A Python Library for Forensic File System Analysis

Bruno Constanzo, Engineer,[12], Ana Di Iorio[12], Engineer, Enzo Nogueira[1], Engineering Student
[1]InFo-Lab, Laboratorio de Investigación y Desarrollo de Tecnología en Informática Forense
[2]Universidad FASTA, Argentina, {bconstanzo, diana}@ufasta.edu.ar

***Abstract*– The search and recovery of information is a basic and foundational part of any digital investigation. In cases where the digital forensics experts work against storage media, filesystem analysis is a cornerstone upon which they will build their investigation. In this work, we present a Python library that lets programmers and examiners interact with the low-level structures and data from a filesystem in a simple, yet powerful manner. We also give a brief discussion on the use of this library in academic environments, to teach and show examples of operating systems concepts, and in specific digital forensics courses.**

Keywords—Digital Forensics; Filesystems; Python; Operating Systems; Digital Evidence.

I. Introduction

Almost every digital forensic analysis involves the analysis of storage media, and consequently a filesystem of some kind. Filesystem forensic analysis requires searching for information, files, metadata, or remnants of these in the content of blocks, or the relevant structures [1, 2].

Operating systems manage user information through filesystems, sets of logic rules and data structures that organize data in a storage medium in an understandable abstraction, both for human users and other programs, while keeping the monopoly over the access to the underlying resources [3, 4]. This guarantees that programs can only access and/or modify the stored information through the abstractions and mechanisms that the OS provides, thus keeping integrity and security.

While these abstractions and mechanisms are useful for everyday use of a computer, digital forensic experts benefit from having a lower-level access to the underlying information and metadata. To begin with, digital forensics experts usually work over image files and not directly over the storage device [7]. Whereas it is possible to mount the image files as a device (either through Linux loop devices [8, 9] or Windows virtual device mounting [5]) and access them as just another file system on the examiners computer, that level of access may hide details, metadata and information which could be relevant to the investigation.

Modified-Access-Change/Creation (MAC) timestamps, the filesystems journal, a files block/cluster/extent allocations, owner and permissions metadata, amongst others, are important bits of information that can only be accessed by tools that work on the same level as the operating system. Usually, this information can be read from an image file using filesystem parsers: programs or libraries that can read the structures raw bytes and represent that information in a meaningful way to the examiner.

Digital Object Identifier: (only for full papers, inserted by LACCEI).
ISSN, ISBN: (to be inserted by LACCEI).
DO NOT REMOVE

In this paper we present a python library that implements parsers for partition tables, a few filesystems, and basic support for VHD files. Through it, users can access forensic images of storage devices to examine them and perform complex analysis with python. As this library only uses the languages standard library, it is portable to different operating systems and has been tested on Windows, Linux and Android.

The development of this library started as a tool that would be used in class to help students of an Operating Systems course better understand real world file systems. It was promptly found, however, that the low-level access it provided could be useful beyond, and that forensic file system analysis could be performed with it.

The following section gives a brief description of theoretical concepts needed for the rest of the paper. Section III discusses the design of the library. Section IV goes into details and examples of its usage. Finally, Section V covers conclusions and future work.

II. Concepts and Theory

From an operating systems perspective, most storage media can be thought of as block-oriented devices which read and write data on a per-block basis, and thus they operate only one whole block at a time[1]. Since storing and retrieving information this way would be cumbersome, file systems provide abstractions that give a simpler mental model for users, programs, and software developers.

When we speak of a filesystem, it can be interpreted in two ways:

- In a broad sense, it is the set of rules, data structures, and conventions that define how to store, retrieve, and manage data for persistent storage. As such, we can speak of FAT, NTFS, or Ext4 as filesystems.
- In a narrow sense, in which we refer to a particular filesystem in a specific storage device: the FAT file system on a thumb drive, or the Ext4 file system on an SSD, to give a few examples.

