

Automatické klasifikace souborů na základě rozpoznávání textových bloků

The Automatic Classification of Files Based on Text-data Blocks
Recognition

Radovan Holík

Bakalářská práce
2012



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2011/2012

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Radovan HOLÍK**
Osobní číslo: **A08042**
Studijní program: **B 3902 Inženýrská informatika**
Studijní obor: **Informační a řídicí technologie**

Téma práce: **Automatické klasifikace souborů na základě
roznávání textových bloků**

Zásady pro vypracování:

1. Zpracujte literární rešerši na dané téma.
2. Analyzujte strukturu standardů PDF.
3. Analyzujte strukturu standardů založených na XML (DOCX, Open Office XML).
4. Analyzujte možnosti praktické aplikace tématu v oblasti Computer Forensics a obnovy dat.
5. Navrhněte ukázkovou aplikaci.
6. Demontrujte výsledky a formulujte závěr.

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. MOHAY, George, Alison ANDERSON a Collie BYRON. Computer and intrusion forensics. Boston: Artech House, 2003. ISBN 15-805-3630-1.
2. DEVI, V. Susheela a N. Narasimha MURTY. Pattern Recognition An Algorithmic Approach. 1. London: Universities Press (India) Pvt. Ltd, 2011. ISBN 978-085-7294-951.
3. ISO 32000-1:2008. Document management Portable document format ? Part 1: PDF 1.7. Geneva, Switzerland: International Organization for Standardization, 2008.
4. ISO/IEC 29500-1:2011. Information technology ? Document description and processing languages – Office Open XML File Formats. Geneva, Switzerland: International Organization for Standardization / the International Electrotechnical Commission, 2011.
5. ISO/IEC 26300:2006. Open Document Format for Office Applications (OpenDocument) v1.0. Geneva, Switzerland: International Organization for Standardization, 2006.
6. VYMĚTAL, Tomáš. Computer forensics. Brno, 2009. Diplomová práce. Masarykova Univerzita.

Vedoucí bakalářské práce:

Ing. Erik Král

Ústav počítačových a komunikačních systémů

Konzultant:

Davide Ariu, Ph.D.

Datum zadání bakalářské práce:

24. února 2012

Termín odevzdání bakalářské práce:

8. června 2012

Ve Zlíně dne 24. února 2012



prof. Ing. Vladimír Vašek, CSc.
děkan



prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

ABSTRAKT

Tato práce spočívá ve studiu typů souborů obsahující textová data za účelem vývoje systému, který je schopen automaticky klasifikovat soubory.

Hlavním cílem práce je automatické třídění souborů obsahující textová data bez podpory informací poskytnutých od souborového systému. V teoretické části je vysvětleno, jak obvykle pracují nástroje pro obnovu dat a Computer Forensics, dále je diskutován problém ohledně stanovení typu fragmentované části souboru. To je často hlavním tématem pro aplikace Computer Forensics. Mimo jiné, vyhledávání řetězců je obzvláště důležité, protože velmi často je důkaz hledán uvnitř textových dokumentů. Dále je v této části také popsána vnitřní struktura souborů PDF a Office XML. Praktická část obsahuje informace o implementaci softwarového nástroje, který je schopen z poskytnutého bloku dat získat informace, které mohou být použity pro automatickou klasifikaci dokumentu.

Klíčová slova: Computer Forensics, klasifikace fragmentovaných souborů, PDF, Office XML

ABSTRACT

The thesis consists in the study of file types containing textual data for the purpose of developing an automatic file classification system.

The main goal of the thesis is the automatic classification of files containing textual data without the support of the information provided by the file system. In the first part is explained how data recovery tools typically work and is discussed the problem of determining the encoding type of a file fragment. This is an important issue for Computer Forensics applications. Among these, string search is particularly relevant since very often crime evidence is searched within textual documents. In addition is described the internal structure of PDF and Office XML-based files. The analysis contains information about implementation of a software tool able to extract from a block of data the features that can be used for automatic classification of the document.

Keywords: Computer Forensics, File-Fragment Classification, PDF, Office XML

ACKNOWLEDGEMENTS

First I would like to thank Dr. Davide Ariu, my bachelor thesis consultant. He gave me a lot of good advice and hints while dealing with challenging tasks. Special thanks to PRA Group at University of Cagliari for collaboration, especially to Prof. Giorgio Giacinto and Dr. Davide Ariu. I would like to also thank Ing. Erik Král, my thesis supervisor.

Especially thanks belong to my family for supporting me throughout my study period.

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....
podpis diplomanta

CONTENT

INTRODUCTION	9
I THEORY.....	11
1 IMPROVED STRING-SEARCH BASED ON STATISTICAL ANALYSIS OF FILE FRAGMENTS.....	12
1.1 INTRODUCTION	12
1.2 THE CURRENT SCENARIO	13
1.2.1 Operating System Native Search Tools.....	13
1.2.2 File Carving.....	13
1.2.3 File Recovery vs. Computer Forensics Tools	15
1.2.4 Byte-level Search	15
1.2.5 File Structure Based Carving	16
1.3 RELATED WORKS	18
2 PORTABLE DOCUMENT FORMAT.....	21
2.1 A BRIEF HISTORY	21
2.2 STRUCTURE OF PDF FILE	22
2.2.1 Header	22
2.2.2 Body	23
2.2.3 A Cross-reference Table.....	23
2.2.4 Trailer.....	24
2.2.5 Incremental Update	25
2.2.6 PDF Data Types - objects.....	26
2.2.7 Stream Object.....	28
2.3 AN EXAMPLE OF PDF RAW FILE	31
2.4 SUMMARY INFORMATION OF PDF	32
3 OFFICE XML-BASED FILES	33
3.1 OPENDOCUMENT FORMAT	33
3.1.1 File Structure	33
3.2 MICROSOFT'S OFFICE OPEN XML.....	36
3.2.1 File Structure	36
3.2.2 Embedded Objects.....	39
3.3 SUMMARY INFORMATION OF OFFICE XML-BASED FILES	40
II ANALYSIS.....	41
4 SPECIFICATION OF THE APPLICATION	42
4.1 MOTIVATION	42
4.2 REQUIREMENTS SPECIFICATION.....	42
4.2.1 The Required Functions	42
4.3 HOW WORKS THE APPLICATION	43
5 IMPLEMENTATION OF THE APPLICATION.....	44

5.1	ALGORITHMS OF RECOGNITION	44
5.1.1	PDF Algorithm	44
5.1.2	Algorithm of Office XML-based Files	48
5.2	THE CONTROL LOGIC WITHIN THE MAIN METHOD	52
5.3	STRUCTURE OF THE APPLICATION	53
5.3.1	PDF Methods	53
5.3.2	OOXML Methods	56
5.3.3	ODF Methods	58
5.3.4	General Methods	61
6	RESULTS.....	64
6.1	OCCURRENCES OF PDF OBJECTS IN TEST FILES	64
6.2	OCCURRENCES OF OOXML FEATURES IN TEXT FILES	65
6.3	OCCURRENCES OF ODF FEATURES IN TEXT FILES.....	66
	ZÁVĚR	68
	CONCLUSION	69
	LIST OF REFERENCES	70
	LIST OF ABBREVIATIONS	73
	LIST OF FIGURES	74
	LIST OF TABLES	75
	LIST OF ATTACHMENTS	76

INTRODUCTION

Development and expanse of the internet and computers causes big growth of textual data and information. Nowadays, newspapers and books are replaced by their electronic versions and all information is converted to an electronic form. This increase of textual data and documents is a big challenge for research and development in their classification and categorization.

Modern text files are containers of content with different nature. The complexity can be exploited for malicious (e.g. PDF can vehiculate viruses) or benign purposes. The information about the file structure can be used to data recovery or Computer Forensics. Data recovery applications are commonly used for business but they only solve the problem of file recovery. In fact, a typical issue in both data recovery and Computer Forensics application is to recover data from which the information provided by the file system is not available anymore. In computer forensics, this is usually done for the purpose of acquiring legal evidence to be eventually used in front of the law court. One of the most powerful techniques in the Computer Forensics is called Carving using which we can restore unknown files. Carving has its main limitation in fragmented or corrupted files, for which recovery may result indeed challenging.

In this work, file types containing textual data have been studied for the purpose of developing an automatic file classification system. The final goal of the thesis is the automatic classification of files containing textual data without the support of the information provided by the file system. A similar system might result of interest in both Computer Forensics and Data Recovery applications. Even if textual data can be found in a plenty of different file types, the activity focused on two main file types:

- Adobe PDF files
- XML-Based Microsoft Office and Open Office files

In the first part it is described how data recovery and forensics tools typically work. In addition, algorithms which have been recently proposed for the automatic classification of files are briefly described.

In following section the structure of both PDF and XML-Based Office (Microsoft and Open Office) files is described. The aim of this study was to identify the features that can be used to assess the various file types and in addition to locate the text within a given file.

In the first section of the analysis a specification of the application is explained in detail. Next section is an implementation of the application where the algorithms and also flowcharts used in this thesis are described more in details. The sixth and final chapter consists of the results of the work. The possible usage of the application is also reviewed in this section.

Finally, a software tool is able to extract features from a given block of data. The tool is able to provide a features vector that can be eventually used by a machine learning algorithm for the automatic classification of files. The tool is also able to extract text from data, if present.

I. THEORY

1 IMPROVED STRING-SEARCH BASED ON STATISTICAL ANALYSIS OF FILE FRAGMENTS

1.1 Introduction

According to ForensicWiki, *Computer Forensics* is defined as the practice of *identifying, extracting and considering evidence from digital media such as computer hard drives*. Basically, a computer forensics expert (the „digital investigator“) try to find digital evidence. Digital evidence can be found not just within a computer hard drive but also on all electronic devices (e.g. mobile or smart-phones, game consoles, GPS navigation device). Digital evidence can be provided by every piece of document (e.g. a PDF file, an email, a picture, a log file) that can in some way confirm or demolish the hypothesis formulated by the investigator.

The type of the document on which the investigator will search for the evidence, basically depends on the scenario of the investigation. Nevertheless, in the vast majority of digital investigations the computer forensics experts is asked to find textual evidence. This is a consequence of the fact that a great deal of the stored digital data is linguistic in nature (e.g. human languages, programming languages, and system application logging conversion) [1]. Textual evidence can be found in the Internet history, in word processing documents, spreadsheets but also within emails, address book, calendars, log files and so on.

Textual evidence is usually found by the digital investigator through “string search” [1]. The computer forensics expert looks for the documents that do contain a certain string. The main problem is represented by the fact that this analysis, in the scenario of a legal action, might not be as simple as that we would perform when looking for a document in our laptop. In fact, there are at least two main problems:

- In order to prevent the evidence from being found, the person under investigation would have erased or renamed the files containing the evidence.
- The file system of the inspected device can be damaged.

1.2 The Current Scenario

In this section, we illustrate how a computer forensics expert behaves when he has to find textual evidence of a crime. Basically, he can:

- Use operating system native search tools
- File Carving
- Byte-level search

1.2.1 Operating System Native Search Tools

The list of all the files contain a certain string can be easily obtained using tools such as “Find” (on Windows) or “Spotlight” (on OSX) that are natively provided by the operating system. The results can be obviously filtered with some restriction on file extensions at during the search. Files containing textual data are usually selected (e.g. .docx, .pdf, .html, .txt). Unfortunately, this kind of approach suffers from two main drawbacks.

First, it is quite simple to evade the search tool. In fact, by replacing the extension of the file to hide (e.g. “.txt”) with an extension which is not commonly associated to text files (e.g. “.avi”) the file will probably go unnoticed. The problem is that almost every operating systems use extensions in file names to help identify what they contain (the file type). For example, file extension consist of a dot at the end of a file name and a few letters (usually three or more) to identify the file type [2]. For example the file extension as .odt, .doc and/or .txt identify a text file.

Second, when such kind of analysis is performed, not all the file formats that might include a text portion are inspected. As an example, we might consider the “.nbu” extension, which is the one associated to the backup file of Nokia mobile phones. The “.nbu” files are not text files, but they typically contain large portions of text, since they can contain the SMSs or the telephone book backed up from the mobile phone. As a consequence, they should be considered during the string search.

1.2.2 File Carving

File carving allows to perform a more in-depth analysis of the storage device. According to DFRWS [3], “Data carving is the process of extracting a collection of data from a larger

data set. Data carving techniques frequently occur during a digital investigation when the unallocated file system space is analyzed to extract files.” File system structure is not used during the process, this means that carving does not care which file system is used for storing files (e.g. FAT16, FAT32, NTFS, ext2, ext3, HSF, HSF+, ISO 9660) [4]. So what exactly is file carving? File carving is the process of finding corrupted or hidden files within a larger file. This larger file could be something as a simple text document, or for example as disk image [5]. Data block is searched block by block in the remaining data by matching the values header and footer that are unique for each file type. Header and footer are a special sequence of bytes that defines the beginning and/or end of the file. Carving is primarily useful in criminal cases where the use of carving can recover the evidence. For instance, file carving can be useful in the fight against child pornography or even terrorism where the forensics expert is able to get more evidence from the suspect’s device [6].

