FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Digital Forensics Testing Images Generator
Student:	Bc. Petr Horák
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2022/2023

Instructions

Digital forensics is a very important area as most of the information is stored electronically. Used in investigations, it has become a powerful method for solving a lot of computer crimes. Training of required skills is possible on many available test disks. But creating these disks and covering all the possible combinations is a complex process. Not to mention that these do not fully cover all possible scenarios and offer poor scalability.

Survey the most used digital forensics scenarios (encryption, deleted files, ...) and create a modular tool to generate source files (disk images) to practice these scenarios. The tool should provide space for easy scalability for possible further development and extensions. Provide supporting documentation so the scenarios could be used in actual digital forensics tutorials.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Digital Forensics Testing Images Generator

Bc. Petr Horák

Department of Information Security
Supervisor: Ing. Jiří Dostál, Ph.D.

May 4, 2023

Acknowledgements

I would like to thank all the friends and family members that supported me during the creation of this thesis. Namely, I would like to thank our volleyball group "Nemůžu to říct" for all the laughts, playing games with them kept me sane in the dark times. The same goes for the volleyball group "Kytičky zleva doprava". I'd like to thank my friend Radek Šmíd for his advice and assistance. He has been of great help during the creation of this thesis.

Last but not least, I would like to thank my supervisor, Ing. Jiří Dostál, Ph.D, for all the guidance and help he provided.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 4, 2023 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Horák, Petr. *Digital Forensics Testing Images Generator*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Digitální forenzní analýza je kritický nástroj nejen při vyšetřování, ale poznatky z této oblasti se používají i např. pro obnovy poškozených systémů. Tato práce poskytuje základní bázi znalostí potřebnou pro provádění těchto akcí. Detail je kladen na diskové oddíly a jejich možné chyby v konfiguraci. Popsány jsou vybrané souborové systémy (konkrétně ext4, FAT32 a NTFS) s ohledem na běžné scénaře, se kterými se můžeme setkat v rámci digitální forenzní analýzy. Na přiloženém CD (zip souboru) jsou některé z nich uvedeny ve formě úkolů včetně popisu řešení. Praktická část je pokryta vytvořením nástroje pro generování disků pro trénink scénářů popsaných v úkolech s možností snadné konfigurace a rozšiřitelnosti pro další potencionální vývoj.

Klíčová slova digitální forenzní analýta, disk, obraz disku, MBR, GPT, diskový oddíl, šifrování, souborový systém, ext4, FAT32, NTFS, BitLocker

Abstract

Digital forensics is a critical tool not only in criminal investigations, but the knowledge gained from it is also essential in system recovery schemes. This thesis provides the essential knowledge base for these tasks. A significant focus is laid on partitioning schemes, with some real-world examples of possible misconfigurations. Further, selected file systems (ext4, FAT32, and NTFS) are described with coverage of basic digital forensics scenarios. Several of these scenarios are described in the form of tasks to complete on the enclosed CD (or zip file), with a solution provided. As for the practical part, a disk image generator for these cases was created, allowing easy configurability for the disk image output. It is built in a scaleable manner to allow possible further development.

Keywords Digital Forensics, disk, disk image, MBR, GPT, disk partition, encryption, file system, ext4, FAT32, NTFS, BitLocker

Contents

List of Figures

Chapter 1

Introduction

Digital forensics is a branch of forensic science that focuses on the identification, preservation, extraction, and analysis of digital evidence. It is a critical tool in criminal investigations, as well as in civil and corporate matters, where electronic devices and networks may contain valuable information related to an incident or event. The goal of digital forensics is to recover and analyze data from electronic devices in a manner that is forensically sound and admissible in a court of law.

There are several tools available for creating disk images from existing hardware disks and for further analysis. However, according to my survey, there are not many available options in the sense of generating these images based on the selected parameters with the pre-built forensic scenarios of the user's choice. This hole is helpfully reduced by the work. Additionally, there is a lot of room for continual improvements, as there is no chance of ever covering all the scenarios.

The structure of this thesis is as follows: Chapter 2 is about data storage, obtaining evidence from storage media, and common types of disk image formats. Chapters 3 and 4 focus on the partitioning schemes; extensive attention is paid to MBR analysis, with some of the scenarios explained. GPT is also covered. Disk and file system encryption are briefly mentioned in the following chapter.

Chapter 6 describes the selected file systems, in particular ext4, FAT32, and NTFS. It pays special attention to disk layouts and internal data structures, and understanding processes of commonly used tasks. Digital forensics in general are then mentioned in Chapter 7. Lastly, some information about the implementation of the image generator tool is provided in Chapter 8.

Throughout the thesis, there are no details provided about Apple devices and APFS (Apple File System). Some details of the file system are still unknown, and there are also limitations when it comes to system data [1]. Based on the nature of the intended practical use, it was omitted from the scope of this thesis.

1.1 Scenarios

The main objective of digital forensics is to examine internal computer or server storage drives to look for digital evidence as part of criminal investigations [1]. This trend tends to change with the use of cloud storage. It can be challenging to get a hold of the data stored in the cloud. It can be physically stored in multiple locations around the world, and access to it cannot be guaranteed, even though most cloud service providers cooperate with law enforcement on issued warrants [2].

Imagine the following scenario. A crime is committed. Law enforcement looks through the evidence and identifies a suspect. They get a warrant and perform a home inspection, during which they find the suspect's laptop. It may be essential for the case to extract the data from the computer to find more evidence, motive, or co-conspirators. The laptop is handed to a digital forensics team to retrieve as much information as possible. Because the data in volatile memories, such as RAM, is erased when the power goes out, they are not guaranteed to be useful. That leaves the laptop's drive (HDD or SDD) as the main source of the information, possibly alongside some portable storage media. That is the scenario I kept in mind during the creation of this thesis.

1.1.1 Testing models

Throughout the thesis, several scenarios had to be tested in practice to assess system behavior, as they were rather specific. These were conducted on an external flash disk with a capacity of approximately 15.5 GB (14.44 GiB). As for the test environments, three different operating systems were used, and on each of them, different partitioning software. These were Manjaro 5.15.81-1-MANJARO (64-bit) with GParted version 1.4.0, Ubuntu 22.04 LTS 5.15.0-52-generic with gnome-disk-utility 42.0, and Windows 11 Disk Management tool version 10.0.22621.1.

Chapter 2

Data storage

There are several types of computer storage. Historically, magnetic tapes or punched cards were used. Today's storage options are far more advanced. As for this thesis, it is not reasonable to cover all the storage types. It is difficult to perform forensics analysis on some types which makes them unsuitable for learning, which is the main purpose of this thesis.

The focus will be paid to NVM (non-volatile memory), specifically HDDs, SSDs, and external storage devices such as flash disks. These are the most common memory types and thus are of great interest. Performing forensic analysis on other types of memory (EEPROM, RAM, etc.) is possible and, in many cases, very desirable, however, it is not suitable for our purpose of creating learning software. The program output of this thesis is scale-able and extensible; there is a possibility to add this functionality later if needed.

From now on, the terms "drive" and "disk" will be used to describe both HDDs and SSDs alike, as well as external storage drives like USB flash disks or SD cards.

2.1 Data extraction

Data extraction is a process of obtaining digital evidence. Sources of the evidence may vary, as can the means of obtaining it; some of them are log exports, copies of storage media, and others.

It is crucial to avoid any possible evidence tampering if the digital information is meant to be admissible in court. Control hashes [3] is one possible method for ensuring data has not been changed.

The original evidence needs to remain unchanged. Forensic specialists have to work with data copies to avoid potential disruptions. This is especially true for SSDs, which use mechanisms like wear leveling and garbage collection [4]. Both hardware and software tools exist to create a copy of digital media, some of them are discussed in the following subsections.

2.1.1 HW tools

There are several devices physically blocking access to connected disks, allowing only one-way communication. An example of such a tool is a forensic disk controller. It can work on either a USB or SATA connection. How exactly the device prevents write access is usually not publicly available information; for example, implementation is protected by patent US 6,813,682 B2 from 2004 [5]. One of such devices is the Tableau Forensic SATA/IDE Bridge T35u [6].

There are also specialized duplicating devices, like the Tableau Forensic Duplicator TD2u [7]. They usually offer more functionality, including hashing for evidence-tamper detection, disk cloning, formatting, and others.

2.1.2 SW tools

Multiple programs are available to create disk images from the existing drives. It is really important not to perform any write operations on the original drive, as this could cause the obtained evidence to be inadmissible in court.

Starting with one of the popular options [8], FTK Imager is accessible as a free-to-use tool created by AccessData [9]. It is one of the most widely used free software packages in the field. Functionality includes creating images, exporting files, previewing, hashing, and more. If we want a full-scale forensics tool, FTK (Forensics Toolkit) from the same company is also an option, although it has a commercial license.

EnCase [10] is a popular digital forensics tool that is widely used by law enforcement agencies, forensic investigators, and other organizations [1]. It is a comprehensive tool that provides a range of features, including data acquisition and imaging, data analysis and search, and evidence presentation. From the beginner's point of view, a disadvantage is its commercial license.

Magnet Forensics' products are among the most widely used commercial software. They are well-regarded among professionals and even used by law enforcement, it is one of the currently best choices for digital forensics analysis [1]. There are lots of other tools, such as Autopsy, Cellebrite Inspector, and X-Ways, to name a few; it comes down to the preferences of individual users.

2.2 Storage drives

Traditional magnetic HDDs are mainly used in high-capacity storage servers and data centers, as their cost-performance ratio is still the best when there is no need for extra high-speed data transmissions. Speed limitations can be partially addressed by cache memory. Still, it is fairly rare to find hard disks in modern computers as their only drive.

Solid-state drives are generally much faster, but that is balanced by a higher price. The storage method fundamentally differs from magnetic drives,

with NAND flash storage being the number one option. Due to their nature, SSDs have a limited lifespan. Therefore, techniques such as wear-leveling and garbage collection are used.

Utilizing all the cells of the drive equally prolongs its service life, but it has a significant negative impact on digital forensics. A leading example could be deleted files. When a file is marked as erased, it can still be retrieved from the magnetic storage (using various techniques) as long as the memory space is not overwritten. This is not guaranteed for flash-based memory, as it can utilize a technique called TRIM for erasing data marked as deleted [4].

2.2.1 Logical structure

To work with the storage space of either internal or external drives in a user-friendly way, it has to be formatted to a certain file system. It is possible to use unformatted disks and access them as raw on low-level, but this is not a common practice. Therefore, it is not suitable for this study. There are many file systems usable when formatting storage devices, and it is not possible to cover them all. The most commonly used (except for macOS) are described later in the thesis.

This section is focused on the organization of the disk as a whole. On the drive, there can be multiple partitions with different file systems. They are organized in a well-defined manner; the older Master Boot Record (MBR) and the newer GUID Partition Table (GPT) are the two mainly used partitioning schemes.

2.2.2 Disk images

A disk image is a digital representation of a disk, a computer file that behaves similarly to a hardware drive. It includes its content and has the same structure. The most common usage is in virtualization, where these images are used for virtual computers (VMware or Oracle VM VirtualBox being examples of such usage), software distribution (OSes, as well as other programs are often provided in the form of ISO images), and digital forensics.

There are multiple types of disk images, only a selected few will be discussed. The selection is based on relevance in this thesis, the author's experience with the images, and overall popularity in the community.

2.2.3 Raw image - .IMG

Raw disk images contain raw, unaltered data in binary format. They are block-by-block identical copies of the source. There is no additional metadata or compression. Especially, compression would be useful for high-capacity disks where a significant amount of the drive is not occupied and, thus, there is no need to store it byte by byte. This format is easy to create (a simple *dd*

command on Linux will suffice). As they do not need any external tools to be read or created, this is the ideal format for our purpose and will be used in the practical part of this thesis.

2.2.4 Raw data - .BIN

There is not much difference between .IMG and .BIN disk images in terms of content. Both contain raw binary data. The only one is that .BIN is often used to store solely file and directory content, not containing any information about the disk layout. In combination with .CUE file, they can create a complete disk image [11].

2.2.5 Optical Disk Image - .ISO

As the name suggests, ISO disk images are complete copies of optical disks like CDs or DVDs. They do not include any compression and are often used to distribute software, such as Linux distributions or Windows OS. The advantage is that ISO images can be mounted the same way regular optical disks would. Also, ISO is well standardized [12], and there should be no variations in the implementation of the format in any software used.

2.2.6 EnCase - .E01

There are several disk images that are originally associated with the software used for their creation. One of the most widely recognized is the EnCase evidence file. Encase Forensic is a very popular digital forensics tool that uses this format when coping and creating disk images. It divides the disk into data chunks of size 640 MB [13] and stores important information separately, including checksums, headers with details like media description, author name, date, and others. Nowadays, FTK Imager can work with this type of file too [14].

Chapter 3

MBR

Master Boot Record (MBR) is a partitioning scheme introduced and originally used in MS-DOS. It resides in the first sector of the storage medium and holds information about booting the device and partition distributions. MBR cannot be found on non-partitioned disks, e.g., on floppy disks [15], which are out of the scope of this thesis.

3.1 Structure

MBR can be found in the first sector (logical address 0x00). Its size is 512 bytes, which is a standard logical disk sector size. It has a table structure, as can be seen in Figure 3.1.

3.1.1 Bootstrap

Section bootloader (bootstrap, Master Boot Code) is the first of the three main parts of MBR with 446 bytes in size. It contains executable code to boot the device. That is usually done by finding an active partition and loading its first sector, which contains additional boot code (446 bytes are not enough to load a whole operating system to memory and transfer control to it).

3.1.2 Partition table

The partition table is the next significant part of the media. It holds all the necessary information about drive partitions. With 64 bytes in size, it is divided into four equal parts of 16 bytes, each for a separate partition. That would mean a maximum of four partitions on the drive. However, the last one can be programmed as an extended one, which could contain more partitions within. The internal structure then looks similar to the main partition table.

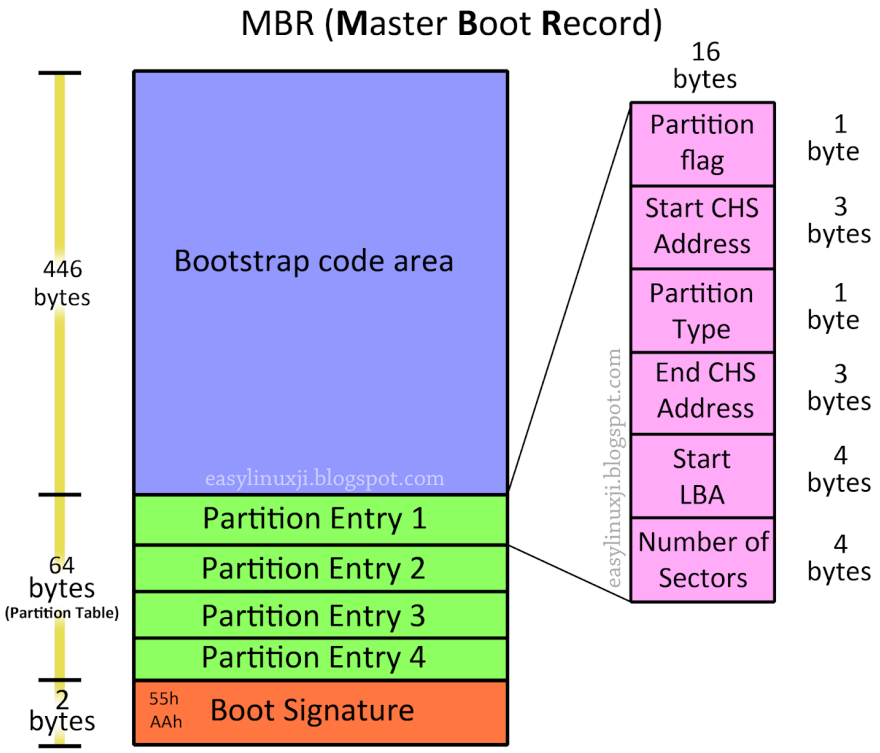

Figure 3.1: MBR structure [16]

Structure

As noted in the Figure 3.1, each partition entry is divided into 6 parts, their respective sizes can also be found in the figure.

Partition flag holds information about partition status. Value 0x80 stands for an active (bootable) partition, the one containing additional code to boot the device. Value 0x00 is standard for partitions, that are valid but not used for booting. All the other values indicate that the partition is invalid.