Modern operating systems work with a layered approach, with the lowest level abstractions (device drivers) sending instructions to a specific device or class of devices, and higher-level abstractions handling different file systems (in the broad sense). This allows to support filesystems over a range of different physical storage mediums, through the same file

[1] Blocks, or sectors in this case, are usually 512 bytes long, though newer drives work with 4096 bytes blocks. These are hardware details that should be hidden from higher abstraction levels, but over the years this has been a leaky abstraction.

system driver. Depending on the operating system, file system support can be implemented through different mechanisms:

- On Windows there are File System Drivers [5, 6], that sit on top of mass storage or local or network drivers.
- Linux can both load kernel modules for specific file systems or use the *Filesystem in USErspace* (FUSE) interface to allow a user-mode process to handle the implementation details [6], such as opening and closing files, reading and writing data, etc.

In a very broad sense, commonly used file systems have a root directory, some sort of mapping (table, indexes, trees), and a metadata structure for individual files. Directories are objects that store files inside them. It is commonplace for them to be implemented as a special kind of file that stores file records, the structure that is finally responsible for storing the files metadata and uniquely identifying information. Additional metadata can be derived from hierarchies and supplementary metadata structures when needed.

Most filesystems will have some header structure in the first sector where they store metadata, bootloader code, and important information to parse the rest of the filesystem. Microsoft file systems usually work with file tables and allocating fixed-sized clusters for files, while Linux filesystems are based on inodes and directories simply hold the files name and inode number to reference the file, with the inode containing the rest of the files metadata.

Journals are a particular kind of file that modern filesystems use, which logs an entry for every metadata change that is about to be made to the filesystem, and then another entry when that change is effectively written to disk. This allows operating systems to quickly verify the integrity of a file system, without having to scan all its structures, speeding up integrity checks for large volumes.

A. Windows File Systems

The File Allocation Table filesystem, or FAT, was developed by Microsoft during the 80’s and 90’s for their MS-DOS and Windows operating systems. During those years, different versions were released, with each new revision including improvements and new features.

The latest version, called FAT32, had support for up to 2 TiB volume sizes, 4 GiB (-1 byte) maximum file sizes and, through the VFAT extension, could handle long file names[2].

The three main structures that describe the filesystem in its entirety are the Volume Boot Record (VBR), the File Allocation Table (FAT), and the Directory Entries.

The VBR works as the filesystem’s header, and stores metadata, values and pointers that guide the OS into loading the rest of the filesystem. The FAT is an array that, for every cluster in the filesystem, tells whether it is free or allocated, and if allocated, which is the following cluster that makes up the file that it is part of.

Directory Entries are 32-byte sized structures that hold the name, MAC times, size, first cluster and attributes for a file. By reading this structure, the OS can read the entirety of the file by following the FAT cluster chain. Directories are simply files that hold Directory Entries inside them and are stored alongside regular files.

On FAT variants file timestamps are limited to 2-, 4- and 5-byte structures. The last access field is only a date, without time information (2 bytes). The modified field does have a time component (4 bytes) however its time resolution is limited to 2 second intervals. The created field is the only full-sized (5 bytes) timestamp, comprised of a data, and time with a granularity of 10 milliseconds.

Through its simplicity, FAT32 has been made a de-facto standard able to interoperate between various operating systems, devices and storage media [10]. While a bit aged, it is still relevant for these uses, and as the filesystem of choice for EFI Boot Partition [11, 12].

The New Technology File System (NTFS) was the filesystem that became integral part of the NT series of operating systems. While still having a file table and being cluster-oriented, it added a host of features and improvements that make it superior to FAT32 as a system-oriented filesystem.

NTFS supports much larger volumes and file sizes than FAT32, Unicode file names, native compression and encryption, sparse files, object permissions, alternate data streams, and journaling, to name the most prominent features.

The Master File Table (MFT) is a metafile that stores instances of FileRecord, the structure responsible for holding each files metadata (analogous to the DirectoryEntry structure of FAT, but larger and more flexible). On NTFS there are no separate structures for administrative data, but metafiles are special files within the file system that hold metadata about it and are identified with a $ at the start of their name. So, for example, there are $MFT and $MFTMirr files for the MFT and its mirror, a $Boot file that holds boot code and filesystem header information, to name a few.