Until data are not overwritten or destroyed, deleted data can be recovered on all storage device, including all electronic device (e.g. mobile or smart-phones), where it could be found. It depends on conditions, but even formatted disks can be restored by using carving techniques. According to McAfee, *With the exhaustive measures of drives since 2006, there is a big chance, and you delete a document from that drive. The disk space reserved for that document will be marked “available,” but it could really take a long time before this address space on the disk is overwritten.* Even, on disk can be found deleted files years ago.

From the previous section should be clear that header and footer (magic numbers) are unique for every file format, and do not change if the file extension is changed in order to hide the file. The file carving tools commonly have a database where for every single file type is described internal structure of the corresponding headers and footers. For example, a carving tool such as PhotoRec [7] offers a database with more than 320 file extensions. On the contrary, Foremost [8] (which is another carving tool) has a smaller database but it being more flexible, as it allows to customize the database by adding new file extensions.

Nevertheless, it remains the limitation that only the files whose extension is within the database, can be carved. Obviously, even if it is almost easy to include well known file types, it is not hard to lose some interesting file during the analysis. Moreover, the analyst is asked to specify to the carving software the type of the files it should look for. Then, it is assumed that the computer forensics expert knows exactly all the file extensions where

a text can be found. In addition, when the tool starts, the analysis of the device is performed searching only for the file types specified. As a consequence, if the analyst want to search for a file type that was not initially included into the analysis (because no evidence has been found with the first attempt or simply because some file types have been forgotten), a new carving of the entire device must be performed. Furthermore, no carving tool can guarantee a 100% accuracy [9]. Thus the analysis is frequently repeated by using several tools, especially in the case that no evidence has been found. This is indeed a fairly inefficient and time consuming approach, since the analysis takes a time which increases with the number of file types considered and given that several scans of the device could be required if the crime evidence is not immediately found.

1.2.3 File Recovery vs. Computer Forensics Tools

There is a big difference between file recovery and typical computer forensics techniques. File recovery uses information provided by the file system which remains after deletion of a file. By using this technique a lot of files can be recovered but is necessary that the information provided by the file system needs to be correct. If information is not correct or even a system is formatted, the files cannot be recovered.

From the previous section should be clear that typical forensic tools deal with the raw data on the media and does not use the file system structure during its process. Almost all Computer Forensics tools do not care about which file system is used to store the files, but it could be helpful to understand how a file system works. For example, let's assume that we have a FAT file system. According to McAfee, *"When a file is deleted, the file's directory entry is changed to show that the file is no longer needed (unallocated). The first character of the filename is replaced with a marker, but the file data itself is left unchanged. Until it is overwritten, the data is still present"* [6].

The main difference is that Computer forensics tools exploit the information provided by the internal structure of a file. File is a block of stored information (binary digits) such as document in a .doc file, an image a .jpg file or a program in an exe file.

1.2.4 Byte-level Search

It represents a kind of last resort and is by far the most powerful analysis that can be performed. In this case, the searched string is matched with the data on the drive at the byte

level, without any regard of the filesystem structures or of the type of the file the string might belong to. Obviously, the match should be performed for both the ASCII and UNICODE character encodings. The offset that gives the position of the string within the file and into the analyzed device can be obtained with an accurate usage of Linux tools, such as ‘string’ and ‘grep’. Once a string has been located, the next step is to trace back the file from which the string originates. As an example, in the case of UNIX systems this can be done by exploiting the inode information. The occurrences of the strings in the erased (from the filesystem) files and into the slack-space zones can be also found.

As a result of the analysis a list of all the occurrences of the string that have been found on the device is produced. Unfortunately this list typically contains a large number of “false positives”, since the string might (accidentally) occur also within non text files. Then, an additional effort is usually required to the digital investigator to filter the results and to remove all the undesired occurrences from the list. As we will see in the following section, several automatic file classification approaches have been proposed that would be useful for the purpose of reducing these false positives. Unfortunately, they are not particularly effective, and none of them has found application in real tools. In the next section we will address this issue by proposing an alternative approach for accurate file fragment classification.

1.2.5 File Structure Based Carving

File carving is powerful technique for the data recovery. However, when file is damaged and/or fragmented it is not so useful, because fragments of file are not stored contiguously, but they are (may be) variously scattered on HDD. It means that classic method based on location header and footer will be unsuccessful. Garfinkel estimated that up to 58% of outlook, 17% of jpegs and 16% of Microsoft Word documents are fragmented [23]. Let’s

assume that files have been stored as in

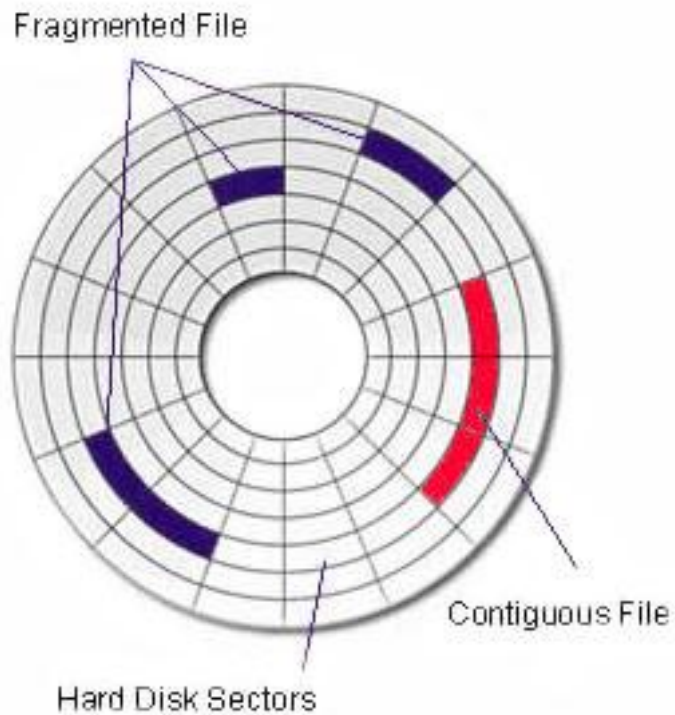


Figure 1. As we can see, Contiguous file is stored in adjacent sectors, whereas “Fragmented file” it is not. In the first case we can use standard carving tools for determining the file structure. For the second, we should analyze every single sector separately. Nowadays, a technique exist which is called "Smart Carving" [24] and it is able to handle fragmented files automatically. Nevertheless, it has a main limitation in the fact that it does not work when the file is corrupted and/or does not contain header.

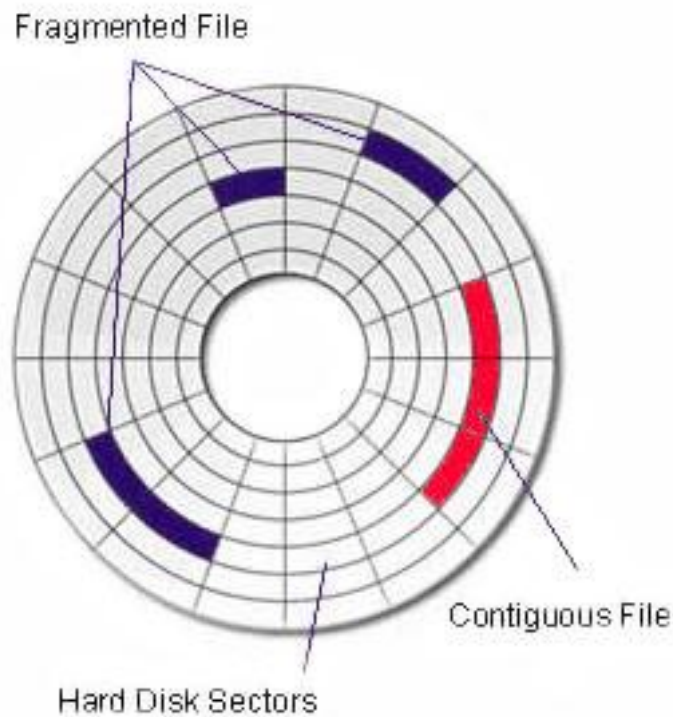


Figure 1 - Harddisk sectors

File Structure Based Carving is based on analysis of the file content of the propose file. Basically, in string searching it means that when the searched string is matched, should be automatic recognized the file structure of the sector. For instance, let's assume we want to find string with name 'for'. We will get a lot of false positives, as we can see in the Figure 2. File Structure Based Carving should be able to automatic recognize the file structure of file fragments. It means that this carving method returns file types and numbers of sectors that was found the string. When a sector's file type is identified, we can compare it to other sectors that contain similar data structure. If we will be successful we can collect file fragments together.

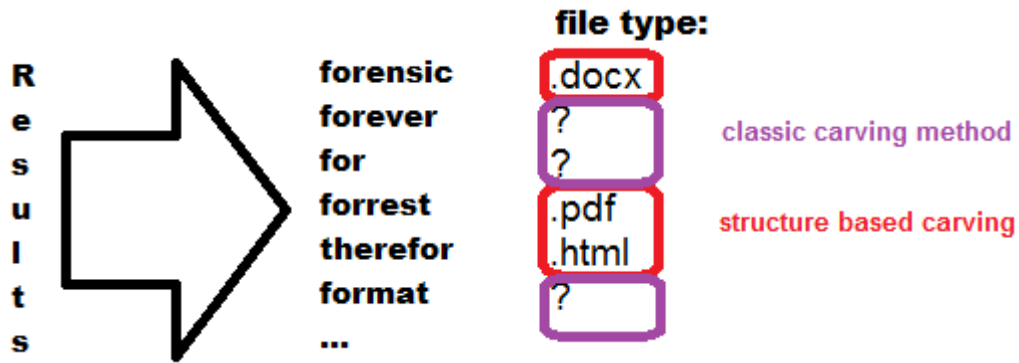


Figure 2 – How structure based carving works

1.3 Related Works

From the previous section, it should be clear that in order to support the computer forensics expert during the analysis, a string search tool should be able to automatically categorize the results depending on the type of the file where the string appeared. As an example, let us assume that the string “hello” has been found in the sector 1234 of the analyzed hard drive after a byte-level search. It would be very nice, if the tool was able to assess if the data fragment in the 1234 sector belongs (or belonged) to a text file or not. In the former case the result should be included in the list presented to the investigator whereas in the latter not.

Several works in the literature addressed the problem of establishing the type of the file from which a data fragment originates. The largest part of them analyses the statistical properties (e.g. byte histograms, byte-frequency correlations) of the distribution of the file bytes [10], [11], [12], [13]. The underground idea is that the properties of the bytes’ distribution for a certain file basically depends on the originating file type. As an example, a (significantly) different bytes’ distribution is expected for an “.exe” file type with respect to those associated with a “.doc” or with an “.avi” file. Based on this assumption, a “statistical fingerprint” is created for each file type. For instance, we could have a model for the “.exe” files, a different one for “.doc” files, another one for the “.avi” files and so on. Then, the type of an unknown file is determined by using a measure of similarity between the file and the statistical fingerprints (the type assigned to the file is that whose fingerprint is the closest).

A pioneering work in this area was that of McDaniel and Heydari [10]. The fingerprints were created by the means of the byte histogram and the byte frequency cross correlation. Veenmann proposed to represent fragments using the histogram of byte values, the entropy derived from the byte histogram and the Kolmogorov complexity [11]. In [12], the authors use both the byte distribution and the “rate of change” (the absolute value of the difference between the values of consecutive bytes). Li et al. propose an application of the n – gram analysis to this problem [13]. They create a different centroid for each file type and calculate the Mahalanobis distance among the file and the centroids.

Unfortunately, to the best of our knowledge, no one of the proposed approaches has found application in a real tool. As Roussev noticed [14], in spite of the promising results achieved, all of these works suffer from two severe drawback. The first concern is related to the size of the dataset used for the experiments. The dataset generally consisted of a number of files of different type, with an overall size of the dataset that was approximately from minus than one hundred (e.g. [12]) up to five hundreds of MB [11]. If compared to the typical size of the modern hard drives, this size appears definitely too small to let these results considered as statistically relevant.

The second (and probably more relevant) issue is related to the way files have been considered during the analysis. Let us assume we have two different PDF files. One contains only text data, whereas the other contains several images without any text. It can be easily expected that from the point of view of the bytes’ distribution, the two files will appear significantly different in spite of being both PDF files. Analogous considerations apply to every file type that can contain data of different nature (e.g. text, images). An evident case is that of the Microsoft Office files.

In order to cope with this problem, Roussev proposed to revise the approach to the file fragment analysis [14]. In particular, he suggested to reconsider the file fragment classification problem paying a more attention to the data encoding methods instead that to the file type. In fact, he observed that file types such as PDF, do not define an encoding data format but instead define a structure that groups together blocks of data with different “primitive” encodings (e.g. a jpeg image, a zlib compressed text, etc.). Then, a correct approach to data fragment classification should be aimed at recognizing primitive encoding

formats instead of file types. In the following section we will illustrate how this idea will be exploited in this thesis.

2 PORTABLE DOCUMENT FORMAT

2.1 A Brief History

Portable Document Format. (hereinafter referred to as PDF) was introduced in 1990 by Adobe Systems, but for public domain it became available in 1993. At present, PDF is one of the most popular formats for visualization and reading document, along with DOC (DOCument – Microsoft Office) and ODT (OpenDocumenT – Open Office).

