FIIT, STU Iľkovičova 2, 842 16 Bratislava

FORENSICS AND COUNTER FORENSICS ON EXT4

Final project

Study: Informatics
Subject: Forensic Analysis of Computer Systems

Contents

1. Motivation

In my project I focused on LINUX forensics and counter forensics on filesystem EXT4. Currently, there are multiple operating systems, using different filesystems across the world (figure 1), some of them are more reliable than others (figure 2) and data from them may be returned with higher recovery rate. As shown in figure 1 currently most used OS is Windows (NTFS or FAT), OS X (APFS or HFS) and LINUX (EXT or UFS). One of my main reasons, why I chose LINUX's default filesystem over Windows's and Apple's is due to more open-source tools, information provided by standards and bigger community of supportive people.

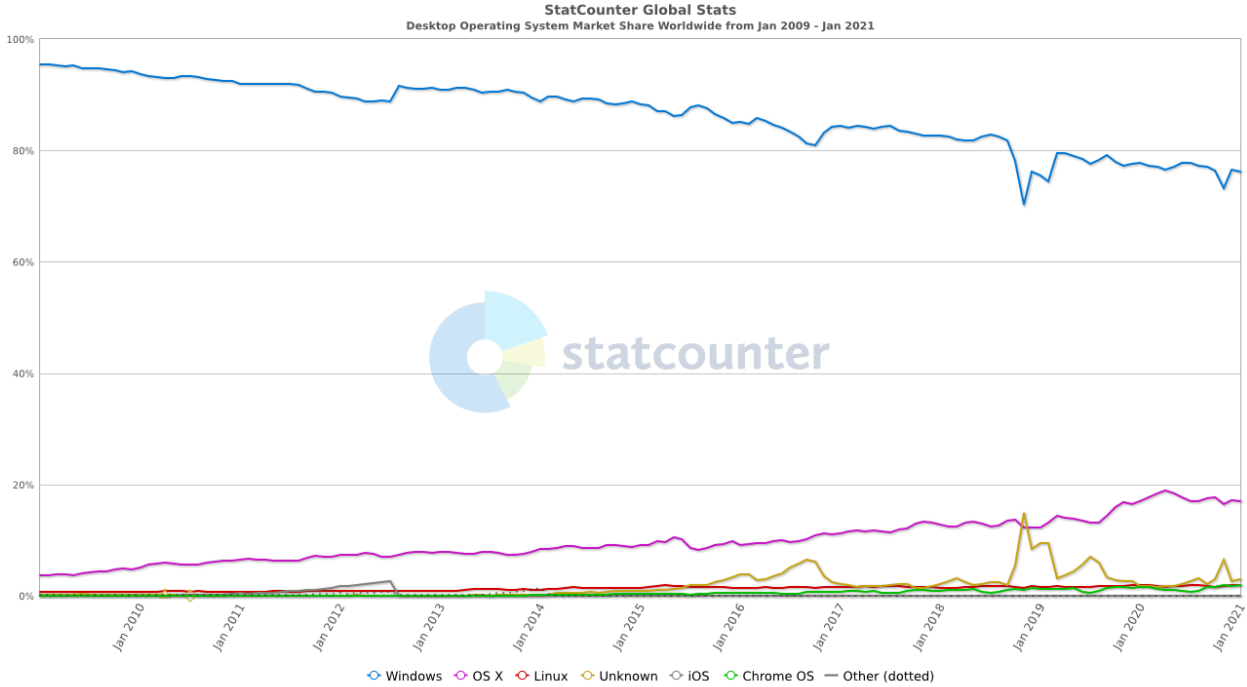
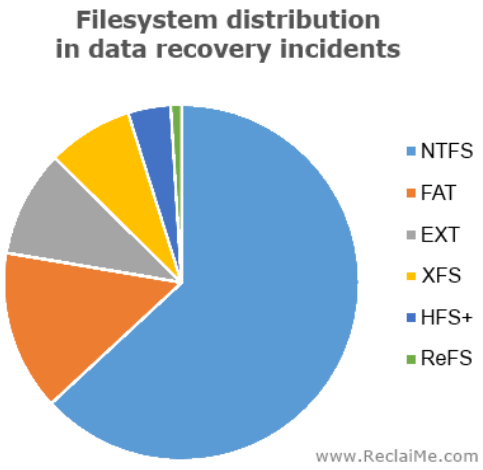


Figure 1 Most used operating systems from 2009 till 2021 (9)



NTFS	63%
FAT	14%
EXT	10%
XFS	8%
HFS+	4%
ReFS	1%

Figure 2 Most filesystems in data recovery incidents according to ReclaiMe

2. Introduction

This project covers 5 different sections of EXT4 filesystem, with different approaches to forensics, counter forensics, and programming simple data recovery tool.

In the beginning are discusses basics about EXT4, for example layout, predecessors, and plans. This section is relevant mainly because of tool that I programmed during this project and for its better understanding. All values in tables from section 3 are only the most important ones, that were needed for programming mentioned tool.

After covering basics, this project focuses on comparison of EXT4 with its rivals, for example NTFS, HFS+, and FAT32. This comparison is done from three different angles. Firstly, EXT4 is compared from overhead, secondly from recovery rate and lastly from performance point of view. Two out of three of these comparisons have been done by me, the last one is from source listed below.

Main part of the project is focus on programming simple data recovery tool. This tool is written in python and uses as little third-party software as necessary to provide better functionality and reliability. Secondly, whole code is published on my GITHUB repository with open-source approach under GNU public license.

Forensics is covered next to better understand how this tool works and what are most common techniques when deeper investigation is needed. In section below is mentioned experiment done on EXT4, about how many data might be restored after wiping data with DD command only.

Ending of this project is focused on counter forensics and steps needed to make life harder for forensic analyst. This section is covered from 3 different views with each containing 3-4 anti-forensic techniques.

3. Introduction to EXT4

As stated above, this section will provide basic information needed to understand EXT filesystem, with knowledge needed to program data recovery tool from section 5. EXT4 is journaling filesystem (fourth **ext**ended filesystem) for Linux, developed in 2006 as the successor to ext3. One of its main features is backwards compatibility with EXT3 and performance, stability, and storage limit improvements. Currently there are multiple well-known companies using mentioned filesystem like Google (in their infrastructure), Android and so on (3).

	EXT2	*EXT3*	*EXT4*
Introduced	1993	2001	2006
Maximum file size	16GB ~ 2TB	16GB ~ 2TB	16GB ~ 16TB
Maximum filesystem size	2TB ~ 32TB	2TB ~ 32TB	1EB
Features		Journaling	- Extents - Multiblock allocation - Delayed allocation

Table 1 represents comparison of EXT4 with its predecessors.

In the table above we can see main strengths of EXT4 over EXT2, EXT3, together with bigger maximum file size, file system size and EXT4 backwards compatible with EXT3. Three main features provided are:

1. Extents – reduce number of metadata needed to keep track of the data blocks for large files.
2. Multiblock allocation – allocation of many blocks in a single call.
3. Delayed allocation - works by deferring the mapping of newly written file data blocks to disk blocks in the filesystem until writeback time (10).

Information needed to understand layout of whole filesystem stands on understanding device partitioning and MBR (master boot record size). Figure 3 below shows simple structure of EXT4 layout.

Group 0 Padding	ext4 Super Block	Group Descriptors	Reserved GDT Blocks	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1024 bytes	1 block	many blocks	many blocks	1 block	1 block	many blocks	many more blocks

Figure 3: Layout of EXT4 main sections (7)

3.1. Superblock

After understanding layout, we can focus on the next parts of EXT4 filesystem. The most important part of EXT4 is the superblock. Layout of EXT4 Super Block is shown in table 2 (only the most important values are mentioned, needed for section 5).

OFFSET	SIZE	DESCRIPTION
0X00	32b	Total inode count
0X04	32b	Total block count
0X0C	32b	Free block count
0X10	32b	Free inode count
0X18	32b	Block size in 2^(10+ this value)
0X20	32b	Blocks per group
0X24	32b	Clusters per group
0X28	32b	Inodes per group
0X38	16b	Magic number, 0xEF53
0X58	16b	Size of inode structure in bytes
0XE0	32b	Inode number of journal file

Table 2: Superblock layout on EXT4 with only the most important values mentioned (7), more details in section 5.

3.2. Group descriptor

Right after superblock are located group descriptors which provide information about position of iNode table which is analyzed next.

OFFSET	SIZE	DESCRIPTION
0X08	32b	Lower 32b of location of inode table
0X28	32b	Upper 32b of location of inode table

Table 3: Group descriptor layout (7)

3.3. iNode table

iNode table consists of separate iNodes representing information about files stored on disks and what filesystem needs to know about file beside file name.

