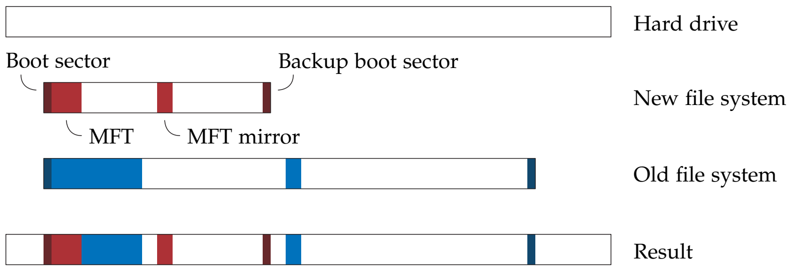


Figure 4.3: Overlapping MFT and boot sector

overwritten MFT entries in the blue file system, severely impacting reconstruction. However, geometry for both partitions can be inferred because their boot sectors are intact.

Finally, Figure 4.3 shows two file systems starting at the same offset. This means that the blue boot sector is lost, because the red one is written exactly over it. Technically speaking, the MFT might be located anywhere on disk, however formatting tools usually implement standard values for its position. Hence, the entries of the red MFT are likely to be written over the corresponding blue ones.

As a consequence, it might be very hard or impossible to tell which MFT entries belong to the old file system and which belong to the new one. Information about the partition geometry regarding both file systems might be gained if the blue backup boot sector is still available, however this does not help to separate files if the MFT position is the same for both file systems.

In this case, reconstruction of the two file systems would not be successful. It may be possible to get a directory tree including a mix of files from the two file systems, however this is not a desirable outcome because it does not reflect their real structure.

The worst-case scenario is that of exactly overlapping file systems, including the same size and MFT position. In this case, it would not be possible to tell that an old file system existed and some of the recovered MFT entries belong to it.

4.3 HARDWARE DAMAGE

A hard disk is acquired by performing an exact copy of the physical drive to an IMAGE FILE, also known as bitstream copy. The content should exactly reflect every byte of the input drive and the *imaging utility* used to perform this operation must be carefully tested [5]. A simple and flexible acquisition tool is called `dd`. It is found in many UNIX versions (including any Linux OS) and it is also available as an executable compiled for Windows [3].

The following command represents the basic usage of `dd`. It performs the copy of device `/dev/sda` to an image file called `copy.img`:

```
dd if=/dev/sda of=copy.img
```

The tool works by copying blocks of bytes in sequential order. This is fine when the drive is not damaged, however it stops abruptly in case one block cannot be read. The `conv=noerror` flag may be passed to avoid interruptions, but this simply skips bad blocks and the data acquired after them will not be located at the correct offset [3].

Another option is to use GNU `ddrescue`, a similar tool that can adaptively change the size of blocks to read recoverable data in a fine-grained fashion. This is the most interesting method from a forensic point of view, because the output copy is as close as possible to the original contents of the drive. Offsets are respected, hence intact portions of file systems and their contents can be read.

However, verification of the image file using a *checksum* becomes a problem. Since `ddrescue` fills unreadable blocks with zeros, the hash of the copy will not match that of the original drive [6]. This can be an issue in validating evidence, because a failing drive might produce different output errors if read multiple times, leading to bitstream copies differing in their checksums.

Hardware can become damaged for a variety of reasons. Any kind of physical media experiences decay over time, however magnetic disk drives are especially subject to damages and data corruption [9]. Furthermore, hardware might suffer harm caused by heat, smoke particles, water and exposure to harsh conditions (e.g. fire, jet fuel and toxic chemicals) [5]. Physical deterioration affects not only hard drives, but also CDs and DVDs, for which a specialized imaging tool called `dvdisaster` may be used [6].

4.4 EVIDENCE HIDING AND DESTRUCTION

Another cause of corruption is human intervention with the goal of hiding or destructing evidence. The most effective way for a user to erase data is the use of WIPING SOFTWARE. These tools are known with different names, including *secure delete* programs, *file shredders* and *disk wipers*. They work by writing zeros over allocated file contents or entire file systems [3].

If a program overwrites a partition fully, its file system is completely gone, including any metadata. This makes recovery impossible, because there are no traces left. It can be considered the most extreme case of corruption caused by human action. Instead, wiping tools working on individual files are less effective. These applications are usually focused on making data unrecoverable, while we are more interested in metadata for rebuilding a directory tree.

In particular, file shredders always try to overwrite the contents of a file, with different degrees of success depending on the OS and other factors [3]. However, different scenarios may be possible regarding the metadata. In the best case, the wiping tool does not try to hide it, hence the MFT entry can be recovered, including the file name and other interesting information. A smarter software might rename the file but leave the file record untouched. The worst case is that of a secure delete program which overwrites the MFT entry as well.

This is not strictly a file system corruption, because the file is effectively gone. Nevertheless, from a forensic point of view we might consider the entry as damaged, since we lose metadata related to that file. Some traces about its existence may still be left in the slack space of index records in the parent directory.

It is very unlikely that a tool can have such a deep control on the file system as to clean the aforementioned slack. Thus, it may be possible to find artifacts of a *ghost* file.

When it comes to hiding evidence, rather than erasing it, the number of techniques is likely limited only by the user's creativity [6]. Many approaches may cause file system corruption, if not done properly. However, the goal of the author is likely to perform modifications which are not easily detected [8]. As a first option, an individual might alter the metadata related to bad clusters in a file system. This alteration prevents the OS from reading and writing data at the locations marked as damaged and it works for NTFS (by changing the `$BadClus` metadata file), FAT and other file systems [6].

For this reason, an analysis tool shall not trust information about bad clusters. Moreover, in most modern hard drives the hardware controller is able to detect bad sectors before the OS [8]. The mere presence of clusters marked as bad should be considered suspicious.

A different approach might target partition information to hide data or whole file systems. A hard disk might contain both an Apple Partition Map (APM) and a Master Boot Record (MBR) [6]. The former is preferred by Apple computer systems while the latter is used by IBM-compatible PCs. A user of OS X might set up partitions in APM format, including a fake MBR listing different partitions (or none). This trick may confuse a forensic examiner if the analysis is performed on a PC. Information found in partition tables should never be trusted without proper verification [6].

Huebner et al. [8] describe additional data hiding techniques, focusing especially on NTFS. The easiest approach is to add a new ADS to a file for storing hidden data, as described in Section 3.2. This operation can also be done to directories, without corrupting the file system [6, 8]. A similar approach is extending the existing `$DATA` attribute of NTFS metadata files by adding more cluster runs, effectively increasing their sizes and leaving room for hidden information.

A third way to exploit file contents is to alter the `$DATA` attribute of the `$Boot` file, i.e. the boot sector of the file system. In NTFS, 16 bytes are allocated for it but less than 1 is used [6]. As a consequence, small amounts of data can be hidden by changing it and using additional space in `$Boot`. When analyzing a file system, the boot sector should be compared with its backup [8].

Part II

FILE SYSTEM RECONSTRUCTION ALGORITHM

5 BOTTOM-UP TREE RECONSTRUCTION

As discussed before, top-down approaches are not suitable for forensic file system reconstruction because they do not work with partially corrupted metadata. In this chapter we introduce an algorithm for rebuilding a file system bottom-up.

In Section 5.1 we start by collecting all available metadata. In Section 5.2 we continue by separating files from different partitions. In Section 5.3 we finish by reconstructing a directory tree as specified in Chapter 2. The discussion focuses on NTFS, however the technique we provide is file system independent. In Section 5.5, special attention is given to improving the set of available metadata by detecting *ghost* files in NTFS index records.

5.1 DISK SCANNING FOR METADATA CARVING

Carrier [3], Jenevein [9], Naiqi et al. [13] suggest *scanning* a hard drive to search for file system metadata, which are analyzed for the purpose of accessing the directory structure. This approach is interesting because it is carving applied to metadata rather than file contents.

Although carving can only be used to recover some files without names or timestamps, it is very successful in the collection of artifacts from key metadata structures in file systems. These pieces can be merged and interpreted later on. Therefore, signatures for interesting kinds of metadata need to be defined. After that, each sector of the drive is compared to the signatures and the position of each candidate is recorded for later.

Algorithm 5.1 Disk scanner for NTFS

```
1:  found_boot = [ ]
2:  found_file = [ ]
3:  found_indx = [ ]
4:  for all sectors $s$ with position $i$ do
5:      if $s$ ends with 55 AA and NTFS $\in s[0,7]$ then
6:          found_boot.append($i$)
7:      end if
8:      if $s[0,3] \in$ [FILE, BAAD] then
9:          found_file.append($i$)
10:     end if
11:     if $s[0,3] =$ INDX then
12:         found_indx.append($i$)
13:     end if
14: end for
```

Algorithm 5.1 contains the definition of a disk scanner for NTFS. The interesting file system metadata artifacts include boot sectors, MFT entries and index records, i.e. contents of `$INDEX_ALLOCATION` attributes.

In the first case, we want to compare the last 2 bytes of the sector to `55 AA`. An optional constraint is added, namely checking for string `NTFS` in the first 8 bytes. This is not strictly a requirement for NTFS boot records, however it is very common and it allows to filter out many false positives, including FAT boot sectors.

For file and index records, a candidate sector is matched if a fixed signature is found in the first 4 bytes. This can lead to false positives as well, however they can be checked later. Regardless of the file system type, at the end of the scanning process a few LISTS OF INTERESTING SECTORS are produced.

5.2 FILE RECORD CLUSTERING

The carved metadata obtained by scanning a disk might be composed of file records and other artifacts of several file systems. As stated in Section 2.3, our goal is to reconstruct any file system of a given type. For example, a scan may lead to the collection of MFT entries belonging to two different NTFS partitions. In this case, we need to cluster file records dividing them in two distinct groups.

The procedure to solve this problem is file system dependent. For NTFS, the following facts can be used:

- The first 12.5% of the file system is reserved for MFT entries [3] as seen in Section 4.2.
- Hence, the whole MFT is usually contiguous.
- The positions of sequential file records should differ by the size of one MFT entry, namely 1024 bytes or 2 sectors.

Given a MFT entry located at offset y on disk (in sectors) we can parse it and read its record number x. Assuming a contiguous MFT, the following formula holds:

$$y = p + sx$$

Where p is the location of the first entry (id 0) in the same file system and s is the size of one file record. In practice, $s = 2$. The formula can be reverted to determine the value of p:

$$p = y - sx$$

We can use this offset to cluster MFT entries. Entries leading to the same value *belong to the same file system*. This simple strategy has the effect of separating file records easily, automatically determining the number of clusters. The rigorous steps of this process can be seen in

Algorithm 5.2 File record clustering for NTFS

```
1: partitions = { }
2: $s$ = MFT_entry_size
3: for all file records $r$ do
4:     $p = r_{pos} - s \cdot r_{id}$
5:     if $p \notin$ partitions then
6:         partitions$[p]$ = { }
7:     end if
8:     partitions$[p][r_{id}] = r$
9: end for
```

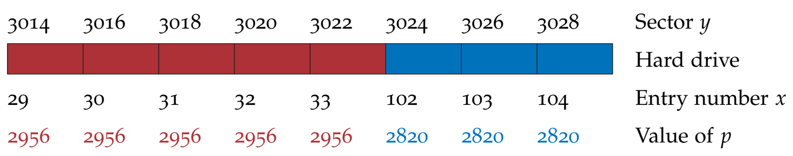


Figure 5.1: Example of clustering of MFT entries

Algorithm 5.2. Essentially, it consists in a loop computing p for every record r, storing results in an associative array.