The physical location on the disk is stored in sections **Start CHS Address** and **End CHS Address**. CHS stands for cylinder-head-sector, it is an older way to specify a location on a hard drive based on its physical structure. There are two related problems. There are only 3 bytes dedicated to these addresses, which limits the total possible capacity of the storage drive. Secondly, flash storage uses a different structure than HDDs and CHS addressing no longer makes sense from a physical point of view.

A newer addressing system, Logical Block Addressing (LBA), is used. **Start LBA** represents the beginning of the partition by sector sequence number, that is, the number of sectors from the beginning of the disk, starting with sector 0. Instead of end LBA, **Number of Sectors** of a partition is

```
00 Empty            27 Hidden NTFS Win  82 Linux swap / So  c1 DRDOS/sec (FAT-
01 FAT12            39 Plan 9           83 Linux            c4 DRDOS/sec (FAT-
02 XENIX root       3c PartitionMagic   84 OS/2 hidden or   c6 DRDOS/sec (FAT-
03 XENIX usr        40 Venix 80286      85 Linux extended   c7 Syrinx
04 FAT16 <32M       41 PPC PReP Boot    86 NTFS volume set  da Non-FS data
05 Extended         42 SFS              87 NTFS volume set  db CP/M / CTOS / .
06 FAT16            4d QNX4.x           88 Linux plaintext  de Dell Utility
07 HPFS/NTFS/exFAT  4e QNX4.x 2nd part  8e Linux LVM        df BootIt
08 AIX              4f QNX4.x 3rd part  93 Amoeba           e1 DOS access
09 AIX bootable     50 OnTrack DM       94 Amoeba BBT       e3 DOS R/O
0a OS/2 Boot Manag  51 OnTrack DM6 Aux  9f BSD/OS           e4 SpeedStor
0b W95 FAT32        52 CP/M             a0 IBM Thinkpad hi  ea Linux extended
0c W95 FAT32 (LBA)  53 OnTrack DM6 Aux  a5 FreeBSD          eb BeOS fs
0e W95 FAT16 (LBA)  54 OnTrackDM6       a6 OpenBSD          ee GPT
0f W95 Ext'd (LBA)  55 EZ-Drive         a7 NeXTSTEP         ef EFI (FAT-12/16/
10 OPUS             56 Golden Bow       a8 Darwin UFS       f0 Linux/PA-RISC b
11 Hidden FAT12     5c Priam Edisk      a9 NetBSD           f1 SpeedStor
12 Compaq diagnost  61 SpeedStor        ab Darwin boot      f4 SpeedStor
14 Hidden FAT16 <3  63 GNU HURD or Sys  af HFS / HFS+       f2 DOS secondary
16 Hidden FAT16     64 Novell Netware   b7 BSDI fs          f8 EBBR protective
17 Hidden HPFS/NTF  65 Novell Netware   b8 BSDI swap        fb VMware VMFS
18 AST SmartSleep   70 DiskSecure Mult  bb Boot Wizard hid  fc VMware VMKCORE
1b Hidden W95 FAT3  75 PC/IX            bc Acronis FAT32 L  fd Linux raid auto
1c Hidden W95 FAT3  80 Old Minix        be Solaris boot     fe LANstep
1e Hidden W95 FAT1  81 Minix / old Lin  bf Solaris          ff BBT
24 NEC DOS
```

Figure 3.2: MBR partition types - output of ”fdisk” command

stored, which determines its size; the final sector dedicated to the partition can be computed from that. The remaining field, **Partition Type**, specifies the file system used. A list of possible values can be found in figure Figure 3.2.

Limitations

From the structure, there are some limitations to be noted. The start and end CHS addresses are only 3 bytes long. There are 10 bits for the cylinder number (0 - 1023), 8 bits for the head (0 - 254), and 6 bits for the sectors (1 - 63) [17]. That means any sector beyond these maximal values is not possible to be represented by the CHS method and has to be indexed by LBA only. Traditionally, there are 512 bytes in a sector, leaving the maximal referenceable size of disk $512 \cdot 63 \cdot 255 \cdot 1024 = 7.84$ GiB. As the number of sectors is 4 bytes long (and so is LBA), there can be a maximum of $2^{32} \cdot 512$ bytes in a partition, that is 2 TiB. MBR will still work for drives with higher storage capacities, but the remaining space cannot be used. This limitation was one of the leading causes that led to the development of GPT, which is described in the section 4.

The apparent limitation of only 4 partitions can be resolved by setting the last partition as extended (there can be only one on the disk), which then can be separated into multiple logical partitions. It cannot be used for booting

but still provides a reasonable level of data security. Similarly to MBR, at the beginning of an extended partition, there is Extended Partition Boot Record (EPBR or EBR). It has the same structure, only Master Boot Code is not used and is usually filled with zeros. Then again, one of these logical partitions can be extended, which can be divided again into logical partitions. There is no limit to the number of logical partitions [18].

3.1.3 Boot signature

The last two bytes of the MBR are the boot signature. Their purpose is verification; a valid boot device will have these bytes set to 0x55 and 0xAA. This is sometimes called the magic number [15].

3.2 Behavioral testing

Only a few reliable sources describe the behavior of the MBR table and partitioning SW when there are some artificial modifications made (e.g., rewriting or changing selected bytes of the MBR table). Firstly, the MBR table is updated immediately after changes regarding partitions are made. Deleted partitions are not easily recoverable by changing the status in the respective MBR field. How partitioning SW would behave if we turned it around and changed some parts of the table is a question that available sources do not cover sufficiently or convincingly. To describe the behavior correctly, these scenarios had to be tested.

Tests were performed in the following way. External flash storage of size 14.44 GiB was formatted in the GParted software for multiple partitions utilizing multiple file systems, particularly FAT32, ext4, and NTFS. Exact space distribution can be seen in Figure 3.3. The corresponding MBR table can be found in Figure 3.3, where separate partitions are indicated by different colors.

3.2.1 Analysis

Let's examine the MBR table generated from Figure 3.3. Skipping the bootstrap code area, the first byte of our focus is 446 (0x1BE) (numbering from 0). That is our first partition. The first byte is 0x00, meaning the partition is not set for booting and is valid. Following are 3 bytes of the starting CHS address: 0x040104. It should be noted that little endian is used. It does not make any difference for this value but should be kept in mind for the next ones.

The following byte is a partition type. According to Figure 3.2, 0x0B is FAT32, which (unsurprisingly) corresponds with what was set in the GParted tool. The end CHS address is 0xFFC2FE, the start LBA is 0x00000800, and finally, the number of sectors is 0x0055F000. Similarly, the second partition

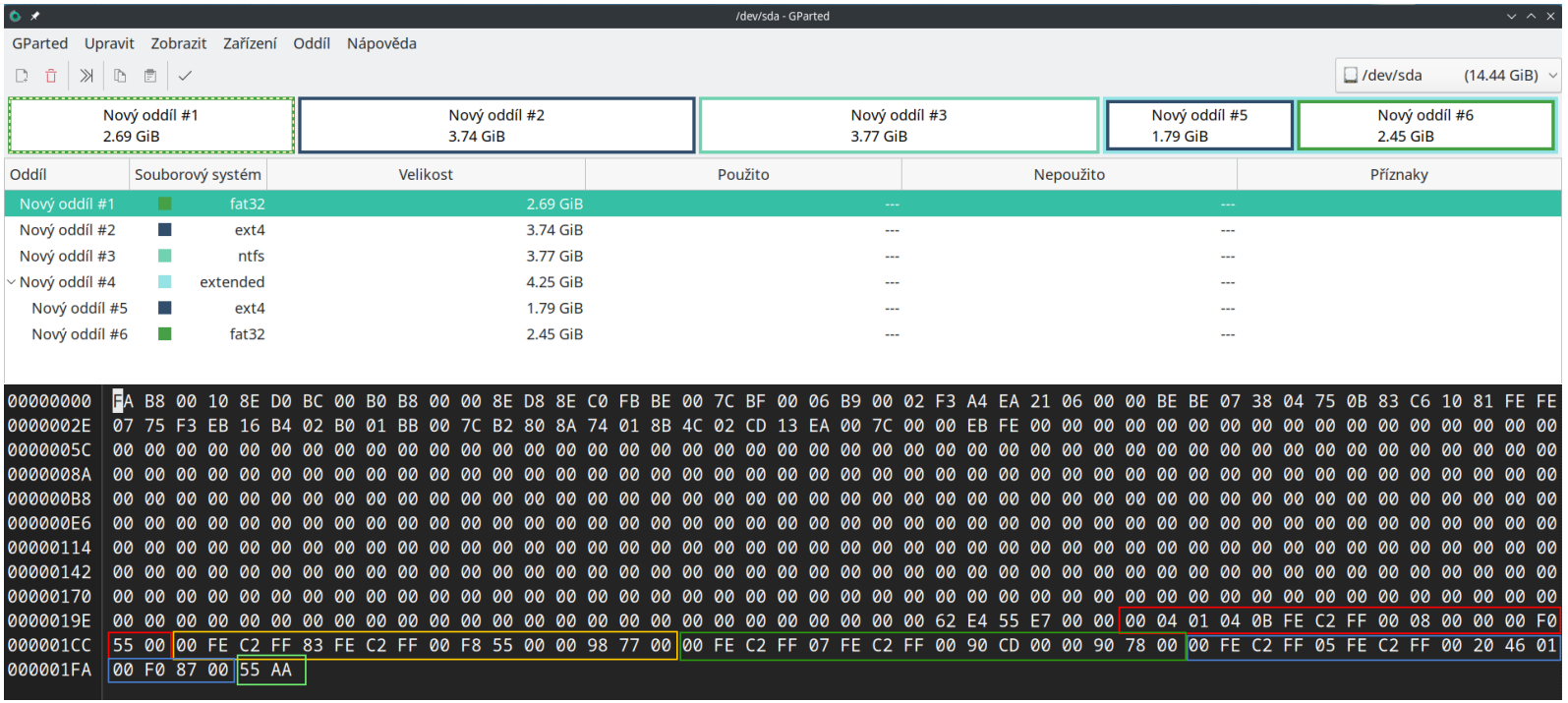

Figure 3.3: screen shot from GParted - partitions for testing

type is 0x83 (Linux), the third is 0x07 (NTFS) and the last is 0x05 (extended). At the end, there is a boot signature, bytes 0x55 and 0xAA, meaning the table is valid.

As one of the objectives of this thesis, a detailed analysis of the individual fields will be provided. It is not usually necessary to perform any MBR-related modifications in ordinary life, but it should be included in the learning process as it is one of the essential parts of the drive and related complications may result in data loss, operating system failures, and other critical outcomes. Also, a malicious actor can modify the table manually to confuse the investigators and to try hide some evidence.

3.2.2 CHS values

There are several things to be mentioned. Firstly, how LBA works. The first partition starts at sector 0x0800 = 1 MiB, which is caused by alignment. The second starts at 0x55F800. The number of sectors between them should be equal to the number of sectors of the first partition, which holds, 0x55F800 = 0x800 + 0x55F000. To verify, 0x55F000 sectors of 512 bytes is 0xABE00000 bytes, which is 2883584000 bytes in decimal, which corresponds to 2.6855 GiB setup in the GParted tool.

Secondly, computation of CHS values. In the case of flash memory, CHS values have no relation to the physical structure of the drive, but these numbers are still calculated and filled in the table. The start CHS value of the first partition is 0x040104. If we look at the level of individual bits, 0b0000 0100 0000 0001 0000 0100. The first 10 bits show the cylinder number, in this case, 0b0000 0100 00 = 16. The head number is following 8 bits, 0b00 0001 00 = 4. The remainder, 6 bits, is the sector number, 0b00 0100 = 4. LBA is calculated in a straightforward manner, that is

$$LBA = (C \cdot HPC + H) \cdot SPH + S \ - 1,$$

where HPC is the number of heads per cylinder and SPH is the number of sectors per head. The calculation for this specific case,

$$LBA = (C \cdot HPC + H) \cdot SPH + S \ - 1 = (16 \cdot 255 + 4) \cdot 63 + 4 - 1 = 257295.$$

This does not make sense, LBA is 0x800. The issue is in the previously mentioned missing connection of CHS to the physical structure. This information is not used anymore. Even altering it to arbitrary values did not change a thing regarding partition distributions. I was not able to find out why these specific numbers appear in the start CHS and the end CHS values, but they are not relevant. Let's examine what happens when the other parts of the table are altered, e.g., by the *dd* command.

3.2.3 Partition flag

The first thing to look at is the partition flag. Valid combinations are 0x80 and 0x00 as mentioned earlier. What happens if we mark a non-bootable partition with an 0x80 flag is a fascinating question. It is a well-known fact that only one disk partition can be marked active. However, marking multiple partitions active on the test flash drive storage did not prove to be a problem, neither for Windows nor for Linux operating systems. It has to be noted that this flag was only technical, as there was no boot code on any partition at all. This may be the reason why, even with the flag properly set, this drive was not even available in the boot device menu.

Our next focus is an invalid flag. Any value other than 0x80 or 0x00 means the partition is in an invalid state. The behavior of different operating systems and partition software varies significantly. After setting the flag to an arbitrary number, the GParted tool was not able to recognize any partitions and displayed the drive as an unallocated space. The Disks utility in Ubuntu recognized different partitions as they were, but was unable to detect used file systems or access files on the drive (note that this only covers automatic mounting and access given by default, there may be some methods to access the data using advanced techniques, but it is out of scope of this thesis). The best solution offered MS Windows 11, which not only detected the partitions correctly but, furthermore, was able to access the files stored on the drive (although there were typical problems accessing files on the ext4 file system).

3.2.4 Partition type

Overwriting the partition type byte can also prove tricky. Let's examine the situation. As main contenders, tested partitions are FAT32, ext4, and NTFS (their codes in Figure 3.2 are 0x0B, 0x83, and 0x07). In this test, Linux came out on top, as both Manjaro and Ubuntu were able to read and write all the partitions, even with interchanged labels. The situation for Windows 11 was considerably worse. I was not able to convince it to open the ext4 partition no matter how it was labeled, but that was to be expected. Furthermore, marking FAT32 and NTFS partitions as ext4 resulted in the same situation even for these systems. For both of those, FAT32 and NTFS labels worked just fine. It shows that modern OSes do not rely simply on the provided labels, as they would be unable to read or write these partitions. It can be considered expected behavior, generally, all the input to the computer should be regarded as invalid or malicious until proven otherwise, creating a presumption of guilt.

There are some interesting partition types to be mentioned further, and those are hidden partitions. A hidden partition is not visible when viewing the list of available drives on a computer. This allows the partition to be used for storing important files, boot files, etc. For this purpose, all hidden NTFS (option 0x17), hidden FAT32 (0x1B), and hidden ext4 (0x93) were

tested. This method does not provide any general protection as the flag can be very easily changed in most of the available partitioning software (that stands for Linux, doing this in Windows is not straightforward and requires a little deeper knowledge), but for inexperienced users or those without admin privileges, this may pose a bit of a challenge. There are more methods for detecting hidden partitions; the most obvious in this context is to look at the MBR table. Also, one can compare the disk capacity to the total capacity of all detected partitions and see the discrepancy.

3.2.5 Start LBA and number of sectors

If two or more partitions overlap, it means that they are using the same space on a drive. This can cause several problems, including data loss (if two partitions are trying to access the same data, one of the partitions may overwrite the other, resulting in data loss) and file system corruption. It is generally a good idea not to have overlapping partitions. In this section, we are going to look at what can happen if this situation occurs.

For the experiment, three equal partitions were created on the USB drive, all of them beginning just after the MBR and alignment (LBA 0x800) and covering the full size of the medium. The situation was tested in four different scenarios; different file systems for separate partitions (ext4, FAT32, and NTFS) and all having the same for each of these three file systems. Behavior differed significantly by the tool used.

GParted prompted the user with a warning on startup about possible overlapping partitions in all the tests. The situation caused the tool to be unstable and frozen when further partition adjustments were needed. All three partitions were mounted without an issue, enabling a user to write to and read from all of them. The output given in the GUI indicated three partitions of the same size (the full size of the drive), see Figure 3.4. As expected, data overwriting happened, and it did not take much effort to lose saved data by writing different files in the same sectors on variable partitions. That stands for the situation when at least one FS differs from the other two. When all the file systems were of the same kind, I was not able to achieve file overwriting, as all the changes were propagated to all the partitions with no differences and no problems.