OFFSET	SIZE	DESCRIPTION
0X00	16b	File type and attributes (0x8000 - Regular file)
0X04	32b	Lower 32b of size in bytes
0X14	32b	Deletion time, in seconds since the epoch
0X1A	16b	Hard link count
0X1C	32b	Lower 32b of block count
0X28	60B	Block map or extent tree

Table 2:iNode structure layout (7)

Data at position 0x28 may be stored in two ways, describing file block's location on disk. One of them is well known *Block map* explained in the figure below. This type of structure is mainly applied in non-regular files (0x00) and currently is being less and less used.

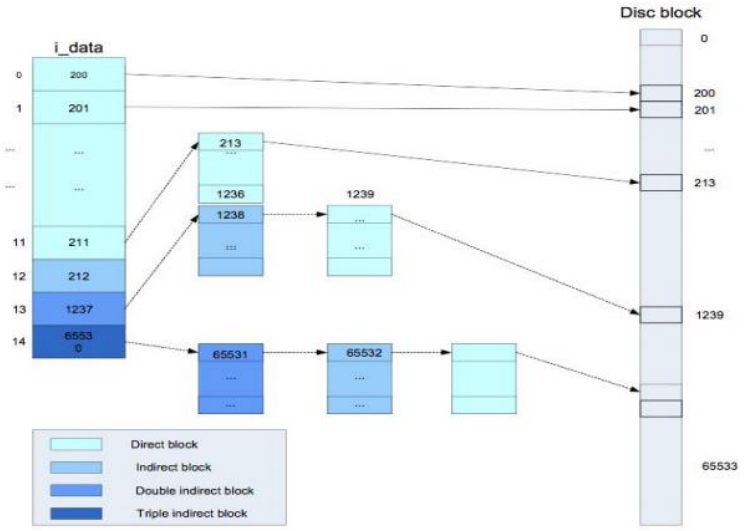


Figure 4: File blocks stored in block map structure (13)

OFFSET	SIZE	DESCRIPTION
00-11	-	Direct map to file blocks
12-12	-	Indirect blocks (block size / 4)
13-13	-	Double indirect blocks (block size / 4)^2
14-14	-	Tripple indirect blocks (block size / 4) ^3

Table 3: Data block types with their offsets in block map structure (7)

Second known structure used in iNode at offset 0x28 is extent tree. This tree has simple layout shown in figure 5. Data blocks are always stored at the leaves of tree and represent data shown in table 4 to table 6.

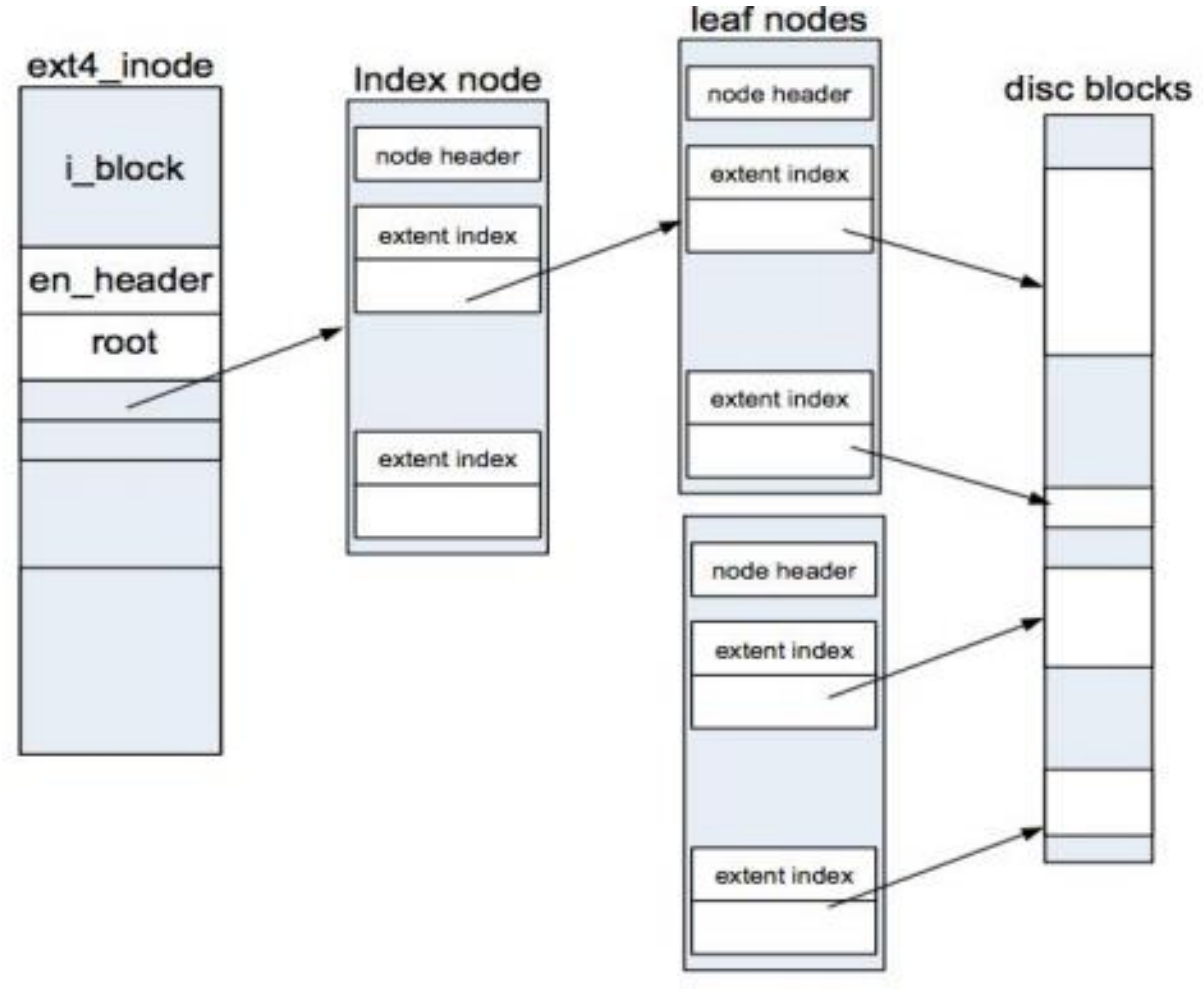


Figure 5: Extent tree structure used for file block location (13)

OFFSET	SIZE	DESCRIPTION
0X0	16b	Magic number, 0xF30A.
0X2	16b	Number of valid entried following the header.
0X6	16b	Depth of this extent node.

Table 4: Extent tree block header (7)

OFFSET	SIZE	DESCRIPTION
0X4	32b	Lower 32bits of the block number of the extent node that is the next level lower in the tree.
0X8	16b	Upper 16bits of field above.

Table 5: Internal node of extent tree, pointing only to next internal nodes or directly to leaf nodes (7)

OFFSET	SIZE	DESCRIPTION
0X4	16b	Number of blocks covered by this extent.
0X6	16b	Upper 16bits representing block number of file.
0X8	32b	Lower 32bits representing block number of file.

Table 6: Leaf node of extent tree pointing directly to data blocks (7)

3.4. Journal

All important information about previous actions of filesystem are stored in journal file located usually on iNode 8 (this can be researched in superblock as mentioned above). Similarly, to whole filesystem, journal has superblock along with blocks of two other different types of blocks. In my project I focused only on descriptor blocks and possibilities for recovery of deleted iNodes from journal file. In figure below is shown layout of journal file and its blocks.

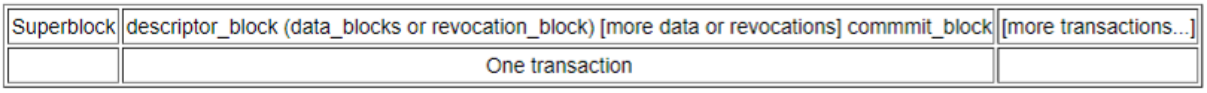


Figure 6: Layout of journal file in EXT4 (7)

Each journal block starts with common header, with information provided in table below. Only one type of block is explained in this project (used in tool from section 5). Descriptor block contains an array of journal block tags that describe the final locations of the data blocks that follow in the journal. These blocks were used when trying to recover deleted iNodes and not wiped data from device.

OFFSET	SIZE	DESCRIPTION
0X0	32b	Magic number, 0xC03B3998
0X4	32b	Description of what this block contains (1 - Descriptor block)

Table 7: Header of journal block

Journal's superblock was not analyzed nor used in the tool, so it is ignored in further explanations.

As mentioned above most important values are stored in descriptor blocks. Header of descriptor block was not used, only data stored after the header. All data were passed through iNode checker to verify if this data may be used to recover any files from previously deleted iNodes.

