Digital Forensics

Federico Conti

2024/25

Contents

Forensic Acquisition

Acquisition is the process of cloning or copying digital data evidence.

- forensically sound (integrity and non-repudiation)
  - the copies must be identical to the original
  - the procedures must be documented and implemented using known methods and technologies, so that they can be verified by the opposite party
- a critical step
  - proper handling of data ensures that all actions taken on it can be checked, repeated, and verified at any time
  - incomplete or incorrect handling of data has the potential to compromise the entire investigation

It is generally recommended to avoid conducting analysis on the original device (namely, best evidence).

Creating a forensic image is typically considered the most effective method for preserving digital evidence (One or more, usually two).

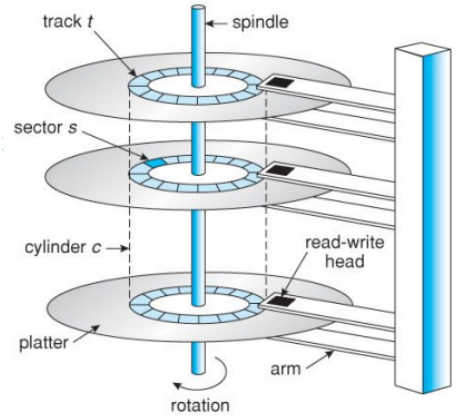
Accessing the original media only once during the acquisition phase can help minimize the risk of altering or damaging the evidence

HDD and SSD technologies

HDD

A **hard disk** is a sealed unit containing a number of platters in a stack:

- each platter has two wothey moverking surfaces
- each working surface is divided into a number of concentric rings called tracks
  - the collection of all tracks that are the same distance from the edge of the platter is called a cylinder.
- each track is further divided into sectors
  - a sector is the smallest unit that can be accessed on a storage device
  - traditionally containing 512 bytes of data each, but recent hard drives have switched to 4KB sectors (Advanced Format)
  - a cluster is a group of sectors (from 1 to 64 sectors) that make up the smallest unit of disk allocation for a file within a file system



The data on a hard drive is read by read-write hands. The standard configuration uses one head per surface and they moves simultaneously from one cylinder to another.

A **low level format** is performed on the blank platters to create data structures for tracks and sectors

- creates all of the headers and trailers marking the beginning and ends of each sector
- header and trailer also keep the linear sector numbers (cf. LBA later), and error-correcting codes (ECC)

All disks are shipped with a few bad sectors (additional ones can be expected to go bad slowly over time).

- disks keep spare sectors to replace bad ones
- ECC calculation is performed with every disk read or write: if an error is detected but the data is recoverable, then a *soft error* has occurred if the data on a bad sector cannot be recovered, then a *hard error* has occurred. A bad sector can be replace with a spare one (but any information written is usually lost)

Older hard drives used a system called **CHS (Cylinder-Head-Sector)** addressing to locate data on the disk. This method relied on:

- Cylinders (C) → The track number (a ring of data on a disk platter).
- Heads (H) → The read/write head that accesses a platter's surface.
- Sectors (S) → The smallest unit of storage on a track.

Eample: CHS (100, 2, 30) would mean: Cylinder 100; Head 2 (indicating the second platter side); Sector 30 (the exact location on that track).

CHS had physical limitations → It could only handle disks up to 504 MB (later ECHS extended this to 8 GB).

Instead of using three values (CHS), **Logical Block Addressing(LBA)** assigns a single number to each sector. LBA starts at 0 and counts up sequentially. The operating system and file system treat the disk as a continuous array of sectors, making it easier to manage.

Example: LBA 123456: The 123,456th sector on the disk.

SSD

A **Solid-State Drive (SSD)** is a non-volatile storage device, meaning it retains data even when the power is off. Unlike Hard Disk Drives (HDDs), which use spinning magnetic platters, SSDs rely on flash memory chips to store data.

- The smallest unit of an SSD is a page, which is composed of several memory cells: the page sizes are 2KB, 4KB, 8KB, 16KB or larger (usually they are 4 KB in size).
- Several pages on the SSD are summarized to a block: the block size typically varies between 256KB (128 pages * 2KB per page) and 4MB (256 pages * 16KB per page)

Operation	Description
Read and Write	An SSD can read and write data at the page level.
Write	Writing is only possible if other pages in the block are empty. Otherwise, it leads to write amplification.
Erase Data	An SSD can only erase an entire block at once due to the physical and electrical characteristics of the memory cells.

Modifying data requires a **Program/Erase (P/E) cycle**.

- During a P/E cycle, an entire block containing the targeted pages is written to memory
- The block is then marked for deletion, and the updated data is rewritten to another block

The erase operation does not happen immediately after data is marked for deletion. Instead, the SSD performs it asynchronously when necessary to optimize performance.

Garbage Collection helps free up space efficiently while minimizing interruptions to read/write operations.

Every flash memory cell has a limited number of P/E cycles before it wears out Wear leveling is a technique used by SSDs to distribute writes evenly across all memory blocks.

Unlike HDDs, SSDs cannot simply overwrite deleted files. If deleted data is not managed properly, unnecessary data copies can cause write amplification, increasing wear on the SSD. TRIM command allows the OS to inform the SSD which pages are no longer needed, enabling it to manage space more efficiently.

Forensic Investigators Face a Big Problem with SSDs|

TRIM permanently deletes data by triggering the garbage collector, making data recovery impossible. The garbage collector operates independently within the SSD controller, meaning:

- Even a hardware write blocker (a forensic tool to prevent data changes) cannot stop it.
- Data can change midway or between acquisitions, complicating forensic investig

Interface standards and protocols

Disks are accessed using standard interfaces that define how they physically and logically connect to a computer system. Each standard improves data transfer speeds and fixes limitations from older technologies. A disk interface consists of two key components:

- Physical Connection → Defines the type of cable and connector used to attach the disk to the system.
- Logical Connection → Defines the protocol that controls how data is transferred between the disk and the computer.

ATA (Advanced Technology Attachment):

a widely used interface standard with multiple versions:

- ATA-1, ATA-2, . . . , ATA-8 → Each new version improves speed and capacity.
- ATAPI (ATA Packet Interface) → Allows removable media (CD/DVD drives) to be connected using ATA but still uses SCSI commands for data transfer.

Type	Description
PATA (Parallel ATA)	Older, also known as IDE (Integrated Drive Electronics). Uses ribbon cables with multiple pins.
SATA (Serial ATA)	Modern standard introduced in ATA/ATAPI-7. Uses thin serial cables, improving speed and efficiency.
SATA 3.0	Supports speeds up to 6 Gbit/s (SATA-600). Common in modern HDDs and SSDs.
ATA-8	Introduced optimizations for SSDs, including TRIM support.

SCSI (Small Computer System Interface):

now replaced by SAS (Serial Attached SCSI) that is based on the SCSI standard, but uses a serial interface to connect storage. Better scalable and faster (supports data transfer rates of up to 24 Gbit/s).

NVMe (Non-Volatile Memory Express):

Uses a PCIe interface to connect storage devices to a computer. Commonly used for high-performance solid-state drives since it supports data transfer rates of up to 32 GB/s.

USB (Universal Serial Bus):

- uses a serial interface to connect storage devices to a computer
- mass storage is the standard protocol used for storage devices
- commonly used for external hard drives, flash drives, and other portable storage devices
- USB 3.1 Gen 1 standard supports speeds up to 5 Gbit/s, while the USB 3.1 Gen 2 standard, supports speeds up to 20 Gbit/s.

ATA specification

Introduced in ATA-3, **hard disk passwords** are an optional security feature designed to restrict unauthorized access to a hard drive. However, it is a lock mechanism, not encryption, meaning data remains unencrypted and could be accessed by other means.

There are two passwords:

1. User Password → Set by the owner.
2. Master Password → was designed so an administrator can gain access in case the user password was lost (every hard disk is initially supplied with an undocumented master password).

If passwords are being used, there are two modes that the disk can operate:

1. high security mode: the user and master password can unlock the disk
2. maximum-security mode: the user password can unlock the disk but the master password can unlock the disk after the disk content have been wiped

A protected HD will require the `SECURITY_UNLOCK` command to be executed with the correct password before any other ATA command. After the password has been entered, the disk works normally until the disk is powered on

Some ATA commands are still enabled on the HD when it is locked (so it may show as a valid disk when it is connected to a computer); however, trying to read data from a locked disk will produce an error

Setting the Password:

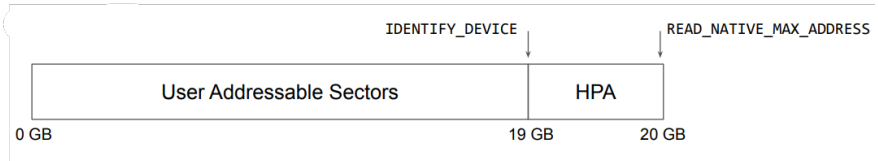
- Can be configured via BIOS settings.
- Linux users can manage HDD passwords using tools like hdparm.

Introduced in ATA-4, the **Host Protected Area (HPA)** is a hidden storage section on a hard disk that is not accessible to the operating system. It is used mainly by hardware vendors for system recovery files, diagnostics, or factory reset tools. A HPA is at the end of the disk and when used, it can be accessed by reconfiguring the hard disk.