From a forensic point-of-view, one of the most significant changes is that the timestamps in NTFS are stored using a 64-bit integer with an epoch of Jan 1st 1601, with 100ns resolution [13]. Also, in addition to the standard MAC timestamps, there is a Record timestamp that stores the last time the File Record was modified.

B. Linux File Systems

The Ext family of filesystems have been the standard on Linux operating systems for almost three decades, with some incremental improvements leading all the way up to Ext4, the currently used version. An important detail about Ext3 and Ext4 is that they have been backwards and forwards compatible with the previous version [14, 23], and Ext4 drivers can mount both Ext3 and Ext2 file systems. This is a helpful feature for forensic filesystem parsers as it allows

[2] Though without it, it only supports 8.3 file names.

previous parsers to keep working on newer versions, and usually requires little work to add the new features. Conversely, it is possible to read older filesystems with newer parsers.

Ext4 supports large file volumes, large files (optimized using extents), implements journaling, is backwards compatible, improved timestamp resolution and range, can use transparent encryption, and implemented a host of performance-related improvements over previous versions.

From a forensics perspective, Ext file systems have their own particularities that must be taken into consideration. Ext2/3 both have modified, last access, and inode change timestamps, with a time resolution of one second, stored in a 32-bit signed integer that follows the UNIX time format. Ext4 adds 32 bits more to the timestamps and adds a file creation timestamp (*crtime*). Both the extended timestamp bits and the file *crtime* are stored in the extended part of the inode, to keep compatibility with previous versions [21, 22]. Through these extended bits, Ext4 timestamps attain nanosecond granularity, and have an extended range of dates (roughly 500 years more than the previous version).

C. *Forensic File System Analysis*

Forensic file system analysis involves processing the low-level data found in a storage medium in search of information that is relevant to an investigation. Based on Ref. [1], it can be split into several categories, depending on what is the object of study in which the examiner focuses:

- File system category, that tries to identify a filesystem uniquely, describing its layout and features. This means identifying all the relevant data structures for a filesystem to be parsed correctly.
- Content category, focused on the data units that store file information themselves (clusters, blocks, etc), analyzing the unused space on these units, or the units that have been marked as defective or unusable.
- Metadata category focuses on the descriptive information about files that is stored, such as timestamps and file locations.
- File name category, as a distinct category from metadata. The focus is placed on the file name and its path, which usually must be constructed from other files metadata (parent directories, recursively).
- Application category focuses on nonessential data, e.g.: journals, and other application specific file types a filesystem might define as important, though not required for the file system to work.

The Sleuth Kit [15, 16] is a library and set of command line tools that implements these categories over a large number of file systems. While it is a standard in the field, in class students found it a bit too complex, and they had to go through a steeper learning curve than expected.

III. Design and Development

Haruspex was developed initially as a teaching tool, to be used in a special assignment on an Operating Systems course, in which students are expected to apply basic concepts of file systems. As development went on, its capabilities and potential became evident, and it was extended beyond its initial goals to accommodate file system exploration and forensic analysis.

In developing the library, we had a few clear design goals from the beginning: it had to be portable across operating systems, easy to install, simple to use, and its code had to be readable, while retaining flexibility and extensibility.

Portability and an easy installation were needed to simplify the environment set up and configuration. When using other programs and libraries, from time-to-time students would find difficulties which interfered with the expected learning process. Having a complex install process, unfortunately, is a not-so-rare trait among open source digital forensic tools and was something we wanted to avoid at all costs.

Ease of use and code readability as goals ensure that anyone can study and verify how the code works. Inside the classroom, that means students can follow how a filesystem works, and develop the functions and tools based on the library that their assignment requires. Outside the classroom, it ensures that developers and digital forensic experts can follow examples and documentation to use Haruspex in a report, or implement new filesystems, features, or new tools on top of the library that they may require.