In Ubuntu, mounting all the partitions was not an issue as well, the experience was very similar to Manjaro. However, utility Disks showed odd manners in displaying the overview. It suggested only two partitions (although 3 were already mounted) and even though they were the same size, on the display one occupied significantly more space, see Figure 3.5. Files overwriting proved to be tricky, as the OS refused to write to more partitions at once, after writing to one, the other two were marked read-only. On user-level experience, I did not find a way to overwrite or corrupt files, including using superuser privileges.

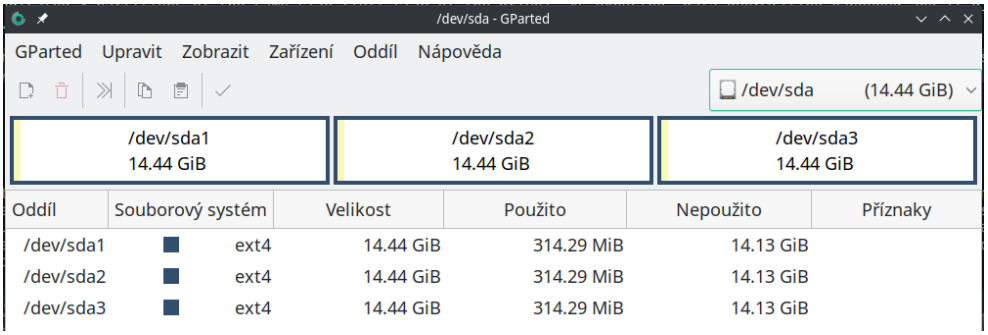

Figure 3.4: Screenshot from GParted - overlapping partitions

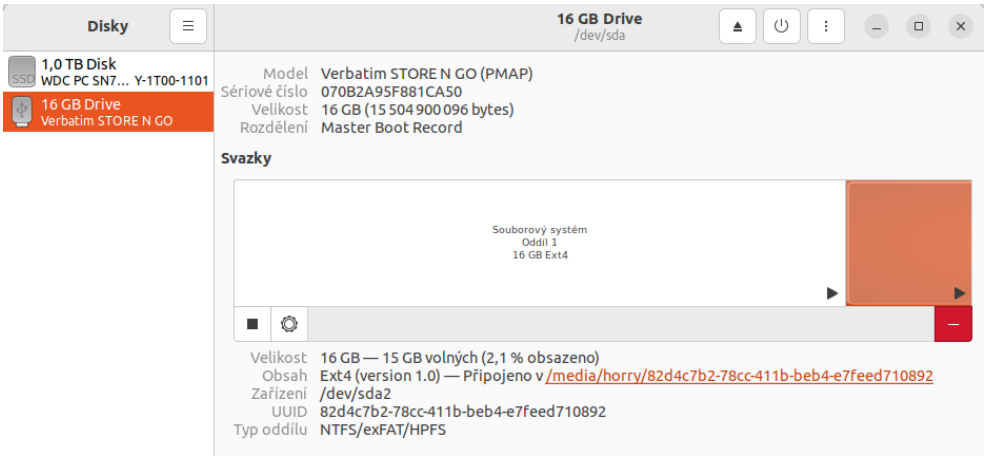

Figure 3.5: Screenshot from Disks - overlapping partitions

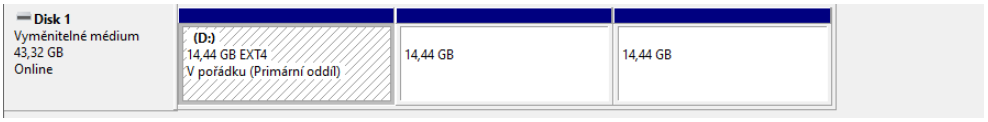

Figure 3.6: Screenshot from Windows - overlapping partitions

Windows proved to be a user-safe system; it was the only one that mounted only the first partition (in case it was not ext4, problems mounting the ext4 file system were already mentioned earlier). Attempts to mount the other two resulted in an error referring to out-of-date information loaded and suggested restarting the program or the whole OS, which for obvious reasons did not help. Talking about the actual partitioning program, it displayed all the partitions with the correct sizes and assumed the overall storage size to be their sum. Details can be seen in Figure 3.6.

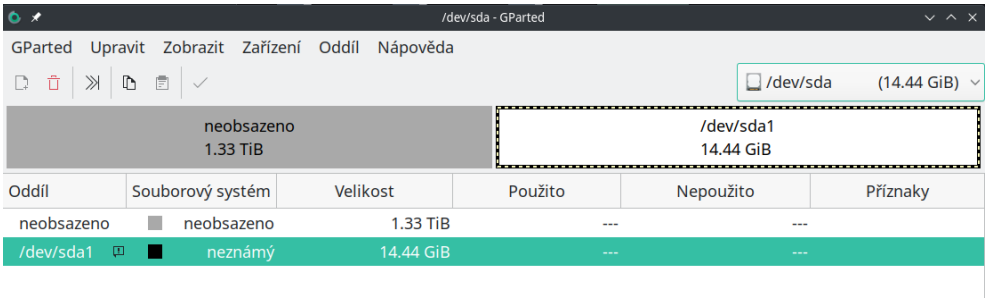

Figure 3.7: Screenshot from Manjaro - setting LBA beyond the size of the drive

There were 3 more tested scenarios, in particular starting LBA inside the MBR table, beyond the size of the drive, and setting the size of the partition higher than the drive. Altering the start LBA value failed in all three testing models. None of the systems was able to mount the drive. Windows required formatting the device before using it, Manjaro and Ubuntu both displayed incorrect readings in the program display output showing unrealistic values, as can be seen in Figure 3.7. Both Linux systems showed an error when trying to perform any operation with the partition, reporting an inability to read the partitioning table or an unfamiliar partition description.

The situation became more interesting when I changed the number of sectors to an off-limits number. All the systems were able to mount the drive. In Manjaro, any operation in GParted resulted in the error message the same as the one above. The reported disk size was consistent with the number of sectors (Figure 3.8). Write and read operations worked as normal. Ubuntu showed the wrong drive size (see Figure 3.9). However, it automatically reloaded, repairing the number of sectors to match drive capacity. All the other operations worked as well. Windows offered an easy but not very intuitive solution. Drive was not accessible immediately, there was no FS recognized. After the formatting operation was issued, it returned an error (of the same nature as in the case of Manjaro). However, the data was reloaded and the correct readings were available right away. Data on the disk could be accessed, and no data loss occurred.

3.2.6 MBR recovery

If the entire MBR table is deleted, it does not mean all the data is lost. If one can successfully recover the MBR, the data on the disk remains unchanged. That is true for HDDs but not necessarily for SSDs. They use garbage collection and level wearing, which can cause overwriting or permanently deleting stored data [4]. There are multiple ways to recover a lost MBR table. The most optimal is recovery from a backup, because if only the files changed on

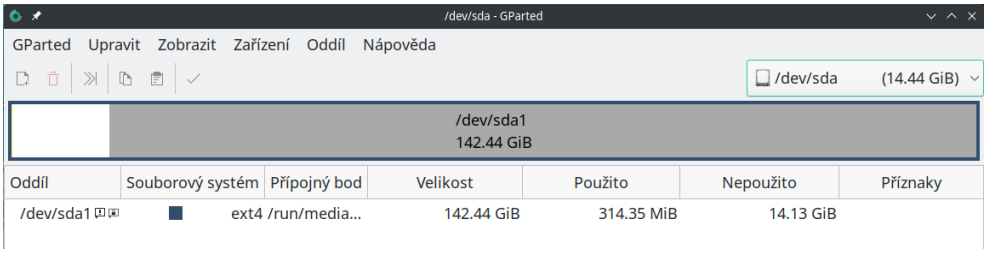

Figure 3.8: screen shot from Manjaro - setting number of sectors off-limits

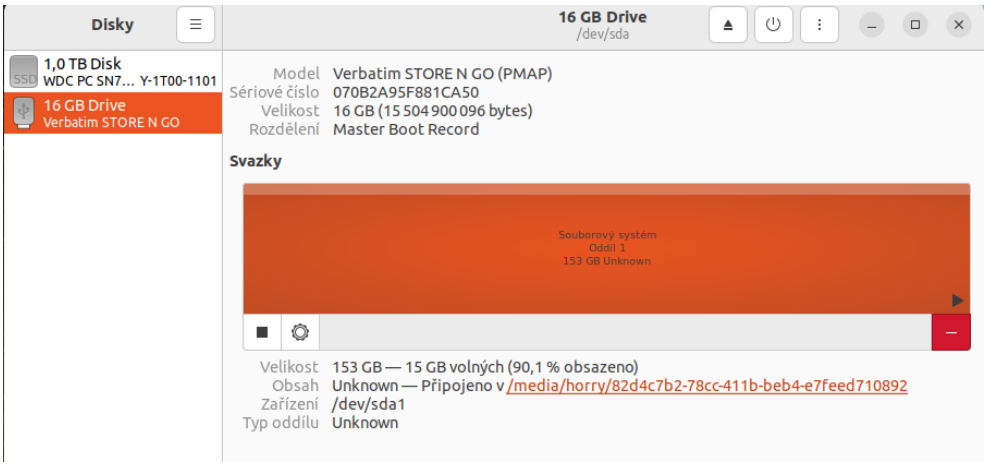

Figure 3.9: screen shot from Ubuntu - setting number of sectors off-limits

the drive, there should be no effect on the MBR table, and thus even old backups can be handy. It would be necessary to back up the MBR table as well and not just the user files, which is not a common practice. Multiple commercial tools claim to be able to recover a partition table, such as the EaseUS Partition Recovery tool [19]. The exact method of recovery is kept secret (that is to be expected), but one can assume it processes raw disk and looks for file signatures and other artifacts to try to identify partitions and used file systems.

3.3 Partition boot sector

The partition boot sector is not a part of MBR, but it is strongly interconnected with it. Therefore, at least a brief introduction is provided in this section. The partition boot record (PBR) (or partition boot sector) is a specific area of a computer's drive that contains the instructions for booting the operating system. It is typically located at the very beginning of the drive partition(s) in logical sector 1.

The bootstrap code area inside the MBR table contains a short program that is used to start the boot process, while the partition boot record is a data structure that is used to store information about the boot process and is used by the bootstrap code to locate and load the operating system and the boot loader. Once the boot loader has been loaded, it takes control of the boot process and is responsible for locating and loading the operating system. The operating system is then initialized and control is transferred to it, it is responsible for managing the resources of the computer and providing a user interface. All the partitions can contain a program to load the operating system, but only one is given control by the Master Boot Record, which is the one specified as active in the partition table entry [20]. Unlike the MBR table, PBR is highly file system dependent.

Chapter 4

GPT

GPT (GUID Partition Table) is the newer and more modern partitioning scheme. Created to assess some previously mentioned limitations of MBR, GPT is a part of the Unified Extensible Firmware Interface (UEFI) specification and was designed as a replacement for the MBR scheme [21].

4.1 GPT vs MBR

GPT offers several advantages over MBR, making it a better choice for the majority of modern drives. For example, GPT supports disks of much higher sizes, up to exabytes, while MBR is limited to disks up to 2 tebibytes in size. This is important because hard drives are getting larger and larger, and many users need a partitioning scheme that can support disks with high capacities that exceed the maximum limit of MBR. Not only in personal computers, but also in data centers, where this size of drive is fairly common. Furthermore, LBAs are 64 bits long instead of the 32 bits in the MBR, increasing the maximum possible disk capacity. It is almost impossible to buy a new PC with the MBR partitioning scheme set up by default.

In addition to supporting larger disks, GPT also allows for more than four partitions on a single disk. MBR is limited to a maximum of four primary partitions, which can be restricting for users who need to access multiple OSes from the drive (without using virtualization). It is partially mitigated by the possibility of using multiple logical partitions, but those cannot be set up as boot partitions. On the other hand, GPT allows users to be more flexible in how they organize their data. According to the specification [21], there is almost no limit on the number of partitions on the drive, but there could be OS restrictions, e.g., Windows limits this number to 128 partitions [22].

GPT also includes improved error-checking and recovery capabilities. It uses cyclic redundancy checks (CRCs) to verify the integrity of the partition table and can detect and repair certain types of corruption. This can help

Block:	Contents:			
LBA 0	Protective MBR			
LBA 1	Primary GPT Header			
LBA 2	Entry 1	Entry 2	Entry 3	Entry 4
LBA 3	Entries 5 – 128			
LBA 34 to LBA -34	Partition 1			
	Partition 2			
	Remaining Partitions			
LBA – 33	Entry 1	Entry 2	Entry 3	Entry 4
LBA – 2	Entries 5 – 128			
LBA – 1	Secondary GPT Header			

Figure 4.1: GPT table [23]

prevent data loss and ensure the reliability of the partition table. Also, a full backup is stored in a different part of the disk, making it unlikely to accidentally get into similar scenarios as were described in the MBR chapter.

Another advantage of GPT is its inclusion of the protective MBR (a MBR table is at the beginning of the disk even if GPT is used). This can help prevent accidental overwriting of the GPT data, which can occur if the user is unaware that the disk is using GPT and attempts to create an MBR partition table on the disk. Also, it provides limited backward compatibility; older programs that do not recognize GPT will read the protective MBR and interpret the disk as having a single partition (which type would probably be unrecognized), but it should prevent the software from addressing the disk as unpartitioned [22]. The used partition flag in the protective MBR is in the case 0xEE, indicating usage of the GPT table.

In addition to the protective MBR, GPT also supports partition labels and type codes. These allow users to assign names and type codes to partitions, making it easier to identify and manage them. This can be especially useful when working with large disks that have many partitions. Finally, GPT is required for the use of certain features, such as secure boot [21] and faster boot times, which are not available with MBR. Secure boot helps protect against malware by requiring that the system only boot using trusted boot loaders, while faster boot times can improve the overall user experience by reducing the time it takes for the system to start up.

4.2 Structure

The structure of GPT can be found in Figure 4.1. As is immediately notable, there are no fields with CHS values. Protective MBR was discussed in the section comparing MBR and GPT; thus, it will be omitted here. Green and blue-marked fields are copies of the same data for redundancy.

Offset	Length	Contents
0	8 bytes	Signature ("EFI PART", 45 46 49 20 50 41 52 54)
8	4 bytes	Revision (For GPT version 1.0 (through at least UEFI version 2.3.1), the value is 00 00 01 00)
12	4 bytes	Header size in little endian (in bytes, usually 5C 00 00 00 meaning 92 bytes)
16	4 bytes	CRC32 of header (0 to header size), with this field zeroes during calculation
20	4 bytes	Reserved; must be zero
24	8 bytes	Current LBA (location of this header copy)
32	8 bytes	Backup LBA (location of the other header copy)
40	8 bytes	First usable LBA for partitions (primary partition table last LBA + 1)
48	8 bytes	Last usable LBA (secondary partition table first LBA – 1)
56	16 bytes	Disk GUID (also referred to as UUID on UNIXes)
72	8 bytes	Partition entries starting LBA (always 2 in primary copy)
80	4 bytes	Number of partition entries
84	4 bytes	Size of partition entry (usually 128)
88	4 bytes	CRC32 of partition array
92	*	Reserved; must be zeroes for the rest of the block (420 bytes for a 512-byte LBA)
LBA size	**Total**	

Figure 4.2: GPT header [23]

4.2.1 Primary GPT header

The primary GPT header is located in LBA 1 (the second logical block). It is a data structure that contains important information about the GPT itself as well as the partitions that are defined within it. A brief description of each field can be found in Figure 4.2.

The first and last usable LBAs identify the range usable by GUID Partition Entries. All the data stored on the drive must reside in this range, with the only exception being the data structures defined by UEFI to manage partitions and drive information.

A cyclic redundancy check is used to identify modifications of the header as well as the partition table. The checksum is computed by the EFI process during the startup sequence and compared to the value stored. It poses challenges for the raw GPT data editing action since the CRC would have to be recalculated each time. It is impossible to use disk hex editors to edit raw GPT data without manual re-computation [23].

If any discrepancies are detected, the GPT header is corrected by the backup header [23]. The two headers have to be the same to successfully work with the disk, the incorrect one is overwritten by the valid one. If a program changes any header entry, it is required to also change the secondary header entry. If both of them are invalid, the disk is usually corrupted, and recovery attempts have to be made to save the data.