The earlier version of PDF did not support some features that are nowadays commonly used. For example hyperlinks, passwording, interactive page elements and forms, mouse events, Unicode etc. The file dimensions were quite high, compared to the modem speeds of the second half of the '90s. Hence, PDF standard met various difficulties in spreading itself, especially compared with some other formats [15].


Portable Document Format	
	 Adobe PDF icon
Filename extension	.pdf
Internet media type	application/pdf application/x-pdf application/x-bzpdf application/x-gzpdf
Type code	'PDF ' (including a single space)
Uniform Type Identifier	com.adobe.pdf
Magic number	⌘PDF
Developed by	Adobe Systems
Initial release	1993

Figure 3 - PDF document's details [15]

Despite the various rivals in this sector, PDF is now widely used with a lot of advantages, as it has been optimized during these years. The actual version is the 1.7, Adobe Extension Level 8, updated for Adobe Reader X (10). Currently PDF version PDF supports text search, random access of data, bookmarks, links, annotations, interactive page elements, passwording, encryption, compression, JavaScript actions and much more [16].

2.2 Structure of PDF File

The PDF File Format is text with some binary data mixed in. Assume that file is opened in a text editor we would see the raw data objects that define the structure and content of the document. According to Thomas, *PDF implements documents as a hierarchy of tagged objects, organized into trees and/or linked lists. The objects, which can be any of seven basic types, can be purely structural in nature or can encapsulate various types of content, or attributes, or pointers to external resources* [16].

The PDF File is composed of four major parts which we can see on Figure 4.

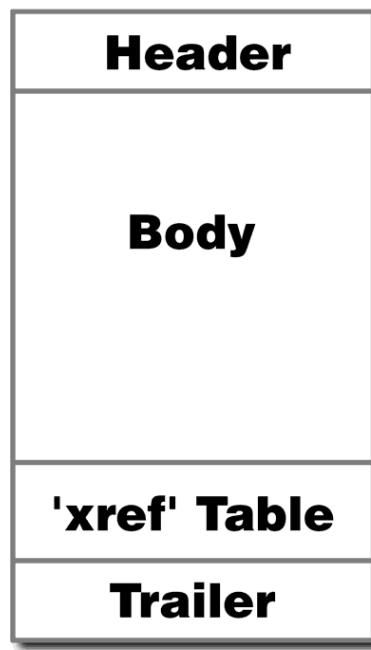


Figure 4 - The structure of PDF File

2.2.1 Header

The first line within the file defines the version of the file by magic numbers. For example:

```
%PDF-1.5
```

The header makes the browser (parser) to determine what version it is. Thus, which version will be follow [17].

2.2.2 Body

The body consists of indirect objects (images, fonts, pages,...) that completely describe appearance of the document. Body can contain invisible objects that are useful for security features or logical structure.

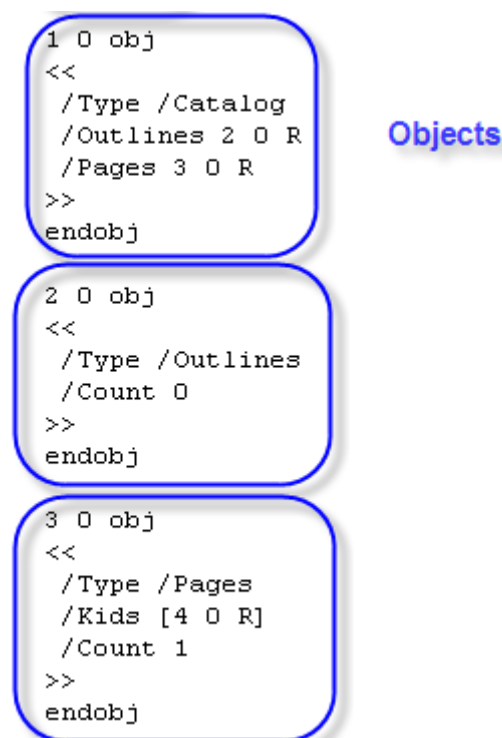


Figure 5 - Body of PDF file

2.2.3 A Cross-reference Table

Cross-reference table is a kind of content. It contains information necessary for quick access to indirect objects. For each indirect reference the table contains one line describing information about its location. Every single PDF file contains just one cross-reference table, which contains one or more sections. When file is updated, always is created new section in cross-reference table. PDF document has just one section if it has not been modified.

Each section cross-references begins within a file by keyword `'xref'`. The following subsections composed of a pair of integers determining the range of used objects. The following items contain information about the object.

Information about object is consist of byte offset indirect object in the file (shows on which byte begins the object), generation number, identifier (`f` – to indicate that the object is free, `n` - the object is in use) and of pair of characters determining end of file.

```
xref
0 8
0000000000 65535 f
0000000012 00000 n
0000000089 00000 n
0000000145 00000 n
0000000214 00000 n
0000000381 00000 n
0000000485 00000 n
0000000518 00000 n
```

*Figure 6 - 'xref' table
(EOL characters are
omitted)*

2.2.4 Trailer

The trailer is a footer of the document and shows the location in the document where begin the cross-reference table. Trailer is always at the end of file. Reading from the end of the PDF document enables fast access to individual objects referenced in the xref table. This ensures particularly rapid identification of the location of the xref table, and thus determine the location of individual objects. Trailer contains more values, some of them are required and some optional. The required values include data on the number of records in the xref table and document identifier, which consists of two string objects. The first is based with the creation of the document and the second is changed with every update of the document. For example, `encrypt` is an optional value containing information for decryption when a document is encrypted. On the other hand, `size` is necessary and shows total number of items in a *cross-reference table*, including sections added as an *incremental table*. The last line of a PDF file contains only the end of file marker, `%%EOF`.

```
trailer
<<
  /Size 8
  /Root 1 0 R
>>
startxref
642
%%EOF
```

Figure 7 - Trailer

2.2.5 Incremental Update

For update is used the incremental table that lies in keeping the original content and adding new elements to the end of the file (body, xref and trailer). PDF file is not overwritten, only added to. From previous section should be clear that PDF document ends with %%EOF, but in update document can be multiple EOF's. From this reason are often confused programs like `foremost` (described in File Carving section) [18].

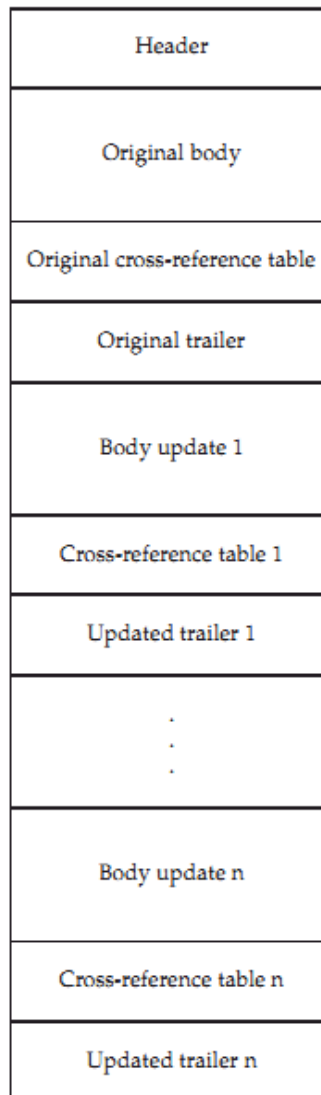


Figure 8 – Incremental update

2.2.6 PDF Data Types - objects

Each type of object can be easily recognized by start and end strings. These strings consist only of ASCII characters. Objects can be even embedded. Specification PDF defines these objects: Boolean, Numeric, String, Name, Array, Dictionary, Stream, Null a Indirect.

- **Boolean**, which is represented by `true` and `false` values
- **Numeric**, which is represented by integers or real numbers

- **String**, which is a succession of characters or hexadecimal character codes.

Examples:

```
( This is a valid string. )
( Two <2B> two <3D> four. )
```

- **Name**, which is represented by a slash /, followed by some text, no white space or punctuation allowed. Examples:

```
/Contents
/Chap6_Section1
/1.5
/.end
```

- **Array**, which is a list of different objects separated by white spaces Dictionary.

Example:

```
[ 2 4 6 8 12.5 ]           % an array of numbers
[ true /Chap9 3.14 (yes) ] % array of misc. objects
```

- **Dictionary**, which is a name object followed by another type of object.

Example:

```
<< /Type /Example /Chapters 9 /Encrypt true (no undo) >>
```

These objects are the most common objects in a PDF file. Fonts and pages are represented through dictionaries [16].

- **Stream**, is the most complex and important type for this thesis. It will be described in next section.
- **Null**, which is represented by `null`
- **Indirect**, which is defined in PDF document using *object number*, *generation number* and by keyword `obj`. At the end of the object is keyword `endobj`. An example of raw object within the file is expressed like this:

```
3 0 obj
(text)
endobj
```

, in which the number 3 is the object reference (*object number*). After this kind of the line, there is something that uniquely identifies the type of object (*generation*

number), as it's pointed out in the table underneath. The object defined here is called indirect since it can be referenced by its number; there are also objects without this reference prefix, called direct objects and are always contained inside other objects. A container object that references another object does so with the syntax '`3 0 R`' to include the previous object defined with '`3 0 obj`', on this object can be referenced from anywhere in document.

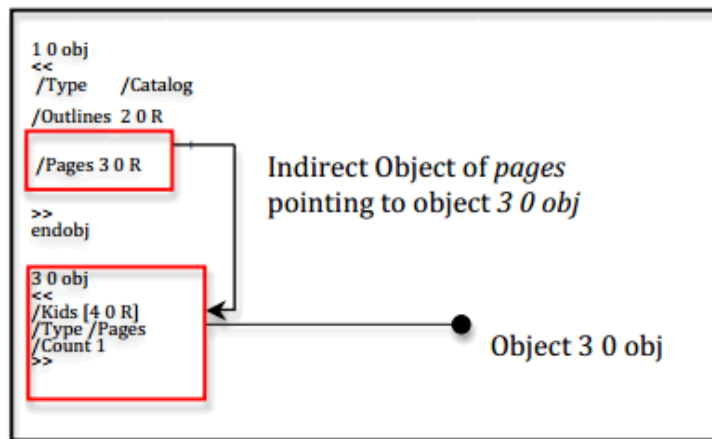


Figure 9 - Relationship of the indirect object

2.2.7 Stream Object

A PDF stream object is a sequence of bytes. This object allows to store strings of bytes like as object `string`. But its length is unlimited. Therefore, it is suitable for long storage of data such as images.

Stream object begins by dictionary object, which stores all necessary information about object (<< especially length of data >>), follows keyword **stream**, sequence of bytes and the keyword **endstream**. Streams are always indirect objects, so they always begin with and object reference.

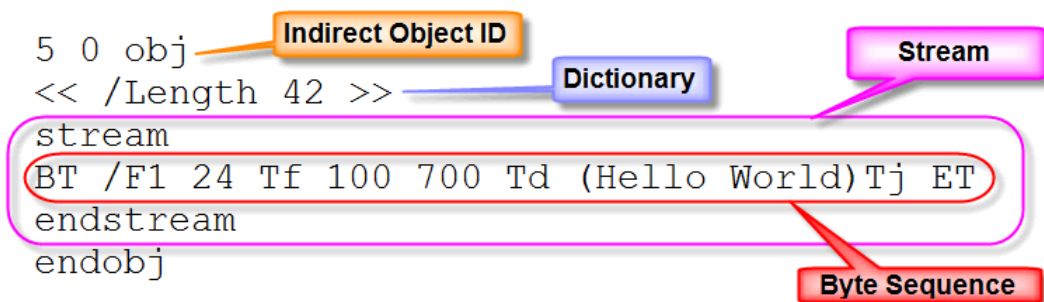


Figure 10 – PDF Stream Object

This stream is indirect object 5 version 0. The dictionary contains a `/Length` entry. It represents the length of (encoded) byte sequence. The `stream` and `endstream` keywords are terminated with EOL character(s) [19]. In this example we can see the string Hello World with special instructions for the PDF reader as given font and precise position. The length of this stream is precisely 42 bytes.

The sequence of bytes is usually encoded. This is done with a stream filter. A stream filter specifies what kind of algorithm has to be used for decoding. Let's have a look at an example in Figure 11.

The diagram shows a PDF stream object with the following text:


```
5 0 obj
<<
  /Length 55
  /Filter /ASCII85Decode
>>
stream
6<#\7PQ#@1a#b0+>GQ(+(u.+B2ko-qIocCi:FtDfTZ).9(%)78s~>
endstream
endobj
```

 Annotations include:

- A blue box labeled "Filter" pointing to the dictionary entry `/Filter /ASCII85Decode`.

 A blue oval highlights the dictionary entry `/Filter /ASCII85Decode`.