The choice of Python as programming language was simple: the team had extensive experience with it, and its ethos of simplicity, the extensive standard library, wide support across many platforms, and its wide use in the digital forensics' community were perfect fits for the project. Since it is very simple to learn to program in it, a short introduction is enough to meet the minimum level required for the assignment for students who are not familiar with Python.

Haruspex has been released under LGPL-2.1, its code is available on GitHub[3], and has been uploaded to the Python Package Index, so it can be pip-installed.

The library implements classes that know how to parse binary structures read from storage media or forensic images. Once the objects are instantiated, the bytes are parsed on initialization and all the data and metadata is made available through attributes. Python double-underscore methods and properties are used extensively to keep parsing code simple, while the setters and getters of the properties handle type and range checks. The languages standard library is a great asset too, in particular the *datetime* module to handle timestamps,

[3] https://github.com/bconstanzo/haruspex.

and the *struct* module to handle the packing and unpacking of binary data.

There is a strong focus on the interactive use of Haruspex, as it brings a quick and reliable interface for users and allows developers and forensic experts to test things on the go as they explore a drive or test code. There is an extensive use of docstrings, *__repr__* methods to provide a brief description of the objects, and *__str__* to provide a detailed view. This posture is actually opposite to the standard python practice, where *repr()* should give detailed information, and *str()* a good-for-presentation string, but through use and tests it was found that following that convention resulted in a suboptimal experience on interactive sessions. The development team constantly tries and tests code and usage patterns on interactive shells to find these rough patches, or places where the interfaces can be improved to provide a better experience for users.

The classes in the library that parse bytes all receive a bytes object (usually read from a file-like), which is stored as received and unpacked into the different attributes of the object the class represents. For example, instantiating an MBR Partition requires 16 bytes read from the correct location of the first sector. There are also higher-level constructs, like the classes for a whole file system (FAT32 and Ext2 currently), which receive an open file(-like) and an offset into it where the filesystem starts. Then the class handles reading the filesystems header, unpacking and parsing the correct structures, and provides a ready-to-use object.

As of writing, Haruspex provides support for the following structures and file systems:

- MBR and GPT partition tables.
- VHD fixed-size images [17].
- FAT32, without long-filename support (the extra directory entries used for LFN are ignored).
- Ext2, as described in [18].
- Parts of NTFS, just enough to parse $I30 files[4].

The MBR and GPT modules both provide *Partition* and *Table* classes, each instantiated from a properly sized bytes buffer. The *Table* class handles reading the header (if any) on the bytes that were passed, and then parses the bytes with the partition list, making them readily available as a list object.

Ext2 and FAT32 modules provide *Ext2* and *FAT32* classes, that handle the reading of structures and initialization of attributes for the whole filesystem. Both have specific attributes, but they share a common *open()* method that can be used to open either directories or files as a way to simplify the exploration of a filesystem.

In both file systems modules, *Directory* and *FileHandle* classes are implemented. The *Directory* classes read a directory structure from the filesystem, and exposes a *files* attribute, a list of the all the file entries found in the directory (including deleted entries). The *FileHandle* class is a bit more complex, as it aims to provide methods compatible with pythons file-like API. This allows the users to open a file from the filesystem and use the resulting object just like they would a file opened with pythons built-in *open()*. The classes handle all the details to implement *read()*, *seek()*, and *tell()* methods consistently with the underlying file system.

Going to a lower level, each filesystem module implements classes for their specific structures. In *Ext2* we find *DirectoryEntry*, *GroupDescriptor*, *Inode*, and *Superblock* classes, whereas *FAT32* has simpey has the *FileRecord* class to parse the Directory Entry structures.

IV. Experiences and Use

As it was stated, one of the design goals for the library was ease of use. Through testing and use in undergraduate and postgraduate courses and digital forensic trainings, we have found this objective is currently met, though it can probably be improved upon.