4. Filesystem comparison

This section is focused on comparison of other filesystems with EXT4.

4.1. Overhead

Filesystem comparison from overhead point of view was done on Virtual Machine with operating system Linux Mint 20.04. This experiment was done in 5 iterations with 7 different filesystems (including EXT4) listed in graphs below. Simple shell script was programmed only for this experiment.

In the script listed on the page 9 can be seen overhead test. This test took image file, made filesystem requested by the input on it and mounted it to testing directory. Afterwards, to this device were written files with size and count listed in the table below. Next was taken output of *df* command, which shows current free and used space of filesystem.

Output:

As could be seen in the graphs, two of the filesystems (HFS, FAT32) were broken after inserting multiple small files or which was not able to allocate more then 3248kB in the last iteration of experiment.

EXT4 is located at the bottom of figure below, with only 232 296kB allocated, so it has the biggest overhead, as HFS and FAT32 would be disqualified.

#	FILESYSTEM SIZE	FILE COUNT	FILE SIZE
1.	256MB	1	256MB
2.	256MB	256	1MB
3.	256MB	1024	256kB
4.	256MB	4096	64kB
5.	256MB	16384	16kB

Table 8: Experiment dataset for comparing overhead on NTFS, FAT32, EXT4, EXT3, EXT2, HFS+, HFS.

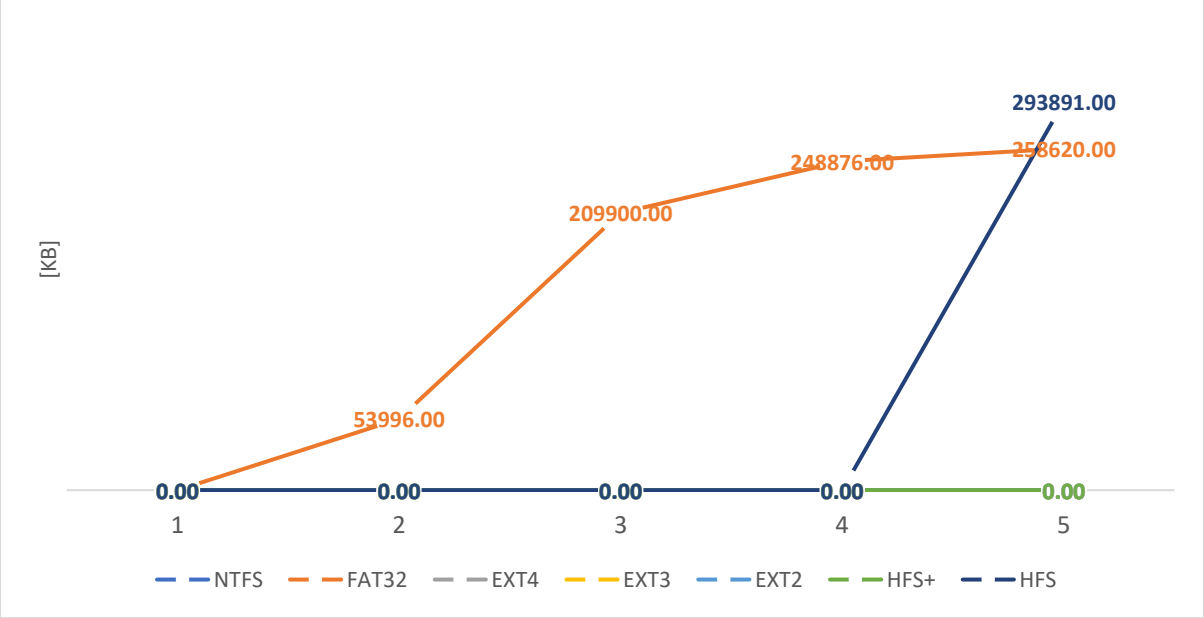


Figure 7: In this graph is shown available space after writing files to the filesystem.

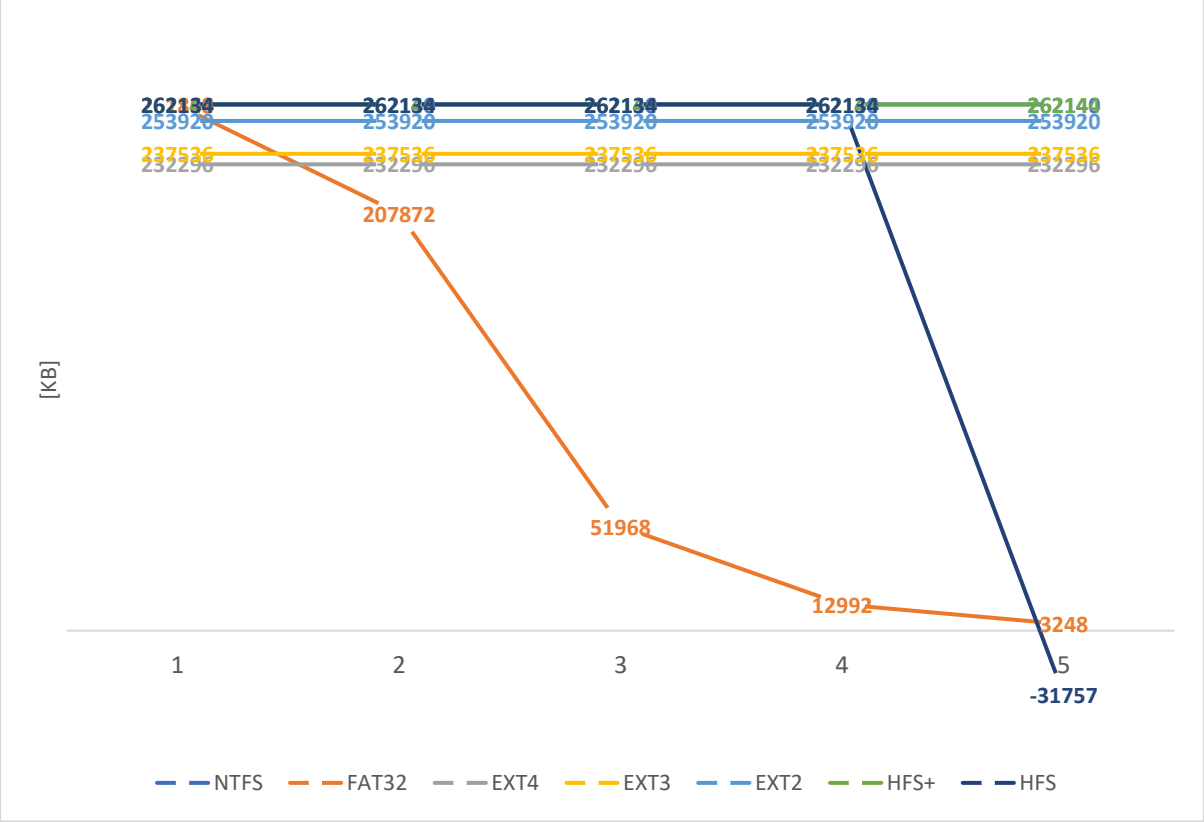


Figure 8: In this graph is shown used space after writing files to filesystem

```
#!/bin/bash

mkfs.$1 file.img
mount=/mnt/media
mount file.img $mount
echo "*****MOUNTED*****"
inf=/dev/zero
ouf=$mount/speed
bytes=$((256*1024*1024))
echo $1 >> df.txt
echo $1 >> dd.txt
for i in 1 256 1024 4096 16384
do
    size=$(($bytes/$i))
    for file in $(seq 1 $i)
    do
        dd if=$inf of=$ouf$file.empty bs=$size count=1 >> dd.txt 2>&1
    done
    df | tail -n 1 | awk '{print $3","$4}' >> df.txt
    rm $ouf*
done
umount /mnt/media
echo "****UNMOUNTED****"
exit 0;
```

Figure 9: Code used for comparing overhead of filesystems

4.2. Recovery rate

Recovery rate and its successes were done with two main tools: ext4magic (11) and ntfsundelete (12). This experiment was done similarly to previous one, but this time with a little bit different shell script with 5 iterations differencing only in filesystem size and total files that were recovered and deleted from the device. Table 8 below represents each iteration of the experiment. This experiment could not be done on the same operating system due to issues with ext4magic and one other tool (extundelete), because they were not able to run on this machine properly. Instead, experiment was executed on Virtual Machine with operating system Kali Linux version 2020.4.

Output:

As could be seen in the figure 10 (above) NTFS was absolute winner with the recovery rate solid 100%. On the other side EXT4's recovery rate was getting lower as the filesystem size was getting bigger. The main point of this section is that file recovery is much easier and reliable on NTFS then on EXT4.