Figure 5.1 shows an example of clustering of MFT entries related to two NTFS partitions. The colored cells illustrate the correct separation of these entries. From the computed value of p, we can see that the position of the first entry is an effective way to determine a clustering. Each MFT entry of the red file system gets a value equal to 2956 and every file record of the blue file system gets 2820.

There are two situations in which this approach may lead to an incorrect clustering. The first one is that of *overlapping partitions with the same MFT offset*. This was already discussed in Section 4.2, motivating the impossibility of separating file records in this case. The other one is the presence of *a fragmented MFT*, which occurs if it becomes bigger than its reserved space because many entries are needed [3].

If a MFT is fragmented, each part has a different offset, leading to the recognition of multiple file systems. Assuming that the `$MFT` file record (id 0) is readable, the pieces may be combined by reading the runlist of its `$DATA` attribute. In the opposite case, user intervention is required to judge if artifacts are spread across fragments of the same MFT or they belong to different file systems.

A similar analysis might be performed to recognize contents of a `$MFTMirr` file, which would otherwise lead to a partition containing a copy of the first few entries in the main MFT.

Each type of file system may require a different approach for clustering. Nevertheless, it is important to note that the strategy should be as precise as possible. Preferably, it should not depend on boot records or other specific metadata structures that might be corrupted. The accuracy of this step is crucial because it determines the quality of the reconstruction for recognized partitions.

5.3 DIRECTORY TREE RECONSTRUCTION

In Section 5.2 we have discussed how to perform file record clustering for NTFS, obtaining one *list of file records* for each file system we want to reconstruct. In this case, records are essentially MFT entries. However, we can assume having a generic *File* object that allows us to access essential metadata as listed in Section 2.1.

The idea of our recovery strategy is to perform a bottom-up reconstruction during which A FILE IS LINKED TO ITS PARENT NODE. This is exactly the opposite of standard top-down file system access discussed in Section 1.3. When the parent node is unknown, the current file is placed in the *Lost Files* directory as required in Section 2.3.

Algorithm 5.3 Bottom-up file system reconstruction

```
Require: partition $p : i \to f$ // mapping id to file
Require: id $r$ of the root directory
 1: $p_{\text{lost}} = \text{File}(-1, \text{“LostFiles”}, \textit{directory}, \textit{ghost})$
 2: if $r \notin p$ then // missing root directory
 3:     $p[r] = \text{File}(r, \text{“Root”}, \textit{directory}, \textit{ghost})$
 4: end if
 5: for all files $f$ with id $i \in p$ do
 6:     if $f_{\text{id}} = r$ then
 7:         $p_{\text{root}} = f$
 8:         $f_{\text{name}} = \text{“Root”}$
 9:     else // $f$ is not the root node
10:         $j = f_{\text{parent}}$
11:         if $j \neq$ null and $j \in p$ then // parent exists
12:             $n = p[j]$
13:         else if $j \neq$ null and $j \notin p$ then // parent is lost
14:             $n = \text{File}(j, \text{“Dir\_”} + j, \textit{directory}, \textit{ghost})$
15:             $p[j] = n$
16:             $p_{\text{lost}}.\text{add\_child}(n)$
17:         else // parent is unknown
18:             $n = p_{\text{lost}}$
19:         end if
20:         $n.\text{add\_child}(f)$
21:     end if
22: end for
```

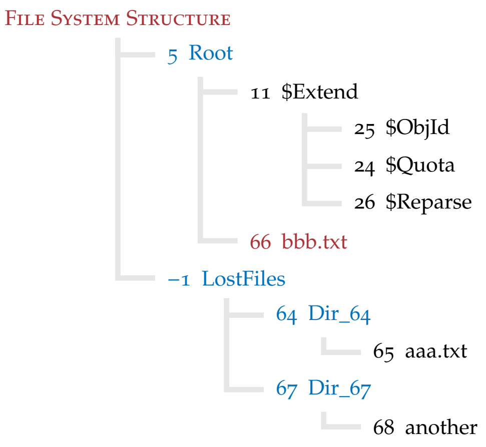
The procedure is shown in Algorithm 5.3. It consists in defining the *Lost Files* directory, ensuring a *Root* directory exists and then iterating over each file to rebuild the directory tree.

There are two requirements:

PARTITION p This data structure represents the file system we want to reconstruct. Its main component is mapping $p : i \to f$ which links each identifier to the corresponding *File* object. Moreover, fields p_{root} and p_{lost} are needed to store a reference to the main directories, namely *Root* and *Lost Files*.

ID	PARENT	NAME
11	5	$Extend
25	11	$ObjId
24	11	$Quota
26	11	$Reparse
65	64	aaa.txt
66	5	bbb.txt
68	67	another

(a) Input file records



(b) Output directory tree

Figure 5.2: Example of NTFS reconstruction

IDENTIFIER r OF THE ROOT DIRECTORY This value is required to determine which file is the root node. It is a parameter because it is file system dependent. For example, in NTFS this is the identifier of the . file which is always 5.

Lines 1–4 are used to initialize the *Lost Files* directory and ensure there is a *Root* directory if its file record was not recovered. After these initial steps, the main process starts on line 5 looping over every file f. Lines 6–8 handle the case when f is the root directory. If this holds, then no parent needs to be considered and p_{root} is updated. The name of f is overwritten to avoid inconsistencies among different file systems. Moreover, the default name of . in NTFS is tricky because it is not a valid filename.

Lines 9–18 cover other files. There can be three different cases, based on the identifier j of the parent node. The first case is when j can be read and there is a *File* object for the parent in p, then the parent node is set as $n = p[j]$ (lines 11–12). The second case is when j is known but it refers to a lost node. The creation of a new *ghost* object is required, and lines 13–16 handle that. The node is inserted in the partition and it is added to the children of p_{lost}. The third case is when j is unknown, therefore the parent node is simply p_{lost}.

Anyway, f is added to the children of n on line 20 and the same technique is repeated for other files. After the rebuilding procedure, the output file system in p can be read in the usual top-down fashion. This concludes the directory tree reconstruction.

The algorithm accurately rebuilds the directory tree of the file system and IT DOES NOT REQUIRE THE METADATA TO BE COMPLETE. Regardless of the amount of corrupted file records, portions of the tree which are not damaged are fully reconstructed. Figure 5.2 shows an example derived from the file system in Figure 2.1, by stripping many standard NTFS file records.

5.4 HANDLING ALTERNATE DATA STREAMS

During our discussion on MFT attributes in Section 3.2, we have seen that *both files and directories* in NTFS can have multiple `$DATA` attributes. However, the abstract model of file system presented in Section 2.3 is fairly simple and does not have the notion of ADS.

The goal of forensic file system reconstruction is to provide an accurate and complete view of file system contents, hence we need to determine a way to add ADS information to the output. The analyst must be able to acknowledge the presence of all data streams and to recover their contents, if possible. At the same time, we do not want to change the simple file system used in our model.

A solution to this problem was mentioned at the end of Section 3.2, namely providing a separate node for each ADS. Figure 2.1 already showed an example of this approach. In particular, the same naming convention used in NTFS is adopted. We can see that file `$BadClus` is listed with id 8 and its ADS called `$Bad` originates another node in the tree, whose identifier also contains the name of the stream.

Algorithm 5.4 Handling multiple ADS in NTFS

```
1: for all NTFS partitions $P$ do
2:     for all file records $r \in P$ do
3:         for all alternate streams $s \in r$ do
4:             $n = r_{\text{id}} + \text{“:”} + s_{\text{name}}$
5:             $P[n] = r$
6:             $P[n]_{\text{name}} = s_{\text{name}}$
7:         end for
8:     end for
9: end for
```

Algorithm 5.4 shows the iterative procedure needed to apply this method to all MFT entries for any detected partition. If record r contains an ADS s, a new name n is created by joining the identifier of r and the stream name with a colon. Afterwards, a new entry $P[n]$ is created, pointing to the same file record r. To keep track of the relevant ADS, its name is stored as field $P[n]_{\text{name}}$.

This process must be done before the bottom-up reconstruction and we describe it only for NTFS. Nevertheless, such a simple approach can be easily adapted to other file systems. To include all data streams, information about the partition geometry must be known. This is because some `$DATA` attributes might be located in non-base MFT entries referenced by a non-resident `$ATTRIBUTE_LIST` attribute.

5.5 DETECTING GHOST FILES FROM INDEX RECORDS

The bottom-up algorithm introduced in Section 5.3 inherently detects the presence of *ghost* directories when the parent node of a file is missing. However, in Section 3.2 we have seen that MFT entries for directories provide two attributes listing the children, namely `$INDEX_ROOT` and `$INDEX_ALLOCATION`. Therefore, the output of Algorithm 5.3 can be improved if the list of files is updated with nodes deduced from these attributes before performing the reconstruction.

In particular, improvement based on the `$INDEX_ROOT` attribute may be achieved regardless of the availability of partition geometry information, because it is always resident [3]. Algorithm 5.5 shows the process of integrating ghost files deduced from `$INDEX_ROOT` attributes.

Algorithm 5.5 Ghost files detection from `$INDEX_ROOT`

```
1: for all NTFS partitions $P$ do
2:     for all records $r \in P$ having $INDEX_ROOT attribute $a$ do
3:         for all $FILE_NAME entry $e \in a$ do
4:             if $e_{\text{id}} \notin P$ then
5:                 $P[e_{\text{id}}] = \text{ghost\_file}(e)$
6:             end if
7:         end for
8:     end for
9: end for
```

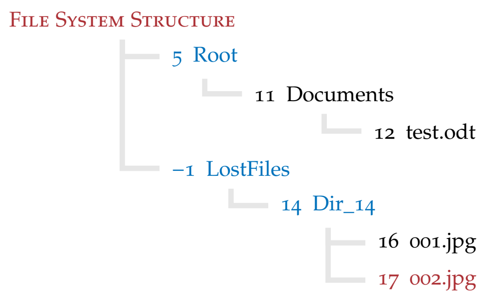
All entries e in attribute a are parsed, however there is a good chance that they refer to files which are already known. Hence, identifier e_{id} is checked to avoid overwriting a *File* node gathered from a MFT entry. If the identifier is new, then a ghost node is created from information contained in the `$FILE_NAME` attribute e.