Two ATA commands that return maximum physical addressable sectors

- `READ_NATIVE_MAX_ADDRESS`: return the maximum physical address
- `IDENTIFY_DEVICE`: return only the number of sectors that a user can access

To create an HPA, the `SET_MAX_ADRESS` command is used to set the maximum address to which the user should have access (to remove it, use `SET_MAX_ADDRESS = READ_NATIVE_MAX_ADDRESS`).



the `SET_MAX_ADRESS` command support different settings, e.g.,

- volatility bit: the HPA exist after the hard disk is reset or power cycled (otherwise the effect is permanent)
- locking command: prevents modification to the maximum address until next reset

when the BIOS requires to read/write some data in the HPA it uses `SET_MAX_ADRESS` with volatility bit and locking.

It is possible to protect settings with a password (different from HD passwords).

The **Device Configuration Overlay (DCO)** was introduced in ATA-6 and allows manufacturers to limit the apparent capabilities of a hard disk. This feature enables backward compatibility with older systems but can also be exploited to hide data.

A computer uses the `IDENTIFY_DEVICE` command to check an HDD's specifications (size, features, supported commands). If a DCO is applied, the `IDENTIFY_DEVICE` command will not show the actual full disk size or features.

This is achieved using two special ATA commands:

- `DEVICE_CONFIGURATION_SET` → Modifies the DCO settings to restrict visible disk space or disable features.
- `DEVICE_CONFIGURATION_RESET` → Restores the original settings, removing the DCO restrictions.

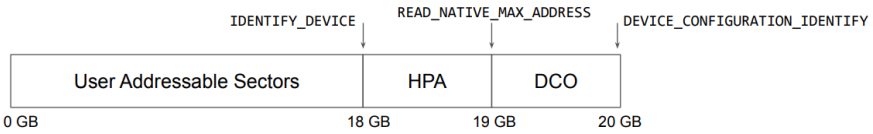
*Example:

A 2 TB hard drive can be made to appear as a 500 GB drive, hiding the remaining 1.5 TB from the operating system.*

The DCO and HPA can co-exist on the same HDD (but DCO must be set first).

The `DEVICE_CONFIGURATION_IDENTIFY` command return the actual features and size of a disk:

- we can detect DCO if `DEVICE_CONFIGURATION_IDENTIFY` $\neq$ `IDENTIFY_DEVICE`



At least three different methods for detecting HPA on Linux: `dmesg`, `hdparm`, and `disk_stat` (https://wiki.sleuthkit.org/index.php?title=Disk_stat)

Disk encryption: Bitlocker

BitLocker is a full-disk encryption feature in Windows that protects data using a multi-layered encryption system.

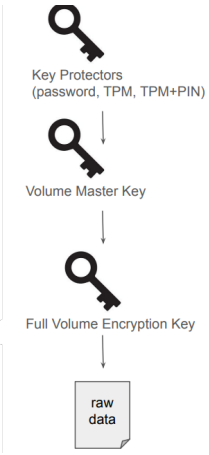
It makes use of symmetric encryption (by default, AES-128).

On modern systems, it is coupled with a Trusted Platform Module (TPM):

- the main functions of TPM are the generation, storage and secure management of cryptographic keys
- on a computer without TPM a password can be used (then BitLocker encryption will be just as secure as the password you set)

Bitlocker uses different symmetric key:

1. row data is encrypted with the Full Volume Encryption Key (FVEK)
2. FVEK is then encrypted with the Volume Master Key (VMK)
3. VMK is in turn encrypted by one of several possible methods depending on the chosen authentication type (that is, key protectors or TPM) and recovery scenarios



The use of intermediate key (VMK between FVEK and any key protectors) allows changing the keys without the need to re-encrypt the raw data in a case a given key protector is compromised or changed.

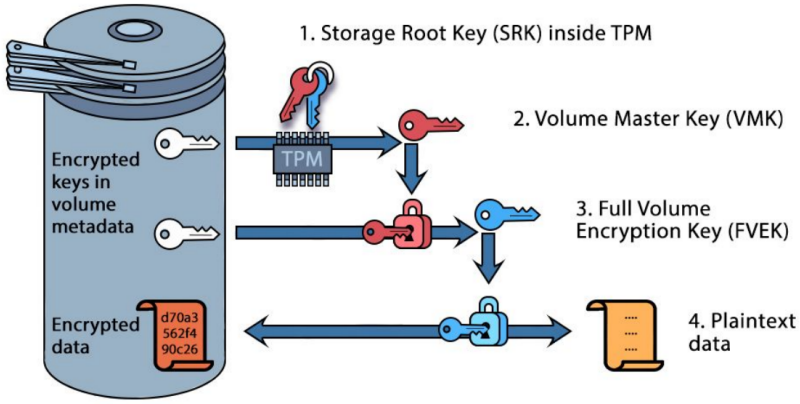
- When changing a key protector, a new VMK will be created and used to encrypt the old FVEK with the new VMK

BitLocker supports multiple key protector options, depending on the security needs and device type.

TPM only

- The TPM module (a hardware security chip) decrypts the VMK using a Storage Root Key (SRK) stored in the TPM.
- The SRK is only released if Secure Boot passes, ensuring the device boots with its original OS and configuration.
- The BitLocker volume is unlocked automatically during boot, before the user logs in.

View article.



TPM + PIN

- The TPM module will only release the VMK if the user enters a correct PIN during the pre-boot phase.
- If too many incorrect PIN attempts occur, the TPM will lock access to the encryption key, preventing brute-force attacks.

Key Takeaways:

- BitLocker is excellent for protecting against physical threats like device theft or unauthorized hard drive access.
- It does NOT protect against malware, ransomware, or unauthorized logins by users on the same computer.
- TPM + PIN is the most secure option to prevent unauthorized access, even if a device is stolen.

BitLocker poses a problem for forensic investigators, as all information on the drive will be encrypted, and therefore unreadable. Some methods for breaking BitLocker password are:

- the RAM dump/hibernation file/page file attack: this attack is universal, and works regardless of the type of protector. It dumps from the computer's volatile memory (and possibly in the page/hibernation file) the VMK that is loaded unencrypted while the volume is mounted
- BitLocker recovery keys: in many situations recovery keys are be stored in the user's Microsoft Account. Extracting those keys from their account allows instantly mounting or decrypting protected volumes regardless of the type of protector

Toolset and examples

A **loop** device is a special kind of block device that does not map to a physical hardware device (such as a hard disk) but instead maps to a regular file stored within a filesystem.

- useful to access a forensic image
- read/only can be forced
- the offset parameter could be useful to directly access a volume
- (can be used to simulate a block device to be acquired)

Key losetup Commands:

- `losetup -a`→ Shows the status of all loop devices.
- `losetup -d [device]` → Detaches a loop device.
- `losetup -f` → Finds the first available (unused) loop device.
- `losetup -o [offset]` → Starts reading data at a specific offset in the file.
- `losetup -r /dev/loop0 [srcfile]` → Sets up a read-only loop device.

dd is the precursor of all acquisition tools, allowing for the acquisition of data bit by bit in raw format.

Key `dd` Options:

- `if=` → Input file (or device to copy from).
- `of=` → Output file (or device to write to).
- `bs=` → Block size (how much data to read/write at a time).
- `conv=` → Specifies conversion options (e.g., noerror to continue on errors).

Example

```
dd if=/dev/sda of=/mnt/dest/image.dd bs=512
```

losetup and dd

In this exercise, we will simulate a block device using a compressed forensic image and interact with it as if it were a real disk.

```
wget https://github.com/enricorusso/DF_Exs/raw/main/acquisition/image.dd.gz
gunzip image.dd.gz

# Before setting up the loop device, find an available one using
losetup -f

sudo losetup -r /dev/loop1 image.dd # Now, set up the image as a read-only loop device

sudo dmesg | tail # To verify that the loop device was correctly attached, check system logs
[84058.342422] loop1: detected capacity change from 0 to 2033664

sudo fdisk -l /dev/loop1 # To inspect the loop device and view partition details

# Since the loop device represents an entire disk,
# Linux does not automatically recognize partitions.
# Use partx to make them available
sudo partx -a /dev/loop1

mkdir -p /mnt/forensic_image
sudo mount -o ro /dev/loop1p1 /mnt/forensic_image

ls -l /mnt/forensic_image

#clean
sudo umount /mnt/forensic_image
sudo losetup -d /dev/loop1
```

In digital forensics, verifying the integrity of a forensic image is crucial to ensure that the data remains unchanged during analysis. This is done by calculating cryptographic hash values (MD5 and SHA1) before and after mounting the image → Before using the forensic image, compute its MD5 and SHA1 hashes.

```
md5sum image.dd
446144a4af914d7e55603b6042f20db1  image.dd

sha1sum image.dd
99540f5aaa170afbab722729e980fd6dc34ff323
image.dd

md5sum /dev/loop1
446144a4af914d7e55603b6042f20db1
/dev/loop1

sha1sum /dev/loop1
99540f5aaa170afbab722729e980fd6dc34ff323
/dev/loop1
```

The dd tool does not calculate hashes during acquisition, so forensic best practices require manually computing hashes before and after imaging to ensure data integrity.

```
# Instead of separately computing hashes before and after, we can stream data from dd to tee, simultane
sudo dd if=/dev/loop1 bs=512 | tee image.dd
| hashdeep -c md5,sha1 > image.src_hash #apt install hashdeep

# dd if=/dev/loop1 bs=512 → Reads data from the loop device.
# tee image.dd → Writes data to image.dd while also passing it to the next command.
# hashdeep -c md5,sha1 → Computes MD5 and SHA1 hashes as data is written.
# > image.src_hash → Saves the computed hashes to image.src_hash.

#Finale verification hash
md5sum image.dd
```

A faulty disk

In this exercise, we simulate a faulty disk by mapping a logical block device and introducing bad sectors. We then attempt to acquire it using dd, handling errors properly to maintain forensic integrity.