4.3. Performance

In the experiment (8) were used multiple SSD and HDD disks in RAID 0 with focus on optimalization of each blocks and partitions used during testing. Operating system chosen was SUSE 12.1 with the latest linux kernel (at the time) 3.1.10. Filesystems tested in this experiment were:

- EXT2 – oldest filesystem, without journaling and little data protection.
- EXT3 – long term standard in many distributions.
- EXT4 – the latest EXT.
- BTRFS – like ZFS, but ZFS was not natively supported on SUSE12.1 (at the time).
- XFS – journaling filesystem developed by SGI.
- ReiserFS – filesystem initiated by Hans Reiser.

All figures below are only recreated figures from linux magazine (8).

Output:

As could be seen in 4 graphs following EXT4 is one of the fastest filesystems in terms of reading from disk (either sequential or random). On the other hand, with writing, it has multiple performance issues even compared to its predecessors. EXT4 should be used in storages with more reads then writes or this could cause significant performance issues.

	FILESYSTEM SIZE (MB)	SIZE OF FILES (B)	TOTAL FILES (COUNT)
1.	64	4	64
2.	128	4	128
3.	256	4	256
4.	512	4	512
5.	1024	4	1024

Table 9: Represeting each itteration of the experiment explained in section 4.2

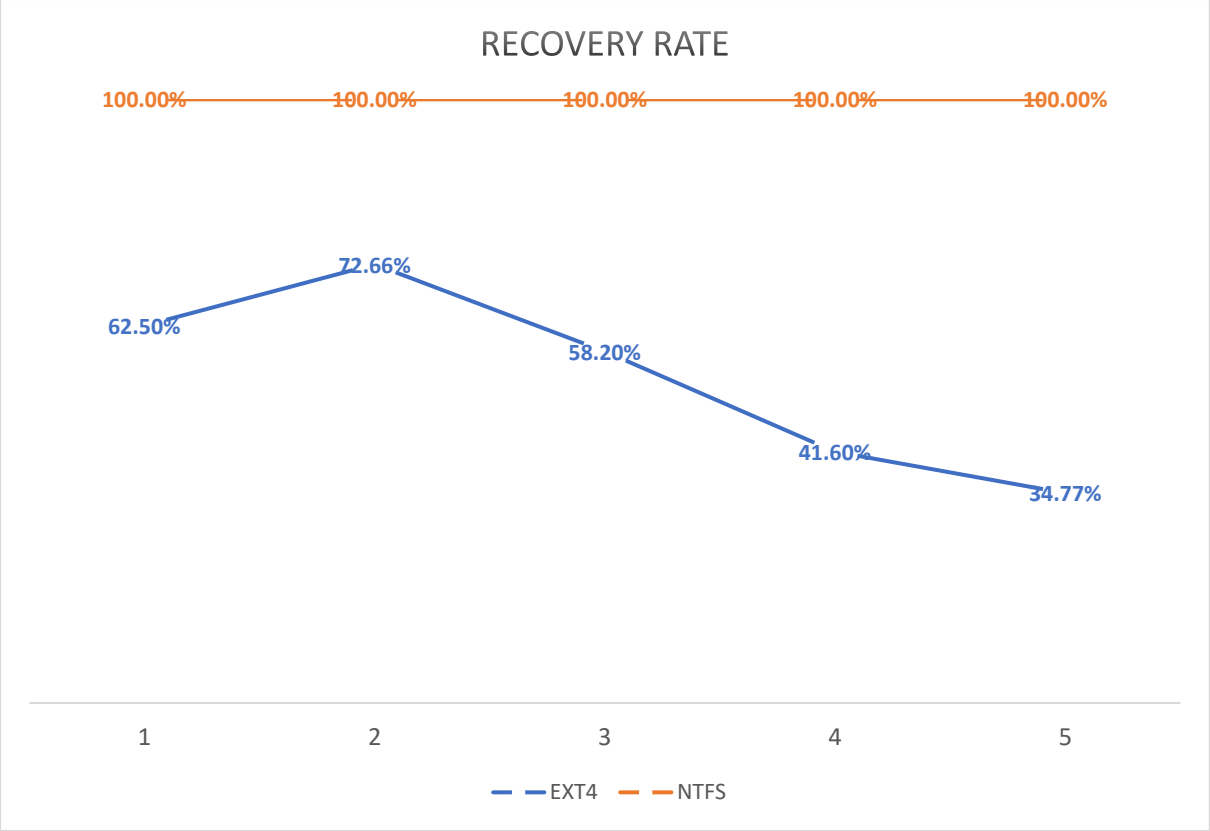


Figure 10: Graph represents absolute value of recovered files divided by the total amount of files deleted from media.

```
#!/bin/bash
file=$1.img; args=$2; size=$3; recover=$4;
mount=/mnt/test
dd if=/dev/zero of=$file bs=1M count=$size
mkfs.$1 $args $file
for i in $size; do
    mount -o sync $file $mount
    sudo rm -rf $mount/*
    for j in $(seq 1 $i); do
        touch $mount/cover$j.txt;
        echo "TEST" >> $mount/cover$j.txt
    done
    files=$(ls -la $mount | wc -l)
    rm -rf ./restored
    mkdir ./restored
    rm $mount/*.txt
    umount $mount
    $recover
    echo "$1: "$(($(ls la ./restored | wc l) - 3))"/"$(($files - 3))
done
exit 0;
```

Figure 11: Code used for comparing recovery rate of EXT4 and NTFS

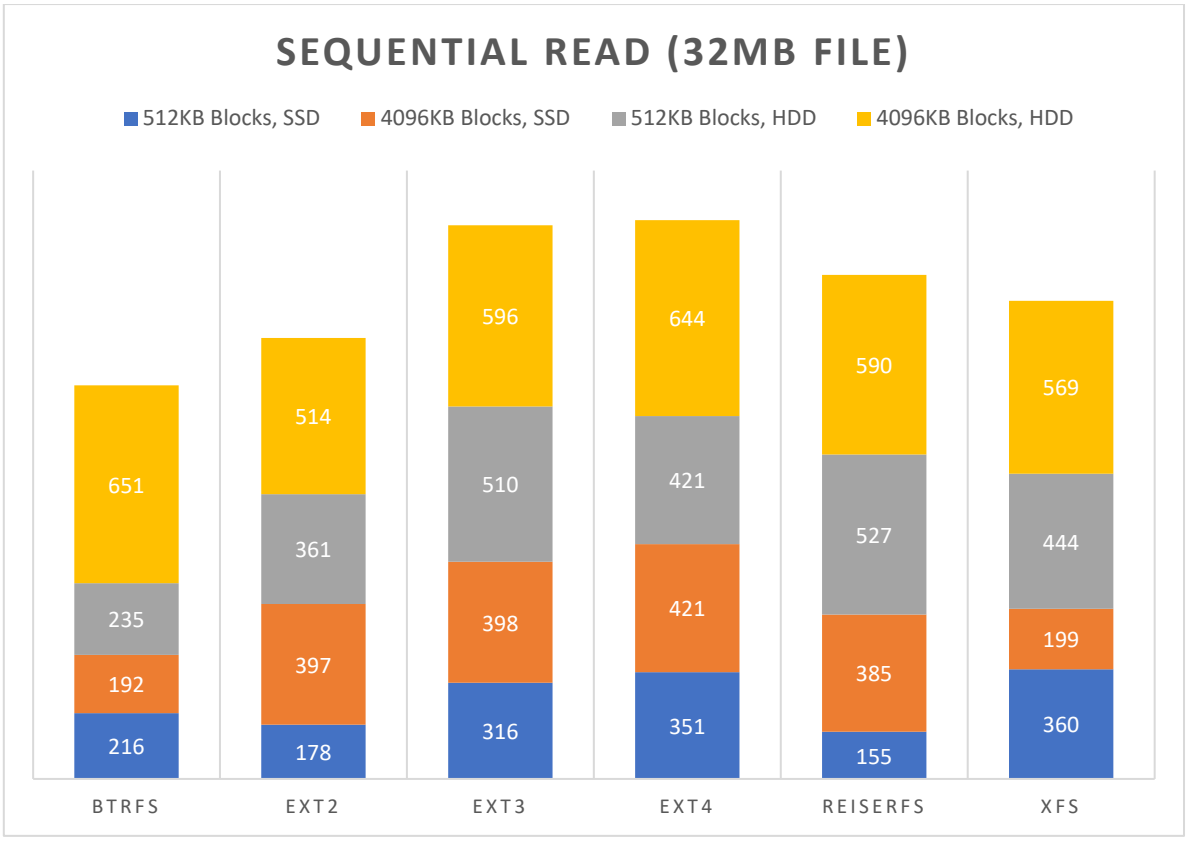


Figure 12: Sequential reading was clearly won by EXT4, with a little improvement against EXT3. Other filesystems are not worth any mention.