Figure 11 – PDF Stream Filter

As we can see for this example is used ASCII85 encoding. The `/Filter` entry specifies how to decode the byte sequence (`/ASCII85Decode`). There are many encoding schemes (ASCII filters and decompression filters), here is a list [19]:

- ASCIIHexDecode
- ASCII85Decode
- LZWDecode

- FlateDecode
- RunLengthDecode
- CCITTFaxDecode
- JBIG2Decode
- DCTDecode
- Crypt

The list is not so long, but many filters, like /FlateDecode take parameters as the compression level that may influence the encoding. Moreover, dictionary can contain more than one filter; it means that the stream must be decoded more than one filter. Here is another example *Figure 12*, where the stream is encoded twice. First is used ASCII85 and then plain HEX:

```
5 0 obj
<<
  /Length 168
  /Filter [ /ASCIIHexDecode /ASCII85Decode ]
>>
stream
36 3C 23 27 5C 37 50 51 23 40 31 61 23 62 30 2B
3E 47 51 28 2B 3F 28 75 2E 2B 42 32 6B 6F 2D 71
49 6F 63 43 69 3A 46 74 44 66 54 5A 29 2E 39 28
25 29 37 38 73 7E 3E>
endstream
endobj
```

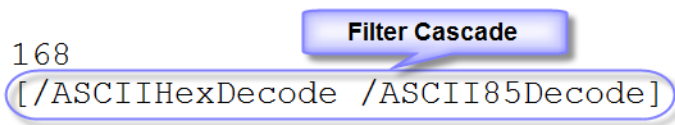


Figure 12 – PDF Filter cascaded

The Stream object is one of the most important parts in this thesis. This object almost always contains textual data. It is really important to know how the stream is encoded if we want to export the textual data from the PDF document.

2.3 An Example of PDF Raw File

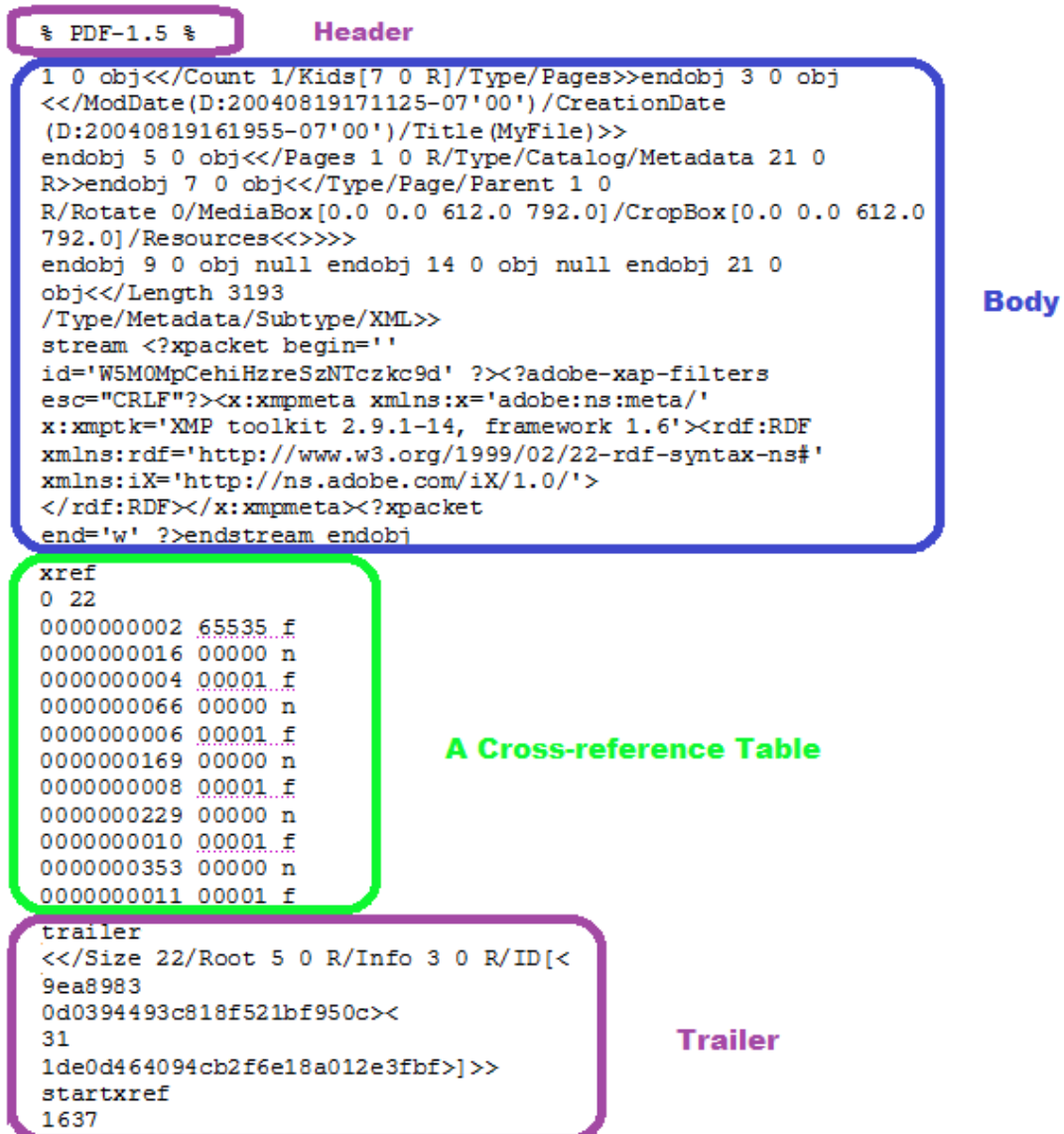


Figure 13 - An example of PDF Raw File

This file has been shortened for display purposes by removing a large piece of the XMP metadata stream. Here we can see several types of objects which are described in section 2.2 Structure of PDF File.

That's enough for our purposes in this thesis. It is not necessary more in-depth analysis of PDF structure, because we use the raw data for automatic files recognition.

2.4 Summary Information of PDF

From forensics aspect is a PDF file very interesting because contains very useful information for investigator. This information can in some way confirm or demolish the hypothesis formulated by the investigator, for instance the date creation or the date modification can help as well. Most important thing in the PDF section is this, when data are modified or deleted in the document by using any PDF editor (Foxit PDF editor etc), data are still present in the document. New information or updates are 'simply' written at the end of the file together with another object (new objects containing textual data, xref table, trailer and so on). For instance, page can be deleted by using PDF editor, but is still present within a file, browser just not read it. Big advantage for forensics expert of this file format is that it can be analyzed even when file is damaged or fragmented to very small pieces.

3 OFFICE XML-BASED FILES

More than twenty years, programs for storing documents such as Microsoft Word have stored all information in binary file format. Developers these tools migrated to new XML-based¹ formats for document files a few years ago. XML is a markup language that is particularly designed to exchanging data between applications. At present, the most widely office documents file formats which use XML are Office Open XML and OpenDocument Format. Both of these file formats have a similar structure which is described in next sections for each single file type [20][21].

3.1 OpenDocument Format

OpenDocument Format (hereinafter referred to as ODF) is open format for office documents created by Sun Microsystems, but has been standardized by the association OASIS². ODF is XML-based format where binary data (mostly images) are not coded as XML but are stored as single files. These files and XML files are packed into a single ZIP archive which represents file in format ODF.

3.1.1 File Structure

ODF structure is tightly defined. For instance, contains single XML files for document content, metadata, and style information or application settings. On the embedded (binary) objects refer to a relative path within the archive; the data are this way bounded together within the ZIP file. Because all the text is compressed, that is impossible to find it by scanning for strings within raw disk or document images. Document stored in this format can be extracted and read with arbitrary programming language with support of ZIP and XML.

As we can see at Figure 14 the basic structure can be opened for example by WinRAR. We can see several XML and binary files.

¹ Extensible Markup Language, its specification is available on <http://www.w3.org/XML>

² Organization for the Advancement of Structured Information Standards

Name ^	Size	Packed	Type
..			File folder
Configurations2			File folder
META-INF			File folder
Pictures			File folder
Thumbnails			File folder
content.xml	5 279	1 320	XML Document
manifest.rdf	532	244	RDF File
meta.xml	998	998	XML Document
mimetype	39	39	File
settings.xml	8 385	1 309	XML Document
styles.xml	11 160	2 066	XML Document

Figure 14 - ODF structure

The Figure 14 shows ODF structure. Description of each part is divided by folders and single files obtained after decompression of the document:

- **META-INF**

This folder contains file with name `manifest.xml`. It is very important file which contains a list of all files in packages, their media type, path and required information for decryption.

- **Thumbnails**

Always contains thumbnail image of the document's first page. The file is stored as `thumbnail.png`. For forensics expert is important that if the thumbnail does not match the document, then someone could modified the thumbnail or the document after the creation [21].

- **Pictures**

Folder where are stored images.

- **content.xml**

The `content.xml` contains the content of the document. It means that within the file is stored textual data.

In the Figure 15 we can see a document that contains only three words of the text by default formatting.

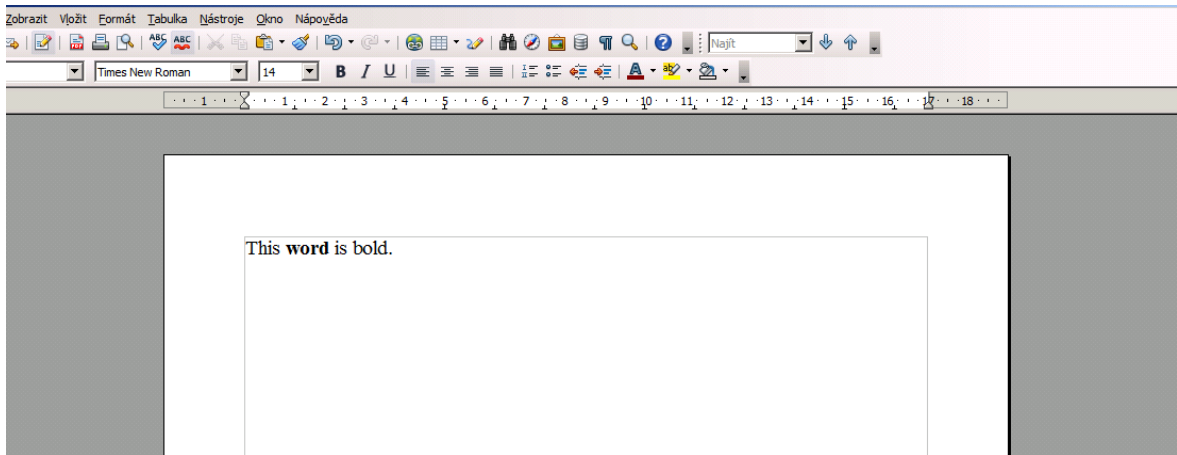


Figure 15 - OpenOffice.org Writer example

Demonstration of content.xml from test file ThisWordIsBold.odt, the file has been shortened. Within the file is written just text 'This **word** is bold'.

```
<?xml version="1.0" encoding="UTF-8"?>
<office:document-content
xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
xmlns:ooow="http://openoffice.org/2004/writer"
xmlns:rpt="http://openoffice.org/2005/report"
xmlns:of="urn:oasis:names:tc:opendocument:xmlns:of:1.2"
interop:xmlns:field:1.0" office:version="1.2"
grddl:transformation="http://docs.oasis-open.org/office/1.2/xslt/odf2rdf.xsl"><office:scripts/><office:font-face-decls><style:style style:name="T1"
style:family="text"><style:text-properties fo:font-weight="bold" style:font-weight-asian="bold" style:font-weight-complex="bold"/></style:style></office:automatic-styles><office:body><office:text><text:sequence-decls><text:sequence-decl text:display-outline-level="0"
text:name="Illustration"/><text:sequence-decl text:display-outline-level="0" text:name="Table"/><text:sequence-decl
text:display-outline-level="0"
text:name="Text"/><text:sequence-decl text:display-outline-level="0" text:name="Drawing"/></text:sequence-decls><text:p
text:style-name="P1">This <text:span text:style-name="T1">word</text:span> is bold.</text:p></office:text>
</office:body></office:document-content>
```

As we can see from the example, the textual data of the document are located between the pair of elements `<text:p...>` and `</text:p>`. The start elements can contain other properties (e.g. font, bold).

- **meta.xml**

This file contains metadata for the entire document. The metadata contained within the file: which version of ODF is used by the document, creation date/time, modification date/time, document title, document subject, keywords, initial creator, printed by, hyperlink behavior, language etc.

- **styles.xml**

File `styles.xml` contains information about visual styles of the document.

- **settings.xml**

This file stores information about document configuration (e.g. window size, printing settings, etc.)

3.2 Microsoft's Office Open XML

As discussed above, Microsoft has stored documents in binary file formats for more than twenty years. Microsoft introduced its own XML-based document file format at 2007. Like ODF, Office Open XML (hereinafter referred to as OOXML) is also an international standard for storing documents (since 2008).

3.2.1 File Structure

OOXML contains XML and binary files (images, graph, other documents, etc) packed into a single ZIP archive. Hence, OOXML is smaller than previous binary version of Microsoft Office (.doc). Because all the text is compressed, that is impossible to find it by scanning for strings within raw disk or document images.

..			File folder
_rels			File folder
docProps			File folder
word			File folder
[Content_Types].xml	1 445	370	XML Document

Figure 16 - OOXML structure

In the Figure 16 we can see OOXML structure. Description of each part is divined by folders obtained after decompression of the document:

- **rels**

Contain file with name `.rels`, that stores information about relationships between items. File `.rels` contains important information for Office 2007 to finding all entries of document.