A similar strategy may be applied to `$INDEX_ALLOCATION` attributes as well, however these are always non-resident. Therefore, partition geometry information must be known. Algorithm 5.6 shows the procedure for deducing ghost files from these attributes.

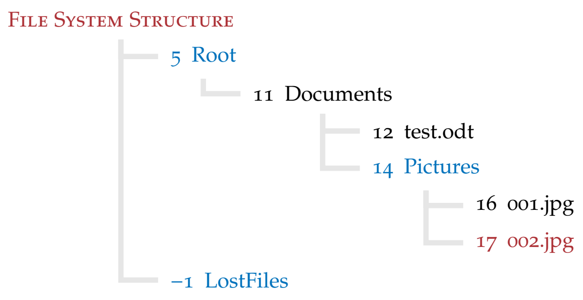
Algorithm 5.6 Ghost files detection from `$INDEX_ALLOCATION`

```
1: for all NTFS partitions $P$ do
2:     for all records $r \in P$ do
3:         for all $INDEX_ALLOCATION attributes $a \in r$ do
4:             for all $FILE_NAME entry $e \in a$ do
5:                 if $e_{\text{id}} \notin P$ then
6:                     $P[e_{\text{id}}] = \text{ghost\_file}(e)$
7:                 end if
8:             end for
9:         end for
10:    end for
11: end for
```

The basic idea is the same, the only difference being that there can be many `$INDEX_ALLOCATION` attributes, each containing several entries. While the inference of *ghost* files from the `$INDEX_ROOT` attribute alone could provide good results for directories with a few children,



(a) Without ghost files



(b) With ghost files

Figure 5.3: Improvement given by index records

the strategy is ineffective for big directories. It is extremely important to consider non-resident nodes as they contain the highest number of entries related to child nodes, thus providing much more information about *ghost* children. This is another reason motivating the search for partition geometry information, discussed in Section 3.4.

Figure 5.3 illustrates a simple example showing the difference that index records can make, especially with respect to *ghost* directories. In Figure 5.3a, no information is known about the directory with id 14. Hence, the reconstruction algorithm stores its children inside `LostFiles/Dir_14`. However, the index of directory `Documents` (id 11) contains the name of directory `Pictures` (id 14) and this information can be used, as shown in Figure 5.3b.

The result is that the whole subtree rooted at node 14 is MOVED TO A BETTER POSITION. In this particular case, this means transferring the directory from *Lost Files* to its correct place inside the tree.

6 INFERENCE OF PARTITION GEOMETRY

In Section 3.4, we argued that information about the partition geometry is crucial for file systems based on clusters. The lack of SPC and CB values makes the contents of a NTFS partition unrecoverable. This information is contained only in the boot sector, hence it is important to develop an algorithm for inferring the partition geometry if it cannot be read from the disk.

In Section 6.1 we briefly cover the simple case, i.e. when the boot sector can be read. After that, we discuss an approach that can be used when the aforementioned values are not available. In Section 6.2 we describe how to fingerprint index records for using their identifiers in an enumeration process, described in Section 6.3.

In Section 6.4 we introduce the approximate string matching problem. In Section 6.5 we adapt it to patterns generated from a set of MFT entries and related index records, improving the Baeza-Yates–Perleberg algorithm [1]. Finally, in Section 6.6 we introduce an algorithm for detecting the partition geometry, even if the file system is partially corrupted.

6.1 GEOMETRY INFORMATION IN BOOT RECORDS

During the metadata carving process described in Section 5.1, positions of sectors probably matching NTFS boot records (or backups) are stored in a list. If a sector is valid, it provides the correct values of SPC and the position of the MFT (see Table 3.1). The CB either coincides with the location of the boot record or it is given by subtracting the number of sectors to it. Since files were clustered by their MFT position in Section 5.2, each record can be easily associated to its partition.

Algorithm 6.1 shows the procedure for associating boot sectors to partitions, therefore determining their geometry. From each record b, the relative position r of the MFT is computed in sectors. After this step, two possible cases are considered. Either the sector contains a boot record ($\delta = 0$) or it contains a backup ($\delta = b_{\text{sectors}}$). Sector i represents the CB after taking into account the value of δ. Address $a = r + i$ is the position of the MFT according to the boot sector.

If the value of a matches a partition p obtained from clustering file records, then its attributes are updated. It is marked as *recoverable* and the values of partition size, SPC and CB are set accordingly to the information provided in the boot record.

The procedure is repeated for all boot records. After this process, all partitions with a matching boot sector are marked as recoverable,

Algorithm 6.1 Partition geometry from NTFS boot records

```
1:  for all records $b \in$ found_boot do
2:      $r = b_{\text{MFT\_position}} \cdot b_{\text{SPC}}$
3:      for all $\delta \in [0, b_{\text{sectors}}]$ do
4:          $i = b_{\text{position}} - \delta$
5:          $a = r + i$
6:          if $a \in$ partitions then
7:              $p = \text{partitions}[a]$
8:              $p_{\text{recoverable}} =$ true
9:              $p_{\text{size}} = b_{\text{sectors}}$
10:             $p_{\text{SPC}} = b_{\text{SPC}}$
11:             $p_{\text{CB}} = i$
12:             break
13:         end if
14:     end for
15: end for
```

while the other ones require more advanced techniques for finding their geometry. In the rest of this chapter we introduce an algorithm that works despite some degree of metadata corruption.

6.2 FINGERPRINTING INDEX RECORDS

Before we discuss our partition geometry inference algorithm, we need to determine a piece of information about index records that is not explicitly stored inside them. As briefly argued in Section 3.3, an index record does not contain the identifier of the directory it belongs to, but this can be guessed from its `$FILE_NAME` attributes.

The list of parent directory identifiers of entries inside an index record may contain different values. This is because NTFS supports HARD LINKS, i.e. multiple directories can have a reference to the same child [3]. However, a file has only one parent directory, hence the attribute always contains its identifier.

Example (Different Parent Nodes). We consider a directory called `misc` with id 27. It contains 6 files, the second one being a hard link to a file actually contained in a directory with id 16. Thus, if we read the attributes in the index record of directory `misc`, we obtain the following list of parent nodes:

$$\text{parents} = [27, 16, 27, 27, 27, 27]$$

The algorithm introduced in the following sections require to know what MFT entry owns a index record. In most cases the values are equal, but we must take this kind of situation into account. Algorithm 6.2 shows the simple procedure required to fingerprint every index record. This is done by assigning to r_{owner} the MODE OF THE PARENT NODES LIST made from `$FILE_NAME` attributes.

Algorithm 6.2 Owner of index records

```
1: for all records $r \in$ found_index do
2:     parents = [ ]
3:     for all entries $e \in r$ do
4:         parents.append($e_{\text{parent}}$)
5:     end for
6:     $r_{\text{owner}}$ = mode(parents)
7: end for
```

In the occasional case of many hard links in a single directory, this approach might fail. However, to infer the partition geometry we only need to have a good amount of correctly fingerprinted index records. If a few MFT entries are not guessed correctly the outcome does not compromise partition geometry detection.

6.3 SECTORS PER CLUSTER VALUE ENUMERATION

Partition geometry requires to determine two parameters (SPC and CB), hence we can solve our problem by using a *model fitting* procedure. The data points used to find correct parameter values are constructed from MFT entries and index records fingerprinted with the procedure shown in Section 6.2.

In principle, fitting two parameters can lead to a very large search space, because both of them may take arbitrary values. The CB might assume any value from 0 to the sector where the MFT is located. However, in NTFS the search space can be reduced because the SPC parameter can only take values in a range of small powers of 2, i.e. 1, 2, 4, 8, 16, 32, 64 or 128 [3].

Therefore, it is possible to detect the partition geometry starting from ENUMERATING THE VALUE of SPC. For each of the few admissible values, we can attempt to find a CB value that makes MFT entries MATCH THE POSITIONS OF INDEX RECORDS in the best way. The matching process takes the most time.

Algorithm 6.3 CB from SPC enumeration

```
Require: set $E$ of MFT entries with $INDEX_ALLOCATION attributes
Require: set $R$ of fingerprinted index records on disk
1: $s = \{\}$
2: text = make_list_from_records($R$)
3: for all $i \in [0 \dots 7]$ do
4:     SPC = $2^i$
5:     pattern = make_list_from_entries($E$, SPC)
6:     (CB, score) = match(pattern, text)
7:     $s$[SPC] = (CB, score)
8: end for
9: return best_solution($s$)
```

Algorithm 6.3 shows the high level pseudocode of the procedure for determining the CB and SPC values of a partition, using index records. This is still a generic approach, because we need to discuss many details. First of all, the requirements are the set E of file records of directories (with `$INDEX_ALLOCATION` attributes) and the set R of fingerprinted index records.

The MFT entries are already clustered, therefore they belong to the partition of interest. Conversely, index records include all those found on disk that can be fingerprinted, because it is not yet possible to deduce which partition they belong to. Both sets are used to generate lists, which might be considered *the same as long strings* used for the matching algorithm.

In particular, the set of index records constitutes the TEXT, while `$INDEX_ALLOCATION` attributes are combined with a SPC value to generate a PATTERN. The text is a list whose size is limited by the number of sectors on disk. Its entries are empty except in the positions where index records $r \in R$ are found. These places contain the value r_{owner}.

Each pattern is the union of all `$INDEX_ALLOCATION` attributes in every MFT entry $e \in E$. Each runlist determines relative positions of index records related to directory e_{id} and the SPC value is used to convert them in sectors.

Example (Text and Pattern Generation). We consider a disk where the following index records have been fingerprinted:

SECTOR	OWNER ID
54	14
62	23
78	14

The resulting *text* list is:

...	∅	∅	14	∅	∅	∅	∅	∅	∅	∅	23	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	14
			54								62																78

The upper row shows the array content, while the numbers on the bottom denote the indexes corresponding to sectors.

Now, let us assume that the following MFT entries of directories with `$INDEX_ALLOCATION` attributes belong to the partition whose geometry is unknown. The values in each runlist are expressed in clusters and they start from 0.

MFT ENTRY	RUNLIST
14	11, 17
23	13

If we take SPC $= 1$, we get pattern:

$$\begin{array}{cccccccccccccccccc}
\hline
\multicolumn{1}{|c}{\varnothing} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & 14 & \varnothing & 23 & \varnothing & \varnothing & \varnothing & \multicolumn{1}{c|}{14} \\
\hline
0 & & & & & & & & & & & 11 & & 13 & & & & 17
\end{array}$$

However, if the value is SPC $= 2$, then the pattern becomes:

$$\begin{array}{cccccccccccccccc}
\hline
\dots & \varnothing & \varnothing & 14 & \varnothing & \varnothing & \varnothing & 23 & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \multicolumn{1}{c|}{14} \\
\hline
 & & & 22 & & & & 26 & & & & & & & & 34
\end{array}$$

In this toy example, the correct value is actually SPC $= 4$. This results in the runlists pointing to sectors 44, 52 and 68, starting from the CB. By looking at the *text*, we can see that the pattern matches it perfectly when CB $= 10$.