The beginning of an interactive session using Library name is shown on Fig. 1:

- The image file “virtual.img”, is opened in binary read mode.
- The first sector of the image is read, and from it, an MBR partition table is instantiated.
- A single partition of FAT32 LBA type is found at the 128th sector in the image.
- A FAT32 filesystem is instantiated, at byte-offset 128*512 of the image file.
- Inspecting the root attribute (which is a *Directory* object), the files on the filesystems root can be accessed, and the rest of the filesystem analyzed.
- Directory objects abstract a simple interface for directories, while still holding a reference to the underlying filesystem structure (in this case, a FAT32 Directory Entry).
- *FileRecord* objects provide the underlying details of the implementation, with all the metadata and a copy of the on-disk bytes.
- Printing the information of a directory entry shows all the metadata of a file (sans the long file name, should the file have one).
- The time metadata of the file show the limitations that were discussed with respect to timestamps on FAT32.

From there on the analysis can be taken to various directions, depending on what is found or being searched for:

- If deleted files are of interest, *FileRecord* objects have a *deleted* property that is consistent with the behavior of FAT32 for deleted objects.
- Collections of files can be sorted or filtered by any of their attributes using pythons *sorted()* and *filter()* built-in function. All metadata for files

[4] For a discussion on their use as forensic artifacts, see [1, 19], some implementation details are available in Willi Ballenthin’s INDXParse [20], however documentation for it, as of writing, has gone offline and only the GitHub repository is available.

has been implemented through pythons built-in and standard library types, so their truth values and comparison behavior are intuitive, following a principle of least surprise.

- Pythons *re* module could be used to match filenames with regular expressions.
- More complex conditions can be tackled using list-comprehensions, user defined functions or any other programming construct the user finds appropriate and is familiar with.

```
>>> import libname
>>> img_file = open("virtual.img", "rb")
>>> mbr = libname.mbr.Table(img_file.read(512))
>>> mbr
< MBR Partition Table @ 2034950680136 >
    < Partition - FAT32 LBA - boot: False @ 128
of 124928 >
>>> fs = libname.fat32.FAT32(img_file, 128 * 512)
>>> fs
< FAT32 @ 65536 of <_io.BufferedReader
name='virtual.img'>>
>>> fs.root.files
[< DirectoryEntry: VIRTUAL.>, < DirectoryEntry:
<DIR> SYSTEM~1>, < DirectoryEntry: TEST.TXT>, <
DirectoryEntry: <DIR> DIR>, < DirectoryEntry:
<DIR> $RECYCLE>]
>>> print(fs.root.files[1])
< DirectoryEntry: <DIR> SYSTEM~1
    size        :           0
    attributes  :      rHSvDa
    cluster     :           3
    created     : 2019-04-25 19:42:25.030000
    last_access : 2019-04-25 00:00:00
    modified    : 2019-04-25 19:42:26
    deleted     :       False
>
```

Fig. 1 Example of an interactive python shell session using Haruspex to open a file image parsing its partition table and filesystem and printing a specific files metadata.

A. *Undergraduate courses*

In the Operating Systems course where we use the library, students are tasked with a special assignment where they have to create a virtual disk, initialize it with a partition table, give it a FAT32 partition, and perform a series of file operations over the volume. At each step, they are required to disconnect the virtual disk, analyze its image file with Haruspex, and take notes of the changes. They have to turn in a report where they detail their findings and explain the changes they see with references to both operating systems bibliography and the referenced documentation they are given for the filesystem and the partition schemes.