- **docProps**

This folder contains two files which stores metadata of document. For instance:

- number of pages, paragraphs, words and characters in the document
- name and version of application that created the document
- keywords and description of the document
- timestamps indicating the creation and last modification of file

This folder may contain thumbnail image of the document's first page. Microsoft stores the thumbnail as a `.jpeg`. OOXML does not save thumbnails by default, it must be checked on the advanced options.

- **word**

Folder `word` can contain own folders `'_rels'` or `media`. In `'_rels'` is file `document.xml.rels` which describe relationships for main entry of document `document.xml`. In `'media'` are stored binary files (images, graph, other documents, etc). Also here can be folder `'theme'`, which contains XML file describing the theme of the document (e.g. colors, fonts, formats). The main content of folder `'word'` is composed of files describing content of the document and other files (which define fonts, styles, header, footer etc.).

The file `/word/document.xml` contains a structure composed mostly minimal of the root element `<document>` and under element `<body>`.

```
<w:document ...>
  <w:body>
  ...
  </w:body>
</w:document>
```

In the Figure 17 we can see a document that contains only three words of the text by default formatting.

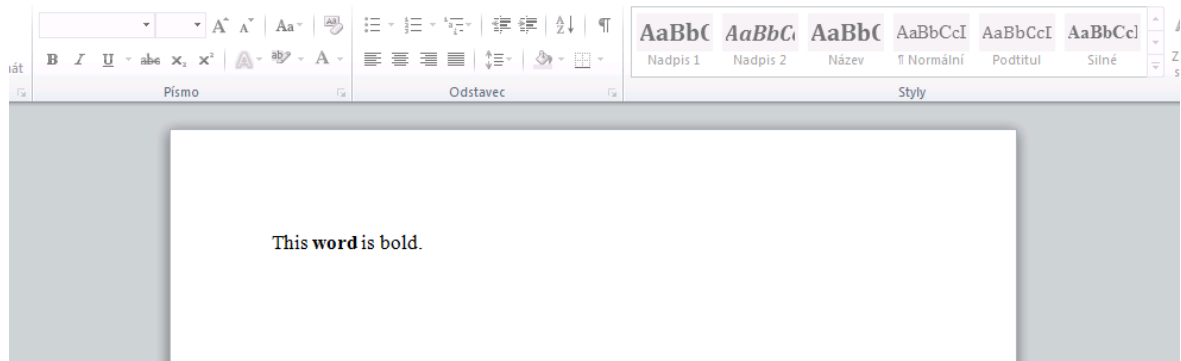


Figure 17 - MS Word file example

Demonstration of document.xml from test file ThisWordIsBold.docx:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<w:document
xmlns:ve="http://schemas.openxmlformats.org/markup-
compatibility/2006"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns:r="http://schemas.openxmlformats.org/officeDocument
/2006/relationships"
xmlns:v="urn:schemas-microsoft-com:vml"
xmlns:w10="urn:schemas-microsoft-com:office:word"
xmlns:w="http://schemas.openxmlformats.org/wordprocessing
ml/2006/main"
<w:body>
<w:pw:rsidR="00F25A57" w:rsidRDefault="00F25A57">
<w:r><w:t xml:space="preserve">This </w:t></w:r>
<w:rw:rsidRPr="00FE579B">
<w:rPr><w:b/><w:bCs/></w:rPr>
<w:t>word</w:t>
</w:r>
<w:r><w:t xml:space="preserve"> is bold.</w:t></w:r>
</w:p>
<w:sectPr w:rsidR="00F25A57" w:rsidSect="00F25A57">
<w:pgSz w:w="12240" w:h="15840"/>
<w:pgMar w:top="1440" w:right="1800" w:bottom="1440"
w:left="1800" w:header="720" w:footer="720" w:gutter="0"/>
<w:cols w:space="720"/>
<w:docGrid w:linePitch="360"/>
</w:sectPr>
</w:body>
</w:document>
```

As we can see from the example, the textual data of the document are located between the pair markers <w:t>, which are under elements of markers <w:r>, which are another

under elements of markers `<w:p>`. Element `<w:p>` defines each paragraphs of the document [22].

- **[Content-types].xml**

File `[Content_Types].xml` contains two types of definitions for content of ZIP archive. One for the file types included in the ZIP archive and the other for parts of XML.

3.2.2 Embedded Objects

Forensics tools are able to extract textual data from the content parts of document. But it is necessary to know that text can be found in other document parts as well. Microsoft Word allows embedded another Word document into by using “Insert/Object...” (menu command) [21]. When another Word document is embedded within other files, ZIP archive will look similar at Figure 18.

Length	Name
-----	----
1527	[Content_Types].xml
735	_rels/.rels
1107	word/_rels/document.xml.rels
4780	word/document.xml
6613	word/media/image1.png
7559	word/theme/theme1.xml
39832	docProps/thumbnail.jpeg
25316	word/embeddings/Microsoft_Word_Document1.docx
2036	word/settings.xml
276	word/webSettings.xml
734	docProps/app.xml
726	docProps/core.xml
15019	word/styles.xml
1521	word/fontTable.xml
-----	-----
107781	14 files

Figure 18 - ZIP archive directory with embedded a Microsoft Word document

3.3 Summary Information of Office XML-Based Files

Both of these file types (described below) are packed into a single ZIP archive. They are two different file types, but they have similar features. Both of them use XML files for storing information. Disadvantage for native browsers is certainly storing all information within one ZIP archive because when file is corrupted and missing any important file, are not able to open the document. Big advantage for forensics expert is that images and textual data are stored separately in different files/directories. Files which contain textual data (ODF – content.xml, OOXML – document.xml) can be opened even when other files are corrupted or missing, for example by XML parser. For forensics expert is important to know that all text is compressed, so that is impossible to find it by scanning for strings within the raw data. Nevertheless, the file structure can be recognized from raw data by using occurrence of features (as features of PDF file are not compressed). Document must be unzipped for finding information (textual data, images, etc.). Moreover, also these file types contain large amounts of metadata which can help to investigator as well.

II. ANALYSIS

4 SPECIFICATION OF THE APPLICATION

4.1 Motivation

The idea of creating an application is based on my own experience with lost data in hard disk. It is not so long ago when my hard disk was corrupted and I lost almost all data. I was not able to recover the most important files for me and I lost a lot of time and nerves with their re-creation. The main reason why I was not able to recover most lost files was that the file system was damaged. Therefore, common applications for file recovery did not work. At that time I did not know the techniques such as File Carving which works without the support of the information provided by the file system. This issue was introduced to me by Dr. Davide Ariu from the University of Cagliari where I stayed for internship last year within Erasmus programme. With Dr. Ariu we focused on a recognition of files based on their content when gave an idea to create application which is able to automatic recognize of text-data blocks without information provided by the file system.

From this idea, it was necessary to move on to considering whether such an application could be useful for more users. Also it was necessary to find out if any similar application already exists and find its functions and drawbacks. Implementation of these actions together with the main requirements for a new application is described in the following chapter.

4.2 Requirements Specification

There are many applications dedicated to data recovery and forensics analysis. However, I did not find any application which would be able to automatically recognize a file type of given block of data when header and/or footer are missing.

These findings indicate that probably not exist planned application that contains the required function. From this perspective, it would certainly make a sense to create the application. The application could be useful in Computer Forensics as well.

4.2.1 The Required Functions

- decide if given block of data is textual file or not (Adobe PDF, XML Office files)
- the tool would be also able to extract text from data, if present

- the application must be able to run under the Linux operating system

4.3 How Works the Application

The principle of algorithm is based on searching characteristic features specifying the file type. Thus, in given block of data are searched (ei. using string search) gradually occurrences of all objects and features that define the structure of file type. These features are not encoded or compressed so they can be found. Each occurrence is counted and recorded. Based on this information we are able to decide if given block of data belongs to the testing file structure or not. When file is automatically recognized, textual data are exported.

5 IMPLEMENTATION OF THE APPLICATION

For application programming I decided to use Java because it is multiplatform language and can be used on any operation system. The actual application programming certainly is not possible neither necessary to describe in detail, but worth it to describe the main parts of the program.

5.1 Algorithms of Recognition

5.1.1 PDF Algorithm

From the theory part we know that PDF consists of objects and their features. So, in this case are searched for these objects defining header, body, x-ref table, trailer and other features of these objects (e.g. stream, endstream, /Filter etc). The objects are not encoded and they can be easily found by string search. All searched objects we can see in Table 1:

Table 1 – Searched PDF objects

obj	endobj
stream	endstream
xref	startxref
trailer	/Page
/Encrypt	ObjStm
/JS	/JavaScript
/AA	/OpenAction
/JBIG2Decode	/RichMedia
/Lunch	/Filter
/FlateDecode	/Length

Frequency of occurrences is recorded into the table and then stored to the hard disk to another analysis. The most important features of objects are 'stream' and 'endstream', because each PDF file containing any data must have these tags as well.

A decision if given file belongs to the PDF or not depends right on these tags. Both of these tags must be equal in frequency of occurrences. When it is not like that it is not the PDF. In the opposite case it is the PDF. When a file is fragmented or damaged, occurrences of 'stream' and 'endstream' may not be equal but if some of them is greater only by one number it means that the block of data probably belongs to the PDF structure. The Figure 19 shows the method which makes the decision.

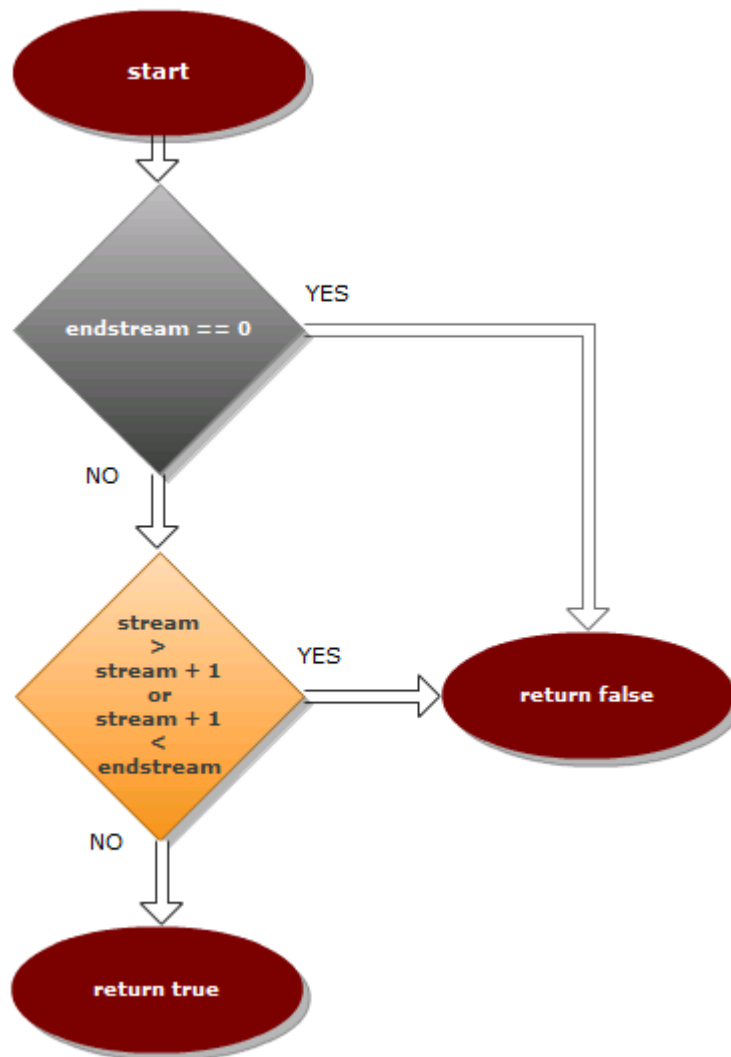


Figure 19 - Flowchart of the decision of the PDF

When the PDF file is damaged or fragmented, there may be three different cases as we can see in Figure 20.

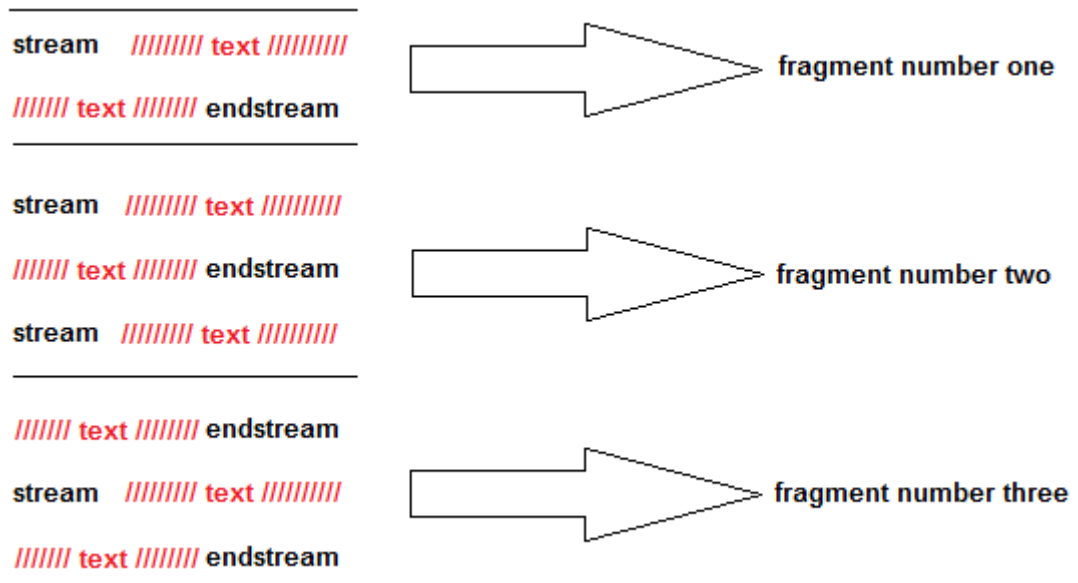


Figure 20 - Three different cases of PDF file fragments

When given block of data is automatically recognized as the PDF, data are automatically exported. Positions of tags 'stream' and 'endstream' and also '/Filter' are stored into a two collections. The '/Filter' represents the coding algorithms which were explained in section 2.2.7. The positions of the algorithms are added into the collections. Then collections are sorted and the text between their items is selected and stored into a new .txt file.

The collections must be equal. If it is not like that, file is fragmented. The Figure 21 and the Figure 22 show the algorithm of treatment in case that the file starts or continues on another cluster.

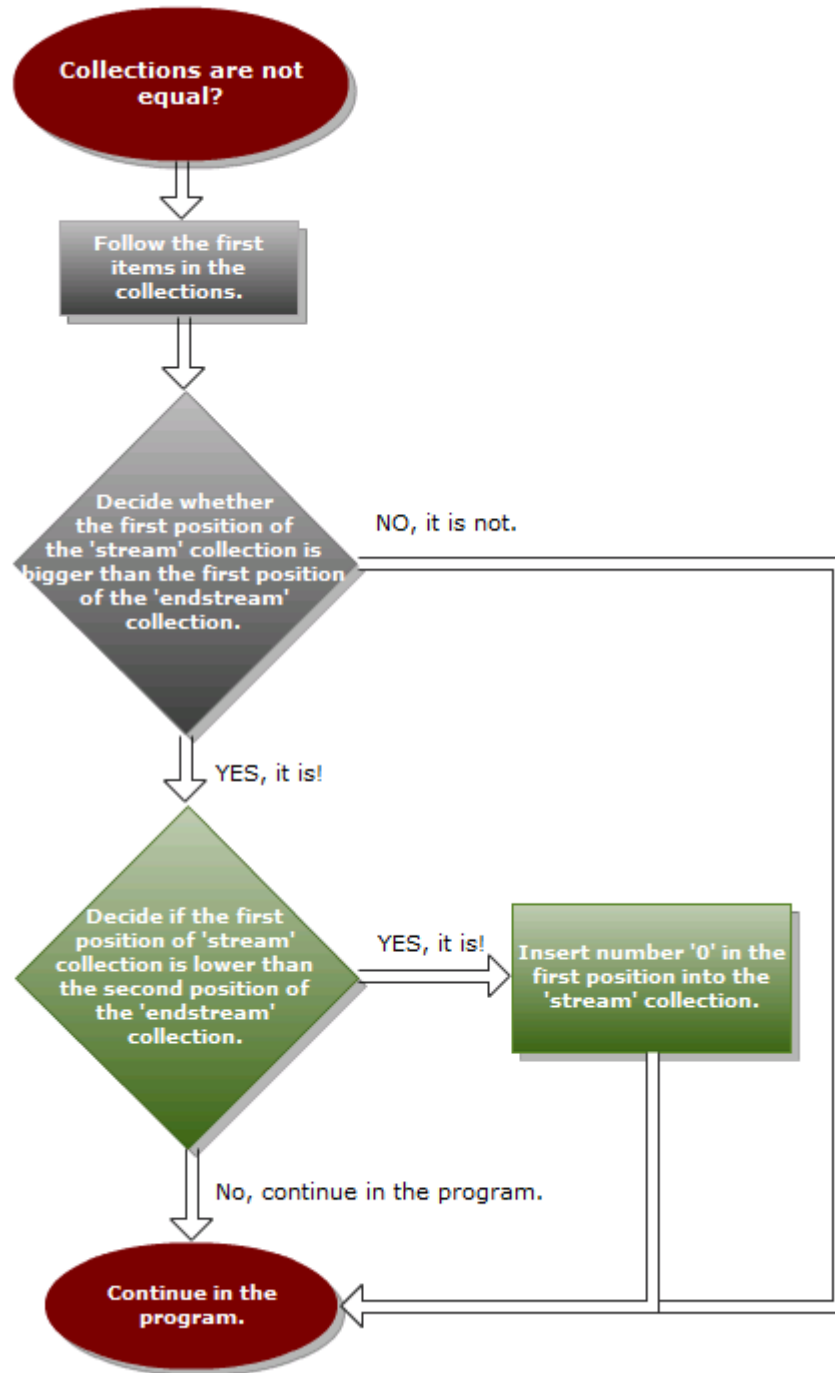


Figure 21 – Treatment if the file starts on another cluster

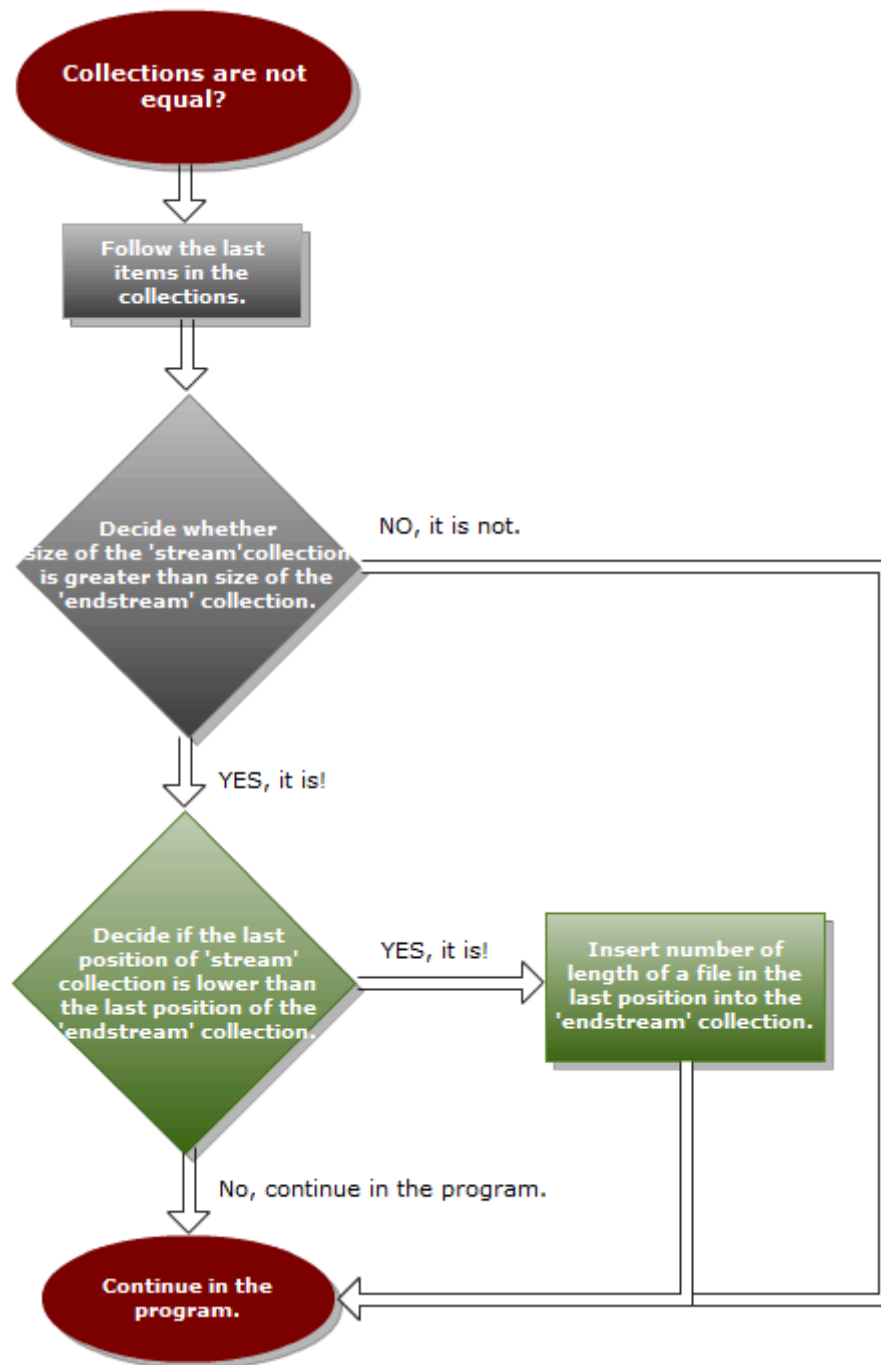


Figure 22 – Treatment if the file continues on another cluster

5.1.2 Algorithm of Office XML-based Files

In the theory was mentioned that both of file types OOXML and ODF are packed into a single ZIP archive. It means XML tags containing some information cannot be found in given blocks of data. However, as we can see in *Figure 23*, names of XML files occur in a raw data.

PK.....!đ!ě}Ž..... [Content_Types].xml~.....N.....
 . _rels/.rels ~..Ś'ÚJ.A.tđ.ĚaČ}7Ú."ŇŮŤH~w"ě.,™ě.w.ĚαÚI~Ł
 řPŮ^ćôçĚÖÖ>. >Ô;Ś<.ŽaYŎ Ř>`Gßkxm·<.PYČ[š,g.GÎ°inoÖ/<`~<Ś1«~âł
 A\$>"f3°Ł\...ČITř..I.SŽ`Ě.őŚ«şIÇôW.š™|ÚY.igd@µÇX6_Ö.]7.~.fdŘĚ%.Č
 .aoŮ.b*1IÍCrřj)ó,.10Ď%t`b~.6ři~ŎóD`_<ž...,..%.Ďó|us.Z^.tŮ~yÇŽ;.!
 Y,.){úC.ł/h>...`..PK.....!..v.``Ž...I.....word/_rels/d
ocument.xml.reln•ÉNĂ0.@d'HüCä;q[,j..@â;...ę.....word/docume
nt.xmlěZÍn.7.I.č; ,öni..š»..súž.Ú@°b.č% v)%. \$HJ.úin,é<u+ú#.dŮ

Figure 23 - An example of OOXML raw data

All searched features of OOXML are presented in Table 2 a Table 3:

Table 2 – Searched occurrences within OOXML

[Content_Types].xml	word/_rels/document.xml.rels
_rels/.rels	word/document.xml
word/media	word/theme
docProps	word/embeddings
word/settings.xml	word/webSettings.xml
word/styles.xml	word/fontTable.xml

Table 3 – Searched occurrences within ODF

manifest.xml	thumbnail.png
content.xml	layout-cache
manifest.rdf	meta.xml
mimetype	settings.xml
styles	

Principle of the application is same for OOXML and ODF. Frequency of occurrences is recorded into a table and then stored to the hard disk to another analysis. For being able to decide whether the given block of data belongs to OOXML or ODF structure is necessary to find occurrences of strings in the format word/document.xml for OOXML and content.xml for ODF. If they are found, file is unzipped and from the file containing textual data (e.i. document.xml or content.xml) is extracted text between markers '<' and '>' by the same way as PDF. Then data are extracted and stored into a new .txt file.

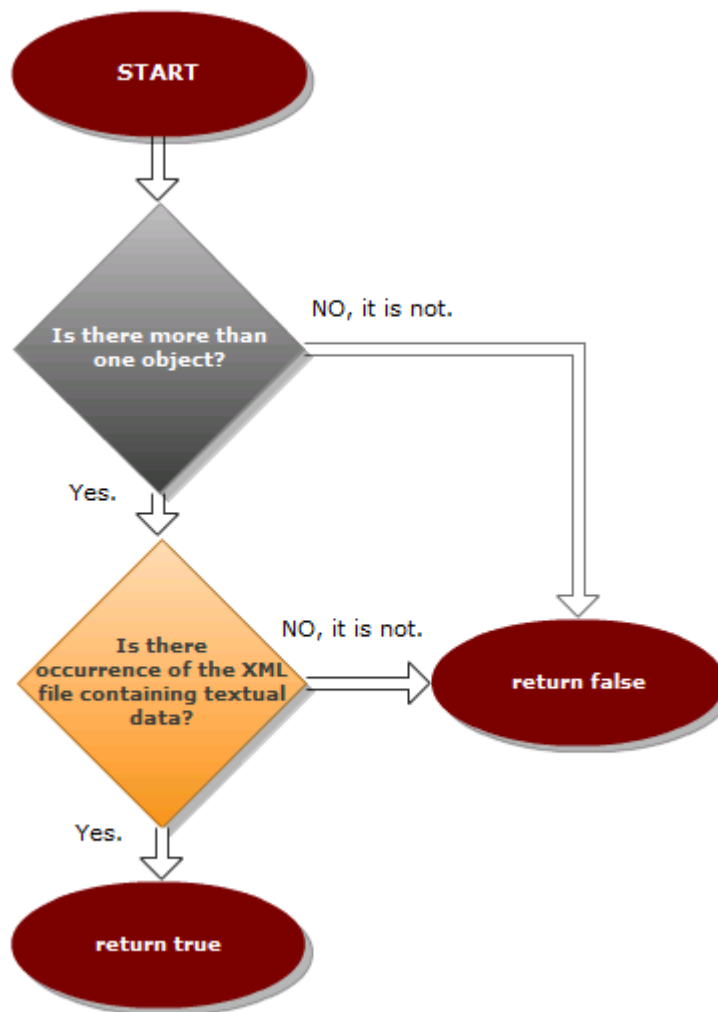


Figure 24 - Flowchart of the decision of the Office XML-based file

Office XML-based files suffer from one great drawback. When a file is corrupted or is stored over several unknown clusters, it may happen that the file containing textual data is not found. Assume that we have a .docx or .odt file contain 4MB of text (without

images, graph etc) stored in several clusters. It will be impossible to unzip the file containing textual data (document.xml or content.xml) from any cluster separately. Hence, when an Office XML-based file is fragmented and contains a lot of textual data, it is not possible to export the text.

5.2 The Control Logic within the Main Method

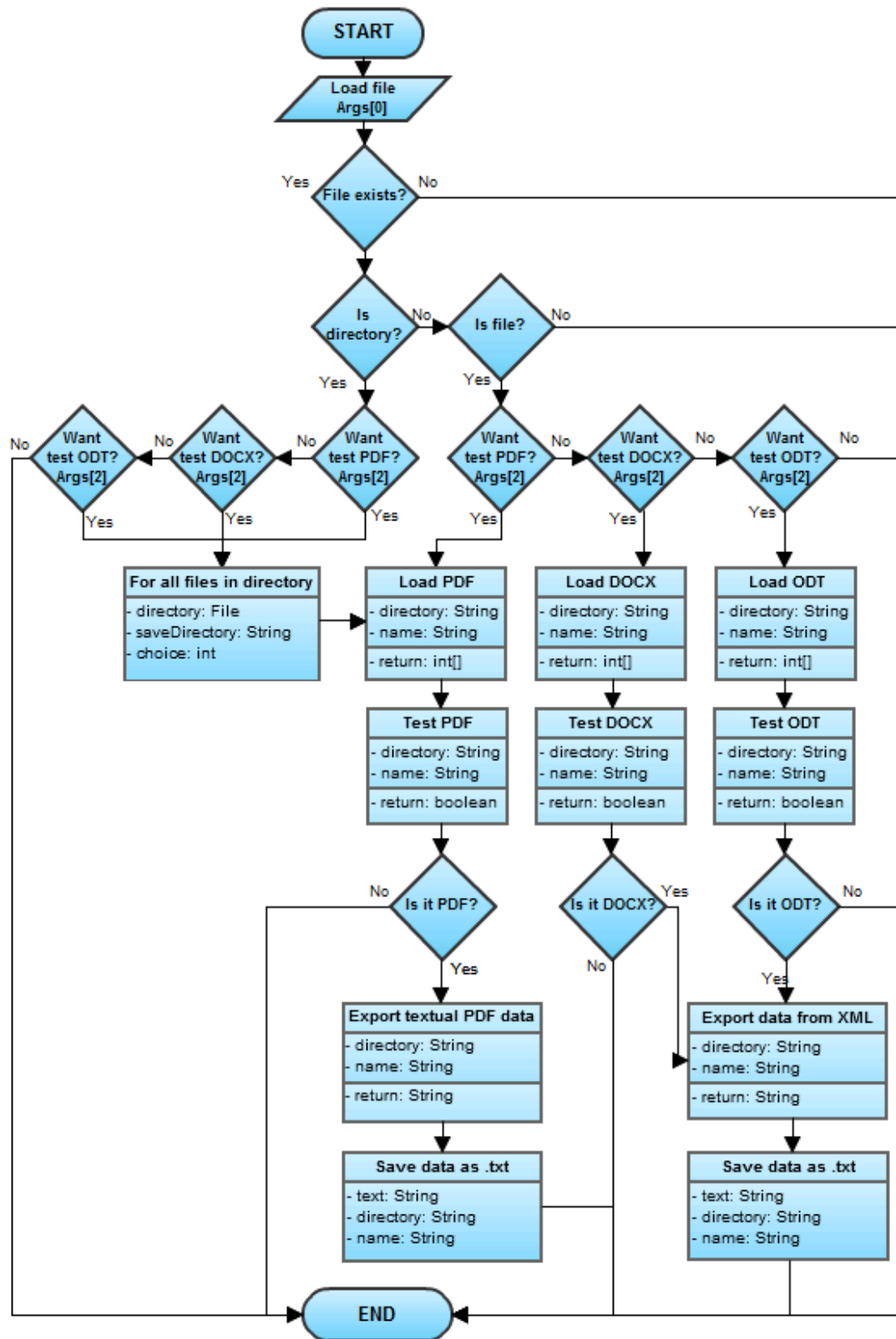


Figure 25 – Diagram of the main method

5.3 Structure of the Application

The whole class is divided to several methods which are then called from the main method. Brief descriptions of each method are listed below. Code of all methods can be viewed in the source files on the enclosed CD.

5.3.1 PDF Methods

Table 4 - List of PDF methods exported from Javadoc

PDF Methods	
static boolean	testSinglePDF (java.lang.String directory, java.lang.String name) Returns "false" in case that given file does not contain the PDF structure, returns "true" when the file was automatically recognized as PDF.
static int[]	loadPDF (java.lang.String directory, java.lang.String file) Returns an array of occurrences of each PDF object.
static boolean	testPDF (int[] loadedPDF, java.lang.String name) Returns "false" in case that given file does not contain the PDF structure, returns "true" when the file was automatically recognized as PDF.
static String	exportPDF (java.lang.String directory, java.lang.String name) When the PDF structure is recognized, returns string with exported textual data of a PDF file.

- **testSinglePDF**

```
public static boolean testSinglePDF(String directory, String name)
```

Briefly, this method tests if a given single file does contain the PDF structure or not. The file is opened and read by the class `InputStream()`. Then, there is used a buffer to store data in a variable. In next step there is a test if the given file contains the PDF objects or not.

If the file contains required PDF objects, the file is automatically recognized and the method returns true.

Parameters:

`directory` - an absolute path of directory

`name` - the name of the file within directory

Returns:

true or false

- **loadPDF**

```
public static int[] loadPDF(String directory, String file)
```

The given file is opened and read. The next step is testing and storing occurrences of PDF objects into the array.

The method returns an array of occurrences of each PDF object. Objects are in order: Name, object, obj, endobj, stream, endstream, xref, startx, trailer, page, encrypt, objStm, JS, javaS, AA, openA, acroF, jbig2, richM, launch, filter, flateD, Length, Number of chars.

Parameters:

`directory` - an absolute path of directory

`file` - the name of the file within directory

Returns:

An array of objects

- **testPDF**

```
public static boolean testPDF(int[] loadedPDF, String name)
```

Briefly, this method tests if the given single file does contain the PDF structure or not. The file is opened and read by the class `InputStream()`. Then, there is used buffer to store data in a variable. In next step is a test if given file contains the PDF objects or not. If the file contains required PDF objects, the file is automatically recognized and the method returns true.

Returns "false" in case that given array does not contain the PDF object "endstream" or if the value of object "stream" is much higher than "endstream". Otherwise, returns "true".

Parameters:

loadedPDF – an array containing occurrences of objects

name – a name of the given file

Returns:

true or false

- **exportPDF**

```
public static String exportPDF(String directory, String name)
```

Briefly, the method returns a string with exported textual data of a PDF file. The file is opened and read by the class `InputStream()`. Then, there is used buffer to store data in a variable (`String`). Then, textual data are selected between tags "stream" and "endstream".

The textual data within the string are encoded and may contain images. In most cases it is encoded by one of these algorithms: `FlateDecode`, `ASCIIHexDecode`, `JBIG2Decode`, etc. When, algorithm is found between tags `"/Filter"` and "stream", name of algorithm is automatically added right before exported textual data found between "stream" and "endstream". Thus, in return string you can find tags which represent determine algorithm for text decoding.

Parameters:

directory - an absolute path of directory

name - the name of the file within directory

Returns:

the string of exported textual data from the PDF file

5.3.2 OOXML Methods

Table 5 - List of OOXML methods exported from Javadoc

OOXML Methods	
static boolean	testSingleDOCX (java.lang.String directory, java.lang.String name) Returns "false" in case that the given file does not contain the DOCX structure, returns "true" when the file was automatically recognized as DOCX.
static int[]	loadDOCX (java.lang.String directory, java.lang.String file) Returns an array of occurrences of features that belong to DOCX file.
static boolean	testDOCX (int[] loadedDOCX, java.lang.String name) Returns "false" in case that the given file does not contain the DOCX structure, returns "true" when the file was automatically recognized as DOCX file.
static String	exportXML (java.lang.String directory, java.lang.String file) Returns string with exported textual data of the DOCX file.

- **testSingleDOCX**

```
public static boolean testSingleDOCX(String directory, String name)
```

Briefly, this method tests if given single file does contain the DOCX structure or not. The file is opened and read by the class `InputStream()`. Then, there is used buffer to store data in a variable. In next step is a test if given file contains specific occurrences of DOCX or not. If the file contains `document.xml`, the file is automatically recognized and method returns true.

Parameters:

`directory` - an absolute path of directory

`name` - the name of the file within directory

Returns:

true or false

- **loadDOCX**

```
public static int[] loadDOCX(String directory, String file)
```

Returns an array of occurrences of features that belong to DOCX file. The given file is opened and read. The next step is testing and storing occurrences of DOCX features into the array. Features are in order: Name, object, [Content_Types].xml, _rels/.rels, word/_rels/document.xml.rels, word/document.xml, word/media, word/theme, docProps, word/embeddings, word/settings.xml, word/webSettings.xml, word/styles.xml, word/fontTable.xml

Parameters:

`directory` - an absolute path of directory

`file` - the name of the file within directory

Returns:

array of objects

- **testDOCX**

```
public static boolean testDOCX(int[] loadedDOCX, String name)
```

Returns "false" in case that the given file does not contain the DOCX structure, returns "true" when the file was automatically recognized as DOCX file. Basically, if the file contains word/document.xml, the file is automatically recognized;

Parameters:

`loadedDOCX` – an array containing occurrences of objects

`name` - the name of given file

Returns:

true or false

- **exportXML**

```
public static String exportXML(String directory, String file)
```

This method works for both XML-based office files (DOCX, ODT). Returns a string with exported textual data of a DOCX or ODT file. The file is automatically unzipped and text exported when the ZIP packet contains word/document.xml (DOCX) or content.xml (ODT). Then, the text is exported between markers ">" and "<".

If the ZIP packet is corrupted, it is not able to export textual data.

Parameters:

`directory` - an absolute path of given directory

`file` - the name of the file within directory

Returns:

the string of exported textual data from the XML-based office file

5.3.3 ODF Methods

Table 6 - List of ODF methods exported from Javadoc

ODF Methods	
static boolean	testSingleODT (java.lang.String directory, java.lang.String name) Returns "false" in case that the given file does not contain the ODT structure, returns "true" when the file was automatically recognized as ODT.
static int[]	loadODT (java.lang.String directory, java.lang.String name) Returns an array of occurrences of features that belong to ODT file.
static boolean	testODT (int[] loadedODT, java.lang.String name) Returns "false" in case that the given file does not contain the ODT structure, returns "true" when the file was automatically recognized as ODT file.
static String	exportXML (java.lang.String directory, java.lang.String file) Returns a string with exported textual data of the ODT file.

- **testSingleODT**

```
public static boolean testSingleODT(String directory, String name)
```

Briefly, this method tests if the given single file does contain the ODT features or not. The file is opened and read by the class `InputStream()`. Then, there is used buffer to store data in a variable. In next step there is a test if given file contains specific occurrences of ODT or not. If the file contains `content.xml`, the file is automatically recognized and the method returns true.

Parameters:

`directory` - an absolute path of directory

`name` - the name of the file within directory

Returns:

true or false

- **loadODT**

```
public static int[] loadODT(String directory, String name)
```

Returns an array of occurrences of features that belong to ODT file. Features are in order: Name, objects, `manifest.xml`, `thumbnail.png`, `content.xml`, `layout-cache`, `manifest.rdf`, `meta.xml`, `mimetype`, `settings.xml`, `styles`.

Parameters:

`directory` - an absolute path of directory

`name` - the name of the file within directory

Returns:

an array of objects

- **testODT**

```
public static boolean testODT(int[] loadedODT, String name)
```

Returns "false" in the case that the given file does not contain the ODT structure, returns "true" when the file was automatically recognized as ODT file.

Parameters:

loadedODT – an array containing occurrences of objects

name - the name of given file

Returns:

true or false

- **exportXML**

```
public static String exportXML(String directory, String file)
```

This method works for both XML-based office files (DOCX, ODT). Returns a string with exported textual data of a DOCX or ODT file. The file is automatically unzipped and the text exported when the ZIP packet contains word/document.xml (DOCX) or content.xml (ODT). Then, the text is exported between markers “>” and “<”.

If the ZIP packet is corrupted, it is not able to export textual data.

Parameters:

directory - an absolute path of given directory

file - the name of the file within directory

Returns:

the string of exported textual data from the XML-based office file

5.3.4 General Methods

Table 7 - List of general methods exported from Javadoc

General Methods	
static void	inDirectory (java.io.File directory, java.lang.String saveDirectory, int choice) Saves and prints occurrences of features of all files in one table.
static void	createFile (java.lang.String filename) Creates file.file.
static void	saveFile (java.lang.String text, java.lang.String directory, java.lang.String name) Saves given textual data to .txt file with the same name.
static void	main (java.lang.String[] args) In this main method is programmed logic of the whole application.

- **inDirectory**

```
public static void inDirectory(File directory, String saveDirectory,
                               int choice)
```

Probably, the most important method in the class. The method includes several main steps. First of all, a content of the directory is added in 'array'. It means that this array contains names of all files in the directory. After this is created 'hash table' (key - String, value - int[]). In another step is created cycle 'for' where is called another method and with 'array' write the values into the hash table. Then the hash table is printed and saved like a new .txt file.

Parameters:

directory - an absolute path of the given directory

saveDirectory - an absolute path to save the table containing occurrences of features of files

choice - represents which file structure should be tested: 1. PDF, 2. DOCX, 3. ODT

- **createFile**

```
public static void createFile(String fileName)
```

Creates file.

Parameters:

fileName - name of a created file

- **saveFile**

```
public static void saveFile(String text, String directory, String name)
```

Saves the given string to the .txt file.

Parameters:

text - the given textual data

directory - an absolute path to save file

name - name of a saved file

- **main**

```
public static void main(String[] args)
```

In this main method is programmed logic of the whole application. Selected methods are called by the given arguments. First of all, there is necessary to find if the given path exists and if it is a file or a directory. After this is clear that we can get three different cases:

1. path does not exist - the application is ended
2. path is directory - test of all files in directory to the specified file structure
3. path is file - test a single file to the specified file structure

When the file is recognized, text is exported and automatically saved by its name.txt.

For run the application is required insert all arguments:

args[0] = an absolute path of given file/directory

args[1] = an absolute path to the save files

args[2] = what kind of file type should be tested (PDF, DOCX, ODT)

Type of args[2] to analyze:

PDF	- testPDF
DOCX	- testDOCX
ODT	- testODT

Parameters:

args - [0] = given directory, [1] = save directory, [2] = file type

In the Table 10 we can see that files with the largest frequency of occurrences of features really belong to ODF and contains .odt file extension. OOXML files can contain several similar features with as `mimetype.xml` or `setting.xml`. The file 190.pdf is also ODF, but it has changed the file extension to .pdf. File 13.odt is corrupted because it does not contain `content.xml` which contains textual data. It is ODT file, but textual data cannot be exported. How works the algorithm is described in section 5.1.2 in detail.

ZÁVĚR

Hlavním cílem této práce byla automatická klasifikace souborů obsahující textová data bez podpory informací poskytnutých souborovým systémem. Důraz byl kladen na pochopení, jak pracují nástroje obnovy dat a Computer Forensics, dále na analyzování vnitřní struktury standardu Adobe PDF a standardů založených na XML (OOXML, ODF). Poté bylo úkolem s využitím těchto informací navrhnout a implementovat softwarový nástroj, který by byl schopen extrahovat z bloku dat charakteristiky, které by mohly být použity pro automatické klasifikace souborů dokumentů.

Myslím, že tyto cíle práce byly splněny. V teoretické části byly podrobně popsány jednotlivé části zaměřující se na vyhledávání řetězců, rozdíl mezi nástroji pro obnovu dat a Computer Forensics, dále související práce a představeny byly také jednotlivé struktury standardů PDF, OOXML a ODF.

Praktickým výsledkem práce je pak funkční aplikace, jež odpovídá vytvořenému zadání a návrhu. Aplikaci by bylo možno doplnit o algoritmy schopné dekodovat textová data exportovaná z PDF. Další vylepšení vidím v rozšíření typů souborů, ač textových nebo i jiných, např. obrázkový jpeg nebo gif.

Výstupem práce je i v praxi použitelná aplikace, která může být užitečná nejen v Computer Forensics při určování modifikovaného, poškozeného nebo neznámého souboru, ale také při obnově dat. Budu velmi rád, pokud tato práce v budoucnu usnadní práci nejen vědeckým pracovníkům.

CONCLUSION

The main goal of the thesis was automatic classification of files containing textual data without the support of the information provided by the file system. It focused on understanding how data recovery and forensics tools typically work, in addition to analyzing the internal structure of both Adobe PDF and XML-Based Office (OOXML and ODF) standards. Then, the main task was to use this information to design and implement software tool capable of extracting the features that can be used for the automatic classification of the document from a block of data.

I think that these goals of the thesis have been met. In theoretical part there were described in detail specific parts of the file focusing on search string, the concept of difference between file recovery and Computer Forensics, in addition related works of pattern recognition. There were also presented structures of PDF, OOXML and ODF standards.

The practical result of the thesis is the functional application that is corresponding with proposal and design. Of course there is still a room for improvement. The application could be supplemented with algorithms that are able to decode the text data exported from the PDF. I can see further improvement in the expansion of the file types (e.g. jpeg, gif).

The thesis could be used not only in Computer Forensics in determining modified, corrupted or unknown files, but also in data recovery. I will be very glad if the thesis will facilitate researcher's work in the future.

LIST OF REFERENCES

- [1] BEEBE, N. L. and CLARK, J. G. *Digital forensic text string searching: Improving retrieval effectiveness by thematically clustering search results*. [online]. 2007, 49 - 54 [cit. 2012-04-12]. Available on: <http://www.dfrws.org/2007/proceedings/p49-beebe.pdf>
- [2] FILE-EXTENSIONS.ORG. [online]. [cit. 2012-04-12]. Available on: <http://www.file-extensions.org/>
- [3] *DFRWS: Forensics Challenge 2006* [online]. [cit. 2012-04-12]. Available on: <http://www.dfrws.org/2006/challenge/index.shtml>
- [4] MEROLA, A.. *SANS Institute: Data Carving Concepts* [online]. 2008. [cit. 2012-04-12]. Available on: http://www.sans.org/reading_room/whitepapers/forensics/data-carving-concepts_32969
- [5] *Technology Flow: File Carving* [online]. [cit. 2012-04-12]. Available on: <http://technology-flow.com/articles/file-carving/>
- [6] BEEK, C. MCAFEE. *Introduction to File Carving* [online]. [cit. 2012-04-12]. Available on: <http://www.mcafee.com/us/resources/white-papers/foundstone/wp-intro-to-file-carving.pdf>
- [7] PHOTOREC [online]. [cit. 2012-04-12]. Available on: <http://www.sgsecurity.org/wiki/PhotoRec>
- [8] FOREMOST [online]. [cit. 2012-04-12]. Available on: <http://foremost.sourceforge.net/>
- [9] KLOET, B. *How much evidence are you ignoring?* [online]. 2010, [cit. 2012-04-12]. Available on: <http://computer-forensics.sans.org/summit-archives/2010/files/eu-digital-forensics-incident-response-summit-bas-kloet-advanced-file-carving.pdf>

- [10] MCDANIEL, M. and M. HEYDARI. JAMES MADISON UNIVERSITY. *Content Based File Type Deletion Algorithms* [online]. 2003 [cit. 2012-04-12]. ISBN 0-7695-1874-5/03. Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1174905>
- [11] VEENMAN, C. J. *Statistical Disk Cluster Classification for File Carving*. [online]. 2007 393 - 398 [cit. 2012-04-12]. Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4299805>
- [12] KARRESAND, M. and SHAHMEHRI, N. *File type identification of data fragments by their binary structure*. [online]. 2006 140 - 147 [cit. 2012-04-13]. Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1652088>
- [13] LI, W.-J., K. WANG, S. STOLFO and B. HERZOG. *Fileprints: identifying file types by n-gram analysis*. [online]. 2005 64 - 71 [cit. 2012-04-13]. Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1495935>
- [14] ROUSSEV, V. and GARFINKEL, S. *File fragment classification-the case for specialized approaches*. [online]. 2009 3 - 14 [cit. 2012-04-13]. Available on: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5341545>
- [15] *Portable Document Format* [online]. 2001 [cit. 2012-04-16]. Available on: http://en.wikipedia.org/wiki/Portable_Document_Format
- [16] THOMAS, K. *Portable Document Format: An Introduction for Programmers* [online]. [cit. 2012-04-18]. Available on: <http://www.mactech.com/articles/mactech/Vol.15/15.09/PDFIntro/index.html>
- [17] ISO 32000-1:2008. *Document Management Portable document format ? Part 1: IDF 1.7*. Geneva, Switzerland: International Organization for Standardization, 2008. [cit. 2012-04-18]. Available on: http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf
- [18] *PDF* [online]. 2011. [cit. 2012-04-19]. Available on: <http://www.forensicswiki.org/wiki/PDF>
- [19] STEVENS, D. *PDF Stream Objects* [online]. 2008. [cit. 2012-04-20]. Available on: <http://blog.didierstevens.com/2008/05/19/pdf-stream-objects/>

- [20] TESARŮK, V. *Převodník elektronických testů*. Brno, 2010. [cit. 2012-04-30] Available on: http://is.muni.cz/th/255591/fi_b/bc_af1ra.pdf. Bachelor Thesis. Masaryk University. Supervisor RNDr. Tomáš Gregar.
- [21] GARFINKEL, S. and MIGLETZ, J. *New XML-Based Files: Implications for Forensics* [online]. 2009 38 - 44 [cit. 2012-04-30]. Available on: <http://simson.net/clips/academic/2009.IEEE.DOCX.pdf>
- [22] MUDRA, T. *Práce s formátem OpenXML na platformě .NET*. Brno, 2010. [cit. 2012-04-30] Available on: http://is.muni.cz/th/255790/fi_b/Prace_s_formatem_OpenXML_na_platfome_.NET.pdf Bachelor Thesis. Masaryk University. Supervisor doc. RNDr. Tomáš Pikner, PhD.
- [23] GARFINKEL, S., R. BEVERLY a G. CARDWELL. *Forensics carving of network packets and associated data structures*. [online]. 2011, 78 - 89 [cit. 2012-05-08]. Available on: <http://www.dfrws.org/2011/proceedings/14-346.pdf>
- [24] *File Carving: Smart Carving* [online]. 2009 [cit. 2012-05-12]. Available on: http://www.forensicswiki.org/wiki/File_Carving:SmartCarving

LIST OF ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
AVI	Audio Video Interleave
CD	Compact Disc
EOF	End-of-file
EXT	Extended file systém
FAT	File Allocation Table
GIF	Graphic Interchange Format
GPS	Global Positioning Systém
HSF	Hierarchical File Systém
HTML	HyperText Markup Language
JPEG	Joint Photographic Experts Group
NBU	Nokia Backup
NTFS	New Technology File Systém
ODT	OpenDocument Text
OOXML	Office Open XML
PDF	Portable Document Format
SMS	Short Message Service
TXT	Text file
XML	Extensible Markup Language
X-REF	Cross-reference table

LIST OF FIGURES

Figure 1 - Harddisk sectors	17
Figure 2 – How structure based carving works.....	18
Figure 3 - PDF document's details [15].....	21
Figure 4 - The structure of PDF File.....	22
Figure 5 - Body of PDF file	23
Figure 6 - 'xref' table (EOL characters are omitted).....	24
Figure 7 - Trailer.....	25
Figure 8 – Incremental update	26
Figure 9 - Relationship of the indirect object	28
Figure 10 – PDF Stream Object.....	29
Figure 11 – PDF Stream Filter.....	29
Figure 12 – PDF Filter cascaded.....	30
Figure 13 - An example of PDF Raw File	31
Figure 14 - ODF structure.....	34
Figure 15 - OpenOffice.org Writer example.....	35
Figure 16 - OOXML structure	36
Figure 17 - MS Word file example	38
Figure 18 - ZIP archive directory with embedded a Microsoft Word document.....	39
Figure 19 - Flowchart of the decision of the PDF.....	45
Figure 20 - Three different cases of PDF file fragments	46
Figure 21 – Treatment if the file starts on another cluster	47
Figure 22 – Treatment if the file continues on another cluster	48
Figure 23 - An example of OOXML raw data.....	49
Figure 24 - Flowchart of the decision of the Office XML-based file	50
Figure 25 – Diagram of the main method.....	52

LIST OF TABLES

Table 1 – Searched PDF objects	44
Table 2 – Searched occurrences within OOXML	49
Table 3 – Searched occurrences within ODF.....	49
Table 4 - List of PDF methods exported from Javadoc	53
Table 5 - List of OOXML methods exported from Javadoc	56
Table 6 - List of ODF methods exported from Javadoc	58
Table 7 - List of general methods exported from Javadoc.....	61
Table 8 – Results of the PDF classification (Rarely used objects are omitted)	64
Table 9 – Results of the OOXML classification (Rarely used features are omitted).....	65
Table 10 - Results of the ODF classification (Rarely used features are omitted).....	66

LIST OF ATTACHMENTS

Attachment 1 The Content of enclosed CD-ROM

ATTACHMENT A I: THE CONTENT OF ENCLOSED CD-ROM

The CD-ROM contains the following directories:

- *Application* – contains an executive file
- *The source files* – contains all source files including Javadoc and Java files.
- *Fulltext.pdf* – contains the thesis in PDF format