Patterns are matched against the text by a *match* function on line 6 in Algorithm 6.3. This is the most important step, because the matches are stored in s and they determine the best solution.

We do not expect to find a perfect match. Rather, the *match* function returns the most coherent result for a pattern (with fixed SPC value), i.e. the one with the highest *score*. In Section 6.4 we discuss how to use an approximate string matching algorithm and how to compute the quality of a match.

6.4 APPROXIMATE STRING MATCHING

The patterns and the text constructed from MFT entries and index records CAN BE REGARDED AS STRINGS, where the alphabet is a subset of all natural numbers. Our goal is to match a pattern against the text. However, we must consider situations where metadata is corrupted. Hence, we need to allow partial matches, giving a score to each match in order to chose the best one.

This situation corresponds to the APPROXIMATE STRING MATCHING PROBLEM, which is extensively described in the literature, with several proposed solutions [1, 7, 12].

Problem (Approximate String Matching). Given:

- a pattern $P = p_1p_2p_3 \dots p_m$
- a text $T = t_1t_2t_3 \dots t_n$
- a parameter k

Find all instances P_i of P in T, such that there are at most k mismatches between P_i and P. The amount of mismatches is the Hamming distance $\delta_H(P_i, P)$, i.e. the number of characters that are not the same [1].

Among all possible solutions, the algorithm proposed by Baeza-Yates and Perleberg [1] has a very interesting property. It runs in $O(n)$ *worst case time* if all characters p_i in the pattern are distinct. If this does not hold, the running time is $O(n + R)$ where "R is the total number of ordered pairs of positions at which P and T match." [1]

Therefore, the Baeza-Yates–Perleberg algorithm is optimized for a specific kind of pattern. We consider this a *good property* because the patterns presented in Section 6.3 show a very small amount of repeating characters. In particular, each element of the alphabet is a file record identifier. The number of repetitions is limited by the amount of entries in the runlist, pointing at different index records. This is very small in many practical cases.

The space requirement is a negative aspect of this algorithm. In fact, the authors store some information for every character in an array, leading to a space occupation of $O(2m + |\Sigma|)$ where Σ is the alphabet. In our case, $|\Sigma|$ is only bounded by the amount of different MFT entries in the partition. However, the process can be tweaked to store only information about $O(m + 1)$ characters, leading to an optimized space occupation.

The algorithm is based on a PREPROCESSED PATTERN. The array *alpha* contains one list for every character in Σ and each list stores the offsets of a certain character in the pattern. When the preprocessing is complete, the search function loops over the text, reading every letter t_i. Each offset $o_j \in \text{alpha}[t_i]$ is used to increment the counter in position $i - j$ of the array [1].

Example (Preprocessing and Search). Let us consider the bogus text "thinner than that" with alphabet $\Sigma = \{␣, a, e, i, h, n, r, t\}$ and the pattern "that". The preprocessing results in the following *alpha*:

CHARACTER	OFFSETS
a	2
h	1
t	0, 3
␣, e, i, n, r	∅

The search process progressively increments the array of counters, resulting in the following outcome:

t	h	i	n	n	e	r	␣	t	h	a	n	␣	t	h	a	t	Text
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Position
2	0	0	0	0	1	0	0	3	0	1	0	0	4	0	0	0	Counters

The final set of matches depends on the value of k.

Counters can actually be implemented with an array of size m accessed in a circular manner, because offsets cannot be greater than m.

The code presented by Baeza-Yates and Perleberg [1] does not strictly adhere to the simple algorithm introduced so far. They include several optimizations, most of which are related to the features of the C programming language.

Algorithm 6.4 Baeza-Yates–Perleberg *preprocess* function [1]

```
Require: pattern $P$ of length $m$
Ensure: initialized arrays count, alpha
 1: for all characters $c_i \in \Sigma$ do
 2:     alpha$[c_i] = [\,]$
 3: end for
 4: for all indexes $i \in [0 \ldots m-2]$ do // initial values
 5:     count$[i] = \infty$ // invalid positions $t_j \mid j < 0$
 6: end for
 7: count$[m-1] = m$ // corresponding to $t_0$
 8: for all characters $p_i \in P$ do
 9:     alpha$[p_i]$.append$(m-i-1)$
10: end for
```

Algorithm 6.4 shows the pseudocode for the *preprocess* function. The main features of this implementation are reversed counters and the fact that errors are counted, rather than correct matches. For this reason, the last counter is initialized as count$[m-1] = m$.

Counter i refers to character $t_{i-(r+1)m+1}$, where r is the number of times the circular array has been rotated. Thus, we set count$[i] = \infty$ for $r = 0$ and $i < m-1$. The function runs in $O(2m + |\Sigma|)$ time [1]. After the preprocessing step, arrays *count* and *alpha* are correctly initialized and the actual pattern matching can begin.

Algorithm 6.5 Baeza-Yates–Perleberg *search* function [1]

```
Require: initialized arrays count, alpha
Require: text $T$ of length $n$
Require: length of pattern $m$
Require: maximum allowed distance $k$
 1: matches $= [\,]$
 2: for all characters $t_i \in T$ do
 3:     offsets $=$ alpha$[t_i]$
 4:     for all $o \in$ offsets do
 5:         decrement count$[i + o \mod m]$ by 1
 6:     end for
 7:     if count$[i \mod m] \le k$ then // new match $r$ found
 8:         $r_{\text{position}} = i - m + 1$
 9:         $r_{\text{mismatches}} =$ count$[i \mod m]$
10:         matches.append$(r)$
11:     end if
12:     count$[i \mod m] = m$ // reset counter
13: end for
14: return matches
```

Algorithm 6.5 shows the pseudocode for the *search* function. The version we present here has been stripped of all low level optimizations related to implementation details, nevertheless the algorithm is unchanged. The only significant difference is in the output. The function shown by Baeza-Yates and Perleberg [1] prints results but it does not save them. Instead, we store the output in a data structure that might occupy up to $O(n)$ space.

The outer loop on line 2 is executed $O(n)$ times, while the inner loop on line 4 is bounded by the length of the longest list in *alpha*. It can be removed if the pattern contains only distinct characters [1].

The best case scenario is that of a pattern without repetitions, leading to $O(n)$ execution time. Conversely, the worst case scenario happens when the pattern is made of a single character repeated m times. In this case the running time would be $O(mn)$, but this situation is neither common nor useful [1].

6.5 IMPROVING THE BAEZA-YATES–PERLEBERG ALGORITHM

In Section 6.3 we have seen that the text and patterns contain many empty places, denoted by $\varnothing$. This results from the combination of two factors: records occupy multiple sectors and the SPC value can be greater than 1. However, the Baeza-Yates–Perleberg algorithm deals with contiguous strings, finding all matches with $\delta_H(P_i, P) < k$.

We want to find only matches with the lowest distance. We also need to avoid performing useless iterations on the text where $t_i = \varnothing$. Empty places occur very often, therefore they also cause a significant waste of space. We can solve the first problem by changing the data structure used for strings, introducing SPARSE LISTS.

Definition (Sparse List). A data structure S similar to an associative array, with a fixed *default* value d. Only values $S[k] \neq d$ are stored in the list. It can be accessed by referring to any key $k \in \mathbb{N} \mid k \geq 0$, with the following property:

$$S[k] = \begin{cases} S.\text{get}(k) & k \in S \\ d & k \notin S \end{cases}$$

The assignment operator has the following semantics:

$$S[k] = v \equiv \begin{cases} S.\text{set}(k, v) & v \neq d \\ S.\text{unset}(k) & v = d \end{cases}$$

The *set* method represents storing the mapping $k \to v$ in S, while the *unset* method removes any value v associated to key k. The latter frees the memory occupied by v and k. The length of S is defined as $\max\{k \in S\} + 1$, or 0 if S is empty.

Algorithm 6.6 shows our optimized *preprocess* function. Array *alpha* is now a sparse list, hence its space requirement is reduced to $O(m+1)$, regardless of the cardinality of the alphabet Σ. Moreover, the initialization of *count* is moved to the optimized *search* function, which is displayed in Algorithm 6.7.

The first important difference between our version and the original Baeza-Yates–Perleberg algorithm is that T and P are now sparse lists, with $d = \varnothing$. The counters are still stored in *count* in a circular fashion, however the variable is a sparse list with $d = 0$. This choice heavily reduces memory utilization.

Our *search* function accepts a new parameter l, to limit the positions in which patterns may occur. A partition cannot start after its first MFT entry and that is usually located in the first part of the disk. Hence, introducing l reduces the time spent on pattern matching by stopping the process before it becomes useless.

Additionally, the matching algorithm becomes faster because positions in T with default value $\varnothing$ are skipped, therefore the outer loop performs $O\left(\min\{n, l\} / \text{SPC}\right)$ iterations. Skipping indexes in the sparse list means that counters might not be initialized correctly when the circular array is rotated. For this reason, on line 11 we wipe all counters in interval $[j, i[$. Method *unset_interval* works in a circular way. If $j > i \mod m$, $S[k]$ is unset when $k \geq j$ or $k < i$.

Skipping has consequences on how matches are checked, as well. We cannot rely on $\text{count}[i]$, because many values of i do not appear in the loop. Instead, on line 17 we check for matches every time a counter is increased. For example, this means that a match with score 4 is matched with scores 2, 3 and 4 if $k_{\min} = 2$. This is not an issue, because we only want matches with the highest score k. As soon as this value increases, the previous matches are discarded.

Finally, we note that our function does not prevent matches to occur in positions $i < 0$ or $i > n - m$, i.e. patterns may be "hanging outside" of the boundaries of T. The image file we are working on might be a partial bitstream copy of a bigger hard drive, hence allowing this situation is acceptable.

6.6 PARTITION GEOMETRY DETECTION

The basic approach shown in Algorithm 6.3 can be improved by using our optimized approximate string matching algorithm, combined with the MFT position to stop the search as soon as possible. Moreover, it is useful to define a minimum score value, i.e. $k_{\min} = 2$. It does not make sense to take $k_{\min} = 1$, because this generates many matches regardless of the correctness of the SPC value. Therefore, at least two "characters" need to match to get a useful outcome.

Algorithm 6.8 shows our strategy for inferring the partition geometry. Before the SPC value enumeration, we build a sparse list for the

Algorithm 6.6 Optimized *preprocess* function

```
Require: pattern $P$ of length $m$ (sparse list)
Ensure: initialized sparse list alpha
1: alpha $=$ SparseList($d = [\,]$)
2: for all characters $p_i \in P$ do
3:     alpha[$p_i$].append($m - i - 1$)
4: end for
```

Algorithm 6.7 Optimized *search* function

```
Require: text $T$ of length $n$ (sparse list)
Require: pattern $P$ of length $m$ (sparse list)
Require: upper limit $l$ for matches
Require: minimum allowed score $k_{\min}$
1: $k = k_{\min}$
2: alpha $=$ preprocess($P$)
3: count $=$ SparseList($d = 0$)
4: matches $= [\,]$
5: $i, j = 0$
6: for all keys $i \in T$ do
7:     if $i > l + m$ then
8:         break
9:     end if
10:    // reset counters from $j$ to $i$ (excluded)
11:    count.unset_interval($j \mod m, i \mod m$)
12:    $j = i$
13:    offsets $=$ alpha[$t_i$]
14:    for all $o \in$ offsets do
15:        increment count[$i + o \mod m$] by 1
16:        $s =$ count[$i + o \mod m$]
17:        if $s = k$ then // new match with score $k$
18:            matches.append($s$)
19:        end if
20:        if $s > k$ then // better match
21:            $k = s$
22:            matches $= [k]$
23:        end if
24:    end for
25: end for
26: return matches, $k$
```

Algorithm 6.8 Inference of partition geometry

```
Require: set $E$ of MFT entries with $INDEX_ALLOCATION attributes
Require: set $R$ of fingerprinted index records on disk
Require: MFT address $A$
 1: $s = \{ \}$
 2: text = SparseList($r_{\text{sector}} \rightarrow r_{\text{owner}} \quad \forall r \in R$)
 3: // $r$ is a file record, $a$ is a $INDEX_ALLOCATION attribute
 4: base = $\{e_{\text{cluster}} \rightarrow r_{\text{id}} \quad \forall e \in a_{\text{runlist}} \quad \forall a \in r \quad \forall r \in E\}$
 5: $k_{\text{min}} = 2$
 6: for all $i \in [0 \ldots 7]$ do
 7:     SPC $= 2^i$
 8:     pattern = SparseList(SPC $\cdot k \rightarrow$ base[$k$] $\quad \forall k \in$ base)
 9:     ($P$, score) = approximate_matching(pattern, text, $A$, $k_{\text{min}}$)
10:     if $|P| = 1$ then
11:         CB $= P[0]$
12:         $s$[SPC] = (CB, score)
13:     end if
14: end for
15: return best_solution($s$)
```

text and another one called *base*. The latter is made by combining every runlist of $INDEX_ALLOCATION attributes in all MFT entries. Its keys are expressed in clusters.

Each pattern is a "translation" of *base* from clusters to sectors, then the matching is performed by specifying the MFT address A as upper limit. If the search is unsuccessful, or the disk is heavily damaged, the set of matching positions P may contain more than one element. We do not accept this case, since we expect a good match to be unique.

If $|P| = 1$, we save the CB value and the score in s. The final step is to select the pair of SPC and CB values with the highest score among all results in s. If the procedure is successful, the geometry is determined and the contents of the partition can be accessed.

Part III

SOFTWARE IMPLEMENTATION

7 RECUPERABIT

We have implemented the forensic file system reconstruction algorithm presented in Part ii, creating a software called RECUPERABIT. The name is a Latin form meaning "he will restore", but it can also be read as "bit restorer" in Italian. RecuperaBit is released as Free Libre Open Source Software (FLOSS) and it can be downloaded from:

`https://github.com/Lazza/RecuperaBit`

In this chapter we describe how it is structured. Section 7.1 introduces the architecture of our software and Section 7.2 is dedicated to the implementation of the abstract file system used by RecuperaBit.

In Section 7.3 we discuss the *unpacker* function, which parses binary data in a way resembling *structs* in the C programming language. In Section 7.4 we describe the NTFS module and in Section 7.5 we explain how file recovery is performed.

7.1 ARCHITECTURE OVERVIEW

RecuperaBit is written in Python. Although several chapters of this thesis are dedicated to NTFS, the idea behind its architecture is that the software should be modular and extensible to other file systems. The program is made of different components, namely:

CORE LOGIC The code for approximate string matching is contained in `logic.py`. This module implements Algorithms 6.6 and 6.7, together with the *SparseList* class implementing the data structure defined in Section 6.5.

UTILITY FUNCTIONS The module in `utils.py` is a collection of functions and procedures used by other components. They include data conversion functions, string processing routines and several procedures for showing the output of RecuperaBit in different textual forms. The most complicated function is *unpacker*, which is described in Section 7.3.

FILE SYSTEM MODULES RecuperaBit makes use of one or more modules stored in the `fs` directory. Each module must implement the interfaces listed in `core_types.py`, i.e. *File* and *DiskScanner*. The *Partition* class is also included, but overriding it is not required, because its main purpose is to implement the bottom-up tree reconstruction of Algorithm 5.3. These classes are described in Section 7.2. The NTFS module is contained in `ntfs.py` and `ntfs_fmt.py`, the latter being a collection of functions and data structures describing MFT entries and index records.

MAIN MODULE This can be found in `main.py`. It provides two essential features, i.e. DISK SCANNING and USER INTERACTION. The main function *feeds* one scanner for every supported file system type, supplying each sector of the disk. A scanner might decide that the sector is not interesting, or it might keep track of it for later usage. After this phase, the main function provides a Command Line Interface (CLI) for visualizing partitions found by scanners. Additional commands can be issued to save results in output files or to restore the contents of partitions.

The modular architecture of RecuperaBit makes its extension very easy. Supporting a new file system requires to override two classes (*File* and *DiskScanner*) and a slight variation in the main module.

Example (Adding FAT Support). At present, `main.py` contains the following lines to support NTFS:

```python
from recuperabit.fs.ntfs import NTFSScanner

# classes of available scanners
plugins = (
    NTFSScanner,
)
```

Let us assume we want to add support for FAT. The first thing to do is implementing a derivative class of *DiskScanner*, which we may call *FATScanner*. This includes method *get_partitions*, returning one or more sets of *FATFile* objects. After that, the main module needs to be made aware of the new scanner, with the following additions:

```python
from recuperabit.fs.ntfs import NTFSScanner
from recuperabit.fs.fat import FATScanner

# classes of available scanners
plugins = (
    NTFSScanner,
    FATScanner
)
```

The scanner for FAT needs to implement a proper way to detect file records and cluster them into partitions. The best way to do this is file system dependent.

However, the bottom-up file system reconstruction algorithm of RecuperaBit works for any derivative of the *File* class. Hence, implementing *FATFile* is necessary only if we want to actually restore the contents of files in a FAT file system.

7.2 ABSTRACT FILE SYSTEM

RecuperaBit expects all modules to work on its ABSTRACT FILE SYSTEM objects. For all supported file systems, a *DiskScanner* class needs to be implemented. This is fed with sectors of the disk, so it can keep track of interesting artifacts. After the scanning process is over, it must provide a set of recognized *Partition* instances, each containing a list of *File* objects (or derivative classes).

DiskScanner (Interface)

The *DiskScanner* interface is pretty much empty, including only two methods and a field pointing to the image file currently scanned. The first method is *feed*, i.e. the action performed by the scanner when a new sector is encountered. The second one is *get_partitions*, which returns the set of detected partitions. Both methods are abstract and must be defined by each scanner.

Partition (Class)

This class can be used for any file system type, but it might also be extended to store additional information. It contains these fields:

- *fs_type*, string representing the type
- *root_id*, identifier of the root directory
- *offset*, its Cluster Base (CB) expressed in sectors
- *root*, reference to the *File* instance of the *Root* directory
- *lost*, reference to the *File* instance of the *Lost Files* directory
- *files*, map of its files referenced by identifier
- *recoverable*, boolean stating if contents can be restored
- *scanner*, reference to its *DiskScanner* instance

This class is mostly composed of getters and setters for the properties listed above. However, the most important method is *rebuild*, i.e. the implementation of Algorithm 5.3.

File (Abstract Class)

The *File* class is almost complete, however it lacks the crucial method *get_content*, used to restore a file. Hence, it may be viewed both as an interface and as an abstract class. This is why we need a derivative class of *File* if we are interested in the recovery of file contents.

This class adheres to the requirements listed in Section 2.3 and it contains the following fields:

- *index*, identifier of the file
- *name*, real or a placeholder (e.g. `Dir_423`)
- *size*, expressed in bytes
- *is_directory*, *is_deleted*, *is_ghost*, boolean flags
- *parent*, identifier of the parent
- *mac*, map of Modification, Access and Creation (MAC) times
- *children*, set of references to the child nodes
- *children_names*, set used to avoid name clashes
- *offset*, sector where the file record can be found

The *add_child* method does not simply insert a new element in *children*, because this would cause issues if there are two or more nodes with the same name. When it comes to file recovery, it is not possible to create two files with the same name in the same directory. For this reason, the name might be changed by adding a suffix such as `_002`. This situation may be caused by deleted file records.

The other two interesting methods are *full_path* and *ignore*. The former provides the path of the file with respect to the *Root* or *Lost Files* directory. The latter is used to avoid potentially unwanted files (e.g. `$BadClus:$Bad` in NTFS) when performing a recursive recovery.

7.3 STRUCT UNPACKER

The process of reading the contents of a file system requires to parse several data structures. As we have seen in Chapter 3, for NTFS the program needs to interpret boot records, MFT entries and attributes, runlists and index records. Each structure is stored in a specific format and is restricted to one type of file system.

RecuperaBit includes the *unpack* function in `utils.py`, to avoid writing a parser for many kinds of common data structures which are similar to *structs* in C. Examples of supported formats include the MFT entry structure shown in Table 3.2 or the index record structure displayed in Table 3.4.

The function takes two parameters: the DATA (raw bytes, e.g. a sector from the disk) and a FORMAT, made of several entries. Each entry has the following fields:

NAME The output of *unpacker* is a Python dictionary that is accessed through keys. This string represents the key under which the parsed value from this entry is stored in the dictionary.

FORMATTER This value can be either a string or a function. If it is a string, the bytes are interpreted in a variety of ways:

- s means that the bytes are simply casted to a string without further processing.
- `utf-8` or `utf-16` tell the function to decode the bytes as a Unicode string represented with the respective format.
- `i` is the default type of number, i.e. the bytes are read as an integer in little-endian format.
- `>i` corresponds to an integer in big-endian format.
- `+i` and `>+i` are similar to the two previous formatters, however the first bit (after accounting for the endianness) is sign-extended to treat the bytes as a signed number.

If the formatter is a function provided by the caller, the bytes are passed to the function as the only argument and the returned value is saved in the dictionary.

FIRST BYTE This can be either an integer or a function. If it is a number, it is used directly as an index for the first byte in the data that needs to be considered for this entry. If it is a function, it must be one that takes as only parameter the record r, i.e. the partial dictionary *obtained by parsing the previous entries*.

LAST BYTE The format is the same as the one for the first byte.

Each file system module in RecuperaBit can use this function to parse a group of bytes, providing a suitable format. More advanced types of parsing can be performed by a dedicated parser, or including a sophisticated function as *formatter* for one entry.