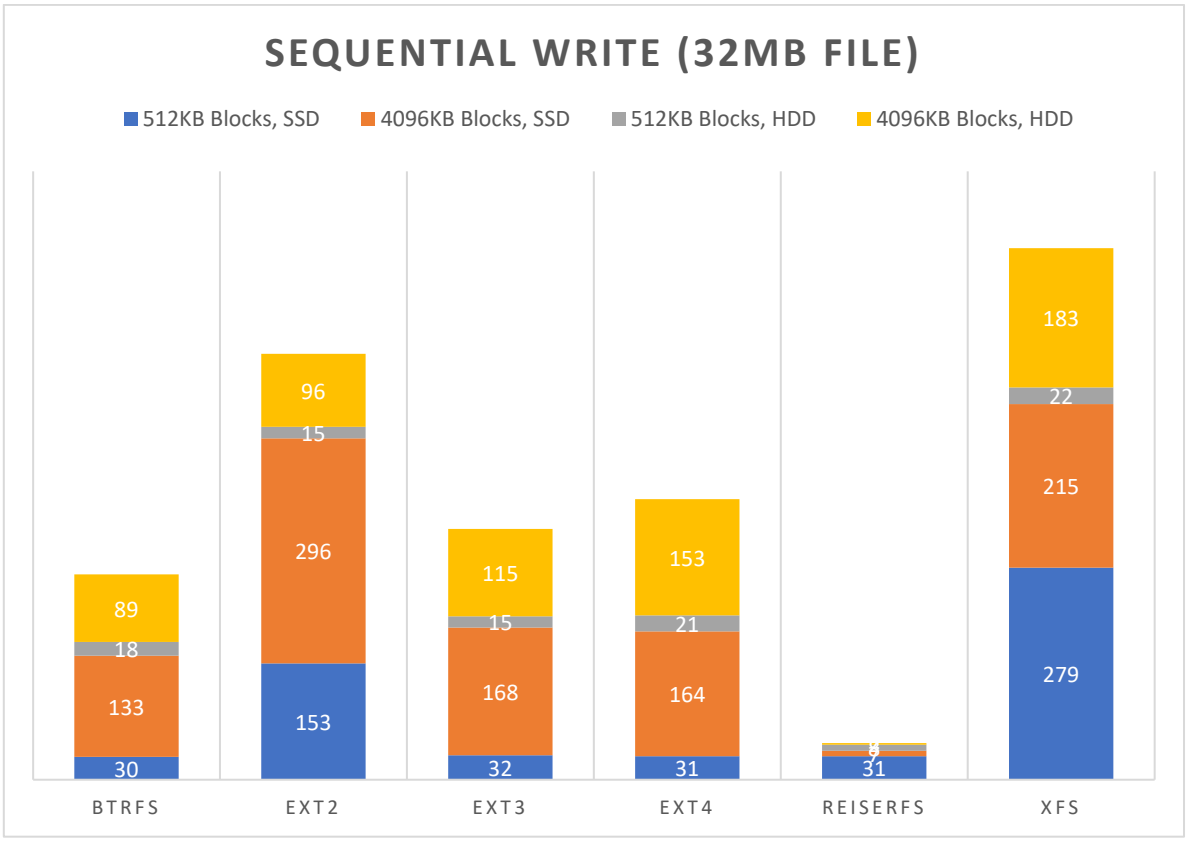


Figure 13: In the sequential write test was EXT4 clearly behind the XFS and EXT2, but compared to ReiserFS or EXT3 is doing much faster job event then BTRFS.

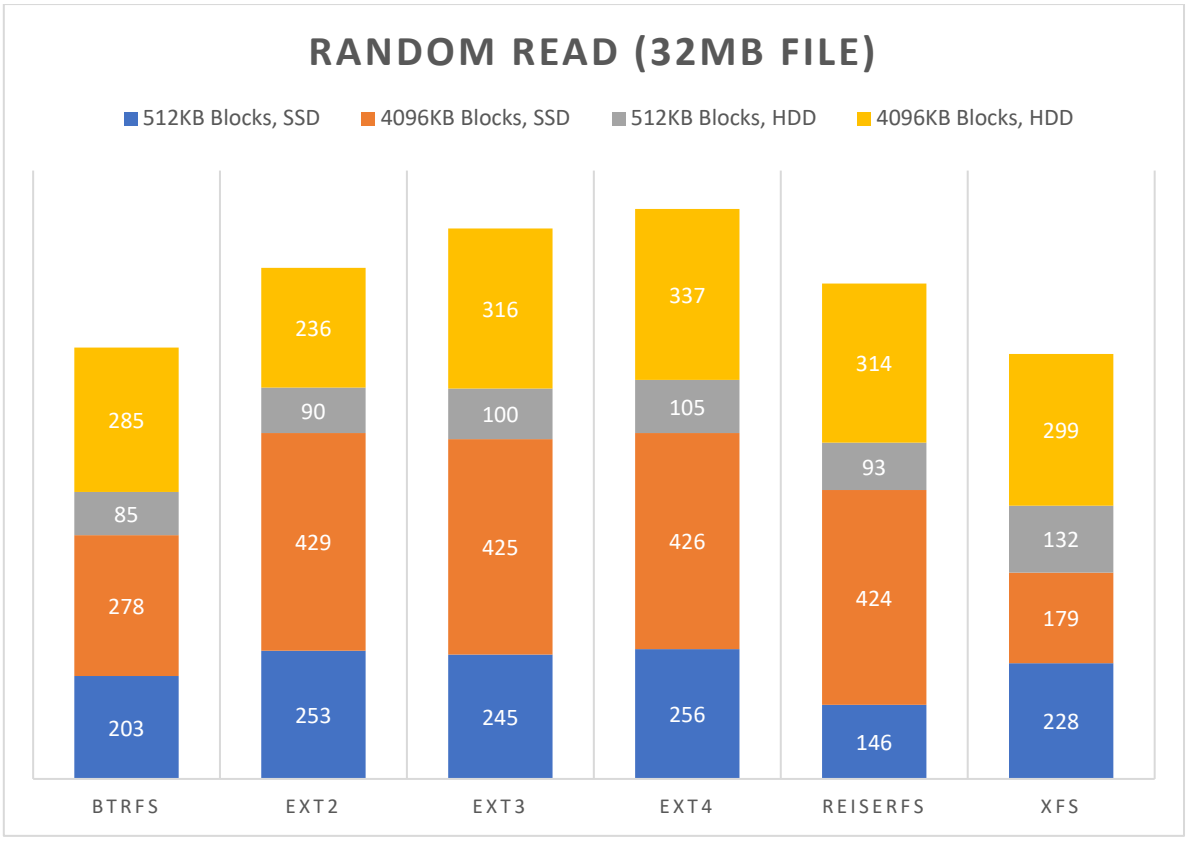


Figure 14: Random reading is dominated again by EXT4 with EXT3 not that far away. Clearly other filesystems are not that far behind.