1. We create a logical "faulty" device (1Kb) with the command dmsetup*
   - `8 8 error` → [starting sector; add sector] maps the next 8 sectors of 512 byte (8 to 16) of the bad_disk device to an error area. This means that any attempt to read or write to bad_disk sectors 8 to 16 will generate an error.
   - `/dev/loop1` is the origin and must be initialized with a .dd (`sudo losetup /dev/loop0 image.dd/`)

   ```
   sudo dmsetup create bad_disk << EOF
   0 8 linear /dev/loop0 0
   8 8 error
   16 2033648 linear /dev/loop0 16
   EOF
   ```

2. Scan the simulated bad sectors.

   ```
   sh sudo badblocks -b 512 -v /dev/mapper/bad_disk
   ```

3. To ensure that all reads go directly to the faulty device (and are not cached), we disable readahead. Then, check the block device size:

   ```
   sudo blockdev --setra 0 /dev/mapper/bad_disk
   sudo blockdev --getsz /dev/mapper/bad_disk
   ```

4. Now, try acquiring the faulty disk with dd

   ```
   sudo dd if=/dev/mapper/bad_disk of=bad.dd bs=512
   ```

   Problem: dd stops when it hits a bad sector, preventing a complete acquisition.

5. To log bad sectors and replace them with zeros, use `conv=sync,noerror`
   - `tee bad.dd` → Writes the output to bad.dd while streaming it to `hashdeep` for hashing.

   ```
   sudo dd if=/dev/mapper/bad_disk bs=512 conv=sync,noerror
   | tee bad.dd | hashdeep -c md5,sha1 > bad_image.src_hash
   ```

6. After acquisition, compare the hash of bad.dd to the hash calculated during acquisition in `bad_image.src_hash`

   ```
   hashdeep -c md5,sha1 bad.dd
   ```

dc3dd

An enhanced version of dd designed specifically for digital forensics. It was developed by the DoD Cyber Crime Center (DC3) and includes several critical forensic features missing in standard dd.

Example

```
sudo dc3dd if=/dev/mapper/bad_disk of=bad.dd ssz=512 log=image.log hlog=hash.log hash=md5 hash=sha1
```

Image file formats

When acquiring digital evidence, the choice of image format is crucial for integrity, compatibility, and efficiency in analysis. There are two main categories of forensic image formats.

The output from dd acquisition is a raw image * it contains only the data from the source device * all the descriptive data about the acquisition (e.g., hashes values, dates, or times) need to be saved in a separate file

An embedded image contains data from the source device and additional descriptive data (metadata).

- **Expert Witness Format (EWF)**

  Joachim Metz (Google) created the libewf project, open source (https://github.com/libyal/libewf, apt install ewf-tools). It provides a library and set of tools to manage the ewf format.

  - `ewfacquire`: reads storage media data from devices and write files to EWF files.
  - `ewfexport`: exports storage media data in EWF files to (split) RAW format or a specific version of EWF files.
  - `ewfinfo`: shows the metadata in EWF files.
  - `ewfmount`: FUSE mounts EWF files.
  - `ewfrecover`: special variant of ewfexport to create a new set of EWF files from a corrupt set.
  - `ewfverify`: verifies the storage media data in EWF files

(FUSE (Filesystem in Userspace) is a Linux kernel module that allows users to mount and manage file systems without requiring root privileges or kernel modifications.) –> alternative `:uses sudo ewfmount`

- **Advanced Forensic Format (AFF)**

  Open Source format developed by Dr. Simson L. Garfinkel

  - Provide compressed or uncompressed image files
  - No size restriction for disk-to-image files
  - Provide space in the image file or segmented files for metadata (unlimited number)
  - Digital signatures
  - Encryption with decryption on-the-fly
  - No patents

  Still lacks wide adoption (software available at https://github.com/sshock/AFFLIBv3).

Guymager

Guymager is a graphical (Qt-based) forensic imager. It is capable of producing image files in EWF, AFF and dd format (apt install guymager).

- AFF is disabled by default (`sudo nano /etc/guymager/guymager.cfg → set AffEnabled=true → sudo systemctl restart guymager`)

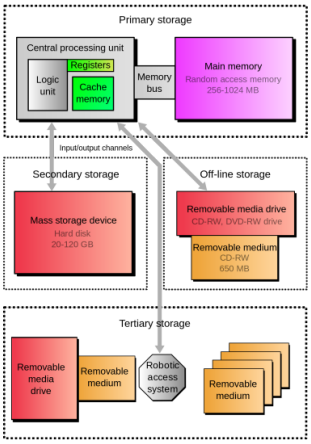
FTK Imager

FTK Imager is a data preview and imaging tool used to acquire data (evidence) in a forensically sound manner by creating copies of data without making changes to the original evidence (https://www.exterro.com/ftk-product-downloads).

File Systems

In a computer, data storage is organized in a hierarchical manner.

At the top, we have fast and directly accessible storage, such as registers and cache, while at the bottom, we have slower but larger storage, like hard drives or SSDs.



We focus on secondary (external) memory storage:

- not directly accessible by CPU –> data must be transferred to main memory (RAM) before it can be processed
- data transferred in blocks –> rather than individual bytes
- significantly slower
- non-volatile –> retains data even when power is turned off

Floppies/HDs/CDs/DVDs/BDs/SD-cards/SSDs/pendrives. . . are all block devices; following File System Forensic Analysis's terminology.

A volume is a collection of addressable blocks

- these blocks do not need to be contiguous on a physical device
- a volume can be assembled by merging smaller volumes

A partition is a contiguous part of a volume

- partitioning is optional: some removable storage does not use it

By definition, both disks and partitions are volumes. In this part of the course we deal with block-device (forensics) images, like the one acquired from actual devices.

Users don't interact directly with storage blocks—instead, they work with files and directories.

The file system creates this illusion; i.e., it handles the mapping between files/directories and a collection of blocks (usually, clusters of sectors). Consists of on-disk data structures to organize both data and metadata. There exist various file-system formats (e.g., FAT, NTFS, . . .)

(high-level) formatting a volume means to initialize those structures.

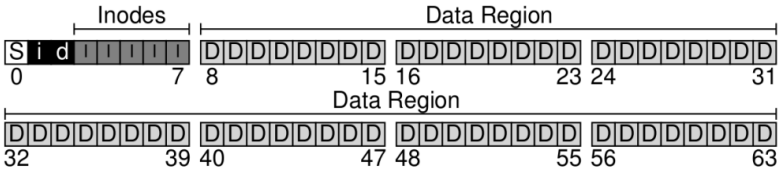
VSFS (Very Simple File-System)

In Unix-like file systems (e.g., EXT4, see `man mkfs.ext4`), each file (or other filesystem object) has an associated inode that stores its metadata. However, inode does not store file names, which are instead kept in directory structures.

Every file system object has an inode, including:

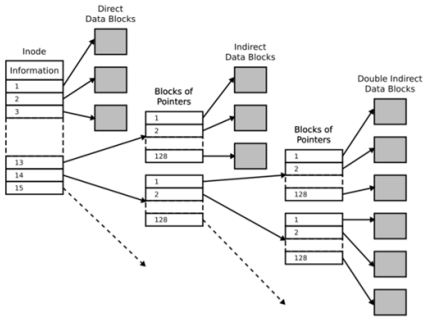
- regular file
- directory
- symbolic link
- FIFO
- socket
- character device
- block device

Formatting means preparing: the superblock, i-node/data bitmaps, i-node table, data region.



An inode contains:

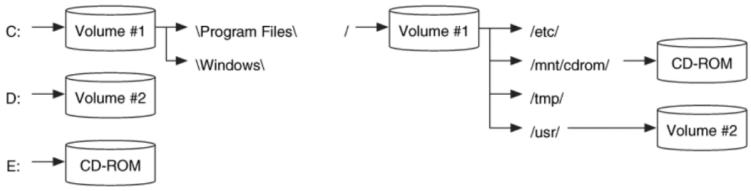
- file type
- UID/GID/permission-bits
- time information
- size in bytes
- number of hard links (AKA names)
- pointers to data blocks



To use the end-user view, a file system, stored on a **block device**, must be mounted (or "parsed")

Most modern operating systems automatically mount external storage devices when they are connected.

- In Unix/Linux there is a single root directory (/), and additional volumes are mounted within this hierarchy
- in Windows each volume (storage device/partition) is assigned a drive letter (C:, D:, E:)



Block devices can be seen as "files" themselves

- Linux special files, typically under /dev
  - various "aliases" in /dev/disk –> /by-id; /by-uuid; /bypath
  - `lsblk` lists information about available block devices

Viceversa, (image) files can be seen as block devices.

1. `losetup` command allows an image file to be treated as a virtual block device (loop device).
   - `--list`
   - `--find [--show] [--partscan] image`
   - `--detach[-all]`
2. Then, we can `mount` them.
   - Instead of manually setting up a loop device, mount can automatically create one:
   - `offset=<byte_offset>` starting point within an image file (use `fdisk -l image_file.img`)
   - `mount [-o loop] image` instead of manually setting up a loop device
   - `ro` read-only

3. Umount and check umount /dev/sda1 fsck /dev/sda1

Example

```
xz -dk two-partitions.dd.xz

# FIRST METHOD

losetup -r -o $((1*512)) /dev/loop0 two-partitions.dd  # First partition
losetup -r -o $((1026*512)) /dev/loop1 two-partitions.dd  # Second partition

# losetup -r -o $((1*512)) --find --show /tmp/two-partitions.dd
# losetup -r -o $((1026*512)) --find --show /tmp/two-partitions.dd

fdisk -l /dev/loop1 # check

mount -o ro /dev/loop0 /mnt/two-partition
mount -o ro /dev/loop1 /mnt/two-partition

# SECOND METHOD

losetup -r --find --show --partscan two-partitions.dd

mount -o ro /dev/loop0p1 /mnt/part1
mount -o ro /dev/loop0p2 /mnt/part2

umount /dev/loop0p1
umount /dev/loop0p1
```

The Sleuth Kit (TSK)

!attention: TSK always uses a term 'inode', but actually the information of filesystem is in the FAT entry directory

The Sleuth Kit (TSK) is a forensic toolkit that provides different layers of analysis for digital investigations. Each layer focuses on specific aspects of a digital storage system, allowing forensic examiners to extract and interpret data at various levels.

- `img_` for images
- `mm` (media-management) for volumes
- `fs` for file-system structures
- `j` for file-system journals
- `blk` for blocks/data-units
- `i` for inodes, the file metadata
- `f` for file names

Typically followed by:

- `stat` for general information
- `ls` for listing the content
- `cat` for dumping/extracting the content

Example

```
img_stat two-partitions.dd
img_cat two-partitions.dd

img_stat canon-sd-card.e01
```

When analyzing file systems, we categorize data into essential and non-essential based on their reliability and importance.

- Essential Data = Trustworthy & required for file retrieval.
  - If name or location were incorrect, then the content could not be read
- Non-Essential Data = Can be misleading & needs verification.
  - the last-access time or the data of a deleted file could be correct but we don't know

The Volume (or Media Management) layer in The Sleuth Kit (TSK) focuses on analyzing and managing disk partitions. This layer is crucial for identifying partition structures, extracting partitions, and verifying file system integrity.

- `mmstat image` displays the type of partition scheme
- `mmls image` displays the partition layout of a volume
- `mmcat image part_num` outputs the contents of a partition

Example

For canon-sd-card.e01

1. Find the type of partition table (mmstat)
2. List the partitions (mmls)
3. Extract the DOS FAT16 partition, by using both mmcat/dd or a dd-like tool

Check whether the SHA256 of their results match Read-only mount the FAT partition and list the files

```
mmstat canon-sd-card.e01
mmls canon-sd-card.e01

    ##OUT##
    DOS Partition Table
    Offset Sector: 0
    Units are in 512-byte sectors

         Slot      Start        End          Length       Description
    000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
    001:  -------   0000000000   0000000050   0000000051   Unallocated
    002:  000:000   0000000051   0000060799   0000060749   DOS FAT16 (0x04)
    ###

# First method (TSK toolkit)
mmcat canon-sd-card.e01 2 > fat16_mmcat.e01

# Second method
ewfmount canon-sd-card.e01 ./rawimage/ # bit a bit copy
sudo dd if=rawimage/ewf1 of=fat16_dd.dd bs=512 skip=51
sudo umount rawimage
```

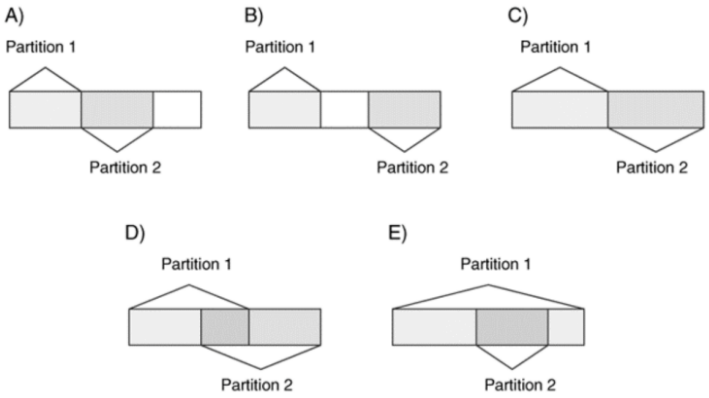
```
sha256sum  fat16_mmcat.dd fat16_dd.dd # equals


# First method (TSK toolkit)
fls -r -o 51 canon-sd-card.e01 #

#Second method
mount -o ro fat16_dd.dd /mnt/fat16_dd
tree /mnt/fat16_dd
```

DOS (or MBR) partition tables

The concept of MBR was introduced in 1983 with PC DOS 2.0.

It contain:

- machine code for the boot loader, which usually loads and executes the active-partition Volume Boot Record
- a 32-bit unique identifier for the disk, located at offset 440 (0x1B8).
- information on how the disk is partitioned –> four 16-byte entries (each at offset 446 (0x1BE)), allowing up to four primary partitions.
- last two bytes of the MBR contain the signature bytes: 0x55 0xAA.



1. Valid Configurations (A, B, and C):
    - These configurations ensure that partitions are either adjacent or properly aligned without overlap.
    - Partitions are defined in a way that does not create ambiguity in data storage.
2. Invalid Configurations (D and E):
    - D and E depict overlapping partitions, which is problematic.
    - Overlapping partitions may cause data corruption, boot issues, or system conflicts because two partitions would claim the same disk space.

CHS (Cylinder-Head-Sector) is the early method for addressing physical blocks on a disk.

It used a 3-byte structure:

- 10 bits for Cylinders (tracks stacked vertically)
- 8 bits for Heads (read/write heads on a disk platter)
- 6 bits for Sectors (sections of a track)

Replaced by Logical Block Addressing in ’90s.

- To convert you need to know the number of heads per cylinder, and sectors per track, as reported by the disk drive
- Yet, many tools still aligned partitions to cylinder boundaries

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Master Boot Code															
…																
1A0																
1B0									Disk Signature						Boot ind[1]	Start head
1C0	start Sect[2]	start Cyl[2]	Sys ID[3]	End Head	End sect[2]	End Cyl[2]	Relative Sectors				Total Sectors					
1D0																
1E0																
1F0															55	AA

1. Boot indicator 0x00 = non-boot, 0x80 = bootable
2. Starting sector & starting cylinder are allocated bits, not bytes (0x1C0-0x1C1) same goes for end head and end sector

BIT	0	1	2	3	4	5	6	7	8	9	A		B	C	D	E	F
Value	Starting sector							Starting Cylinder									

3. Common partition values.

0x01	FAT12 <32MB
0x04	FAT16 <32MB
0x05	MS Extended partition using CHS
0x06	FAT16B
0x07	NTFS, HPFS, exFAT
0x0B	FAT32 CHS
0x0C	FAT32 LBA
0x0E	FAT16 LBA
0x0F	MS Extended partition LBA
0x42	Windows Dynamic volume
0x82	Linux swap
0x83	Linux

0x84	**Windows hibernation partition**
0x85	Linux extended
0x8E	Linux LVM
0xA5	FreeBSD slice
0xA6	OpenBSD slice
0xAB	Mac OS X boot
0xAF	HFS, HFS+
0xEE	MS GPT
0xEF	Intel EFI
0xFB	VMware VMFS
0xFC	VMware swap

A Master Boot Record (MBR) is typically 512 bytes and laid out like this:

```
Offset (hex) | Size | Description
--------------------------------------------
0x000        | 446  | Bootstrap code area
0x1B8        | 4    | Disk signature (sometimes called "unique MBR signature")
0x1BC        | 2    | Usually 0x0000 or may be used for copy-protection, etc.
0x1BE        | 16   | Partition entry #1
0x1CE        | 16   | Partition entry #2
0x1DE        | 16   | Partition entry #3
0x1EE        | 16   | Partition entry #4
0x1FE        | 2    | MBR signature (0x55AA)
```

Each 16-byte partition entry has the structure:

```
Byte | Description
--------------------------------------
0    | Boot indicator (0x80 = bootable; 0x00 = non-bootable)
1-3  | Starting CHS (Head-Sector-Cylinder) - often unused in modern disks
4    | Partition type (ID)
5-7  | Ending CHS (Head-Sector-Cylinder)
8-11 | Relative sectors (start in LBA)
12-15| Total sectors in this partition
```

Example

Use ImHex, writing proper patterns, to extract disk and partition information from mbr{1,2,3}.dd. Then, answer the following questions:

1. What are the three disk signatures?

2. Is there any MBR with inconsistent partitioning?
3. Are there MBRs without bootable partitions?
4. What is the largest FAT (id=4) partition?
5. Are CHS information always present?

(fdisk -l mbr2.dd)

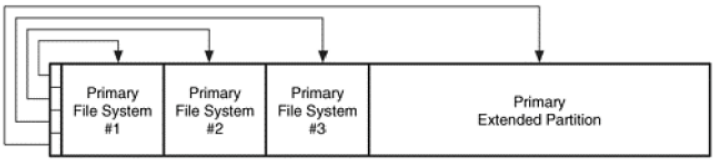
```
tar -tJf MBR123_and_GPT.tar.xz
tar -xJvf MBR123_and_GPT.tar.xz mbr1.dd

xxd -s 0x1B8 -l 4 mbr1.dd
```

Pattern editor

```
// fdisck give a same informations

#include <std/mem.pat>
struct PartitionEntry {
  u8  bootIndicator;
  u8  startCHS[3];
  u8  partitionType;
  u8 endCHS[3];
  u32 relativeSectors;
  u32 totalSectors;
};

struct MBR {
  u8 bootCode[0x1B8];       // 446 bytes
  u32 diskSignature;        // offset 0x1B8
  u16 reserved;             // offset 0x1BC (often 0x0000)
  PartitionEntry partitions[4];  // 4 partition entries, each 16 bytes
  u16 signature;            // offset 0x1FE, should be 0x55AA
};

 MBR seg[while(!std::mem::eof())] @ 0x00;
```

Extended partitions

MBR has only 4 slots for primary partitions.



To work around this limitation, one slot can be used for the primary extended partition, a partition containing other partitions.

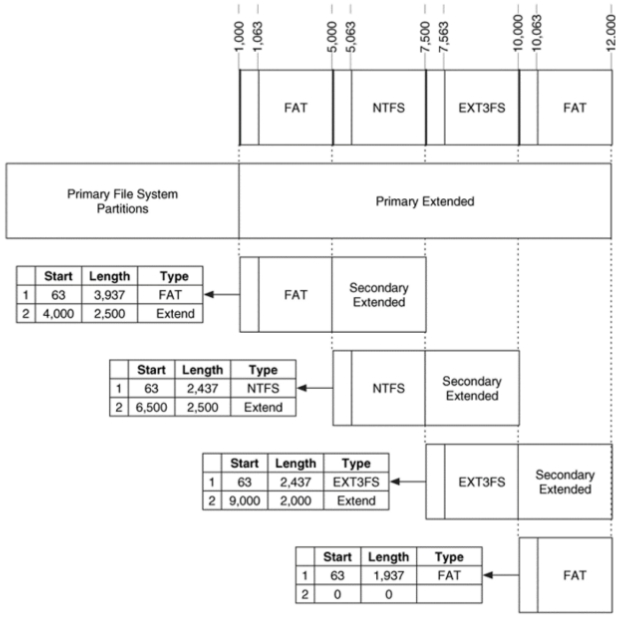
Beware of logical-partition addressing, which uses the distance from the beginning of a partition (vs the physical-addressing, from the beginning of the whole disk).

Inside the primary extended partition we find secondary extended partitions, containing

- a partition table t (with the same 512-byte structure)
- a secondary file-system partition p (logical partition), which contains a FS or other data

The partition table (t) describes:

1. Location of p (logical partition) relative to t.
2. Next secondary extended partition (if any), w.r.t. the primary extended partition



Example

ext-partitions.dd (SHA256: b075ed83211…) contains three partition tables: one primary, two extended. Analyze them with ImHex, and compare the result w.r.t. fdisk/mmls Source: (https://dftt.sourceforge.net/test1/index.html)

```
mmls ext-partitions.dd
```

```
      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000062   0000000063   Unallocated
002:  000:000   0000000063   0000052415   0000052353   DOS FAT16 (0x04)
003:  000:001   0000052416   0000104831   0000052416   DOS FAT16 (0x04)
004:  000:002   0000104832   0000157247   0000052416   DOS FAT16 (0x04)
005:  Meta      0000157248   0000312479   0000155232   DOS Extended (0x05) #15724*512 = address
006:  Meta      0000157248   0000157248   0000000001   Extended Table (#1)
007:  -------   0000157248   0000157310   0000000063   Unallocated
008:  001:000   0000157311   0000209663   0000052353   DOS FAT16 (0x04)
009:  -------   0000209664   0000209726   0000000063   Unallocated
010:  001:001   0000209727   0000262079   0000052353   DOS FAT16 (0x04)
011:  Meta      0000262080   0000312479   0000050400   DOS Extended (0x05)
012:  Meta      0000262080   0000262080   0000000001   Extended Table (#2)
013:  -------   0000262080   0000262142   0000000063   Unallocated
014:  002:000   0000262143   0000312479   0000050337   DOS FAT16 (0x06)
```

Example

Someone purposely damaged the partition table of hidden-truth.dd (SHA256: 5f39a8965ec…)

1. Can you (ro) mount the partitions?
2. Can you repair the broken MBR and mount the deleted partition?

hint (ROT13): Lbh pna ernfba nobhg gur ynlbhg be trg fbzr uryc jvgu fvtsvaq (sebz GFX)

3. Can you recover the password protected "secret"?

```
fdisk -l hidden-truth.dd

    ##OUT##
    Device           Boot      Start        End   Sectors  Size Id Type
    hidden-truth.dd1               2       2050      2049    1M  4 FAT16 <3
    hidden-truth.dd2      1751214177 2311246017 560031841  267G 74 unknown
    hidden-truth.dd3            3076       8191      5116  2.5M  4 FAT16 <3
    Partition table entries are not in disk order.
    ###


 dd if=hidden-truth.dd of=hid2.dd bs=512 skip=3076 count=5115
 dd if=hidden-truth.dd of=hid1.dd bs=512 skip=2 count=2048

 ls -l hidden-truth.

    ##OUT##
    -rwxrwxrwx 1 vagrant vagrant 4194304 Mar 13 19:37 hidden-truth.dd
    # echo $((4194304/512)) == 8192 sectors
    ###

# try to find a sector 1 (0 is a boot sector)
dd if=hidden-truth.dd bs=512 skip=1 count=1 | xxd -g1

    ##OUT##
    .....
    000001e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
    000001f0: 34 37 36 39 36 66 37 36 33 34 36 65 36 65 32 31  47696f76346e6e21
    # only clue ...
    ###

# try a "brute" mount
mount -o ro, offset=$((2051*512)) hidden-truth.dd /mnt/hidden-brute

    ##OUT##
    not_so_secret.zip # a zip with pass
    ###


# does the fs cover the whole prtion in the middle?
# it is true that there is some space between the first and third partition but
# we do not know if all the space in between has been used
fsstat  hidden-truth.dd -o 2051
```

```
    ##OUT##
    File System Layout (in sectors)
    Total Range: 0 - 1023 # 3075-2051 == 1024, okey seems fair
    ###

# using a hex brefore and cyberchef
47696f76346e6e21 --> Giov4nn!
```

GPT - GUID PArtition Tables

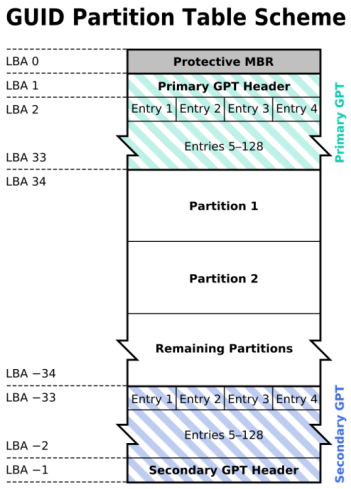
A Universally/Globally Unique IDentifier (UUID/GUID) is a 128-bit label.

- Uniqueness: Properly generated UUIDs are statistically unique, meaning the probability of duplication is extremely low.
- Standard Format: UUIDs are typically written in a 32-character hexadecimal format divided into five groups: 8-4-4-4-12, separated by hyphens.

```
uuidgen

bdeec955-b1b8-44a2-8034-15507d431aca
```

The GPT format, used by the Extensible Firmware Interface (EFI), which replaced BIOS, is the current standard on PCs; it

- starts with a protective MBR
- supports up to 128 partitions
- uses 64-bit LBA addresses
- keeps “mirrored” backup copies of
- important data structures



Example

Use ImHex, writing proper patterns, to extract disk and partition information from gpt.dd. Then, answer the following questions:

1. What is the disk GUID?
2. How many partitions are there?
3. What are the partition names?
4. Can you find the partition type GUIDs in the previous table?

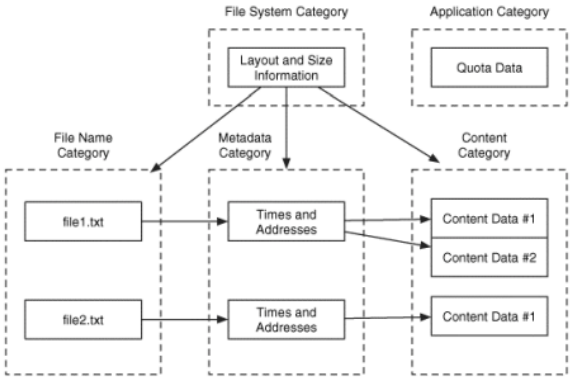
```
gdisk -l gpt.dd
mmls -t gpt gpt.dd
```

File System Analysis

A reference model for a file system based on different categories of data that are involved in file storage and management.

- File System Category: layout and size information about the entire file system, such as: file system parameters (e.g., block size, total size) the structure or mapping of data storage
- Content The actual data, stored in clusters/blocks/data-units
- MetaData: Data that describes files: size, creation date

- File Name Data that assign names to files
- Appliccation: Data not needed for reading/writing a file; e.g., user quota statistics or a FS journal



To get the general details of a file-system - `fsstat [-o sect_offs] image`

Example

1. Find the OEM Name and Volume Label (Boot Sector) in canon-sd-card.e01

```
mmls canon-sd-card.e01
```

```
    ##OUT##
        Slot      Start        End          Length       Description
    000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
    001:  -------   0000000000   0000000050   0000000051   Unallocated
    002:  000:000   0000000051   0000060799   0000060749   DOS FAT16 (0x04)
    ###
```

```
fsstat -o 51 canon-sd-card.e01
```

2. Check whether the partition types are correctly set inside two-partitions.dd

```
mmls two-partitions.dd
```

```
    ##OUT##
        Slot      Start        End          Length       Description
    000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
    001:  -------   0000000000   0000000000   0000000001   Unallocated
    002:  000:000   0000000001   0000001025   0000001025   DOS FAT16 (0x06) # wrong
    003:  000:001   0000001026   0000002047   0000001022   DOS FAT12 (0x01)

    ### Partition table can be modified
    ###
```

```
fsstat -o 1 two-partitions.dd # FAT12
fsstat -o 1026 two-partitions.dd # FAT12
```

Example

inside the image file two-partitions.dd

1. look for the strings

- “didattica”
- “wDeek”"
- "tool
- “secret”

```
strings two-partitions.dd | grep -E "didattica|wDeek|tool|secret" # secret,wDeek,didattica
or
strings two-partitions.dd | ag "didattica|wDeek|tool|secret"
or
xxd -g1 two-partitions.dd| grep -C 3 ecre
```

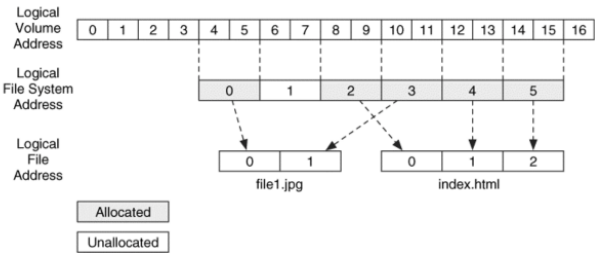
2. (ro) mount its partitions, and look for the same strings inside the contained files

```
losetup -r --find --show --partscan two-partitions.dd

mount -o ro /dev/loop0p1 /mnt/part1
mount -o ro /dev/loop0p2 /mnt/part2

grep -rE "didattica|wDeek|tool|secret" /mnt/part1 # null
grep -rE "didattica|wDeek|tool|secret" /mnt/part2 # tool
```

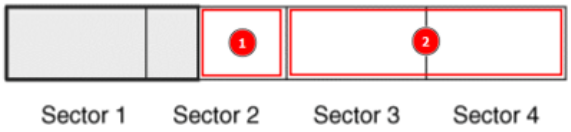
Do some string appear only in one of the two searches? Can you guess why?

Each sector can have multiple addresses, relative to the start of the…

- storage media: physical address
- volume: (logical) volume address
- FS [data area]: (logical) FS address AKA (logical) cluster number
- file: (logical) file address AKA virtual cluster numbers



When writing a 612-byte file in a file system with 2K clusters (where each sector is 512 bytes), the way data is allocated creates slack space—unused but allocated storage that may contain remnants of previous data.



When investigating deleted files, forensic analysts use two major approaches:

1. Metadata-based

   If the file is deleted but metadata still exists, we can recover:

   - File size, timestamps, and allocated sectors/clusters.
   - Orphaned files (files with no full path reference)

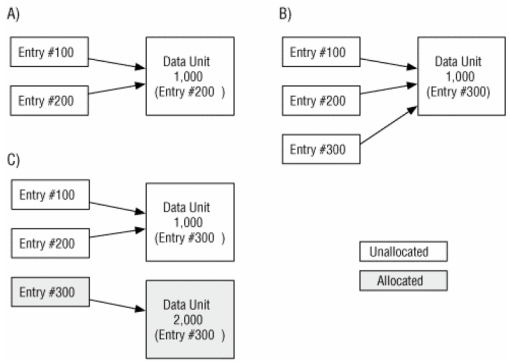
2. Application-Based

Used when metadata is unavailable:

- Typically from un-allocated space
- Does not need any FS information

Example

Where do data-units come from?



A.

- Entry#100 initially points to Data Unit 1,000.
- Entry #200 is created after #100 is deleted and reuses the same data unit.

This means that even after deletion, old data might still be recoverable unless overwritten.

C.

- Entry #300 is now assigned a completely new Data Unit (2,000).
- Entry #100 and #200 had used Data Unit 1,000, but it is now unallocated.

The original content in Data Unit 1,000 might still be present but no longer linked to any active file. (Carving).

TSK metadata commands

The Sleuth Kit (TSK) provides powerful commands to analyze file system metadata, particularly focusing on inodes, which store key file attributes.

1. `ils [-o sect_offs] image` - list inode information
    - `-r` → Lists only removed (deleted) files
    - `-a` → Lists only allocated (active) files
    - `-m` → Displays inode details in a format compatible with mactime (used for timeline analysis)
2. `istat [-o sect_offs] image inum`- dumps detailed metadata of a specific file or inode

!!! TSK uses the inode abstraction even for file systems that do not natively have them.

- Some file systems (e.g., FAT32) do not have inodes, but TSK emulates them to allow a consistent analysis approach

3. `ifind [-n filename] [-d data-unit] [-o offset] image` - viceversa, to find the inode corresponding to a data-unit or file name
    - `strings -t d disk-image.dd | grep "password"`- gives you an offset (e.g., `123456`) where the data appears.
    - `ifind -d $((123456/4096)) disk-image.dd`- $((n/block-size)) returns the inode number

4. `ffind [-o sect_offs] image inum` - lists the names using the inode (useful when names are not inside the "inode")
5. `icat [-o sect_offs] image inum` - extracts and displays the contents of a file based on its inode number.
   - `-s` → Includes slack space (unused space in the last cluster of a file)
   - `-r` → Attempts to recover deleted files

   Note: deleted content may be present in unallocated data (without metadata pointing to it). To check/dump blocks:
   - `blkstat image block` - displays metadata about a specific block (e.g., allocation status, timestamps, etc.)
   - `blkcat image block [how-many-blocks]` - outputs the raw content of a specific block
   - `blkls` - lists or outputs blocks too
6. `fls [-o sect_offs] image [inum]` - list files inside the directory corresponding to the inode number
7. `ffind [-o sect_offs] image inum` - lists the names using the inode (useful when names are not inside the "inode")

Example

Let's find out why some of the following strings appear in one search and not the other

```
mmls two-partitions.dd
```

```
##OUT##
     Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000000   0000000001   Unallocated
002:  000:000   0000000001   0000001025   0000001025   DOS FAT16 (0x06)
003:  000:001   0000001026   0000002047   0000001022   DOS FAT12 (0x01)
###
```

```
strings -t d two-partitions.dd | grep -E "didattica|wDeek|tool|secret"
```

```
##OUT##
20514 and I have a secret message ;)
547396 wDeek
547436 /home/gio/didattica/file-systems/vol_fs_analysis/examples/pp/test
###
```

- End of first partition = (1025*512) = 524800 –> "secret" is in the firts partition.
- Start of second partition = (1025*512) = 525312 –> "wDeek" is int the second partition.
- End of second partition = (1025*512) = 1048064 —> "didattica" is in the seconf

1. Find "secret"
   - Sector number of "secret" = (20514/512) = 40 **at begining of the disk, but partition start at sector 1**
   - Offset = (40-1) = 39

```
ifind -d 39 -o 1 two-partitions.dd # get 4 (a indo of block 39 f partition start 1)

istat -o 1 two-partitions.dd 4
```

```
##OUT##
```

```
    Directory Entry: 4
    Allocated
    File Attributes: File, Archive
    Size: 34
    Name: HELLO.TXT

    Directory Entry Times:
    Written:        2023-03-16 08:57:32 (EDT)
    Accessed:       2023-03-16 00:00:00 (EDT)
    Created:        2023-03-16 08:57:32 (EDT)

    Sectors:
    39 0 0 0 # use only one sector
    ###

icat -s -o 1 two-partitions.dd 4

    ##OUT##
    Hi there! ...
    ###

# We note that the size of ls hello is (34B) < oh the size dd rows (64)

dd if=two-partitions.dd bs=512 count=1 skip=40 | hexdump -C
ls -l hello.txt
```

2. Find “wDeek” and “didattica”
    - Sector number of “secret” = (547396/512) = 1069 **at begining of the disk, but partition start at sector 1026**
    - Offset = (1069-1026) = 43

```
ifind -d 43 -o 1026 two-partitions.dd # get 6 (a indo of block 43 of partition start 1026)

istat -o 1026 two-partitions.dd 6

    ##OUT##
    Directory Entry: 6
    Not Allocated # DELETED --> not mounted by OS
    File Attributes: File, Archive
    Size: 4096
    Name: _EST~1.SWP

    Directory Entry Times:
    Written:        2023-03-16 09:04:10 (EDT)
    Accessed:       2023-03-16 00:00:00 (EDT)
    Created:        2023-03-16 09:04:10 (EDT)

    Sectors:
    43 44 45 46 47 48 49 50
    ###

icat -o 1026 two-partitions.dd 6 | strings
```

```
    ##OUT##
    b0VIM 8.2
    root
    wDeek
    /home/gio/didattica/file-systems/vol_fs_analysis/examples/pp/test
    3210
    #"!

    ###
```

3. Find “tool”

```
fls -rp two-partitions.dd -o 1026

    ##OUT##
    r/r 4:  wikipedia.txt
    r/r * 6:        .test.swp
    r/r * 8:        test
    v/v 16083:      $MBR
    v/v 16084:      $FAT1
    v/v 16085:      $FAT2
    V/V 16086:      $OrphanFiles
    ###


istat -o 1026 two-partitions.dd 4

    ##OUT##
    Directory Entry: 4
    Allocated
    File Attributes: File, Archive
    Size: 3934
    Name: WIKIPE~1.TXT

    Directory Entry Times:
    Written:        2023-03-16 09:04:24 (EDT)
    Accessed:       2023-03-16 00:00:00 (EDT)
    Created:        2023-03-16 09:04:24 (EDT)

    Sectors:
    39 40 41 42 55 56 57 58
    # we see that the cluster is not consecutive and string "tool"
    # is fragmented in "to...ol"
    ###

icat -o 1026 two-partitions.dd 4 | strings | hexdump -C | grep -C 6 tool
```

Carving

Is a method of recovering files without relying on metadata (like file names, paths, or inodes). It works by identifying file signatures (headers & footers) and extracting the data between them.(E.g., 0xFF 0xD8 and 0xFF 0xD9 for JPEG files).

Example

Inside eighties.dd (cc121c3a037f904a4fa5ef51263df9fdb800d89af7330df22615802b81821f9d) there is a FAT file system with some deleted content. In particular, there were files with the following SHA256 hashes:

- 4410aaee5ae15917c064f80a073ec75260482b7035fad58c85f1063d0b795733
- 1b756ad00ad842c3356c093583e2e4fab2540e15ca88750606f45f7efd1f4d26
- 592f47dfcbeda344fc394987b6e02a65a35d4d849d35d2fc821e5be1889c645d
- 8a461036c70736eb4ca83e9062318c8293be2baad1c475c41c1945221559048e
- 0d176b77f6b81468eb7ba367d35bdcbd8fdfc63445c2cc83c5e27c5e0b4c1a14

Can you recover and identify them?

```
fls -rp eighties.dd or ils -r eighties.dd
    ##OUT##
    # seven deleted (*)

    r/r * 3:        _
    r/r * 4:        _
    r/r * 5:        _8.gif
    r/r * 6:        _8.txt
    v/v 523203:     $MBR #
    v/v 523204:     $FAT1
    v/v 523205:     $FAT2
    V/V 523206:     $OrphanFiles
    -/r * 517:      $OrphanFiles/_live.jpg
    -/r * 518:      $OrphanFiles/_8k.jpg
    -/r * 581:      $OrphanFiles/_monty.tzx
    ###

icat eighties.dd 6 | sha256sum # ok
icat eighties.dd 5 | sha256sum # icat fail sha

icat eighties.dd 5 | xxd -g1 # magic is gif, but there is some text
icat eighties.dd 5 > fake.gif # mmm...blurred ...

# wait ... two files cannot share the same sector

istat eighties.dd 5
    ##OUT##
    Sectors:
    108 109 110 111 112 0 0 0
    ###

istat eighties.dd 6
    ##OUT##
    Size: 2426
    Name: _8.gif
    Sectors:
    112 113 114 0
    ###
```

```
# Two possible cases:
# 1. the file may have been overwritten (nothing can be done)
# 2. the gif was framed,
    # first it was created a txt after gif ...
    # the gif used space that was previously empty in the txt file and continued to use other space

# This information is saved in FAT, but when a file is deleted, the cluster chain is lost.
# then we can recover the first cluster and thanks the length  we can find other cluster

# cluster size is 4 sectors
fsstat eighties.dd
    ##OUT##
    Cluster Size: 2048 # 2048/512=4
    ###

img_cat -s 108 -e 111 eighties.dd > fake2.gif
img_cat -s 116 -e 119 eighties.dd >> fake2.gif # okey noe its clear
sha256sum fake2.gif # but not yet, because here we have taken two clusters
# the file must be size: 2426, these bytes actually make up the .gif file
dd bs=1 count=2426 if=fake2.gif of=speriamo.gif
sha256sum speriamo.gif # OK 1b756...
```

The FAT File System Family

The FAT (File Allocation Table) File System is one of the earliest and simplest file systems, first developed in 1977/1978. Over time, it evolved into three main versions: FAT12, FAT16 and FAT32.

The number indicates the # of bits used to identify clusters

- FAT12 can address 212 = 4096 clusters. Windows permits cluster sizes from 512 bytes to 8 KB, which limits FAT12 to 32 MB
- FAT16 can address 216 = 65, 536 clusters
- FAT32 can address 228 clusters (top 4 bits used for other purposes)

Actually, first 2 & last 16 are reserved: usable clusters are slightly less.

Uses the MSDOS 8.3 filename format – Only 8 characters for the name + 3-character file extension (e.g., FILE1234.TXT).

VFAT (Virtual FAT) extends FAT to support long filenames with Unicode, maintaining backward compatibility.

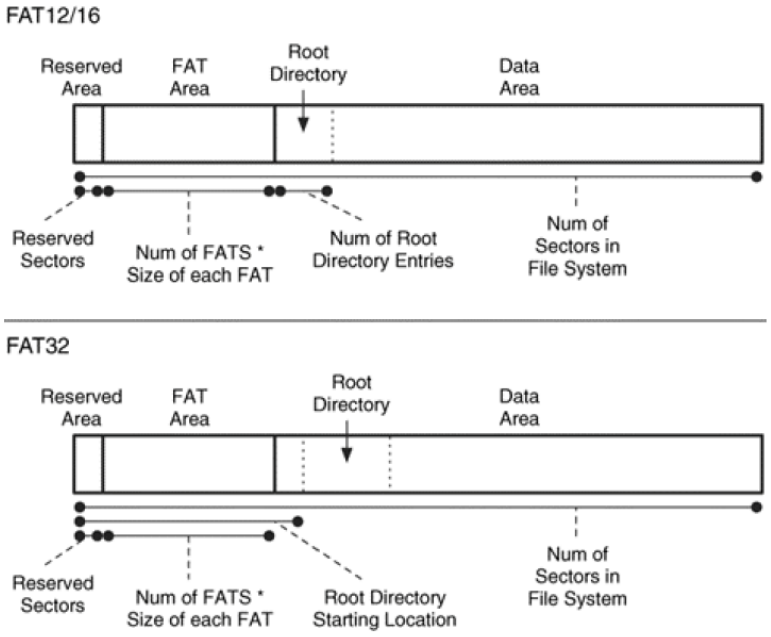
File sizes are stored as 32-bit integers, meaning the largest file size FAT32 can handle is 4 GB.

File sizes are stored as 32-bit integers.

Volume Organization

In FAT file systems, the storage device is divided into specific regions, each serving a defined role:

- The Volume Boot Record (VBR) contains the so-called BIOS Parameter Block
- The root directory of FAT12/16 has a fixed location and size
- FAT32 boot sector includes the locations of the root directory, FSINFO structure (that keeps track of free clusters, to optimize allocations), and boot-sector backup (should be 6)



1. Reserved Area

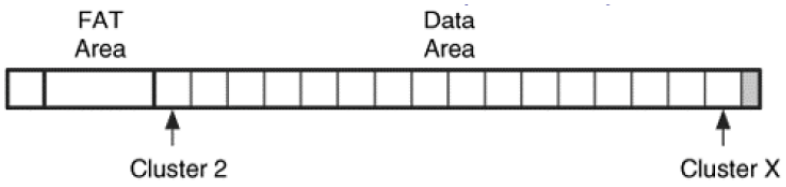
- Starts at sector 0.
- Contains the Volume Boot Record (VBR or Boot sector), which holds key information about the file system.
- FAT12/16: Usually 1 sector (only the VBR); FAT32: Larger because it includes FSINFO structure (helps track free clusters).

2. FAT Area

- follows the reserved area, and its size is calculated by multiplying the number of tables by their size

3. Data Area

- Clusters are only in Data Area, numbered from 2 (!!!) and after the root directory for FAT12/16
- Data could be also hidden after the last valid entry in a FAT table



The “Small Sectors” and “Large Sectors” fields represent the total number of sectors in the volume. Only one of these fields is used, and the other is set to zero.

Green (BIOS Parameter Block - BPB):

- Essential fields required for the basic operation of the file system.
- Defines sector sizes, cluster sizes, and disk structure.

Yellow (Extended BIOS Parameter Block - EBPB):

- Additional metadata introduced in later FAT versions.
- Includes details like the Volume Serial Number, Boot Signature, and Volume Label.

Boot sector FAT12/16

FAT16 Boot Sector

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Jump Instruction			OEM ID								Bytes / sect		sect / cluster	reserved sectors	
10	no / FATS	Root entries		Small Sectors		Media descriptor	Sectors / FAT		Sectors / Track		Number / heads	Hidden sectors				
20	large Sectors				Physical drive number	reserved	ext boot sig	Volume Serial Number				Volume Label (deprecated)				
30	Volume label						File System Type									
40	OS Boot Code															
50																
60																
70																
…																
1D0																
1E0																
1F0															55	AA

Boot sector FAT32

FAT32 Boot Sector

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Jump Instruction			OEM ID								Bytes / sect		sect / cl uster	reserved sectors	
10	no / FATS	0x0000		0x0000		Media descr iptor	0x0000		Sectors / Track		Number / heads	Hidden sectors				
20	large Sectors Total sectors in volume						Sectors / FAT		0x0000		File System Version	Root(first) Cluster Number				
30	FS Info sector		Backup boot sector	Reserved												
40	Phys Drive num	0x00	Extd boot sig	Volume Serial Number			Volume Label (deprecated) normally "NO NAME "									
50	Vol Label			System ID "FAT32"					Boot code							
60	Boot code															
70																
80																
90																
…																
1D0																
1E0																
1F0															55	AA

https://www.writeblocked.org/resources/FAT_cheatsheet.pdf

Files and Directories

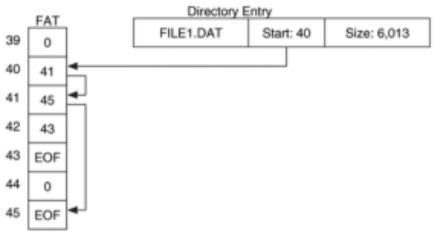
A directory entry in FAT file systems is a 32-byte record that stores metadata about a file or directory.

FAT Directory Entry

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	File name								Extension			attr ibute	reserved	10ms create time[1]	create time	
10	create date		last access date		unused		modified time		modified date		start cluster		File Size			

1. The 10millisecond create time is technically only used in FAT32.

The File Allocation Table (FAT) keeps track of file storage using cluster chains. Each file's data is stored in clusters, and the FAT table links these clusters together to form a chain.



- Each FAT entry points to the next cluster in the file's.
- Clusters marked EOF (End of File) indicate the last cluster of a file.

`fsstat` decodes this chains (in sectors); special values:

```
0 → not allocated
0xf...ff0-0xf...f6 → reserved
0xf...ff7 → damaged
0xf...ff8-0xf...fff → EOF
```

Note: FAT entries start at 0, but:

- The first addressable cluster #2.

- Entry 0 typically stores a copy of the media type, and entry 1 stores the dirty-status of the file system

Example

In eighties.dd (SHA256: cc121c3a…) and eighties-all-files.dd (SHA256:e5f16884…) you'll find two very similar FAT16 (not VFAT) file systems. In the former all files have been deleted. Using ImHex.

1. find out: Sector and cluster sizes Number of reserved sectors Locations of: FAT1, FAT2, Root Dir. (=Data Area), first cluster (#2)

compare these results with the output of fsstat

2. check the FAT entries for 48.gif in the two dd-images, and compare the results of istat on "inode" 5

```
fls -r eighties-all-files.dd
  ##OUT##
  d/d 3:  jpgs
  + r/r 517:      clive.jpg
  + r/r 518:      48k.jpg
  d/d 4:  games
  + r/r 581:      mmonty.tzx
  r/r 5:  48.gif
  r/r 6:  48.txt
  v/v 523203:     $MBR
  v/v 523204:     $FAT1
  v/v 523205:     $FAT2
  V/V 523206:     $OrphanFiles
  ###

istat eighties-all-files.dd 5
  ##OUT##
  Sectors:
  108 109 110 111 116 0 0 0 #l'ha recuperato dentro la FAT
  ###

fsstat eighties-all-files.dd

  ##OUT##
  FAT CONTENTS (in sectors)

  100-103 (4) -> EOF
  104-107 (4) -> EOF
  108-111 (4) -> 116 # cluster chain
  112-115 (4) -> EOF
  116-119 (4) -> EOF
  120-331 (212) -> EOF
  332-1211 (880) -> EOF
  1212-1279 (68) -> EOF
  ###
```

When a file name exceeds the 8.3 format, the file system creates additional directory entries to store the name in Unicode (2 bytes per character).

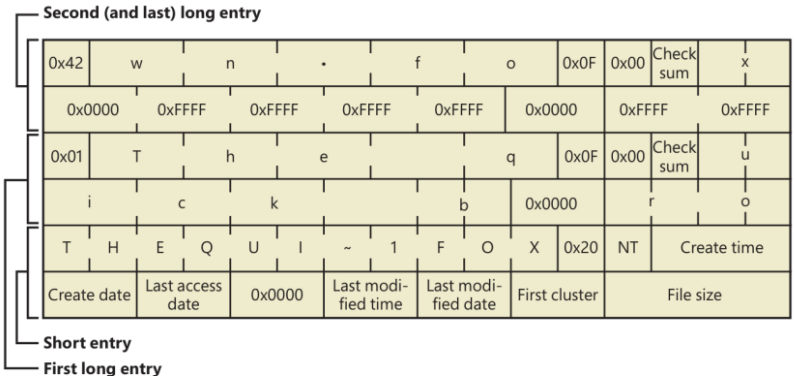
These LFN entries are linked together and precede the main directory entry (which still stores the short 8.3 name for compatibility).

- Each LFN entry is marked with the attribute 0x0F, meaning it is not treated as a normal file entry.
- The last LFN entry in the sequence has its sequence number OR-ed with 0x40; or 0xe5 if unallocated.

Long File Name

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	file name (Unicode 2 bytes/char)											0x0F	reserved	Check sum	file name	
10	file name										0x0000		file name			

The quick brown.fox", as THEQUI~1.FOX in 8.3 convention.



When a new directory is created, it contains

. and ..

Cluster 110

Name	Created	Cluster
dir2	3/30/04 01:29:01	128
dir1	4/03/04 11:47:40	196
file8.dat	3/30/04 20:41:12	112

Cluster 196

Name	Created	Cluster
.	4/1/04 09:27:00	196
..	4/1/04 09:27:00	110
file1.dat	4/3/04 12:58:23	297

those entries can be helpful for carving deleted directories

A)

Cluster 45

.	45
..	26
bad.jpg	105
dir1	210
good.jpg	153

Cluster 210

.	210
..	45
file1.txt	211
file2.txt	250
file3.txt	273

B)

Cluster 45

.	45
..	26
bad.jpg	105
newfile	**400**
good.jpg	153

Cluster 210

.	210
..	45
file1.txt	211
file2.txt	250
file3.txt	273

Since the size of a directory is always 0, the only way to know how many cluster to read is following the cluster chain

Example

In eighties-vfat.dd (SHA256: 62258f92ebb42226…) and eighties-vfat-all-files.dd (SHA256: fe46141b98d227cb…) you'll find two very similar, and familiar, VFAT FAT16 file systems. As with the previous exercise, in the former all files have been deleted.

Yet, fls -rp eighties-vfat.dd can show the full, long name, for some deleted files but not for others, that are listed under $OrphanFiles.

1. Can you explain why? Hint: eighties-vfat-all-files.dd contains some clues

In some cases, even if the file is cacelled, we have the full name, and it is strange because in the fat one byte '_' is put above the first character (eighties.dd). Since we have a vfat there are the entries with the long name and we can trace the original name. OrphanFiles is a standrd used by TSK when it does not have babstanz ainomraizons to know where that file is.

2. Using ImHex, can you manually recover the full names from eighties-vfat.dd?

```
fls -rp eighties-vfat.dd
  ##OUT##
  r/r * 3:        _
  r/r * 4:        _
  d/d * 6:        Games
  r/r * 583:      Games/Mutant Monty.tzx
  r/r * 7:        _8.gif
  r/r * 8:        _8.txt
  v/v 523203:     $MBR
  v/v 523204:     $FAT1
  v/v 523205:     $FAT2
  V/V 523206:     $OrphanFiles
  -/r * 519:      $OrphanFiles/_LIVES~1.JPG
  -/r * 520:      $OrphanFiles/_8k.jpg
  ###
```