The CRC (to be exact, the CRC32) is calculated based on the generating polynomial [24]. The hash can be computed using various programming li-

Mnemonic	Byte Offset	Byte Length	Description
PartitionTypeGUID	0	16	Unique ID that defines the purpose and type of this Partition. A value of zero defines that this partition entry is not being used.
UniquePartitionGUID	16	16	GUID that is unique for every partition entry. Every partition ever created will have a unique GUID. This GUID must be assigned when the GPT Partition Entry is created. The GPT Partition Entry is created whenever the *NumberOfPa rtitionEntries* in the GPT Header is increased to include a larger range of addresses.
StartingLBA	32	8	Starting LBA of the partition defined by this entry.
EndingLBA	40	8	Ending LBA of the partition defined by this entry.
Attributes	48	8	Attribute bits, all bits reserved by UEFI (*Defined GPT Partition Entry — Partition Type GUIDs*.
PartitionName	56	72	Null-terminated string containing a human-readable name of the partition.
Reserved	128	*SizeOf PartitionEntry* - 128	The rest of the GPT Partition Entry, if any, is reserved by UEFI and must be zero.

Figure 4.3: GPT Partition Entry [21]

braries (e.g., Boost C++ libraries [25]). Computing the hash manually would be extremely inefficient.

Having in mind all the checksums and the header redundancy, it is very unlikely to either intentionally or unintentionally alter GPT header fields to cause any significant issues with the drive. Therefore, the testing of the alteration of these fields is omitted from the scope of this thesis.

4.2.2 Partition Entry Array

All the information about partitions is stored in the Partition Entry Array. A detail of the table can be seen in Figure 4.3. There are 8 bytes reserved for each LBA entry. That allows the maximum number of blocks to be 2^{64}, with a logical block size of 512 bytes resulting in a maximum capacity in the order of exabytes, leaving plenty of room for disks in the years (and probably even decades) to come.

The starting and ending LBA pints point to the physical storage location, where the data within the partition is stored. Also, there are no PBRs in the partitions. For all the computers with UEFI, boot code and other necessary information for starting up the machine are stored in a special partition called the EFI System Partition (ESP) [21]. It is stored as a regular partition on the drive with an adequate entry in the partition table. As the drive forensics covered in this thesis do not include operating systems, it is beyond the scope of this thesis to further investigate them.

OEM, also known as the vendor partition, is another common special partition. There are usually device-specific files stored, such as firmware, drivers, or diagnostic tools.

4.2.3 Secondary GPT header

Located in the last logical sector, the secondary GPT header acts as a backup for the primary one. It contains the same fields, and for the system to work properly, the two headers must be the same. As the CRC is mainly for error detection, this header acts as the backup to restore the primary header in the case of need.

Chapter 5

Disk encryption

Physical security has been one of the bottlenecks in securing computers for a very long time. It does not matter how strong users set their passwords in the account profiles; if an attacker can just open their PC case and steal their drive, all the data would be accessible to him (of course, there are some limitations, such as separate file or folder encryption). Luckily, security evolves, and nowadays all the mainstream operating systems support some kind of full disk encryption mechanism.

5.1 BitLocker

The most widely-used full disk encryption software is BitLocker, as it is embedded in the newer version of Windows OS. This Microsoft-developed tool is for Windows users only. It uses the Advanced Encryption Standard (AES) algorithm with a 128-bit or 256-bit key to encrypt data on the disk. When BitLocker is enabled, it ciphers the entire disk, including system files, system recovery files, and user files. This means all the data on the disk is encrypted and cannot be accessed without the correct password or passphrase in a special pre-boot environment. This password is used to unlock the encryption key, which allows data on the disk to be read and accessed. If the password is entered incorrectly, the data on the disk remains encrypted and inaccessible.

Encryption works on the fly. When data from a drive is required, it is decrypted and then fed to the operating system (or any other program). There are two main points to be made. Firstly, separate storage blocks (units) of the drive have to be encrypted separately, but more importantly, decrypted separately, which limits the number of possible block cipher modes. Secondly, as the data must be decrypted (or encrypted, in the case of writing) before usage, there is a little overhead that may result in a performance drop. The fact remains that modern CPUs have a dedicated instruction set for AES encryption (at Intel, there is AES-NI) and overall hardware support for encryption

and decryption. Also, CPU speeds are much higher than storage read and write operations. As reported by many users, the overall performance impact is negligible.

The drive must have at least two partitions, one formatted with NTFS and the other with NTFS or by FAT32. The second has to be different from the OS partitions, it is not locked by BitLocker and contains files needed to load Windows. Partitions encrypted by BitLocker cannot be marked as "active" [26].

There is a recovery key attached to the Microsoft account of the user; it could be stored in his profile. AD Users and Computers has a recovery password view option; additionally, a domain controller can include all recovery passwords across a domain forest [26]. The recovery key is required when a computer detects a possibly unauthorized attempt to access data. The user is prompted by display of the BitLocker recovery screen in the pre-boot environment.

To get the most out of this tool, usage along with TPM (Trusted Platform Module) is advised. That is the special chip on the motherboard for generating and storing the encryption key and performing authentication checks on the firmware. Its advantage is that it is nearly impossible to be accessed by any malicious software [26]. It can provide an additional level of security by ensuring that the drive is still plugged into the same computer and there has been no tampering with critical system components.

The biggest problem with IT security is users. If someone encrypts their data using a weak password that can easily be guessed, it does not matter how much encryption is built on top of it. The same stands for this case, knowing the secret password, there is no problem in accessing user data. However, as BitLocker is Microsoft proprietary software, there is no easy way to encrypt or decrypt it without Windows utilities, thus it is not a candidate suited for scenarios in this thesis.

5.2 Linux

Majority of Linux systems, including Ubuntu, support LUKS (Linux Unified Key Setup) for disk encryption. The basic behavior is similar to BitLocker, except LUKS can support several passwords for decryption [27]. Other very popular open-source (or freeware) utilities are dm-crypt, VeraCrypt and TrueCrypt.

The most basic way to detect encryption is to use the *file* command on Linux, which should output any used encryption (the encryption must be recognized by the command, which is not a problem for all the common encryption types).

5.3 Partition encryption

There are several scenarios in which we want to encrypt disk partitions separately. In some cases, it is necessary due to technical reasons, e.g., in the case of dual booting. The process is the same (analogical) as for the full disk encryption.

Chapter 6

File systems

Until now, there were presented means to format a partition for a specific file system and other disk-wise utilities. This chapter is focusing on the implementation of selected file systems, their behavior, and internal data structures to keep files, directories, and free space organized. There will be some critical knowledge foundations required to successfully pass some of the forensic scenarios.

6.1 FAT32

The FAT (File Allocation Table) file system was developed in the 1970s for use on floppy disks and has since been adapted for use on hard disks and other types of storage media. Nowadays, it is mainly used for external flash disks and SD cards. There are several different versions of the FAT file system, including FAT12, FAT16, and FAT32. The main difference between these versions is the size of the files and disks they support.

6.1.1 Overview

At the heart of the FAT file system is the File Allocation Table (FAT), a structure that gave name to the whole FS its name. It is a table of blocks, stored at the beginning of a partition, that is used to track the location of each file on the disk. When a file is saved, FAT is updated accordingly to reflect the location of the file's data on the disk. When a file is deleted, FAT is updated to reflect that the blocks of disk space that the file was using are now available for use by other files.

In addition to FAT, the FAT file system also includes a root directory, which is a special directory that contains the names and locations of all the files and sub-directories on the disk. The root directory is usually located near the beginning of the partition and is used to quickly locate files and directories. An overview of the file system structure can be seen in Figure 6.1.

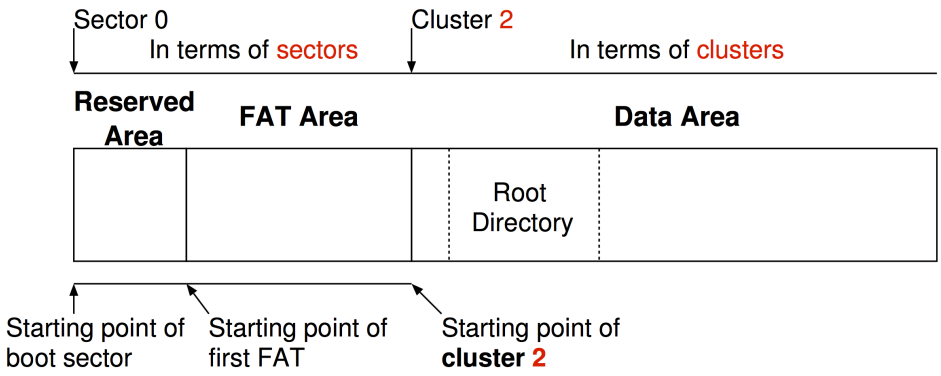

Figure 6.1: FAT32 structure [28]

The FAT file system uses a simple structure to store and organize files on the disk. It divides the disk into multiple groups of blocks (sectors) called clusters, which are the smallest unit of disk space that can be allocated and used to store a file. Each cluster is typically a few kilobytes in size and can hold a portion of a file. When a file is saved, FAT is used to track the location of each block of the file on the disk.

The simplicity of the system and ease of understanding make it widely supported on a variety of operating systems, including Windows, macOS, and Linux. However, it has limitations in terms of the size of files and disks that it can support, and it does not have many of the advanced features found in more modern file systems, such as support for file permissions or disk quotas.

6.1.2 Structure

There are three areas created in a partition formatted as FAT32. The first is reserved for the file system information, including the boot code. There is also a file system signature. The second part is dedicated to the FAT; it contains the primary allocation table and possibly even a backup table. Its size is determined by the size of individual FAT entries and the total number of clusters on the partition. The allocation table is typically duplicated for redundancy and error correction.

The final part is meant for data, here are the clusters for files and directories. Also, the root directory is located here. In the older systems, FAT8 and FAT16, the root directory was immediately after the allocation table. In FAT32, the root directory can be located anywhere in the data area [17], although it is commonly at the beginning as well.

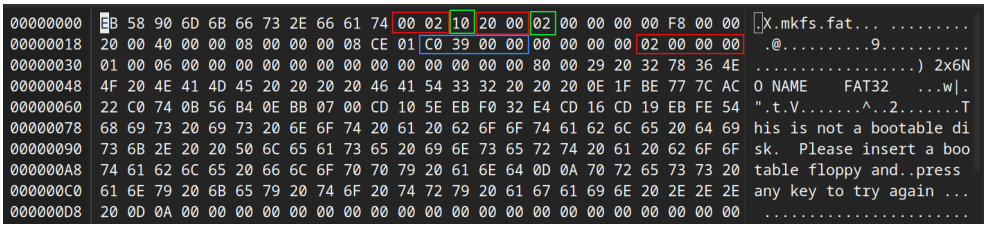

Figure 6.2: FAT32 structure in hex

6.1.3 Partition boot sector

The first part of the reserved area (first 512 bytes) is meant for a boot sector. There are several critical pieces of information about the FAT32 file system that should be known. Analyzing the whole structure is not suitable for this thesis; focus will be paid only to the fields directly connected to our forensic scenarios later on.

The complete structure of the partition boot sector can be found in several sources, e.g., in [17] or even on Wikipedia. The usage of these fields should be evident from the brief description provided. Important values in scope are highlighted in Figure 6.2 and they are also listed below (the first part is data position - byte range in hex, followed by a short description, and value from Figure 6.2, keep in mind that little endian is used):

- 0B-0C - bytes per sector - 0x0200, 512 bytes
- 0D-0D - sectors per cluster - 0x10 - 16 sectors
- 0E-0F - size of the reserved area in sectors - 0x0020, 32 sectors
- 10-10 - number of FATs used (usually 2) - 0x02 - 2 FAT tables
- 24-27 - size of each FAT structure in sectors - 0x000039C0 - 14784 sectors
- 2C-2F - root directory cluster (typically 2 - third cluster in the data area) - 0x00000002 - sector 2

6.1.4 FAT

The File Allocation Table is the structure from which the entire file system got its name. It is located after the reserved area, which size is mentioned above. The key part of this file system is understanding how the allocation table works in combination with the root directory and directory entries. The position of both can be calculated from the information in the boot sector; details are provided above.

Available storage capacity is divided into identically sized clusters, whose sizes are defined in the boot sector. A cluster is a basic allocation space for a file. For large files, more clusters may be used in a chain (linked list). Each entry in the FAT corresponds to one cluster. The number in the file system name represents the number of bits dedicated to identifying storage clusters. In the case of FAT32, only 28 bits are used; the rest are reserved. As a result, FAT32 can only support partitions up to 2^{28} clusters [29]. Cluster size depends on the size of a drive as well as the operating system used.

If a cluster is unallocated, the corresponding value in FAT is 0x0, meaning any file can allocate its content into the corresponding cluster. If the cluster is damaged, the entry value is 0x0FFF FFF7 (there are a maximum of 2^{28} clusters, which corresponds to 0x0FFF FFF6 clusters after subtracting the 11 blocks not used for data storage. Any higher value means the end of file (EOF), ending the cluster chain; more on that later.

Any other value is a link to the next cluster. If a file is too big to be stored in just one cluster and others are needed, the entry in the FAT corresponding to the first cluster of the file contains a link to the entry of the next cluster where the content of the file can be found, which can link to another cluster, and so forth, creating a chain of clusters ended by the EOF flag.

An example of FAT can be seen in Figure 6.3. The first 8 bytes are reserved (2 entries). The first data area cluster is third (cluster 2) and it is typically used by the root directory. To show its functionality, a picture was copied to the drive to show how allocation behaves in hex values. And according to our expectations, the picture needed more than one data sector, so unallocated sectors had to be linked. In this example, the allocation was linear, the fourth entry (number 3) points to number 4, and so forth. The end of the file is marked by the EOF value 0x0FFFFFFF in block number 64.

6.1.5 Secondary FAT

For redundancy, there is often a backup FAT, and values in it are updated alongside the main one. The total number of FATs can be found in the boot sector. They are located directly next to each other, right after the boot sector. In the situation remarked by Figure 6.2, there are two FATs present, each of size 14784 sectors. The first table can be found after the boot sector, which means in sector 32 from the beginning of the partition, and the second one can be found in sector $32 + 14784 = 14816$ from the beginning.