In a practical annex of the report, they are tasked with writing a few simple functions using the library, picked form a pool of different functions of interest. Fig. 2 shows the two functions a student programmed to figure out if a file was fragmented, or contiguously allocated.

```
def list_clusters(fs, record):
    ncluster = record.cluster
    clusters_list = []
    while ncluster < 0x0ffffff0:
        clusters_list.append(ncluster)
        ncluster = fs.fat1[ncluster]
    return clusters_list


def number_of_fragments(list_of_clusters):
    fragments = 1
    iterclusters = iter(list_of_clusters)
    prevc = next(iterclusters)
    for currc in iterclusters:
        if (prevc + 1 != currc):
            fragments += 1
        prevc = currc
    return fragments
# usage:
c_list = list_clusters(fs, record)
num = number_of_fragments(c_list)
print(f”The file has {num} fragment(s).”)
```

Fig. 2 A student's code for an assignment where they had to find out if a file was fragmented, and how many fragments it had. While it does not explicitly import functions or classes from Haruspex, it does rely on the attributes of the *FAT32* and *FileRecord* classes to work.

The experience so far has found that students who work on this assignment achieve a better understanding of the theoretical concepts applied in file systems, and they greatly value the opportunity to relate concepts and theory that is seen in class with real world scenarios.

B. *Postgraduate courses and digital forensics trainings*

Haruspex has also been used in courses teaching digital forensics (usually abbreviated DFIR, for Digital Forensics and Incident Response) for postgraduate students, and in trainings for DFIR experts. In these courses, it was found to be even more positive than on undergraduate level. In Argentina the digital forensics field is still nascent, despite having had important developments in that past 10 years. In this scenario, many DFIR experts are graduate professionals in computer sciences, software engineering or electronic engineering who have steered their career into the field, self-taught, and learning on a need-to-know basis. Historically there has been no postgraduate education in the field, something that has only changed recently.

When teaching digital forensics courses, it is usually needed to give a review of operating systems concepts, and particularly file systems. Most professionals have not revisited the concepts involved for years, and a small number have never taken an operating systems course, and it is their first time viewing these topics in a formal academic environment.

It is unfortunately common in our region to find so called experts simply being users of tools, who barely know how to set up a standard workflow and follow a set of instructions without understanding the mechanisms at work. In this context, our library is yet another tool that complements The Sleuth Kit and hexadecimal editors, giving students the opportunity to explore and inspect on a deep level the results that they obtain using other tools, that tend to hide details and complexity from the user.

Haruspex helps illustrate and analyze scenarios of data deletion, file and data recovery, anti-forensic techniques, and file carving, to name a few. It also helps to show specific behaviors regarding timestamps and how they are updated on different file systems. In this way, students achieve a better understanding of data recovery, forensic file system analysis, and file and data carving.

C. Use as a forensic tool

As development progressed past the early versions and the library saw more use, it quickly became evident that it was useful not just for teaching. As it implements many of the categories defined by Carrier, it can provide valuable data in a forensics analysis, providing a different interface that can result easier to use under certain situations.

The higher-level abstractions that allow the user to open and read from files and directories in a straightforward way, while still providing access to low-level details and metadata makes for a very powerful combination. In addition, Haruspexs seamless integration with Python's built-in functions and standard library allows anyone with programming experience in the language to quickly obtain information, filter, and perform advanced searches from an image file.

The library is also a good platform on top of which new tools can be built, or other file systems implemented. Haruspexs lack of dependencies other than a Python 3.6 (or newer) installation allow it to run practically on any platform.

V. Conclusions and Future Work

The design of a library for forensic file system analysis has been an interesting experiment, both in software engineering and in computer science teaching. The design goals and requirements imposed by the courses where we planned to use the software were a major influence, and we think were instrumental in the success that it has achieved so far. Feedback from students helped tailor and improve code and interfaces, and further refine the design.

Choosing Python as the programming language to develop this project gave it the platform portability that we required: Haruspex has been tested under Windows, Linux and even Android (using Termux). Not only that, but it also helped with implementing complex features through the integration with the language built-in functions and standard library.

Work is currently under way to extend the limited NTFS support that we have so far, and include other file systems, such as Ext3 and Ext4 (using the current Ext2 implementation as a base), exFAT and some custom file systems that are used in digital video recorders. There are also some prototypes in the works to build triage and data recovery tools that use Haruspex to bypass the operating system in accessing the filesystems.