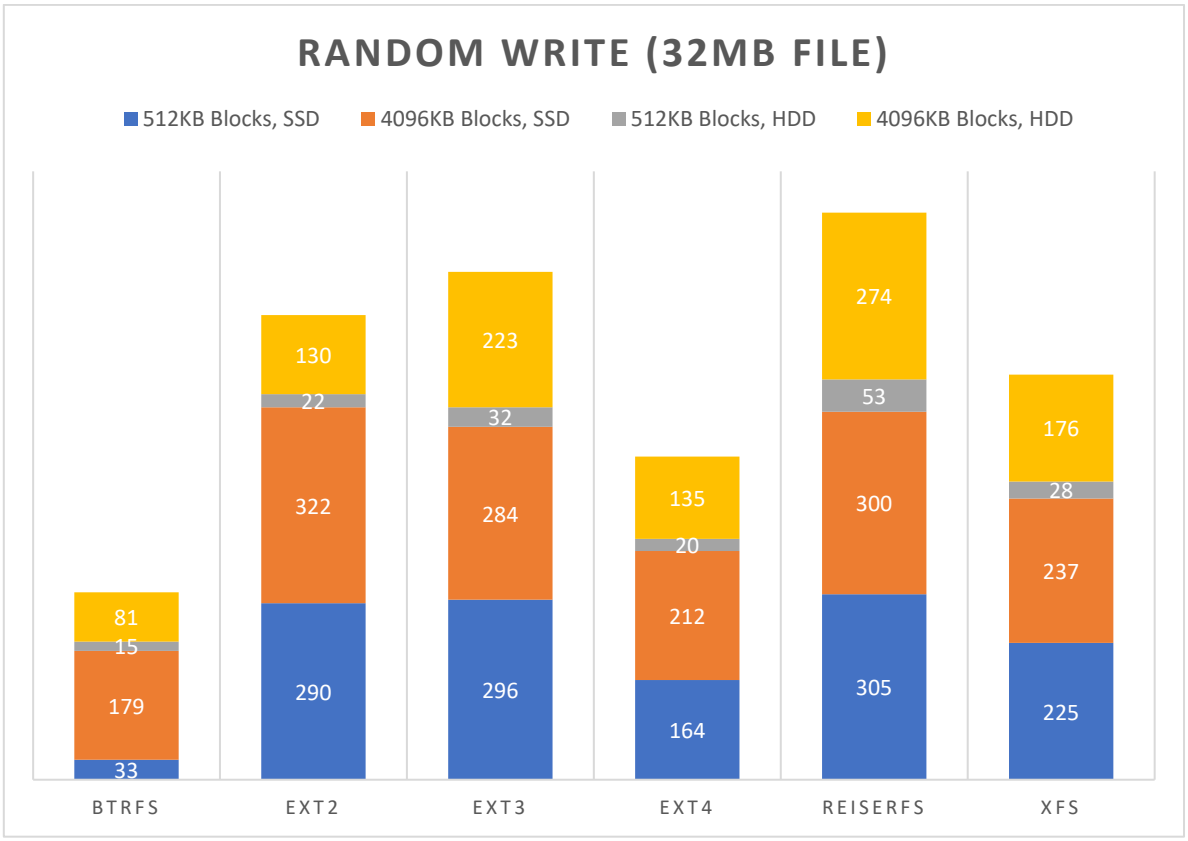


Figure 15: In last test (random write) is EXT4 second from the bottom, outperformed even by EXT3 or EXT2, not writing about ReiserFS or XFS.

5. Data recovery tool

This section writes about data recovery tool, that I programmed in python during this project. Currently has this tool significant limitations in maximum file size to recover, maximum journal size, file name or directory structure recovery (all of this is explained below).

Code is structured in main 3 directories, each representing part of code inherited inside. *Main.py* file is wrapper for starting whole program. Secondly, most important part is in *scripts* folder containing each section of EXT4 layout mention in section 3. *Userinterface* folder contains only few files for basic command line interface and progress for user. Last is *testing* folder with one shell script that writes random files to mounted partition after which my tool may be started and properly tested. In text below is discussed only *scripts* folder which is the most important one.

Recover.py

Starting with class recovery, which wraps all device information and groups of EXT4 filesystem and parses them into my custom classes. First needed are device information which are provided with class *device*. Secondly are provided information for progress (only for user to know which group is currently being processed). Afterwards, when device is validated, are groups of EXT4 parsed. Number of groups to be parsed is constant and is provided in global variable *GROUPPOSITIONS*, as I could not find a way of programing this sequence into my code. When parsing by class group is done, output is printed out with five levels (values zero to four) of verbosity (this functionality is currently turned off/ but may be easily added in the future, default value for now is 0).

Device.py

Device class is mostly done with the help of *pyparted* (6) module (in code is called *parted*). All necessary information needed to parse partitions of device or file provided are processed only by this module due to its easy use and my time needed to spend over this project was already too high to even program my custom partition parser. First, helping variable *valid* is set to *True* which is supposed to check if user inut from *userinterface* is correctly parsed into this class. Next is called method *getDevice()*, which through *userinterface* gets requested device or file to be processed by the code. Afterwards, *getPartition()* does almost the same, but now for choosing partition on selected device. Lastly, *getDirectory()* is called to specify output folder of recovered file (if any). Codes represented in error values are explained in *message* variable and code number is explained in **recovery.py**.

Group.py

After knowing device, partition, and destination folder, comes parsing each block in loop from **recover.py**. Each block consists of *superblock*, *groupdescriptor*, *inodetable*, and *journal* iNode information (each of these classes are described below) as could be seen in section 3. Simple method *print* is created to provide currently only basic output to user.

Superblock.py

Into the class *superblock* are passed variables *chunk* (chunk of bytes, length 1024B, extracted from **group.py**) and *block_group* (used only in *print* method to provide output). After initiating class, parsing is done with most important values shown in section 3.1. Firstly, is checked *magic number*, if this is wrong basic error output is provided and validation variable (*valid*) is set to *False* to indicate error in this group for further parsing. Next, only *64-bit* filesystems are supported which is checked at *chunk* positions 0xFE-0xFF. If both values are as expected, then important and needed variables are extracted from provided chunk and saved to correspondingly named variables.

Groupdecriptor.py

Simple class *groupdescriptor* is needed only to extract two values as mentioned in section 3.2. *iNode_table* variable is needed to know on which device block is iNode table located (without this variable it is almost impossible to find iNode table, due to fact that iNodes do not have any magic numbers). Secondly is variable *free_inode_count* extracted. This value does not currently have any implementation but could be helpful in future.

Inodetable.py

Again, simple class *inodetable* is only a wrapper for iNodes that loops through all possible iNodes located in the table and calls class *inode* to parse themselves.

Inode.py

Most valuable and used class is *inode* class. This class parses important values from section 3.3 and extracts data blocks from extent tree only (for now, but support for parsing block map structure is simple to add). This class is also responsible for recovering data with method *getData* which stores data in destination folder provided by user at the start and saves them into file with custom file name (for now there is not implemented any method to extract filename, so custom method *name* is called when saving data into file). Method *getData* uses variable *blocks* which stores parsed blocks from extent tree (again currently only extent tree with depth 0 is being parsed, but for future purposes method *getBlocksExtentTree* may be extended to support also bigger depths). Method *valid* is used only when parsing descriptor blocks from *journal* and trying to recover data from it. *getInodeType* is simple method to check if current iNode uses extent tree or block map structure. *getBlocksExtentCount* extracts number of blocks used by file from leaf node of extent tree. *parseExtentIdx* is currently unused method for future use if implementation of bigger depths of extent tree wanted to be added into this tool.

First experimental process of recovery was done in this step, by checking variables *deleted_time* and *hard_links* (representing number of hard links pointing to this file). This method was very unsuccessful and was deleted from the code.

Journal.py

For now, only recovery is done via *journal* class, which stores few last blocks of data that were written to disk. In my code I worked mostly with descriptor blocks as they contain information about data, which may be stored on the disk. Firstly, is journal file parsed by *block_size* variable (representing block size of filesystem – not partition). Each block is checked for magic number provided in each journal block and then appended to the final file representing only valid blocks of journal. Next is each block checked to be either descriptor, superblock or commit block (in my code only descriptor blocks are further processed). Parsing descriptor block consists of testing whole block divided into iNode sizes and passing them through iNode validation (due to lack of magic number in each number is this process a bit tricky but can recover at least some data, sometimes even more then may have been deleted). If this validation passes, then the data are recovered and stored by process described in *inode* class (there is no implementation to prevent duplicated data recovery or recovery of non-deleted files, this process is easy to add).

Second experimental process of data recovery provided by functionality described above was successful (only if there were at least some relevant information stored in journal file). Testing was done on multiple .img files (64MB, 1GB and 32MB file with 2 separate partitions, after creating file and deleting were recovered multiple files, with one of them that was the target).

6. Forensics on EXT4

Second main output mentioned in this project are forensic techniques and experiment done on EXT4 after wiping data only by using *DD* command. My main motivation why to include forensics in this project was to better understand how EXT4 works and find alternative ways of recovering files with my tools. This motivation was success because I would not know about journal as much as possible.

6.1. Data residues

Experiment from (1) suggested that after creating fresh image of EXT4 with the size of 500 megabytes it is possible to get data even after zeroing the whole filesystem. On the filesystem were created 4 kilobyte files containing only word "TEST" multiple times in a row. These files covered 100% of available space of the filesystem. Surprisingly after using *dd* command about 22 megabytes of data were left untouched by zeroing process. This can be used in multiple forensic implications in case of attackers or hackers are using *dd* as their 'wiping tool' (but they should not use it as explained above).

6.2. Deleted files

One of my two approaches for recovering files was to capture iNodes that were still valid but had deletion time greater than zero or number of hard links less than one. This tactic uses information provided in source (2) and explains knowledge needed to understand this type of forensic technique.

In (2) is stated that file iNodes are not deleted upon file deletion, but this might and might not happen as experiments show. The biggest factor is file size and depth of extent tree structure and if the iNode is reused soon or is only left marked as free.