6.1.6 Files and directories

The root directory is located in the cluster defined in the boot sector (bytes 0x2C-2F), typically cluster 2. It contains directory entries for the root folder, and all the searches through the file tree start here.

```
F8 FF FF 0F FF FF FF 0F F8 FF FF 0F 04 00 00 00 05 00 00 00
06 00 00 00 07 00 00 00 08 00 00 00 09 00 00 00 0A 00 00 00
0B 00 00 00 0C 00 00 00 0D 00 00 00 0E 00 00 00 0F 00 00 00
10 00 00 00 11 00 00 00 12 00 00 00 13 00 00 00 14 00 00 00
15 00 00 00 16 00 00 00 17 00 00 00 18 00 00 00 19 00 00 00
1A 00 00 00 1B 00 00 00 1C 00 00 00 1D 00 00 00 1E 00 00 00
1F 00 00 00 20 00 00 00 21 00 00 00 22 00 00 00 23 00 00 00
24 00 00 00 25 00 00 00 26 00 00 00 27 00 00 00 28 00 00 00
29 00 00 00 2A 00 00 00 2B 00 00 00 2C 00 00 00 2D 00 00 00
2E 00 00 00 2F 00 00 00 30 00 00 00 31 00 00 00 32 00 00 00
33 00 00 00 34 00 00 00 35 00 00 00 36 00 00 00 37 00 00 00
38 00 00 00 39 00 00 00 3A 00 00 00 3B 00 00 00 3C 00 00 00
3D 00 00 00 3E 00 00 00 3F 00 00 00 40 00 00 00 41 00 00 00
42 00 00 00 43 00 00 00 44 00 00 00 45 00 00 00 46 00 00 00
47 00 00 00 48 00 00 00 49 00 00 00 4A 00 00 00 4B 00 00 00
4C 00 00 00 4D 00 00 00 4E 00 00 00 4F 00 00 00 50 00 00 00
51 00 00 00 52 00 00 00 53 00 00 00 54 00 00 00 55 00 00 00
56 00 00 00 57 00 00 00 58 00 00 00 59 00 00 00 5A 00 00 00
5B 00 00 00 5C 00 00 00 5D 00 00 00 5E 00 00 00 5F 00 00 00
60 00 00 00 61 00 00 00 62 00 00 00 63 00 00 00 64 00 00 00
FF FF FF 0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 6.3: FAT structure

For each file or directory, there exists a 32-byte directory entry containing essential data like a name, dates, and size; a full list can be seen in Figure 6.4. It also contains the position of the first data cluster, where the actual file content is stored. There is no significant difference between files and directories, as they are looked at as a special kind of file.

If a file is bigger than one cluster, additional space is needed. An unallocated cluster (FAT entry 0x0) is chosen and linked to from the FAT entry corresponding to the first data cluster of the file. That is, the FAT entry for the first cluster will contain the number of the next cluster. This way, a chain of arbitrary length can be formed. One of the possible allocation algorithms is ”next available” [17]. It allocates the first empty cluster, starting from the previously allocated one. Unallocated clusters are recognized by the allocation table entry.

The content of a directory is 32-byte long directory entries, which are stored in linked clusters. Each directory entry contains a link to the FAT value, where additional attributes can be found. Each of these entries can link to another directory, making a directory tree. The root directory is in the

Root Directory SFN Entry Data Structure

Bytes	Purpose
0	First character of file name (ASCII) *or* allocation status (0x00=unallocated, 0xe5=deleted)
1-10	Characters 2-11 of the file name (ASCII); the "." is implied between bytes 7 and 8
11	File attributes (see File Attributes table)
12	Reserved
13	File creation time (in tenths of seconds)*
14-15	Creation time (hours, minutes, seconds)*
16-17	Creation date*
18-19	Access date*
20-21	High-order 2 bytes of address of first cluster (0 for FAT12/16)*
22-23	Modified time (hours, minutes, seconds)
24-25	Modified date
26-27	Low-order 2 bytes of address of first cluster
28-31	File size (0 for directories)

File Attributes

Flag Value	Description
0000 0001 (0x01)	Read-only
0000 0010 (0x02)	Hidden file
0000 0100 (0x04)	System file
0000 1000 (0x08)	Volume label
0000 1111 (0x0f)	Long file name
0001 0000 (0x10)	Directory
0010 0000 (0x20)	Archive

** Bytes 13-22 are unused by DOS*

Figure 6.4: Directory entry [30]

known location, and all the files and directories are accessible by walk-through using the file or directory name as the key.

6.1.7 File creation

Let's say we want to create a file in a specified directory. Firstly, the root directory is located using sector boot data. Following a link to the corresponding cluster, individual directory entries are looked at to find the name of the desired directory. This process continues until we get to the directory we want to create our file in. If any directory along the way does not exist, we can create it first by adding a new directory entry to the parent directory. It is added to the end of allocated space or in the place of a previously unallocated file. If there is no free space left, another cluster is added to the chain, and the process remains the same.

With the new directory entry (which entries/clusters are filled in depends on the operating system implementation), an unallocated cluster is chosen from the FAT, and its status is changed to EOF. The content of the file is written to the cluster. If more of them are needed, the next unallocated cluster is marked with EOF, a link to it is added to the first FAT entry, and the file is divided into several parts (logical, it can still be continuous data area).

6.1.8 File deletion

Analogically to file creation, the file deletion process starts with locating the directory and FAT entries of the file to be deleted. A walk through the cluster chain is conducted till the EOF is found. All the processed clusters are marked as unallocated in FAT (the value is set to 0x0). Finally, the directory entry is unallocated. That is done by setting the special flag to the first byte of the entry (the short file name), which is usually 0xE5 [17].

Deleted files can be recovered as long as the directory entry or cluster is not overwritten, which depends on the allocation algorithm. Also, other procedures can force deleted files to be unrecoverable, such as defragmentation or wipe deletion (deleting the data by overwriting it with random values of zeros).

6.2 Ext4

Ext4 (Extended File System) is the most widely-used file system on many Unix-like and Linux operating systems, including popular distributions such as Ubuntu, Manjaro, and others. Windows support is limited; for example, default disk management utilities have difficulties recognizing it and refuse to format drives with ext4 at all. The file system itself is generally the same as the two previous versions, ext2 and ext3, only some features have been added; no major redesign was made.

6.2.1 Structure

At the beginning of a partition formatted as ext4, there is a space available for the boot code, in particular 1024 bytes. This does not have to be used by boot code and can contain hidden data [31] [32]. The remainder of the drive is divided into groups of blocks. At the beginning, there is a superblock with some important information about the file system. For redundancy, a copy of the superblock can be replicated to multiple block groups [3], which can help in the event that the beginning of the disk is corrupted.

Next in order is the group descriptor table, a list of group descriptors with 32 or 64-byte entries for each group block in the file system. It may be replicated in multiple group blocks alongside the superblock to provide redundancy. In terms of blocks, they are the default space allocation unit (if fragmentation is ignored), and their typical size is 4 KiB [33].

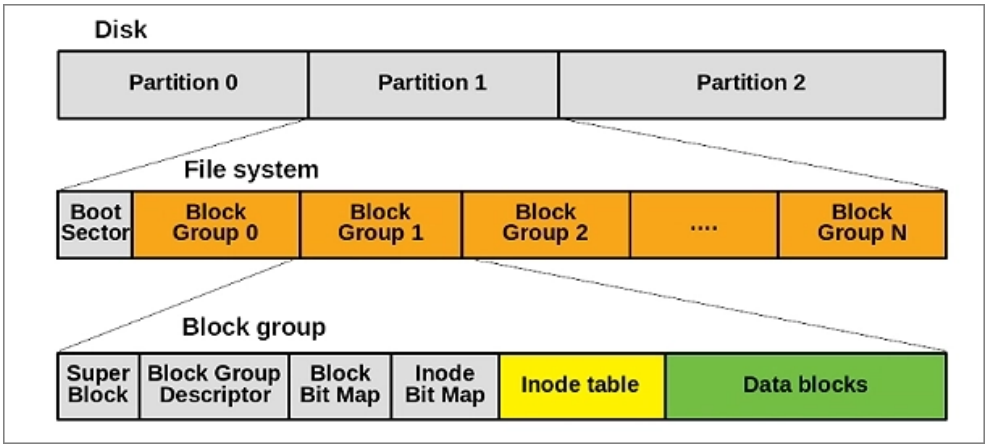

Figure 6.5: ext4 structure [34]

After some reserved blocks for possible future group descriptors, there are two bitmaps. A bitmap is a bit array, with each bit corresponding to a different item. It can hold two values for each area, either zero or one. In this case, the Data Block Bitmap holds information about the allocation status of individual data blocks within the block group. The second one is the Inode Bitmap, which keeps of inode allocation status. Then, inode tables and data blocks follow. A structure overview is displayed in Figure 6.5.

6.2.2 Block groups

In ext4, storage space is divided into blocks (and fragments, but for simplicity, they are not covered in the scope of this thesis, meaning fragment size is considered the same as block size), which are just consecutive drive sectors. The typical block size for ext4 is 4 KiB [35]. Blocks are then joined into same-sized block groups; their amount depends on the specific implementation. This number can be artificially adjusted by the file system creator, e.g., with option *-g* in the *mkfs.ext4* command on Linux.

Typical block group structure can be seen in Figure 6.5 too. The only positions that are strictly defined are superblock and block group descriptor. The position of all the other areas is defined inside the group descriptor, as are free blocks (and inodes), checksums, and others.

6.2.3 Superblock

After the first 1024 bytes of the partition, the first block group begins. At the beginning of it, there is a superblock of the same size. A superblock is a data structure that contains important information about the file system as a whole, such as the size of the file system, the number of inodes (data structures

that store information about files and directories), block size, the number of blocks (units of storage) in the file system, as well as features supported.

A superblock is stored in a fixed location on the file system, and there is a copy of it at the beginning of multiple block groups. This allows the file system to recover from certain types of errors or corruption by using one of the copies of the superblock as a reference.

As the superblock is pretty huge, there is no need to describe it in its entirety, only the main fields relevant to our scenarios are listed; a full list can be found e.g., in [17]. The relevant ones for our scenarios are:

- 12-15 - number of unallocated blocks
- 10-13 - number of unallocated inodes
- 14-17 - starting block of block group 0
- 18-1B - block size
- 20-23 - number of blocks in a block group
- 28-2B - number of inodes in a block group
- 54-57 - first non-reserved inode in the file system
- 58-59 - size of the inode structure
- E0-E3 - journal inode
- E8-EB - head of the orphan inode

6.2.4 Block Group Descriptor Table

Right after the superblock, there is a group descriptor for each block group in the whole file system. It is either a 32 or 64-byte data structure that contains information about the block groups, such as the location of the group's blocks and inodes on the file system and the number of free blocks and inodes in the group. This structure is replicated into the various block groups alongside the superblock.

Its size can be computed by dividing total number of blocks by the group block size to get number of group blocks. For each group, there has to be an entry, so space occupated can be computed by multiplication of the entry size and their number. It is always aligned to full blocks.

6.2.5 Bitmaps

The allocation status of all the blocks in a group is kept in the block bitmap. The corresponding bit for each block can be found by finding the relative position of the block to the start of the group. Similarly, the inode bitmap keeps track of the status of the allocation of individual inodes. In both cases, 1 means block/inode is in use, 0 that block/inode is available for allocation. For each bitmap, there is a full block reserved.

Regarding HDDs, block and inode allocations were created with the highest locality property possible, meaning the disk would not have to spin much [17]. Even for data transfers in modern SSDs, the close locality has an advantage concerning cache hits. In practice, that means related blocks are stored as close as possible. There are several tricks to achieve that. One of them is not to store data on the disk immediately but to wait till all the write operations and dirty flags are resolved to more accurately assess the drive capacity requirements for the file. Also, inodes are ideally placed in the same group as the data and the directory, assuming they are related and could be processed at the same time. The exact algorithmic details are OS-dependent and are out of the scope of this thesis.

6.2.6 Inode table

Inodes have existed since the early days of Unix [36]. It is a data structure of a typical size of 256 bytes for describing files (a directory is a special type of file). There are multiple metadata fields, such as timestamps, link counter, permissions, and data type; the complete structure can be seen in Figure 6.6. There are traditionally some reserved inodes, e.g., the root directory (inode 2), journal (inode 8), and others. Typically, the first non-reserved inode is number 11.

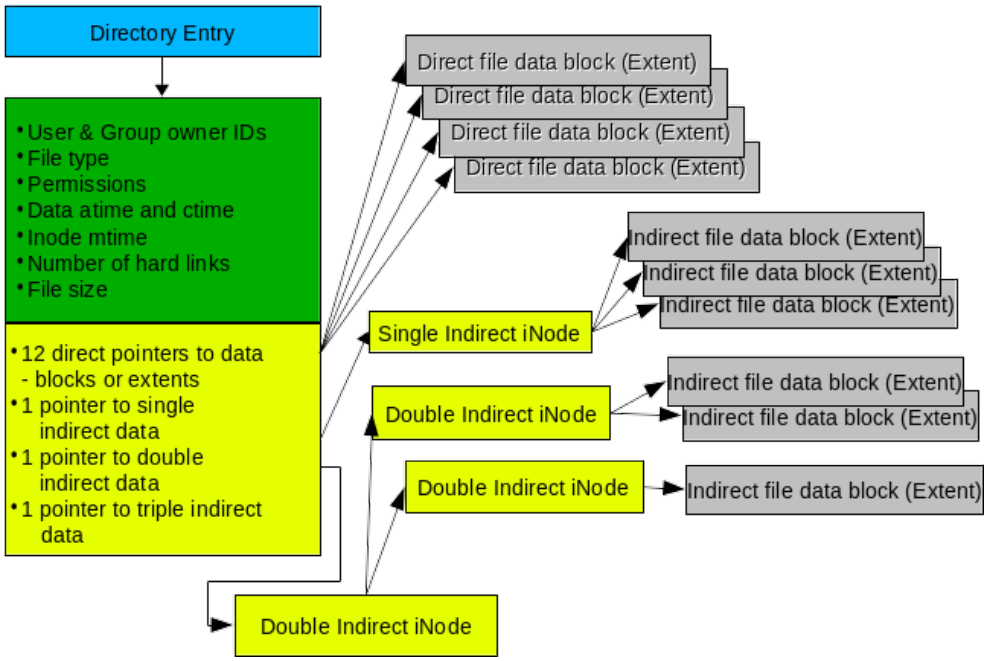

Figure 6.6: inode [37]

Most importantly, there are links to data storage, where the actual file content is stored. The file system of Unix as well as its successor, ext4, were both designed to work effectively with small files [36]. Data space can be addressed either directly or indirectly. A direct link means it contains the address of a block where the file content is stored. Indirect, on the other hand, points to a block where direct pointers are stored. There are also double and triple indirect pointers, which link to data blocks where indirect (respectively double indirect) pointers are stored. There are 12 direct pointers, so it allows storing files up to 48 KiB directly. Additionally, one indirect, one double indirect, and one triple indirect pointer are available.

Pointers are 32 or 64 bits in size, depending on the architecture. Assuming 32-bit size, 4 KiB block can hold $4 \cdot 2^{10}/4 = 2^{10}$ addresses, each pointing to 4 KiB blocks. It is very efficient for small files, as most of them can be saved by direct pointers only. However, storing large files in this manner is inefficient because a large amount of metadata must be retained. This traditional design used in ext2 and ext3 was replaced by extent addressing [38]. Instead of individual blocks, a continuous range of physical blocks is used, and only the border values are kept. Extents are organized into a tree; leaves represent individual ranges of blocks covered.

6.2.7 Directory entry

As can be seen in Figure 6.6, inode does not contain a filename. The directory entry is a structure that maps a filename to its inode. It has a dynamic size as the length of a filename can be in the range of 1 to 255 characters, and size is rounded up to the 4 byte boundary. A directory is a file in which data blocks consist of directory entries. The first two entries are "." (current directory) and ".." (parent directory). Because of the possibly variable filename size, each entry contains a link to the next directory entry. When deleting an entry, simply remapping the link to the next entry would do [17]. In the opposite case, link offsets are compared to actual name lengths to determine if there is a free space that can be used, which also depends on the previous filename length because the new entry would have to fit in the old frame.

Multiple directory entries can point to the same inode; they are called links. Two types of links exist: hard ones and soft ones. A hard link is a directory entry that has a pointer to the inode of an existing file; the counter of links in that inode is incremented. The inode is unallocated after all the links pointing to it are deleted. The second type, soft links, work differently. Instead of linking directly to the same inode, they have their own, which points to the original directory entry. Therefore, when the original file is deleted, the soft link remains but refers to unallocated space.

A walk-through is used to locate a specific file or directory. An inode of the root directory is well known; it is the second inode (number 2, counted starting with 1). From there, we can follow a path of directory entries, looking for the desired name. That is not a very efficient way to handle unsorted lists; thus, the second option exists. Starting with ext3, hash trees are used to help directory processing. If the directory content occupies more than one block, a tree is created to sort all the saved entries. Each filename is hashed, and a search is performed in the tree, where all the hashes are stored and sorted for quick searching.

6.2.8 File creation

In a file system, creating a file is a common task. In ext4, the process is as follows. Firstly, the directory to put a file in has to be found. From the superblock, information about block sizes, reserved space, and other important data can be obtained. We proceed to the group descriptor to get the layout of each block group. The location of the file has to be found. Following inode 2, a search from the root directory down the file system is performed. That is, processing each directory entry, comparing its name, and, in case of success, following the link to its inode.

When the desired destination is found, we process the directory to find a free spot for a new directory entry. If there is none, a new block is allocated and added to the inode chain of the directory. After the directory entry is

created, a corresponding new inode is also created. According to the file size, the corresponding allocation scenario is used; data blocks are linked from within the inode, and the file content is written to these blocks.

6.2.9 File deletion

In the same fashion as in file creation, we locate the desired directory entry. It is unallocated by setting the pointer of the previous entry, from the file to be deleted, to the next file/directory. Then, the linked inode is updated, and the number of links is decreased. If it reaches 0, a deallocation procedure occurs, which includes setting the corresponding bitmap entry to 0 and updating the free inodes number in the superblock. Deleting data blocks is performed by clearing bits in the bitmap as well.

In some cases, deleted files can be recovered. Iterating over directory entries, we can compare file names with the length of the jump to the next entry to determine if there are any hidden (deleted) entries. If there is any, it should still contain a link to the old inode. As for the inode, it should stay in the same place as it was until it is overwritten by a new one. If we are lucky, the inode was has not been overwritten yet. In the inode, data block links can be found, and we can try to recover the file content. We can also skip the directory entry phase and look for unallocated inodes or data blocks directly through the bitmap values.

There is a problem locating unallocated data blocks. As was pointed out in [39] and [3], starting from ext3, inode data links are set to 0 after the inode is unallocated. That can aggravate the whole process. In this case, there are several options for how to proceed. The main one is to go through all the unallocated data blocks and look for their structure and try to find a file signature.

6.2.10 Journal

Apart from ext2, ext3, and ext4 support journaling. Any data operation is not written to the disk until it is finished in its entirety [40] [3]. This helps the recovery process after a system crash (file system is changed in transactions, they are accepted as a unit or not at all), and also it is effective for data storage, as data are written only after all the operations are finished allowing a better space management. All the changes can be viewed as transactions, and the journal stores them along with their order until the commit command is issued, meaning all the temporary changes are performed and it is cleared from the journal. Thus, it can be used to recover recently deleted files and revert other recent changes.

6.2.11 Orphans

An orphan inode is an unallocated inode that is opened in a process. If the pointer counter inside it drops to zero, the inode should be deleted. If a process is still using the inode (e.g., a cleaning procedure), it cannot be deleted right away. Therefore, it is marked and stored as an orphan, and it is cleared just after the process stops using it.

When a inode is marked as "orphaned", it is added to a list of orphan inodes [35]. The blocks used by the file are freed, so that they can be used by other files. Periodically, the system will scan the list of orphan inodes and try to delete them. If the inode cannot be deleted, it remains on the list until a later time when the system can try to delete it again. Orphan inodes can also be deleted manually by an administrator using the *e2fsck* utility, which is a tool for checking and repairing ext4 file systems.

6.3 NTFS

NTFS, or New Technology File System, is a proprietary file system developed by Microsoft for use on personal computers running the Windows operating system. It is the successor to the FAT (File Allocation Table) and HPFS (High-Performance File System) file systems, which were used on earlier versions of Windows. As the system is proprietary, not all the implementation details are known [1].

NTFS was introduced in 1993 with the release of Windows NT 3.1. It was designed to address the limitations of the FAT and HPFS file systems, providing support for larger volumes of data, longer filenames, and advanced features such as file compression and encryption. Over time, NTFS has been updated and improved, and it is now the default file system for all versions of Windows, including Windows 10 and 11.

6.3.1 Structure

At the beginning of the partition formatted as NTFS, like in the cases of the previous two file systems, there is a boot sector containing some crucial information about the file system. One of them is the location of the MFT (Master File Table), an array containing entries for every file and directory in the file system. It does not have a predefined size and can be scaled to almost arbitrary sizes [17]. Similarly to FAT, disk sectors are joined into clusters of a typical size of 4 KiB. File details are stored in the form of attributes, some of them are listed later. Each directory contains a list of its sub-directories and files. These are stored in B-trees to allow quick searches for file names [41]. In NTFS, everything is looked at as a file, sometimes a special type of file.

6.3.2 Boot sector

The first 512 bytes of the partition are reserved for the partition boot record (PBR). Some essential information is stored there, including bytes per sector, sectors per cluster, the position of the $MFT file (0x30), and the position of the mirror $MFT (0x38), for backup of the most essential data.

6.3.3 MFT

From the boot sector, we know the position of the $MFT file. In its $DATA attribute (type 0x80), sectors used by MFT are stored. That way, we can retrieve the full MFT table. Another useful attribute is a bitmap keeping track of the allocated and unallocated MFT entries. MFT is an array of file records. Each file has at least one MFT entry assigned. The first 16 file record segments are reserved for special files, such as $MFT file (0), the root directory (5), and the bitmap for clusters (6), see Figure 6.7.

Each file entry is 1024 bytes in size, its structure can be found in [42]. Big part of the entry is designated to the file attributes. Each attribute has predefined header [43]. It includes, among others, attribute type (some of the most important are $STANDARD_INFORMATION - 0x10, $FILE_NAME - 0x30, $DATA - 0x80, and $INDEX_ALLOCATION - 0xA0), its size and non-resident flag. After the header, structure is different for each attribute, refer to [44] for details. The size is defined in the header so it is easy to iterate over all the attributed to find the desired one.

There are two types of attributes: resident and non-resident. Their difference can be described as in-place and out-of-place attributes; resident ones are stored directly in the MFT entry, while non-resident attributes are stored separately in a different cluster and are linked to from within the entry. This link is implemented in the form of data runs [45]. Each data runs contains one byte header - first 4 bits are used to determine offset length in bytes, the other 4 are for size information length in bytes. Following the header, there is the attribute length in clusters. After that, offset in clusters, where the attribute is stored on the partition. There can be multiple consecutive data runs, each with the same structure.

Mirror MFT is stored on another part of the disk to avoid potential first-sector corruption; its position is saved as the second MFT entry. Usually, it is located at the end of the partition.

6.3.4 Directory entries

Searching for the file name in the whole MFT to find a specific one would be ineffective. Therefore, there is a directory structure containing INDX ($INDEX_ALLOCATION attribute) [43], linked to by MFT entry 5, which represents the root directory. Inside, all the sub-directories and files are stored by their names. A structure used for the organization is a B-tree, a self-balanced

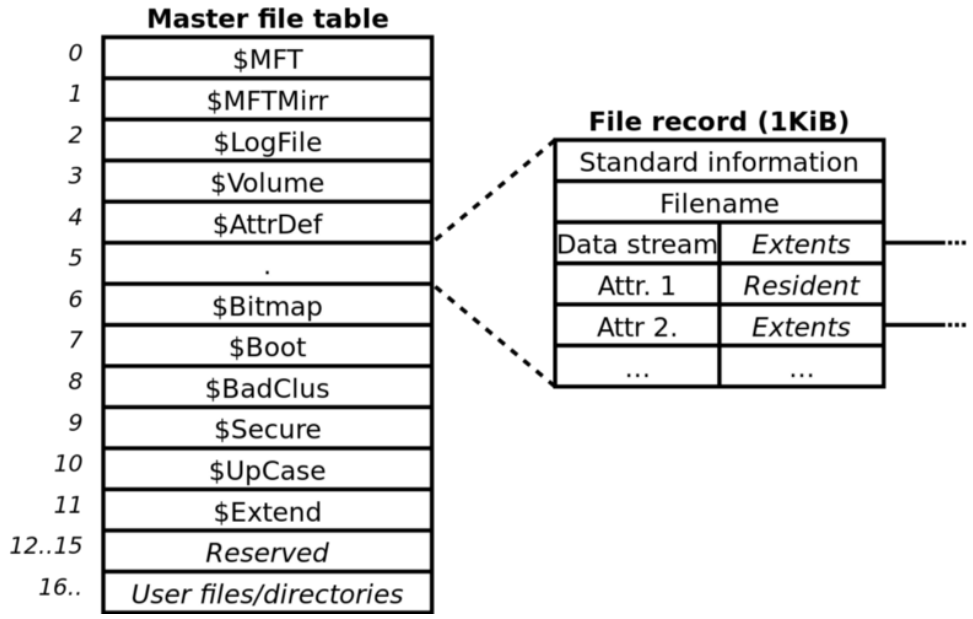

Figure 6.7: MFT Structure [46]

data structure; simply put, it is an ordered tree with as many possible children as values stored in a node plus 1. In this manner, the order can be maintained and searching can be completed quickly. In the leaves, links to MFT records are stored, so an association with the file content is very straightforward. Recursively, sub-directories have their trees in their attributes.

When a hard link is created, no entry is added to MFT. Instead, a new record is inserted in the directory tree, which refers to an existing MFT. This procedure is very similar to that of the ext4 file system.

6.3.5 File creation

If we want to create a new file, firstly, the boot sector has to be processed to determine cluster size, location of the MFT, and size of MFT entries. From $DATA attribute of the $MFT file (the first entry), we get additional information about the layout of the MFT. The next step is an allocation of a new MFT entry, free space is determined from the bitmap stored in the $Bitmap file; an entry is created, and metadata are filled in. Based on the file size, the necessary clusters are allocated (again, free clusters are known from the $Bitmap). The allocation method is usually the best fit; clusters are selected for the highest possible storage efficiency [17]. Links to the data clusters are saved inside the $DATA attribute.

The next step is a filename entry. In the root directory (MFT 5), there is a B-tree organizing all the existing names. The file's destination directory is

found in the tree, and its record is added to the tree there. It may be necessary to perform tree balancing to keep things organized and to keep search times quick.

NTFS supports Alternate Data Streams (ADS) [29]. A data stream is an attribute of the MFT entry. The unnamed data stream, which can be viewed in file explorer, is used by default. Named data streams are hidden from common users; they are not even reflected on the file sizes in the explorer. There are several legitimate purposes to have data associated with files, but unfortunately, it has become one of the favorite places of hackers who use them for malicious purposes.

6.3.6 File deletion

Let's examine a file deletion scenario. As in the file creation process, we locate all the needed structures. The file is located by a walk-through in the root directory and finding its associated MFT record. It is unallocated by clearing the corresponding bit in the bitmap and clearing its non-residential attributes in the same way. No shrinkage of MFT occurs, meaning it can only grow in size [17].

Recovering deleted files is possible as the data are not overwritten; they are only marked as unallocated and can be recovered. The only thing removed is the filename and MFT link in the parent directory index, and the tree could be re-sorted, which can lead to information loss. However, all the unallocated entries are in the MFT table, and they have their name as an attribute, so even the full path can be recovered. Corresponding MFT entry would have to be found manually by an educated guess.

6.3.7 Journal

In a very similar fashion as in the ext4 file system, the journal is used to keep the file system in a consistent state. Recent actions are recorded in the journal and are finished when all the associated activities are resolved in a transaction-like manner.

6.3.8 Shadow copy

Volume Shadow Copy Service (VSS) can be used for data backup. Copy-on-write is one of the mechanisms [47]; when data is to be written to storage, a copy of the original data is stored to allow roll-back. As it stores only changes in data, it is effective for low-fluctuation systems. When changes are more frequent, this method becomes quite expensive. Shadow copies are kept in a special storage area at the end of the partition, after the data area.

6.4 File system independent operations

There are multiple operations that do not depend on the file system used. That means the situation is the same or analogical for all the common file systems, assuming the same tools are available. Almost all the file level operations belong here; the actual content of files is not affected by the storage mechanism. Such operations include data hiding (e.g., steganography), file encryption, compression, and others. Also, data carving belongs to this category, as it can be performed on any disk, regardless of the file system used.

6.4.1 File encryption

File encryption is a technique for protecting data by encoding it in such a way, that it can only be accessed by someone with the correct decryption key. File encryption is typically implemented at the file level, so it is independent of the file system that is used to organize the data on a storage device.

This means that the same file can be encrypted and decrypted on any file system, as long as the necessary encryption and decryption tools are available. For example, a file that has been encrypted on a Windows system using the NTFS file system can be transferred to a Linux system using the ext4 file system and decrypted using the correct decryption key.

It is worth noting that some file systems, such as NTFS and ext4, support native encryption capabilities, which can be used to encrypt individual files or entire directories. In these cases, the encryption is still independent of the file system, but the encryption and decryption processes are handled by the file system itself rather than a separate tool.

Let's look at how file encryption works using the GnuPG (*gpg*) program, which comes standard with most Linux distributions [48] and can also be used on Windows. There are multiple modes of operation; the basic functionality is symmetric encryption, which proceeds as follows: The program takes a file as a parameter and prompts the user for a passphrase. After it is entered, a new encrypted file is created. It can be identified, e.g., by the *file* command as encrypted by *gpg*. The physical structure of the file is well defined in RFC 4880 [49] and contains all the information necessary to successfully decrypt the file when a user knows the password.

6.4.2 Data carving

Data carving is a digital forensics technique that involves searching through a storage device or file for specific patterns or data structures and extracting the data from those patterns or structures. Data carving is often used when traditional forensic techniques, such as analyzing the file system, are not possible or have been unsuccessful in recovering the data.

There are several approaches that can be used for data carving, depending on the type of data being sought and the conditions of the storage device or file. One common approach is to use a hex editor to search for specific patterns of bytes that are known to be associated with certain types of data, such as image or video files. These patterns are called the magic numbers [29]. The data can then be extracted by copying the bytes from the hex editor and saving them to a new file.

Another approach is to use specialized data carving software, which is designed to search for and extract specific types of data based on their known patterns or structures. These tools can be configured to search for specific file types, such as JPEG images or MP3 audio files, or to search for data structures that are common to a particular file format, such as the header and footer of a PDF document.

Data carving can be a useful technique for recovering data that has been deleted or lost due to file system corruption or other issues. However, it can also be a time-consuming and resource-intensive process, as it requires a thorough understanding of the data structures and patterns being searched for.

Chapter 7

Digital Forensics

Digital forensics is a process of analyzing digital traces to identify or reconstruct the events that led to them [1]. Most common folks have it connected to criminal investigations, as it was popularized by many TV shows. Indeed, digital forensics can play a big role in finding evidence and presenting it to court in legal matters. But that is not the only usage. In incident response scenarios, which rely on identifying attack vectors to avoid future problems, forensics are key elements to detect when and how the incident occurred. Not all applications of digital forensics have to be connected with wrongdoing. There could be, for example, an unexpected behavior of a valid program causing problems in the system or an accidental intervention by the user causing system instability, among other things.

The analysis' main goals are to preserve digital evidence integrity (original data must remain unchanged, especially if evidence is to be presented in court), correctly document and identify all data used in the process, and ensure that the analysis results are undisputed [1].

There are many sources of data used in the analysis. Not just computer disks or similar media, but also network data or physical access to the device can prove to be crucial parts of an investigation. To be able to properly perform the analysis, deep knowledge of the systems in question is needed. That is not useful just for the analysis, but this information can be used in, for example, the software development of recovery tools, incident response automation, and others.

7.1 Typical process

Every investigation is different; each case has its specifics, and it is hard to generalize the process of digital forensics in much detail on the technical level. On the other hand, from a methodology perspective, six steps have to be conducted in almost every case (of course, deviations exist, but in this case,

they are rather rare). These steps are not strictly separated; in fact, they mingle and coexist.

7.1.1 Preparation

Reconnaissance is an essential part of the process. When an investigator gets to the crime or incident scene, he/she should be as aware of the situation as possible. What happened, when, how, who did it, who detected it, what is affected, and why it happened are some of the most important questions. Not all of them can be answered instantly, but the information is key to success, and getting some of the answers immediately can significantly speed up the whole process.

Also, time at the incident scene is not unlimited. It may be a good idea to prepare in advance. Bringing along hardware equipment to acquire data is one of the tasks, work organization and scheduling are also important to allocate manpower and work efficiently [1].

7.1.2 Data acquirement

It may sound easy to obtain relevant data - just get everything from every workstation and from every server near the incident and analyze it. That would be way too much data. Only relevant data should be gathered; it is a really slow process to go through terabytes and terabytes of unrelevant storage. On the other hand, no information should be omitted. It is important to keep in mind that some of them are time-limited (e.g., network logs, which can be overwritten fairly quickly, volatile memory,…). Specialists are entrusted with identifying which specific data is case-relevant and should be gathered.

7.1.3 Preservation

In court, a defendant would gladly use all the possibilities to deny evidence being presented against him. Either by pointing out possible evidence tampering or investigator bias. Original data must be preserved, ideally by computing hashes and control sums to demonstrate integrity. All the work is typically done on a copy of the original data, which is sealed and available in case there are any objections or doubts about the analysis process or whether the data were processed according to criteria.

7.1.4 Analysis

When the data is obtained and its integrity is secured, an analysis can be performed. There are multiple automatic tools (briefly mentioned at the beginning of the thesis) that can help, but in some cases, it needs to be analyzed by hand. Each step has to be properly documented if we want results to be reliable. Anyone should be able to replicate the investigators' conclusions by

taking the steps described from the original data. Also, keep in mind the unwritten rule of computer security; work that is not documented is not done.

7.1.5 Reporting

The final results are presented. It can be evidence in court; it can be a PowerPoint presentation to the company's management about the incident; it can even be a bug report to software authors. The only way for humanity to move forward is by learning from its own mistakes; this phase is a great opportunity for realizing that and passing the knowledge obtained along.

7.2 Digital forensics tools

Many different tools that are used in digital forensics, and which one to use depends on the specific needs and requirements of the investigation. Some common digital forensics tools include Magnet Axiom, EnCase, FTK, X-Ways Forensics, and Autopsy. These tools are widely used by forensic investigators to analyze data from digital devices and to extract and present evidence in legal proceedings. They often provide multiple functions, which is why they have already been covered in the disk image cloning section earlier.

7.3 Anti-forensics techniques

There are multiple way to defend against digital forensics techniques. The first and most important one is disk encryption. When a disk is completely encoded by a strong passphrase, options for investigators are extremely limited.

Another example of an operation that can be handled in a forensically secure manner is file deletion. Normally, deleting files from a file system leaves behind traces such as metadata, pointers to unallocated space, and others. There are a number of wiping tools, which not only unallocate the space used by a file but also overwrite its data blocks with zeros or random values, leaving no information for investigators to look for.

Data hiding is another technique used. There are numerous places to hide data so that it cannot be easily found [50] [31]. Many of these places are well-known, and automatic tools can find the data inside them. Apart from traditional spaces, like in the partition boot sector, there are also new techniques, such as uninitialized block groups [32].

There also exist many anti-forensic tools for special-purpose manipulation. File system timestamps can be altered to confuse the investigators. To prevent data carving techniques, one can manipulate file signatures to mislead the investigators [51].

Chapter 8

Disk generator tool

The main goal of the practical part of this thesis was to create a tool to generate disk images for selected digital forensics scenarios that could be used in training. To my knowledge, there is no such tool available. There are many testing images available online, but they are not feasible for scalability and creating new scenarios, each image would have to be created manually. It is impossible to create a generator of all possible scenarios, so only a few selected ones were implemented. Application design allows others to implement their scenarios and add them to the tool.

8.1 Application Design

At its core, the application needs to create a disk and then manipulate it to create a desired scenario. Therefore, it is reasonable to use existing tools for the disk creation part as well as for the alteration process.

The application functions as a wrapper for basic Linux commands, such as *dd*, *fdisk*, *sfdisk*, *mount*, *mkfs*, and others. Details can be found in the source code. These commands are open-source and distributed under the GNU GPL license. One of its key points is copyleft; thus, this program also follows the GNU GPL license. Details can be viewed on the enclosed CD or zip file.

As the Linux OS is written in C, and so are the tools used under the hood of this program, the C/C++ programming language was selected for implementation. Application design as well as implementation details can be viewed in the enclosed source code. There are comments explaining the core functionality. Also, it is believed that the code is easy enough to read that anyone wanting to develop an extension or scale the application would not have a problem understanding the program and doing so. Also, program usage from the user perspective is rather intuitive, and instructions are displayed on the screen during the process. Note that the program is not fault-proof in terms of a user trying to break its functionality, as it should be used in good faith only.

8.1.1 Program structure

There were several design choices to be made in the process of creating this thesis. As a result, there were two program versions in total, both of which are described below. As the first one did not fulfill the requirements and objectives for its functionality and scalability, the program was redesigned to address these discrepancies. The description of the first version is intended to enlighten some changes and design decisions, specifics and more detailed documentation are provided only for the current version.

8.1.2 Earlier version

The first version of the program was written as a proof of concept, creating scenarios and investigating possibilities of implementation. As such, it was written as one file only, containing the function *main()* and several supporting functions to delegate tasks to smaller units.

There were no classes implemented; a specific scenario was generated based on the user input into the console, which was processed in the *switch()* branches. From them, support functions were called to at least limit code redundancy. Several of them were supporting, with no direct impact on the generation process itself.

As for the abilities of the first version, it supported five scenarios (the first five of the current version, details are written in the section below). Creating a scenario means creating a disk image with wrappers (mostly the command *dd*), formatting it with partitions (commands *fdisk* and *sfdisk*), and then altering it in some way.

To be able to see if the goal of a scenario was achieved and the problem was fixed, there had to be some indication of success. For this reason, pictures were inserted into these disk images, and the goal of each basic scenario would be to recover the image(s). In order to do that, file systems had to be added to the disk images. This proved to be a significant problem. When designing a program, it is always a good practice to follow the least-privilege principle. A program should be running with the least amount of privileges, only with those necessary for its functionality.

There are tools for editing file system on a disk image for FAT32, specifically *mtools* library [52]. However, there are limitations to managing file systems ext4 and NTFS without mounting them. It is possible to create an arbitrary directory in the newly created ext4 file system (an option when using the command *mkfs.ext4*), but copying files and other tasks are not easily achievable. NTFS is even more complicated, options are very limited without mounting the disk image.

The mounting process requires superuser privileges. Following the least-privileges principle, the first version of the program avoided using mounting

so it could run with only basic privileges. That meant not using NTFS and only limited usage of ext4 file systems in the scenarios.

Further problems arise when deleting FAT32 files using *mtools*, it proved to be problematic and caused failures. To address this issue and allow recovering a deleted file as a possible scenario, I implemented the file deletion process as it was described in the theoretical section earlier, manually overwriting data at the byte level.

8.1.2.1 Necessary improvements

The application, by design, was not easily scalable. Adding further *switch* branches and adding more and more functions would soon make source code unreadable. Therefore, a complete redesign was necessary. Also, with classless design, there were no rules or standards that would have to be followed when adding new functionalities.

It was also necessary to add some options to the application. Limitations for only one file system seemed not enough; there was also no information available about the implemented scenarios from within the program; their description was only provided as an attachment.

Having these in mind, a new program version was created to address these issues and bring more clearance and tidiness. Details of the current implementation are described in the following section.

8.1.3 Current version

The first program version was rewritten in the object-oriented paradigm, which gave birth to the current one. There are several significant improvements and additions to the original program to make the new one more usable.c Also, several new scenarios were added.

8.1.3.1 Class Schema

As for each C++ program, the entry point is the function *main()*. This file includes only code for printing the starting screen (clearing the terminal window and printing the introduction and main choice menu). After grabbing input from the console, if the user's option is to exit the program, then the execution ends here. For any other choice (including invalid input, which is not validated at this point), an instance of *DiskImage* object is created, and control is handed to its method *loop* with the user option as its argument.

DiskImage class is the core of the application. Based on the option selected by the user, method *loop* validates the user input. It is either an interactive creation of a new disk image or a creation of one of the predefined scenarios. Disk creation is included in the process of creating scenarios.

User choice determines which scenario is created. Class *Scenario* provides API for them, as it is the abstract parent class. Each scenario is required

to override method *generateScenario(DiskImage*)* to create a training disk. Other class member functions are *getDescription()* to provide a quick summary of the task, *getHint(int hintNum)*, to provide the selected hint, and *getHintsTotal()* to get the number of hints available. As these three are "getter" functions, their implementation is provided by the parent class, and the actual data returned by them is initialized by the constructors of the child classes. There was a hesitation about including function *getSolution()* to reveal solutions of scenarios but based on the complexity of later scenarios and consistency across all of them, it was decided against it.

The specifics of each scenario, as well as their implementation details, are covered in the later section and in the file *scenarios.md* enclosed. What is common for all of them is the creation of disk images and working with a file system. So let's get to this first. A disk image is created by the function *createDiskImage(std::istream &)*, which is a member function of the class *DiskImage*, by calling the system command *dd* to create a file of the desired size, either filled with zeros or with random data based on the option selected. The next available selection is the partitioning table, which is created by the system command *sfdisk* with the option either dos for MBR or GPT, support of GPT is limited and has not been tested properly, it is not used in the scenarios.

The next in order is creating disk partitions, which is done by the wrapper of commands *fdisk* and *mkfs*. The user can specify the size and file system. There is currently only support for primary partitions; working with extended ones can be added in the future. There is a check implemented to ensure partitions fit on the disk; if the selected partition size exceeds the disk capacity, the volume is shrunk to fit it. The parameter *std::istream &* is to allow the creation of scenarios. For interactive situations, when input is handed interactively by a user through the terminal, that would mean *std::cin* as the parameter. However, for pre-defined images, it is more convenient to use *std::istringstream* with firmly defined input.

Speaking of file systems, the abstract class *Filesystem* defines their API. Specifically, all the file system implementations are required to override functions *createFS(...)*, which creates a file system on a given partition, *copyFile(...)* to insert files into the file system structure; and lastly, *deleteFile()* to delete a given file. There were several significant choices to be made as mentioned before. Working with NTFS and copying and deleting files on ext4 requires running the application with elevated privileges, but as the program is not that extensive, everybody should be able to trust that it does not try to do any malicious things.

The implementation of FAT32 is a little different story, it stayed mostly the same as in the previous version; there was no reason to change it. That means the application uses the mtools library alongside manual file deletion. These operations can be done with basic-level privileges. In addition, if the program is run with elevated privileges, it uses mounting to avoid any inconveniences.

Mounting is done via the system command *mount -o loop* to be mounted as a loop device. Then, working with the disk image file system is the same as working with the file system on a regular disk. The application currently supports FAT32, ext4, and NTFS.

There are many general tasks needed throughout the application. For this purpose, there is a file **utility** (to be exact, two files: source and header files), which provides an implementation of some support functions, most notably function *systemCommand(str::string)*, which serves as a bridge between the application and the system beneath for executing system commands by function std::system(...), and its alternatives to return system command error code (*systemCommandReturnErrorCode(std::string*) and return command output to standard output (*systemCommandReturnOutput(std::string)*). The other significant function is *loadUserInput(...)*. This function loads input from the provided input stream and checks its validity against the input type and possible values provided as parameters. Also, file encryption function *encryptFile(...)* is in this file, allowing to encrypt provided file using *gpg* tool. Password encryption can be either selected manually or randomly from rockyou.txt file enclosed (only from the first 55555 entries to avoid extensive search times when trying to crack the password).

8.2 Available scenarios

There are 7 implemented scenarios provided in the application. Details can be found in the descriptions below and in the markdown file **scenarios.md** enclosed along with the source code. The markdown file should be maintained along with the program to provide documentation and details necessary for all potential future scenarios. There are many of them in digital forensics. As the purpose of this thesis is to create training material, there is no reason to cover highly advanced situations. As a result, only the fundamentals are covered. The list is put together based on the author's experience and consultations. It should be noted that all the tasks can be solved with the assistance of advanced forensics tools; however, every specialist should know the basics and be able to go through these scenarios manually on his/her own. Also, provided solutions are based on the disk images generated by the program on the author's computer - Manjaro, these images can be seen in folder **images_example**. Details and specific values can differ on other platforms, general steps to fulfill the scenarios should remain the same.

8.2.1 Scenario 1

The first case study is about manipulating MBR table records for partition size and position. In real life, it can be caused intentionally or even unintentionally by the misbehavior of a program or by a system error. As for implementation details, a disk image of size 11 MiB with an MBR partitioning table is created,

in this case, consists of only one partition, FAT32, and is filled with zeros. Then, a picture is copied into the file system.

The forensics work is required for the next part, where the start LBA value is overwritten by a random value out of bounds, in this case, 70707070. It is achieved by the command *dd*. This confuses partitioning software; an inserted value would mean that the partition begins after the disk. Furthermore, even the partition size is overwritten to be bigger than the disk size. The quest is to recover the picture originally inserted into the file system.

8.2.1.1 Solution

The image cannot be mounted right away. Let's start with the simple fdisk command. Reported sector sizes and first-sector positions do not correspond with reality. The size of the image is 11 MiB; the reported size and first sector are hundreds of gigabytes away. From that, we can assume there are some problems with the partitioning scheme; open the disk image in a hex editor and examine the MBR table.

The first 446 bytes are dedicated to bootstrap code. Refer to the theory section for MBR layout. The MBR entry for the first and, presumably, only partition begins at address 0x1BE. As CHS values are deprecated and not of much interest, we can skip the following 8 bytes. Start LBA is where things get bad, its value is 0x70707070. The next value is not right too, the reported number of sectors 0x10101010. Starting with the total size in sectors, the file is 11 MiB long. If we take the reserved 1 MiB at the beginning into account, 10 MiB has to be divided into 512 sectors, meaning multiplied by 2048 (binary shift left 11 bits); $10 \cdot 2048 = 20480 =$ 0x5000, little endian is used, so value 0x10101010 is replaced with 0x00500000, which is the correct partition size.

Start LBA is 1 MiB (2048 sectors = 0x800 sectors), 0x70707070 is replaced with 0x00080000, which is the correct start point. Now the disk image should be able to mount.

8.2.2 Scenario 2

The second scenario is also about manipulating MBR fields. In this case, there are two file systems on the disk image. Its size is 21 MiB, and the available space (20 MiB) is divided equally between FAT32 and ext4 file systems. In this case, MBR fields are manipulated so that both partitions appear to start at the beginning of the disk and cover full space.

8.2.2.1 Solution

If we try to mount the image, two disks appear, but there are no files visible on either of them. GParted as well as the *fdisk* tool both report the same: 21 MiB disk image has two partitions of 20 MiB each. The MBR table shows two partitions: FAT32 and ext4, both from the very beginning (after the first

reserved MiB) all the way to the end. This does not seem right. We must divide the partitions so that they do not overlap.

There are several options here: guess it (e.g., use the binary division method) or do something similar like look for FS artifacts, in this case, artifacts of the ext4 file system. Some hex editors can search for the given string or just process until you get any non-zero values, as there should be no more data on the FAT32 partition. If someone wants to make their life difficult, they can try to find the exact division by hand. For the others, as stated in the hint provided, it is divided in the middle.

For the first partition - Start LBA remains the same, number of sectors should be reduced to half - from 0x00A00000 to 0x00500000. The second partition - Start LBA has to be corrected - reserved space + size of the first partition: 1 MiB + 10 MiB = 0x800 + 0x5000 = 0x5800 -> write 0x00580000 (little endian is used) as Start LBA; the size change is the same as for the first one. The disk image can be mounted now and the picture can be seen on the second partition.

8.2.3 Scenario 3

The third case study is fairly easy; there are two partitions on the image, both FAT32 and one of them is marked as hidden. It is really easy to spot and correct using any partitioning software, but the task is aimed at finding documentation and understanding fields in the partitioning table.

8.2.3.1 Solution

The image is mountable right away, and there is a file that should be found. However, there is one more small hidden partition at the end of the disk. Either replace the type in the MBR table (recommended to ensure understanding of the principles) or change it, e.g., inside GParted. There is another file hidden inside it.

If the MBR approach is chosen, the offset to the flag is (in bytes) 446 + 16 (first partition) + 1 (flag) + 3 CHS value. The current value is 1B (hidden FAT32), replace it with 0B (FAT32).

8.2.4 Scenario 4

This task deviates from the previous; it is aimed at checking the ability to decrypt files. A picture is copied to the FAT32 file system; the picture is encrypted, and the goal is to get it back. As an encryption program, *gpg* was selected. The password is randomly selected from the standard "rockyou" file. There is also an option to select the password manually.

8.2.4.1 Solution

For cracking passwords, one of the standard tools is John The Ripper, in this case, along with the utility *gpg2john*. The following 3 commands should be issued to decrypt the file.