Example (MFT Attribute Header). RecuperaBit contains the following format for MFT attribute headers, as shown in Table 3.3a:

```
attr_header_fmt = [
    ('type', ('i', 0, 3)),
    ('length', ('i', 4, 7)),
    ('non_resident', ('i', 8, 8)),
    ('name_length', ('i', 9, 9)),
    ('name_off', ('i', 10, 11)),
    ('flags', ('i', 12, 13)),
    ('id', ('i', 14, 15)),
    ('name', (
        printable_name,
        lambda r: r['name_off'],
        lambda r: r['name_off'] + r['name_length']*2 - 1
    ))
]
```

All fields except *name* are integers in little-endian format, located at fixed positions. The *name* field, instead, is formatted by the *printable_name* function which strips invalid UTF-16 characters. Each one requires 2 bytes, therefore we need to format those in interval $[r_{\text{name\_off}} \ldots r_{\text{name\_off}} + 2r_{\text{name\_length}} - 1]$.

7.4 NTFS SCANNER

The *NTFSScanner* class implements the *DiskScanner* interface for NTFS. Its *feed* method is a straightforward implementation of Algorithm 5.1. Instead, the *get_partitions* method is more involved as it combines various algorithms presented in Chapters 5 and 6.

The first part of the method iterates over all MFT entries having a `$FILE_NAME` attribute. The entries are parsed and clustered using their identifiers, as shown in Algorithm 5.2. One *NTFSFile* instance is created for each ADS, conforming to Algorithm 5.4. Additional *ghost* elements are searched inside `$INDEX_ROOT` attributes with the procedure of Algorithm 5.5. The parsed entries that include `$ATTRIBUTE_LIST` or `$INDEX_ALLOCATION` attributes are stored for later usage.

The second part of the method regards index records. Each one is fingerprinted to find its owner as shown in Algorithm 6.2 and a *text* is built with the procedure discussed in Section 6.3.

The last part deals with partition geometry. Each available boot record is used to associate CB and SPC values to the respective partition using Algorithm 6.1. The geometry of those without a boot sector is inferred through our improved approximate string matching strategy, shown in Algorithm 6.8. The MFT entries in partitions with known geometry are extended using their `$ATTRIBUTE_LIST` attribute. Finally, additional *ghost* files are derived from `$INDEX_ALLOCATION` attributes with Algorithm 5.6.

Before returning the list of partitions, the NTFS module exploits the information available in MFT backup copies to improve the results. If the first entries of a MFT are not available, they are extracted from the backup. After this step, the `$DATA` attribute of the `$MFT` file is used to check if the MFT is fragmented. If this is the case, multiple partition pieces generated from simple file record clustering are merged.

7.5 RECURSIVE FILE RECOVERY

This thesis is about forensic file system reconstruction, focusing on metadata rather than file contents. Nevertheless, the *analysis phase* of a digital forensics investigation requires to inspect file contents, including deleted elements [2]. Therefore, a forensic tool should provide functionality to recover file contents.

The *File* class contains abstract method *get_content*, which must be implemented by derived classes. In the case of *NTFSFile*, the method works only if the `$DATA` attribute is resident or the partition geometry is known. Currently, files using NTFS compression are not supported and encrypted files are recovered but not decrypted.

RecuperaBit contains the *recursive_restore* procedure in `logic.py`, that can be used to recover both single files and directories together with all their children recursively. The files are saved in the output directory provided by the user. By default, all elements are recovered, including deleted files and *ghost* files (with empty content).

If the content is very small, we expect that *get_content* provides a string of bytes, otherwise it must return an iterator of strings. When a file is very big, it is not possible to load it fully in RAM before restoring it. Therefore, the method shall provide an iterator of chunks limited in size by the *max_sectors* value stored in `constants.py`. The procedure loads and writes one chunk at the time.

When RecuperaBit restores a directory, the children are tested using their *ignore* method, skipping nodes that return a positive value. If we want to force recovery for an ignored file, we must call the procedure directly on it. For NTFS, the ignored elements are `$BadClus:$Bad` and `$UsnJrnl:$J`, because they are mostly not very useful but they require a significant amount of space.

8 EXPERIMENTS

The procedures used by software to produce digital evidence should be tested and validated [2]. In this chapter, we present several experiments conducted on different test cases, comparing RecuperaBit with numerous commercial and open source programs.

In Section 8.1, we describe the software used for comparison and the disk images chosen as test cases. In Sections 8.2, 8.3 and 8.4 we present results for some experiments related to file system detection, reconstruction and recovery. In Section 8.5 we compare the reporting ability of different tools in terms of output formats. In Section 8.6 we analyze how the corruption level of a disk impacts the quality of results produced by RecuperaBit.

8.1 TESTED SOFTWARE AND DISK IMAGES

If a forensic program is released under a FLOSS license, it improves the quality of its testing process, because a code review can uncover bugs and flaws [2]. However, proprietary analysis software has existed for a much longer time [2] and there are many commercial forensic and data recovery programs.

For this reason, we compare RecuperaBit with both open source and proprietary programs. The main purpose of RecuperaBit is the detection of file systems on damaged hard drives and their forensic reconstruction despite corrupted metadata. Therefore, our focus is on programs searching for lost partitions and data recovery software.

The following is a list of the software products we consider. Their features are summarized directly from what is advertised by the producers. The open source programs are:

- Gpart[1], a tool aiming to guess partitions on a MBR disk with a damaged partition table. It includes various file system guessing modules, but it does not perform data recovery.

- TestDisk[2], a data recovery software designed to find lost partitions and restore data. Its purpose is to repair damage caused by software errors. It can recover partitions and backup boot sectors, fix FAT tables and partially restore a MFT from its backup. It can also undelete files from several file systems.

1 Gpart version 0.2.1, by Michail Brzitwa. URL: `https://github.com/baruch/gpart`
2 TestDisk version 7.0, by Christophe Grenier.
URL: `http://www.cgsecurity.org/wiki/TestDisk`

- Autopsy[3], a GUI to The Sleuth Kit and other digital forensics tools. It includes a wizard for creating cases and importing disk drives. It performs an analysis of various file systems, allowing to extract files, keywords and web artifacts. It can generate different kinds of reports.

- Scrounge-NTFS[4], a data recovery program for this kind of file system. It is designed to extract files from damaged partitions, but it requires the user to provide the SPC, start sector (i.e. CB), end sector and MFT offset values. If the latter is not provided, the files are restored in a single directory.

The commercial programs are:

- Restorer Ultimate[5], an application for the advanced recovery of various file system types. It provides a wizard for assisting in the analysis phase and it features a "smart scan" technology to restore files from corrupted hard drives.

- DMDE[6], a software for data searching, editing and recovery. It supports several file systems and it provides a partition editor. It applies thorough search and analysis algorithms in case of complex damage, to reconstruct the directory structure.

- Recover It All Now[7], an intuitive data recovery utility. It is designed for accessing data lost through file system corruption or slight hardware damage. It includes a "forensic scan" feature to assess which files contain bad sectors.

- GetDataBack[8], a product marketed as proven by millions of successful recoveries. It features a wizard in three steps and it is capable of restoring file names and the directory structure.

- SalvageRecovery[9], a partition recovery program dedicated to NTFS and FAT. It uses a library of algorithms to work with different scenarios, e.g. hardware malfunctioning and system failure. It can recover data in case of damaged boot records, formatted partitions, or corrupted file systems.

3 Autopsy version 4.0.0, by Brian Carrier and contributors.
URL: `http://www.sleuthkit.org/autopsy/`

4 Scrounge-NTFS version 0.9, by Stef Walter.
URL: `http://thewalter.net/stef/software/scrounge/`

5 Restorer Ultimate Pro version 7.8.708689 (licensed), by BitMart Inc.
URL: `http://www.restorer-ultimate.com/`

6 DMDE version 3.0.4 (free edition), by Dmitry Sidorov. URL: `http://dmde.com/`

7 Recover It All Now version 2.0 (trial), by DTI Data Recovery.
URL: `http://dtidatarecovery.com/product/recover-now/`

8 GetDataBack for NTFS version 4.3.3 (trial), by Runtime Software.
URL: `http://www.runtime.org/data-recovery-products.htm`

9 SalvageRecovery for Windows version 2.5 (trial), by SalvageData Recovery Inc.
URL: `https://www.salvagedata.com/recovery-software/`

Each of these programs can analyze bitstream disk images. To allow for a better comparison with respect to RecuperaBit, we are interested only in NTFS support. Whenever possible, tools supporting more file system types or additional data carving approaches have been tested with these features turned off.

Any kind of advanced file system analysis technique available in the software has been switched on, to test the full potential of each program in the detection and reconstruction of NTFS partitions.

We consider various test cases with different levels of difficulty:

- *SimpleIntact* is a disk image containing a working partition table with one NTFS partition. Its file system is intact and readable by normal OS tools. It contains four JPEG images, but one has been deleted from the trash bin on a Linux system. The file system has a size of 15 MB. CB $= 2048$ sectors and SPC $= 8$.

- *BrokenTable* was used as an example in Chapter 5 and it is displayed in Figure 5.2. The boot sector is intact, however the MFT has been intentionally damaged by removing the file record of the root directory (id 5) and that of many other NTFS metadata files. There are two text files (one deleted) but they are empty. The file system has the same geometry of that of *SimpleIntact*.

- *HardToFind* is a scenario built especially for testing the detection of partition geometry. It is a 1 GB disk image with a NTFS file system inside it. The partition table has been completely overwritten with zeros, together with all the MFT entries related to metadata files (including the backup MFT).

  Both boot sectors have been wiped as well, hence no partition geometry information can be located on the drive. It contains several executables and shared objects from a Linux system, with some pictures and various text files in a few directories. They amount to a total of 517 files and directories, excluding the root. The file system has a size of 847 MB and its geometry values are CB $= 223232$ sectors (109 MB) and SPC $= 16$.

- *RealLife* is a bitstream copy of a 80 GB hard drive from a computer running Windows XP. The drive suffered from physical damage leading to some unreadable sectors or parts thereof. The file system has a size of 76.32 GB and it contains thousands of files and directories, including the OS and personal data of a few users. Most of the MFT is still readable, however it is split into two fragments. CB $= 63$ sectors and SPC $= 8$.

8.2 FILE SYSTEM DETECTION

The first step for reconstructing a file system is the detection of its presence, or at the very least reading information stored in the par-

SOFTWARE	#1	#2	#3	#4
Gpart	OK	OK	Nothing	Partial[1]
TestDisk	OK	OK	Nothing	OK (+1)
Autopsy	OK	Partial[2]	Nothing	OK
Scrounge-NTFS	OK[3]	OK[3]	Nothing	OK[3]
Restorer Ultimate	OK	OK	OK	OK
DMDE	OK	OK[3]	OK	OK (+3)
Recover It All Now	OK[4]	Nothing	Nothing	OK[4]
GetDataBack	OK[4]	OK[4]	Nothing	OK (+1)[5]
SalvageRecovery	OK[4]	OK[4]	Nothing	OK[4] (+1)
RecuperaBit	OK	OK	OK	OK $\times$ 2 (+302)[6]

1 Crashed before completing the output
2 File system type not recognized
3 From the Master Boot Record (MBR)
4 Same file system found twice due to the MFT mirror
5 One file system of slightly different size, but same contents are shown
6 Two overlapping recoverable partitions, traces of 302 possible partitions

Table 8.1: Test results for partition detection

tition table. In this section we provide a summary of the tests performed to check whether the tools effectively recognize NTFS artifacts contained in possibly corrupted bitstream images.

This is the only experiment where we can take Gpart into account, because its purpose is limited to partition detection. The results are listed in Table 8.1 and they are discussed below.

SimpleIntact (#1)

Gpart, TestDisk, Autopsy, Restorer Ultimate, DMDE and RecuperaBit correctly found one 15 MB wide NTFS partition, with a CB of 1 MB (2048 sectors). Recover It All Now, GetDataBack and SalvageRecovery detected two copies of the same file system, considering both the MFT and its backup, nevertheless the output is correct.

Scrounge-NTFS was able to detect the file system using the `-l` flag. However, this is performed by reading the partition table in the MBR, therefore the file system cannot be considered *detected*.

BrokenTable (#2)

Gpart, TestDisk, Restorer Ultimate, and RecuperaBit detected the file system correctly as in the case of *SimpleIntact* which has the same size and position. Autopsy detected the partition geometry correctly but returned “Error detecting file system type”.

GetDataBack and SalvageRecovery detected the file system twice, similarly to *SimpleIntact*. Scrounge-NTFS and DMDE recognized the information stored in the partition table. In particular, DMDE did not recognize any NTFS presence after a "full scan" of the bitstream file. Recover It All Now did not recognize the file system.

HardToFind (#3)

Finding the file system in this test case is intentionally hard. Nevertheless, Restorer Ultimate, DMDE and RecuperaBit inferred the partition geometry and the file system type correctly.

Gpart, TestDisk, Autopsy, Scrounge-NTFS, Recover It All Now, GetDataBack and SalvageRecovery did not detect the file system. Autopsy returned the error "Failed to add data source", preventing the user from working with the image file. GetDataBack provided many log messages related to MFT entries while scanning, but no partition was identified. Moreover, the program even suggested to install and run GetDataBack for FAT.

RealLife (#4)

This image file contains one *main* NTFS partition (the object of this test) and some traces related to previous NTFS boot records. For this reason, the results varied greatly among different programs.

Despite the presence of a valid partition table in the MBR, Gpart provided a message about a "possible partition" of 76.32 GB before crashing due to a "fatal error" in the backup boot sector. TestDisk, GetDataBack and SalvageRecovery detected two partitions starting at CB $= 63$ sectors. The former was 160055532 sectors in size (i.e. the *main* one) and the latter 160071592 sectors. GetDataBack displayed the same contents for both, while SalvageRecovery displayed the *main* one twice but reported the other one as empty.

Scrounge-NTFS found only the *main* partition, by reading the MBR. Restorer Ultimate and Recover It All Now found only that one as well, but they scanned the entire drive. In the case of Autopsy, it is unclear whether the same result derived from reading the partition table or scanning the disk image.

DMDE results were confusing. Despite finding the *main* file system correctly, after scanning the drive it also displayed three other partitions. Each one appeared to be related to one of the two parts of the whole MFT, but the CB values were completely off.

RecuperaBit found the highest number of partitions, namely 304, of which two are overlapping and *recoverable*. The former is the *main* one, with MFT starting at 6291519 sectors. The latter has the MFT located at 60929375 sectors. The MFT of the *main* file system is split in two parts containing entries 0–59167 and 59168–74021. At the same time,

SOFTWARE	#1	#2	#3	#4
TestDisk	Perfect	*Error*	—	*Error*
Autopsy	Perfect[1]	No files	—	Good[2,3]
Scrounge-NTFS	Partial[4]	Terrible[5]	Terrible[5]	Terrible[5]
Restorer Ultimate	Perfect[6]	Partial[7]	Perfect	Good[3,8]
DMDE	Perfect[6]	*Error*	Perfect	Good[3]
Recover It All Now	Terrible[9]	—	—	No files
GetDataBack	Perfect[10]	Good[11]	—	Good[3,12]
SalvageRecovery	Perfect[13]	Terrible[14]	—	Perfect
RecuperaBit	Perfect	Perfect[15]	Perfect	Perfect[16]

1 Showing 8 orphaned files (ids 16–23)
2 It shows some *ghost* files not found by other tools, it finds the name and location of `Root/WINDOWS/PeerNet` (id 143)
3 Many orphaned files
4 No deleted files
5 With the `-m` flag it returns an error, without the flag it discards directory information
6 NTFS metadata files are shown in a separate directory
7 The output includes the name of `interesting` (id 64) but does not show `Dir_67` (id 67) nor `another` (id 68)
8 The directory with id 143 is shown as `Root/?`
9 It loops infinitely when trying to open `Root/.Trash-1000`, then it crashes
10 Double entries shown for some metadata files, due to the MFT mirror
11 The *ghost* directories with identifiers 64 and 67 are shown as "virtual" under *Root*
12 Orphaned files are shown directly under *Root*
13 No NTFS metadata files
14 Only `bbb.txt` and `$Extend` are displayed, with a wrong size of 69 KB
15 The real name of `Dir_64` is not detected
16 The directory with id 143 is shown as `LostFiles/Dir_43`

Table 8.2: Test results for directory tree accuracy

records in the second one start with identifier 59168. Hence, the other partition is probably the result of leftover MFT entries which were not wiped after the MFT has been defragmented. RecuperaBit correctly identifies this set of over 14000 file records as a partition on its own, because the entries are placed in a way that does not match the MFT of the *main* file system.

Regarding the other 302 possible partitions found by our program, these are mostly formed by sets of 2 or 3 MFT entries that cannot be clustered together. The largest one contains 9 files. Since the identifiers of these records do not match the numbering used in the largest partitions, they must be kept separate.

8.3 DIRECTORY TREE ACCURACY

In Section 2.3, we focused on the forensic file system reconstruction problem. The main goal is to obtain a tree which is as accurate as possible, therefore it is very important to assess the output quality of the tested programs. Table 8.2 shows a summary of our test results, which are discussed below in greater detail.

SimpleIntact (#1)

In this example we expected all tools to provide a correct outcome, since the file system is intact. TestDisk, Autopsy, Restorer Ultimate, DMDE, GetDataBack, SalvageRecovery and RecuperaBit displayed the contents of the file system perfectly. Autopsy also detected 8 orphaned files related to unallocated reserved MFT entries.

Restorer Ultimate and DMDE displayed the NTFS metadata files in a special directory, but this is a feature of these tools. GetDataBack showed double entries for some of these files due to the MFT mirror, while SalvageRecovery completely hid them.

Scrounge-NTFS is programmed to recover only allocated elements, therefore it did not detect deleted files. Recover It All Now failed completely on this test, showing an infinite directory loop when trying to open `Root/.Trash-1000` and crashing shortly after.

BrokenTable (#2)

The expected solution for this disk image is shown in Figure 5.2. Only GetDataBack and RecuperaBit found all the files correctly. With the former, the two *ghost* directories (id 64 and id 67) were listed directly under *Root* and marked as "virtual".

Restorer Ultimate was able to identify the real name of `interesting` (id 64) from an unreferenced index record, however it did not detect the elements with identifiers 67 and 68. For this reason, we consider the result to be a partial output. SalvageRecovery found only `$Extend` (id 11) and `bbb.txt` (id 66). Moreover, it displayed them with a wrong size of 69 KB, i.e. the MFT size.

Scrounge-NTFS was provided with the correct partition geometry, including the `-m` flag pointing to the MFT offset. In this setting, it did not find any file. Without the `-m` flag, it ran in carving mode detecting only `aaa.txt` (id 65) and `another` (id 68) with no directory structure. TestDisk and DMDE returned an error due to the damaged file system. Autopsy showed an empty drive and Recover It All Now could not be tested because it did not recognize the file system.

HardToFind (#3)

Restorer Ultimate, DMDE and RecuperaBit reconstructed the directory tree perfectly. Scrounge-NTFS was tested with the correct partition geometry parameters found by RecuperaBit, however it returned the dubious message "couldn't read mft: Success". Using its carving mode (with no `-m` flag) it detected the files discarding directory information. We could not test the other tools on this sample.

RealLife (#4)

For a better comparison, we tested the tools only for the *main* file system, despite the fact that some programs found more than one. TestDisk could not access the partition because of the slightly damaged MFT and Scrounge-NTFS showed the same problems seen with *BrokenTable* and *HardToFind*. Recover It All Now displayed no files.

Autopsy, Restorer Ultimate, DMDE and GetDataBack provided a good result. However, all of them included many extra orphaned files mostly coming from other partitions detected by RecuperaBit, as discussed in Section 8.2. This means that MFT entries were not clustered thoroughly and they were included as soon as they were found inside the boundaries of the *main* file system. This kind of wrong assumption is called ABSTRACTION ERROR [2].

On the positive side, Autopsy correctly found the name and location of directory `Root/WINDOWS/PeerNet` (id 143), together with some *ghost* files not detected by other tools. Restorer Ultimate placed the former at path `Root/?` and GetDataBack showed all orphaned files under the *Root* directory.

RecuperaBit and SalvageRecovery reconstructed the directory tree perfectly. RecuperaBit did not find `Root/WINDOWS/PeerNet` (id 143), therefore it placed the *ghost* directory `Dir_143` under *Lost Files*.

8.4 RECOVERED FILE CONTENTS

The recovery of file contents is not directly related to file system reconstruction, but it is a useful feature. Table 8.3 shows our test results after using the programs with several disk images. We considered different kinds of NTFS files.

All programs were able to extract the contents of standard files, both fragmented and non fragmented. Therefore, we do not include this case in our comparison. We compare the output of the tools on sparse, compressed and encrypted files.

Definition (Sparse, Compressed and Encrypted Files). A file is called SPARSE if it has a `$DATA` attribute with some runlist entries having no offset. In this case, the corresponding sectors are not written on disk but they shall be considered filled with zeros. A COMPRESSED file is stored using NTFS compression, however Microsoft does not provide the actual algorithm [3].

An ENCRYPTED file is not saved in a readable format. Its content is encrypted using a random key, which is then stored in the `$LOGGED_UTILITY_STREAM` attribute encrypted with a key based on the owner's password [3].

SOFTWARE	SPARSE	COMPRESSED	ENCRYPTED
TestDisk	OK	OK	Empty
Autopsy	Empty	OK	OK
Scrounge-NTFS	OK	Unsupported	OK
Restorer Ultimate	OK	OK	OK
DMDE	OK	OK	Unsupported
Recover It All Now	OK	Wrong	OK
GetDataBack	Empty	OK	OK
SalvageRecovery	Empty	Wrong	OK
RecuperaBit	OK	Unsupported	OK

Table 8.3: Test results for recovered file contents

Most tools were able to extract sparse files, however Autopsy, GetDataBack and SalvageRecovery did not restore them. Regarding compressed files, Recover It All Now and SalvageRecovery extracted their contents without performing any decompression, leading to broken output. Scrounge-NTFS and RecuperaBit displayed a warning to remind the user that they do not support this kind of files. All the other tools decompressed files correctly.

The programs should be able to extract encrypted files normally, because they are stored in the same way as standard files. Decryption is outside the scope of these tools, therefore we tested if the software could extract contents in encrypted form. Most programs restored encrypted `$DATA` attributes. However, TestDisk created empty files and DMDE is described as not supporting NTFS encryption.

8.5 OUTPUT FORMATS

Forensic analysts must be able to document and report their actions and findings [3], hence programs should provide a way to save reconstructed file systems in one or more commonly used formats. When testing the tools, we discovered that many of them do not provide sufficient output capabilities for reporting purposes.

In fact TestDisk, Scrounge-NTFS, DMDE, Recover It All Now, GetDataBack and SalvageRecovery can only restore file contents. They do not provide any functionality for saving the list of files with their attributes. Gpart, Scrounge-NTFS, DMDE and GetDataBack are limited to exporting partition geometry information in plain text.

Only Autopsy and RecuperaBit provide effective reporting features. Autopsy can export a list of files as Comma Separated Values (CSV) and it can produce a body file[10], i.e. an intermediate format used by

10 The body file format specification is available on SleuthKitWiki.
URL: `http://wiki.sleuthkit.org/index.php?title=Body_file`

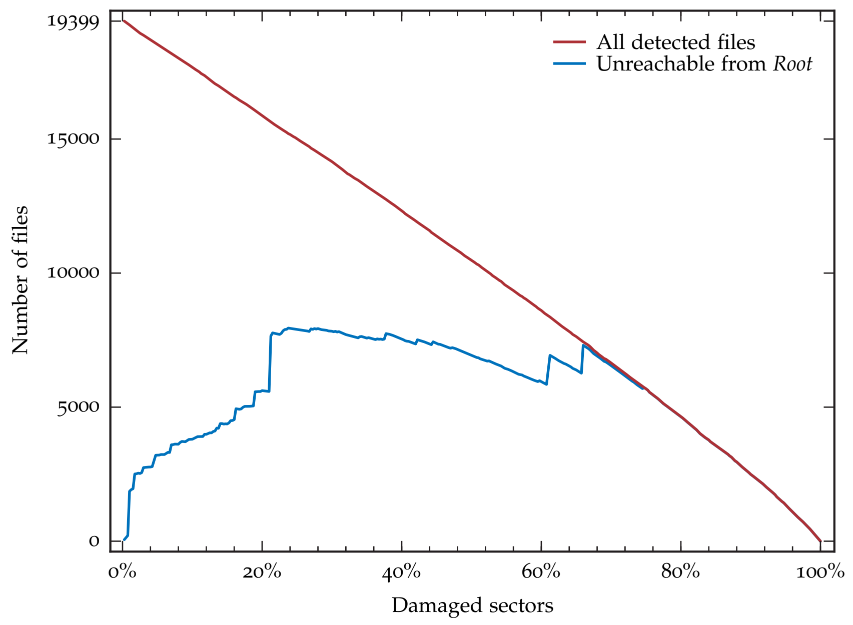


Figure 8.1: Detected files and corruption level

`mactime` and other tools to create timelines based on MAC times. There is also an option for exporting reports in HTML and Excel formats, but these are more limited.

RecuperaBit can export file lists in CSV and in body file format. Moreover, it generates valid Ti*k*Z code for embedding directory tree drawings in LaTeX documents (e.g. see Figures 2.1, 5.2 and 5.3).

8.6 OUTPUT QUALITY VERSUS CORRUPTION LEVEL

In our last experiment we tested the reconstruction capabilities of RecuperaBit on an increasingly corrupted drive. We started from a bitstream image containing a clean installation of Windows XP, shrinked to a 3 GB NTFS partition completely full of files.

We damaged the hard disk by randomly wiping sectors, up to 100% of them. At every iteration different sectors were wiped, sampling 401 data points. Figure 8.1 shows the number of files found by RecuperaBit in the biggest partition found at each iteration.

The red line displays the total amount and the blue line shows only the elements which were unreachable from the *Root* directory and ended up under the *Lost Files* node. We can see that, despite the variable corruption level, RecuperaBit effectively scans the drive and it detects readable MFT entries.

With a moderate number of damaged sectors, we obtain a good portion of the directory tree. When the corruption level becomes severe, many files start to be detached from the main tree. This happens because there is not enough information to find their path. Still, if their presence can be inferred they are listed in *Lost Files*.

9 CONCLUSION

With our work, we have focused on solving the forensic file system reconstruction problem when metadata is partially corrupted or missing. As a first contribution, we have provided a generic bottom-up algorithm for rebuilding the directory tree of a damaged file system, including subtrees unreachable from the *Root* node.

This allows to effectively correlate and analyze the metadata that is still available in a corrupted file system. Metadata is useful because it contains file names and MAC timestamps. These pieces of information provide additional context with respect to mere file contents restored using carving techniques.

Our second contribution is a technique for detecting the geometry of NTFS partitions, based on an optimization of the Baeza-Yates–Perleberg approximate string matching algorithm. Although not essential for obtaining a reconstructed directory tree, this information is needed to restore file contents correctly.

These two approaches can be used together to obtain a complete picture of the metadata and the data contained in a corrupted drive making use of NTFS. Therefore, forensic analysis techniques that rely on metadata can still be applied, even if parts of it are damaged.

We have described the algorithms needed for all recent versions of NTFS and we have published a FLOSS implementation called RecuperaBit. Hence, our techniques may be independently verified, tested and improved. Currently, our partition geometry detection strategy is limited to NTFS. However, the basic idea can be adapted for other file systems having addresses expressed in clusters and containing data structures similar to index records.

We empirically compared the output quality of RecuperaBit against other open source and commercial data recovery programs. Many of these tools implement undocumented techniques that cannot be easily scrutinized. Nevertheless, our results show that our algorithms are effective and the outcome is good. In some cases, RecuperaBit produced a better output than commercial software.

9.1 FUTURE WORK

There are several opportunities for further improvement on the reconstruction of damaged file systems. Regarding NTFS, we would like to investigate how to support its older versions, namely those having MFT entries with no stored identifier. If the file record number is not available, our current clustering approach cannot work. It should be possible to adapt the approximate string matching technique for determining file identifiers using file names as “characters”, thus we should explore this idea further.

Additionally, we do not consider remnant entries in the slack space of index records. More *ghost* files may be discovered if these traces are taken into account, however we shall also check carefully if their usage has negative consequences on the output quality.

RecuperaBit does not support the extraction of files stored using NTFS compression. We would like to add support for them, because they can be found in many cases.

Most importantly, support for more file systems should be added. Our bottom-up reconstruction algorithm does not depend on the underlying file system type, therefore we should test it in other common scenarios such as drives using FAT. The source code of RecuperaBit is available, thus additional tests and improvements can be performed with the help of users and digital forensics professionals.

In particular, we plan to seek further collaboration with the CAINE project[1], a Linux-based digital forensics environment with numerous active users. The potential inclusion of RecuperaBit in the platform could broaden the pool of contributors, improving and validating our algorithms for forensic file system reconstruction.

1 CAINE (Computer Aided INvestigative Environment), developed by Nanni Bassetti. URL: `http://www.caine-live.net/`

BIBLIOGRAPHY

[1] Ricardo A. Baeza-Yates and Chris H. Perleberg. Fast and practical approximate string matching. *Information Processing Letters*, 59(1):21–27, 1996. ISSN 00200190. doi: 10.1016/0020-0190(96)00083-X. URL http://www.sciencedirect.com/science/article/pii/002001909600083X.

[2] Brian Carrier. Open Source Digital Forensics Tools: The Legal Argument. Technical report, 2002. URL http://digital-evidence.org/papers/opensrc_legal.pdf.

[3] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley, 2005. ISBN 0321268172. URL http://digital-evidence.org/fsfa/.

[4] Harlan Carvey. *Windows Forensics and Incident Recovery*. Addison-Wesley Professional, 2005. ISBN 0321200985. URL http://dl.acm.org/citation.cfm?id=1208511.

[5] Eoghan Casey, editor. *Handbook of Computer Crime Investigation: Forensic Tools and Technology*. Elsevier Academic Press, 2001. ISBN 0121631036. URL https://www.elsevier.com/books/handbook/casey/978-0-12-163103-1.

[6] Andrea Ghirardini and Gabriele Faggioli. *Computer Forensics*. Apogeo Editore, 2007. ISBN 8850325932. URL http://www.apogeonline.com/libri/9788850325931/scheda.

[7] Patrick A. V. Hall and Geoff R. Dowling. Approximate String Matching. *ACM Computing Surveys*, 12(4):381–402, 1980. ISSN 03600300. doi: 10.1145/356827.356830. URL http://dl.acm.org/citation.cfm?id=356827.356830.

[8] Ewa Huebner, Derek Bem, and Cheong Kai Wee. Data hiding in the NTFS file system. *Digital Investigation*, 3(4):211–226, 2006. ISSN 17422876. doi: 10.1016/j.diin.2006.10.005. URL http://linkinghub.elsevier.com/retrieve/pii/S1742287606001265.

[9] Roy M. Jenevein. Method and apparatus for recovering data from damaged or corrupted file storage media, 2001. URL http://www.google.com/patents/US6173291.

[10] Zhang Kai, Cheng En, and Gao Qinquan. Analysis and Implementation of NTFS File System Based on Computer Forensics. *2010 Second International Workshop on Education Technology and Computer Science*, pages 325–328, 2010. doi: 10.1109/ETCS.2010.434. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5458951.

[11] Jason Medeiros. NTFS Forensics: A Programmers View of Raw Filesystem Data Extraction. Technical report, 2008. URL http://grayscale-research.org/new/pdfs/NTFSforensics.pdf.

[12] P. D. Michailidis and K. G. Margaritis. On-Line Approximate String Searching Algorithms: Survey and Experimental Results. *International Journal of Computer Mathematics*, 79(8):867–888, 2002. ISSN 00207160. doi: 10.1080/00207160212111. URL http://dx.doi.org/10.1080/00207160212111.

[13] Liu Naiqi, Wang Zhongshan, and Hao Yujie. Computer Forensics Research and Implementation Based on NTFS File System. *2008 ISECS International Colloquium on Computing, Communication, Control, and Management*, pages 519–523, 2008. doi: 10.1109/CCCM.2008.236. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4609565.

[14] Tony Sammes and Brian Jenkinson. *Forensic Computing*. Springer London, London, 2007. ISBN 978-1-84628-397-0. doi: 10.1007/978-1-84628-732-9. URL http://link.springer.com/10.1007/978-1-84628-732-9.

[15] The Linux-NTFS Project. NTFS Documentation. URL http://flatcap.org/linux-ntfs/ntfs/index.html.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured at:

`http://postcards.miede.de/`