During development and testing of my data recovery tool, I have not seen this type of EXT4 behavior and was not able to recover any data using this technique. My tests consist of creating .img file with different size and writing random sequence of words that I could recognize in *xxd* tool. After trying different sized filesystems, files with this recovery technique I was not able to get any data back from deleted files, even though raw data were still present in .img file.

6.3. Metadata

All metadata of files are stored in iNode as stated in sections before multiple times. Although, iNodes do not have any magic number, an extent trees have. This knowledge may be used in finding all data blocks associated to file. Following extent tree header with various depth exactly describes blocks of file where to find them and this technique may be also used to recover data. However, I have not used it due to performance issues and searching whole filesystem that could have more than terabytes of data could lead to significant issues.

6.4. Data

As suggested by (2) it is possible to recover data from extent trees in one more way then described before. Reused iNodes (that were not zeroed up after deleting file) may contain previous pointers to blocks in extent tree leaves or internal nodes. This technique suggests checking all iNodes and searching blocks which they are pointing to, with some hope to recovery at least some parts of previously deleted file. I have not used this method due to uncertainty how much file data I would be able to recover using my tool.

6.5. Journal

As stated, multiple times in this project journal can be source of previous data or metadata stored on disk. My focus on this recovery technique was due to fact that journal file is always multiple times smaller than whole filesystem (technique from section 6.3) and there is always a chance to recover either whole deleted data or not recover them at all.

7. Counter forensics on EXT4

The growth of forensics comes together with anti-forensic or hostile forensic. These methods grow in hacker communities that do not want investigators to gather evidence against them and they are trying to find various techniques for hiding data, exploiting forensic tools (for example by finding buffer overflow errors causing tools not to show data properly – luckily, this type of technique was not successful yet) or destroying data before law enforcement agents can gather them. These sections write about most common techniques used against forensic steps needed to provide evidence to courts or police after crime is done. (5)

7.1. Subversion and Denial of Service

One of many hostile forensic strategies is to exploit a vulnerability on a forensic tool preventing evidence gathering or to launch a denial-of-service attack which again prevents tools from functioning properly. In source (5) is for example mentioned EnCase v4.15 (current version is v6.16.1 which has this bug already fixed) in which the maximum directory depth is only 253 folders. In case of greater depth, the tool crashes with a fatal error preventing deeper analysis of data in such folders. Most honorable mentions in this type of counter forensics are manipulation of timestamps, insertion of compression bombs, use of sparse files or modifying magic numbers. (5)

7.1.1. Timestamps

Most common techniques involving gathering evidence use timestamp information provided by filesystem. This of course is commonly used by hackers or anti-forensic specialists to manipulate modification, access, and creation (MAC) times of files. This method is so useful because analyst cannot estimate timeline and order in which each possible attack or data leak happened.

Even some native linux tools provide functionality to overwrite timestamps. Such an example is *touch* command which allows file timestamps to be set to the current time and to reset MAC times to any values. Another tool for modifying timestamps is fileTweak which can change timestamps on linux, FAT and NTFS filesystems.

As stated in (5) filesystem or file timestamps cannot be trusted, and this should lead to not filter files according to their timestamps when gathering evidence.

Experiment from (4) suggested hiding data in EXT4 timestamps and was quite successful. They suggested a prove of concept for such data hiding software with results that were impressive. In table below are shown examples of required allocated iNodes depending on size of file needed to be hidden. Their experiment involved for example hiding 1 megabyte file within 183190 iNodes (which needed filesystem with about 3 gigabyte size).

SIZE OF THE INPUT FILE	REQUIRED ALLOCATED INODES	SIZE OF FILESYSTEM
1 KB	186	< 1 GB
100 KB	18325	< 1 GB
500 KB	91595	> 1 GB
1 MB	183190	> 1 GB
5 MB	915923	> 10 GB
10 MB	1831846	> 10 GB
50 MB	9159219	> 100 GB

Table 10: Representing data hiding within timestamps with theoretical filesystem size allowing this to be done (4)

7.1.2. Compression Bombs

The method of creating compression bombs is to damage or crash operating system. Typical compression bomb is a normal for example .zip file that expands into massive files. Such an example is the 42.zip (5) bomb which in compressed form has about 42 kilobytes. Unfortunately, after uncompressing this zip has 4.5 petabytes of data (in modern cloud storages and solutions is such amount of data not that massive, but I think no ordinary forensic analyst would have for example SIFT operating system with more than 4.5 petabytes of free storage). Example of such compression bomb is shown in figure below.

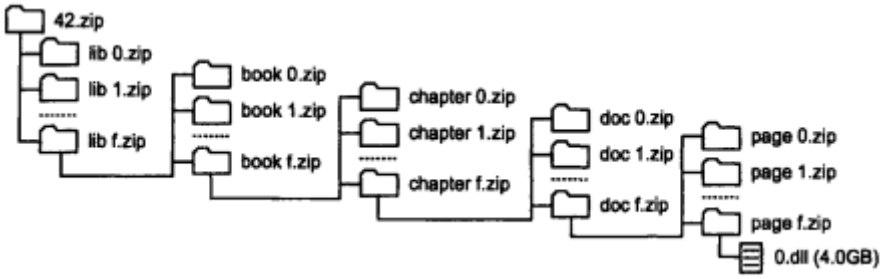


Figure 16: Uncompressed version of the 42.zip compression bomb (5)

According to source (5) there are multiple forensic tools that react inappropriately and crash or freeze. Such an example is The Forensic Toolkit (FTK) (v1.42) which freezes and becomes practically unusable. As mentioned above EnCase v4.15 had issues with maximum depth of directories and thus crash on this bug.

Such zip file with high compression ratios is created by compressing files composed entirely by zeroes. It is also worth mention that there are rarely any data or evidence stored within file compression bombs.

7.1.3. Sparse Files

Second type of special files that prevent common forensic methods are sparse files. These types of files involve similarly to compression bombs “a lot of zeroes”. Such file is created with a simple command (for example DD) and data may be stored anywhere within that file. One of the most interesting parts is that older versions of FTK, EnCase or ILook are in some cases unable to properly handle these files and thus make investigation harder. On the other hand, important for investigator is to look for hidden data in the whole file not just in the begging or at the end, because data may be at any location, but are lead and followed by zeroes.

7.1.4. Magic Numbers

In each filesystem headers may be found some kind of 2,8 or multiple byte values called magic number (as discussed above for EXT filesystems it is 0x53EF, FAT32 has 55AA, NTFS has ‘NTFS ‘ – ntfs with 4 spaces following filesystems name). Some forensic tools need this number to be correctly written in the filesystem or they will not be able to determine filesystem type stored on that partition (for example it is possible to use DD command to completely wipe position of magic number). This issue may be fixed by manually verifying and guessing filesystem and writing proper magic value on its original place. Another commonly used technique associated with magic number is modifying or wiping partition table of devices and thus tools are taking whole device as one partition causing significant issues (this method may be fixed by using *gpart* – tool for reconstructing partition tables).

7.2. Data Concealment within Filesystems

Beside active attack or attempt to modify data to prevent detection and gathering evidence, there is much more used technique – data hidden within the filesystem. This method allows to hide data and thus prevent forensic tools from collecting them. Such techniques discussed below are involving data streams, using slack space, or using reserved locations for filesystem. (5)

7.2.1. Alternate Data Streams

This method is mostly used on NTFS filesystem that allows using it natively. Files in NTFS are interpreted by the operating system as streams of data associated with a filename. In most cases file has only one stream of data, but that does not block user to add also other alternatives of this data streams. For better understanding I used figure from (5), which represents how these data may be in NTFS stored.

```
C:\>echo hello world > file.txt
C:\>echo this data is hidden > file.txt:secret
C:\>more < file.txt
hello world
C:\>more < file.txt:secret
this data is hidden
```

Figure 17: Alternate data stream creation (5)

7.2.2. Slack Space

Filesystems usually divide partitions into blocks of equal size (simple example would be having a file with 14 characters = 14 bytes, this file is taking whole filesystem block – in EXT4 most commonly 4096 bytes, or another example would be for 5000 characters file using two of these blocks, thus 8192 bytes). Of course, section from 14th to 4096th may be used for hiding data as program reading from that file knows exactly how big is that file and when should it stop reading from it.

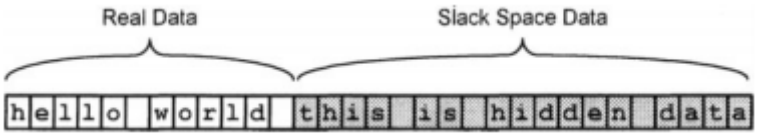


Figure 18: Slack space used to hide data (5)

7.2.3. Reserved Locations

Last examples in this category of anti-forensic techniques are reserved locations, which use hundreds of unused filesystem locations that are not actively used by the operating system or the filesystem itself, as can be seen below and in section 3 where I described my usage of group descriptors (my program uses only two values from this 26-byte long structure – even though I was programming the tool for EXT4 this structure has not changed very much yet).

```
struct ext2_group_desc
{
        __u32   bg_block_bitmap;        /* Blocks bitmap block */
        __u32   bg_inode_bitmap;        /* Inodes bitmap block */
        __u32   bg_inode_table;         /* Inodes table block  */
        __u16   bg_free_blocks_count;   /* Free blocks count   */
        __u16   bg_free_inodes_count;   /* Free inodes count   */
        __u16   bg_used_dirs_count;     /* Directories count   */
        __u16   bg_pad;
        __u32   bg_reserved[3];
};
```

Figure 19 EXT2 structure representing group descriptors (5)

7.3. Data Concealment outside Filesystems

Last popular method to prevent forensic analyst from gathering data is to hide them in not typical medias, that are not usually imaged while gathering evidence. Such places written below are for example concealing data within random access memory, obscure disk locations or overwriting BIOS chips. (5)

7.3.1. Random Access Memory

Usually used method when capturing data is to turn them off and analyze only non-volatile data. This technique denies this method due to possible data loss of precious files located only in system memory (RAM). Although it is dangerous to analyze data on running machine with multiple disadvantages, at least this method should be used when possible. To reduce the amount of gathered evidence, some individuals may even try to destroy as much data as possible before law enforcement can collect data by using remote access tools. Commonly used *The Tinfoil Hat Linux* operating system is designed perfectly for this job mainly because it leaves almost no evidence after user uses it and encrypts all data written to persistent memory (some other examples of leaving as little evidence as possible is to use live distributions of operating systems, Knoppix or bootable CDs/USBs). (5)

7.3.2. Hard Drives

Hard drives themselves have more memory than they claim to the computer (example would be Host/ATA protected area at the end of HDD which cannot be read nor written to using standard system calls). Special tool to access this space is X-Ways Replica. (5)

Another technology with tremendous number of advantages has in forensic application some disadvantages. SMART is another feature that modern HDDs use, but so do hackers or attackers. This tool can display various information about media health, along with time since the disk was mounted on the system, helping him with the discovery that his drives might have been imaged and used to gather evidence so he would be able to erase or tamper with other computers or servers that were not yet imaged. (5)

7.3.3. BIOS

Each computer has a **B**asic **I**nput/**O**utput **S**ystem (BIOS for short), which is represented as a chip, code and data with memory varying from 128K to 512K required to boot the operating system. Out of the available memory about 28K to 100K may be unused space or space that will allow normal operation of the system even after modification. A lot of malicious software (viruses) uses this space or data may be hidden here. Uniflash is one of the utilities allowing data to be read from and written to BIOS chip. Thankfully, there are devices like BIOS Savior which provide necessary hardware and software to fix onboard BIOS chip if anything goes wrong. Simmilarly, AwardMod is one of the forensic utilities used to read whole chip memory and this data may be further investigated by forensic analyst. (5)

8. Project outcome

My project was mainly about EXT4 described from different points of interests. The biggest project outcome was comparing and understanding EXT4, programming data recovery tool and describing various forensic and anti-forensic techniques used on this filesystem.

Filesystem comparison

This section outcome will be represented by simple table summing up all results and benefits of filesystems used during each comparison.

	FAT32	NTFS	HFS	HFS+	EXT2	EXT3	EXT4	BTRFS	REISERFS	XFS
OVER-HEAD	✗	✓	✗	✓	✓	✓	✓	-	-	-
	✗	✓	✗	✓	✓	✓	✓	-	-	-
	✗	✓	✗	✓	✓	✓	✓	-	-	-
	✗	✓	✗	✓	✓	✓	✓	-	-	-
	✗	✓	✗	✓	✓	✓	✓	-	-	-
RECOVERY RATE	-	✓	-	-	-	-	✗	-	-	-
	-	✓	-	-	-	-	✗	-	-	-
	-	✓	-	-	-	-	✗	-	-	-
	-	✓	-	-	-	-	✗	-	-	-
	-	✓	-	-	-	-	✗	-	-	-
SPEED	-	-	-	-	✗	✗	✓	✗	✗	✗
	-	-	-	-	✗	✗	✗	✗	✗	✓
	-	-	-	-	✗	✗	✓	✗	✗	✗
	-	-	-	-	✗	✗	✗	✗	✓	✗
Σ	0	10	0	5	5	5	7	0	1	1

As could be seen in table above although not all filesystems were used in all tests EXT4 came out of this competition as decent filesystem although it does not always provide the best speeds or recovery rates. On the other hand, experiments show that at least some data may be recovered from EXT4 and overhead was too showing some issues, but filesystem was always able to use whole space and store most of the data required by user.

Data recovery tool

As mentioned above the biggest achieved throughout this project was programming data recovery tool that is at least in some cases able to recover data. Tool was programmed for future implementations of more complicated structures with the biggest issue is that currently is implemented only usage of extent trees to maximum depth of 1. This tool was able to recover data recently deleted and may be used in simple recovery of recently deleted files.

Forensics and counter forensics

In both sections I covered multiple techniques used for forensics and counter forensics. Main forensic techniques used during investigations are data residues (as shown by an experiment), deleted files, metadata and of course journal. On the other hand, much more in numbers are represented anti-forensic techniques like subversion or denial of service (attacking forensic toolkits), data concealment within filesystem (via different techniques for example using slack space or reserved locations) and data concealment outside filesystem (using only RAM or BIOS chips as storage).

References

(1) Kevin D. Fairbanks, Christopher P. Lee, and Henry L. Owen. 2010. Forensic implications of Ext4. In Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '10). Association for Computing Machinery, New York, NY, USA, Article 22, 1–4. DOI: https://doi.org/10.1145/1852666.1852691

(2) Kevin D. Fairbanks, An analysis of Ext4 for digital forensics, Digital Investigation, Volume 9, Supplement, 2012, Pages S118-S130, ISSN 1742-2876, https://doi.org/10.1016/j.diin.2012.05.010 (https://www.sciencedirect.com/science/article/pii/S1742287612000357)

(3) Page name: Ext4, Author: Wikipedia contributors, Publisher: Wikipedia, The Free Encyclopedia. Date of last revision: 18 December 2020 10:43 UTC, Date retrieved: 3 April 2021 13:51 UTC, Permanent link: https://en.wikipedia.org/w/index.php?title=Ext4&oldid=994946165 Primary contributors: Revision history statistics Page Version ID: 994946165

(4) Göbel, Thomas & Baier, Harald. (2018). Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. Digital Investigation. 24. S111-S120. 10.1016/j.diin.2018.01.014.

(5) Mark Davis, Scott Piper, & Sujeet Shenoi. (2006). COUNTERING HOSTILE FORENSIC TECHNIQUES. https://link.springer.com/content/pdf/10.1007/0-387-36891-4_7.pdf

(6) D. (2021). dcantrell/pyparted. GitHub. https://github.com/dcantrell/pyparted

(7) Ext4 Disk Layout - Ext4. (2013). Kernel. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

(8) Kromer, M. (2021). Filesystems Benchmarked ». Linux Magazine. https://www.linux-magazine.com/Online/Features/Filesystems-Benchmarked

(9) Operating System Market Share Worldwide. (2021). StatCounter Global Stats. https://gs.statcounter.com/os-market-share

(10) Community, C. (2021). Frequently Asked Questions - Ext4. EXT4 Wiki Kernel. https://ext4.wiki.kernel.org/index.php/Frequently_Asked_Questions#What_is_delayed_allocation_.28delalloc.29.3F_What_are_its_advantages_in_Ext4.3F

(11) C. (2021a). Ubuntu Manpage: ext4magic - recover deleted files on ext3/4 filesystems. Ubuntu Manpage. http://manpages.ubuntu.com/manpages/xenial/man8/ext4magic.8.html

(12) C. (2021b). Ubuntu Manpage: ntfsundelete - recover a deleted file from an NTFS volume. NTFSUndelete. http://manpages.ubuntu.com/manpages/bionic/man8/ntfsundelete.8.html

(13) Zoulas, C. and Hrishikesh, 2016. Addition of Ext4 Extent and Ext3 HTree DIR Read-Only Support in NetBSD. 2nd ed. [ebook] https://www.netbsd.org/gallery/presentations/hrishikesh/2017_AsiaBSDCon/abc2017ext4_final_paper.pdf: netbsd. Available at: https://www.netbsd.org/gallery/presentations/hrishikesh/2017_AsiaBSDCon/abc2017ext4_final_paper.pdf [Accessed 10 April 2021].