```
- gpg2john cat_enc.gpg > enc
- john --wordlist=rockyou.txt enc
- gpg --pinentry-mode=loopback --passphrase "PASSWORD"
      --output res.jpg --decrypt cat_enc.gpg
```

8.2.5 Scenario 5

The last three prepared scenarios are about recovering a deleted file, in this case from the FAT32 file system. A disk image with a single partition is generated, and a file is created and then deleted by corresponding class functions for the respective file system. The task is to recover the file.

8.2.5.1 Solution

The solutions for all three file deletion scenarios are fairly complex and require multiple steps, only brief description is provided here. Please refer to enclosed file **scenarios.md** for more details.

Firstly, let's try to mount the file as a classical disk. It mounts without a hitch, but a file explorer displays no files. We can see whole file in a hex editor and discover that it is using the MBR table. The *fdisk* command shows the start block of the partition, which is 2048 (block size is 512, thus, 1048576 bytes). From the first sector of the partition - Partition Boot Sector, we can get all the necessary information about the disk layout. Let's jump into the hex editor to start with the first sector of the partition - Partition Boot Sector. Based on this, we can now compute the beginning of the data area:

$$\begin{aligned} offset &= start\_sector + reserved\_sectors + FATs\_num \cdot FAT\_size = \\ &= 1\ MiB + 16\ KiB + 2 \cdot 12\ KiB = 1089536\ bytes \end{aligned}$$

That is the offset of root directory. If we jump to that location, we can see directory entry for the deleted file, from which we can get the file name. The first byte is invalid (0xE5), so we correct it from the rest of the file name, which is clear: cat5.jpg. Offset 28 of the entry is designated for the file size, in this case 0x012FB8 (little endian) = 77752 bytes. There are $8 \cdot 512 = 4096$ bytes per cluster, 19 data clusters are needed for the storage.

The first cluster position can be gained from the directory entry as well. The next step is to reallocate data structures. The FAT table is placed right behind the reserved sectors. The first two entries are reserved, and the third belongs to the root directory; the deleted file was on cluster 3. We know from

the theory that linear allocation was used, which means consecutive blocks. The next step is the creation of the cluster chain. Each FAT entry should link to the next one until enough clusters (19) are connected, the last one with EOF flag.

The disk image can be mounted now, and the file can be viewed in a file viewer. There are two possible scenarios to be created. If the program was run with elevated privileges, deleting the file was achieved through mounting and the command *rm*. It runs with basic rights, there is manual implementation of deletion on FAT32. The solution is practically identical for both cases.

8.2.6 Scenario 6

In this case, a file was deleted from the ext4 file system. A disk image with a single partition is generated, and a file is created and then deleted by corresponding class functions for the respective file system. The task is to recover the file.

8.2.6.1 Solution

Mounting the disk shows no files. Ext4 is a bit tricky, so there are multiple ways to recover deleted data. The first one is data carving - searching for file signatures on the whole disk. This can be done easily for this task.

For ext4, we can use a little trick to get all the necessary structural information without searching through the binary data (of course, that is also a possibility). It is the following sequence of commands:

```
- sudo losetup -Pf sc6.img - mount disk as a loop device
- sudo losetup -j sc6.img - show mounting directory
- sudo dumpe2fs /dev/loopXp1 - get information
```

The same information can be gotten manually from the superblock on offset 1 MiB (reserved - MBR) + 1024 bytes (boot code) = 1049600 bytes. Further information about the layout can be obtained from Block Group Descriptor located in the block right after the superblock. Each has 64 bytes. In this case, the disk image size is relatively small so there are only two block groups. Details about the structure of Group Descriptor can be found in the theoretical section of the thesis or in the documentation. We get from it block numbers of data block bitmap, inode bitmap and inode table.

Even from the root directory, there are no links to the correct inode of the deleted file, if interested, it can be examined by following the data extent in the root inode (inode num. 2, counted from 1). Focus should be paid to the data block bitmap. A deleted file would have set bitmap entries to 0. We can guess the file location by aiming for the first unused blocks. In this case, the first not-FF value is at the offset 0xE0 = 224 bytes = 1792 bits, which corresponds to block 1792. Using the formula above, we get an offset

of 2883584 bytes. It should be near our deleted file. And indeed, the actual offset to the deleted file is 2887680 bytes.

The only thing remaining is the size of the original file. One method of doing so is to search for consecutive 00, which are to be expected after the end of the file. This way, we can find the end of the file at an offset of 2972160 bytes giving us a file size of approximately 84480 bytes. The deleted file can be retrieved by the *dd* command.

8.2.7 Scenario 7

In this case, a file was deleted from the NTFS file system. A disk image with a single partition is generated, and a file is created and then deleted by corresponding class functions for the respective file system. The task is to recover the file.

8.2.7.1 Solution

Mounting the disk shows no files. Let's open it in a hex editor and as for the previous two scenarios, partition begins at offset 1 MiB where partition boot sector begins. We are interested in the BPB (BIOS parameter block) table beginning at offset 0x0B (refer to documentation for details about the layout), from which we can get the position of MFT table, which is at the offset (from the beginning of the disk image) of $1024 \cdot 1024 + 16 \cdot 1024 = 1064960$ bytes. Each MFT entry has a standard size of 1024 bytes. Most notable are the root directory (entry 5) and user files, which begin at entry 16.

To get to the actual MFT entry of the deleted file, it is sufficient to proceed through all the entries and looking for their names until the correct one is found, which is located at the offset of 1130496 bytes (the last MFT entry allocated). There are several notable attributes of this entry, the most important for our purpose is $DATA (type 0x80) beginning at the offset 0x114158 (1130840) from the image beginning.

This attribute is out of place (refer to the theoretical section for more information), meaning it contains data runs. In this case, there is only one, 0x21371D04 - 1 byte length and 2 bytes offset - length 0x37 clusters and offset 0x041D = 1053 clusters, which gives position $1024 \cdot 1024 + 1024 + 1052 \cdot 1024 = 2126848$ bytes, which is indeed the beginning of the deleted file's data. Its size is 0x37 = 55 clusters = 556320 bytes. That is enough to issue *dd* command to retrieve the file.

Chapter 9

Conclusion

The goal of this thesis was to create a disk generator for digital forensics training and to provide several training scenarios and the theory needed for their completion. As computer forensics is a quite broad area, it was not possible to describe all the situations or scenarios in this thesis. I believe that the theory and training provided here are fundamental to getting started in the area of digital forensics. There are lots of materials available for subsequent studies; some of them can be found in the bibliography section.

Many of the case studies provided were concluded based on the tests performed by the author and his observations. As the test sample size was not representative of all the possible systems and situations, there may be some variations encountered on different systems. Some of the behavior I came across was quite surprising, e.g., the behavior of the partitioning software when manipulating the MBR record.

There are numerous avenues for future research to further this thesis. Adding further scenarios is the main one. Even though the program was written with scalability and extensibility in mind, there are several flaws to be aware of. Even if an application redesign is required or desired in the future, I believe the principles used can serve as an inspiration (or even a framework) for future works.

Bibliography

1. NOVÁK, Dominik. *Úvod do digitální forenzní analýzy [Introduction to digital forensics]*. 2021.
2. BHUIYAN, Johana. *Apple says it prioritizes privacy. Experts say gaps remain* [online]. The Guardian, 2022 [visited on 2022-12-18]. Available from: `https://www.theguardian.com/technology/2022/sep/23/apple-user-data-law-enforcement-falling-short`.
3. ÅRNES, André. *Digital forensics*. John Wiley & Sons, Ltd, 2017. ISBN 9781119262442.
4. JOSHI, Binaya Raj; HUBBARD, Rick. Forensics analysis of solid state drive (SSD). In: *2016 Universal Technology Management Conference (UTMC)*. The Society of Digital Information and Wireless Communications (SDIWC), 2016, vol. 2016, pp. 1–12. ISBN 978-1-941968-29-1.
5. Steven BRESS; Mark Joseph MENZ. *Write protection for computer long-term memory devices*. Inventor: Steven BRESS; Mark Joseph MENZ. Publ.: 2004-11-02. Patent US006813682B2.
6. *Tableau Forensic SATA/IDE Bridge T35u* [online]. OpenText Security [visited on 2022-12-18]. Available from: `https://security.opentext.com/tableau/hardware/details/t35u`.
7. *Tableau Forensic Duplicator TD2u* [online]. OpenText Security [visited on 2022-12-18]. Available from: `https://security.opentext.com/tableau/hardware/details/td2u`.
8. AKBAL, Erhan; DOGAN, Sengul. Forensics Image Acquisition Process of Digital Evidence. *International Journal of Computer Network & Information Security*. 2018, vol. 10, no. 5.
9. *Product downloads* [online]. AccessData [visited on 2022-12-18]. Available from: `https://accessdata.com/product-download/ftk-imager-version-4-5`.

10. *OpenText EnCase Forensic* [online]. Open Text Corporation [visited on 2022-12-18]. Available from: https://www.opentext.com/products/encase-forensic.

11. DURYEE, Alexander. An Introduction to Optical Media Preservation. *The Code4Lib Journal* [online]. 2014, no. 24 [visited on 2022-12-18]. Available from: https://journal.code4lib.org/articles/9581.

12. *Volume and File Structure of CDROM for Information Interchange: Standard ECMA-119*. ECMA International, 2019. Standard. Available also from: https://www.ecma-international.org/publications-and-standards/standards/ecma-119/.

13. *E01 (Encase Image File Format)* [online]. FORENSICSWARE. [visited on 2022-12-18]. Available from: https://www.forensicsware.com/blog/e01-file-format.html.

14. *Imager User Guide*. AccessData Group, Inc., 2020.

15. AL SADI, Ghania. Analyzing Master Boot Record for Forensic Investigations. *International Journal of Applied Information Systems*. 2016, vol. 10, pp. 22–26. Available from DOI: 10.5120/ijais2016451541.

16. RAXESH. *Disk partitioning: MBR VS GPT* [online]. Blogger, 2018-12 [visited on 2022-12-18]. Available from: https://easylinuxji.blogspot.com/2018/12/what-is-disk-partitioning-disk.html.

17. CARRIER, Brian. *File system forensic analysis*. Addison-Wesley Professional, 2005.

18. CROSS, Michael; SHINDER, Debra Littlejohn. *Scene of the Cybercrime*. 2008. ISBN 9781597492768.

19. *EaseUS Partition Recovery 9.0* [online]. EaseUS [visited on 2022-12-18]. Available from: https://www.easeus.com/partition-recovery/index.htm.

20. *DOS Boot Record (DBR) / DOS Boot Sector* [online]. Pro Data Doctor. [visited on 2022-12-18]. Available from: http://www.p-dd.com/chapter3-page17.html.

21. *Unified Extensible Firmware Interface (UEFI) Specification* [online]. UEFI Forum, Inc., 2022. Version 2.10 [visited on 2022-12-18]. Available from: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf.

22. *Windows and GPT FAQ* [online]. [visited on 2022-12-18]. Available from: https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-and-gpt-faq?view=windows-11.

23. *GUID Partition Table (GPT)* [online]. LSoft Technologies Inc. [visited on 2022-12-18]. Available from: https://www.ntfs.com/guid-part-table.htm.

24. WILLIAMS, Ross N. A Painless Guide to CRC Error Detection Algorithms. 1993.

25. WALKER, Daryle. *CRC Library* [online]. 2001. [visited on 2022-12-18]. Available from: `https://www.boost.org/doc/libs/1_66_0/libs/crc/`.

26. *BitLocker* [online]. 2022-09-12. [visited on 2022-12-18]. Available from: `https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview`.

27. BOSSI, Simone; VISCONTI, Andrea. What Users Should Know About Full Disk Encryption Based on LUKS. In: REITER, Michael; NACCACHE, David (eds.). *Cryptology and Network Security*. Cham: Springer International Publishing, 2015, pp. 225–237. ISBN 978-3-319-26823-1.

28. *Overview of FAT32* [online]. [visited on 2022-12-18]. Available from: `https://eric-lo.gitbook.io/lab9-filesystem/overview-of-fat32`.

29. LIN, Xiaodong. *Introductory Computer Forensics*. Springer, 2018. ISBN 9783030005801.

30. *The FAT File System* [online]. [visited on 2022-12-18]. Available from: `http://www.c-jump.com/CIS24/Slides/FAT/lecture.html`.

31. PIPER, Scott; DAVIS, Mark; MANES, Gavin; SHENOI, Sujeet. Detecting hidden data in ext2/ext3 file systems. In: *IFIP International Conference on Digital Forensics*. Springer, 2005, pp. 245–256.

32. GÖBEL, Thomas; BAIER, Harald. Anti-forensic Capacity and Detection Rating of Hidden Data in the Ext4 Filesystem. In: *IFIP International Conference on Digital Forensics*. Springer, 2018, pp. 87–110. ISBN 9783319992761.

33. MATHUR, Avantika; CAO, Mingming; BHATTACHARYA, Suparna; DILGER, Andreas; TOMAS, Alex; VIVIER, Laurent. The new ext4 filesystem: current status and future plans. In: *Proceedings of the Linux symposium*. Citeseer, 2007, vol. 2, pp. 21–33.

34. KING, Tracy. *What is Ext2/Ext3/Ext4 File System Format and What's The Difference [Full Guide]* [online]. 2022-09-22. [visited on 2022-12-18]. Available from: `https://www.easeus.com/partition-master/ext2-ext3-ext4-file-system-format-and-difference.html`.

35. *Ext4 Disk Layout* [online]. 2019-08-26. [visited on 2022-12-18]. Available from: `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`.

36. RITCHIE, Dennis M. The evolution of the Unix time-sharing system. In: *Symposium on Language Design and Programming Methodology*. 1979, pp. 25–35.

37. BOTH, David. *An introduction to Linux's EXT4 filesystem* [online]. 2017-05-25. [visited on 2022-12-18]. Available from: https://opensource.com/article/17/5/introduction-ext4-filesystem.

38. CAO, Mingming; SANTOS, Jose Renato; DILGER, Andreas; KUMAR, Aneesh. Ext4 block and inode allocator improvements. In: *Proceedings of the Linux Symposium*. 2008, vol. 1, pp. 263–274.

39. LEE, Seokjun; SHON, Taeshik. Improved deleted file recovery technique for Ext2/3 filesystem. *The Journal of Supercomputing*. 2014, vol. 70, no. 1, pp. 20–30.

40. PATE, Steve D. *UNIX Filesystems: Evolution, Design, and Implementation*. Wiley Publishing, Inc., 2003. ISBN 0471164836.

41. KAI, Zhang; EN, Cheng; QINQUAN, Gao. Analysis and Implementation of NTFS File System Based on Computer Forensics. In: *2010 Second International Workshop on Education Technology and Computer Science*. IEEE, 2010, vol. 1, pp. 325–328. Available from DOI: 10.1109/ETCS.2010.434.

42. *MFT Entry Format* [online]. [visited on 2023-05-04]. Available from: http://www.c-jump.com/bcc/t256t/Week04NtfsReview/W01_0220_mft_entry_format.htm.

43. MEDEIROS, Jason. NTFS Forensics: A Programmers View of Raw Filesystem Data Extraction. 2008.

44. RUSSON, Richard. *NTFS - Attributes* [online]. [visited on 2023-05-04]. Available from: https://flatcap.github.io/linux-ntfs/ntfs/attributes/index.html.

45. *Concept - Data Runs* [online]. [visited on 2023-05-04]. Available from: https://flatcap.github.io/linux-ntfs/ntfs/concepts/data_runs.html.

46. FORTUNA, Andrea. *How to extract data and timeline from Master File Table on NTFS filesystem* [online]. 2017-07-18. [visited on 2022-12-18]. Available from: https://andreafortuna.org/2017/07/18/how-to-extract-data-and-timeline-from-master-file-table-on-ntfs-filesystem/.

47. *Volume Shadow Copy Service* [online]. 2022-07-12. [visited on 2022-12-18]. Available from: https://learn.microsoft.com/en-us/windows-server/storage/file-server/volume-shadow-copy-service.

48. LAUBER, Susan. *Getting started with GPG (GnuPG)* [online]. 2020. [visited on 2022-12-18]. Available from: https://www.redhat.com/sysadmin/getting-started-gpg.

49. CALLAS, J.; DONNERHACKE, L.; FINNEY, H.; SHAW, D.; THAYER, R. OpenPGP Message Format. *www.rfc-editor.org* [online]. 2007 [visited on 2022-12-18]. Available from DOI: 10.17487/RFC4880.

50. WEE, Cheong Kai. Analysis of hidden data in NTFS file system. *Edith Cowan University*. 2006.

51. YAACOUB, Jean-Paul A.; NOURA, Hassan N.; SALMAN, Ola; CHEHAB, Ali. *Digital Forensics vs. Anti-Digital Forensics: Techniques, Limitations and Recommendations*. arXiv, 2021. Available from DOI: 10.48550/ARXIV.2103.17028.

52. *Mtools* [online]. Free Software Foundation, Inc., 2023 [visited on 2023-03-22]. Available from: https://www.gnu.org/software/mtools/.

Appendix A

Acronyms

FS file system
HW hardware
SW software
OS operating system
HDD hard disk drive
SSD solid state drive
KiB kibibyte
MiB mebibyte
GiB gibibyte
TiB tebibyte
ISO Optical Disk Image
MBR Master Boot Record
CHS cylinder-head-sector
LBA logical block addressing (address)
GPT GUID Partition Table
GUID globally unique identifier
UEFI Unified Extensible Firmware Interface
CRC cyclic redundancy check
FAT File Allocation Table
NTFS New Technology File System
MFT Master File Table
GNU GNU's Not Unix
GPL General Public License

Appendix B

Contents of enclosed CD (zip file)

- `README.txt` ..... file with brief CD (zip) contents description
- `thesis/` ..... text of the thesis
  - `DP_Horak_Petr_2023.pdf` ..... thesis in PDF format
  - `latex/` ..... LaTeX source files
- `DiskImageGenerator/` ..... program and source code directory
  - `build/` ..... folder for object files generated
  - `images_example/` ..... sample disk images generated by the program
  - `src/` ..... source code folder
  - `DiskImageGenerator` ..... program binary
  - `LICENSE.txt`
  - `Makefile`
  - `README.txt` ..... program notes
  - `scenarios.md` ..... file with all the scenarios provided