Acknowledgment

We would like to thank Universidad FASTA, Ministerio Público Fiscal de la Provincia de Buenos Aires, and Municipio de General Pueyrredon, for their joint effort in creating and maintaining InFo-Lab. Without their support this work would not have been possible.

We also want to thank Ariel Podestá, Roberto Giordano Lerena, Martín Castellote, and Juan Iturriaga, for the work we've done together and their support and feedback over time, as it has helped us shape our work and careers.

References

[1] B. Carrier, "File System Forensic Analysis", Adison Wesley Professional, 2005.
[2] V. Roussev, "Digital Forensic Science: Issues, Methods, and Challenges", Morgan & Claypool, 2017.
[3] A. Tanenbaum, "Modern Operating Systems, 3rd edition", Pearson, 2009.
[4] W. Stallings, "Operating Systems: Internals and Design Principles, 6th edition", Pearson, 2009.
[5] M. Russinovich, D. Solomon, A. Ionescu, "Windows Internals, 6th Edition, Part 2", Microsoft Press, 2012.
[6] Microsoft Corporation, "File systems driver design guide", online documentation, 2022, https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/.
[7] Linux FUSE (Filesystem in Userspace) reference implementation, various authors, https://github.com/libfuse/libfuse.
[8] B. Nikkel, "Practical Forensic Imaging: Securing Digital Evidence with Linux Tools", 2016, No Starch Press.
[9] Linux man pages, losetup(8), online version: https://linux.die.net/man/8/losetup.
[10] Standard of the Camera & Imaging Products Association, "Design rule for Camera File system: DCF Version 2.0", 2010, online copy: https://web.archive.org/web/20130930190707/http://www.cipa.jp/english/hyoujunka/kikaku/pdf/DC-009-2010_E.pdf.
[11] Microsoft Corporation, "Microsoft Extensible Firmware Initiative FAT32 File System Specification – FAT: General Overview of On-Disk Format", 2000, online version: http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc.
[12] Unified EFI Inc., "UEFI Specification Version 2.5, Section 12.3 File System Format", 2015, online version: https://uefi.org/sites/default/files/resources/UEFI%202_5.pdf#page=536.
[13] Microsoft Corportation, "File Times", online documentation https://docs.microsoft.com/en-us/windows/win32/sysinfo/file-times.

[14]M. Jones, “Anatomy of ext4”, IBM Developer content, 2009, online: https://developer.ibm.com/tutorials/l-anatomy-ext4/.

[15]The Sleuth Kit, software, http://www.sleuthkit.org/.

[16]The Sleuth Kit, git repository, https://github.com/sleuthkit/sleuthkit.

[17]J. Metz, A. Albertini, “Virtual Hard Disk (VHD) image format”, online documentation for libvhdi, online https://github.com/libyal/libvhdi/blob/main/documentation/Virtual%20Hard%20Disk%20(VHD)%20image%20format.asciidoc.

[18]D. Bovet, M. Cesati, "Understanding the Linux Kernel, 2nd Ed", O'Reilly Media, 2002.

[19]C. Tilbury, “NTFS $I30 Index Attributes: Evidence of Deleted and Overwritten Files”, blog post for SANS, 2011, online: https://www.sans.org/blog/ntfs-i30-index-attributes-evidence-of-deleted-and-overwritten-files/.

[20]W. Ballenthin, “INDXParse”, software, online repository: https://github.com/williballenthin/INDXParse .

[21]H. Pomeranz, “Understanding EXT4 (Part 1): Extents”, blog post for SANS, 2010, online: https://www.sans.org/blog/understanding-ext4-part-1-extents/.

[22]H. Pomeranz, “Understanding EXT4 (Part 2): Timestamps”, blog post for SANS, 2011, online: https://www.sans.org/blog/understanding-ext4-part-2-timestamps/.

[23]A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, “The new ext4 filesystem: current status and future plans”, Ottawa Linux Symposium, 2007, online: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf.