# Real time detection and analysis of PDF-files

Knut Borg

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2013

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

# Real time detection and analysis of PDF-files

Knut Borg

2013/06/02

# Abstract

The PDF-file format is a very popular format to perform attacks with due to the format being quite versatile. A PDF-file can be used as direct attacks against specific targets like the government, the army or other high value targets. These kinds of attacks may be performed by foreign intelligence or by organised crime because they have the most to gain by a successful attack. The attacks are often well obfuscated which makes it easy for users to unintentionally execute the malware on his/her machine. A PDF-file may for instance contain a well written report with important information to the user [1], but do also contain malicious code in order to perform reconnaissance on the target's network.

This master thesis is a continuation of the results of Jarle Kittilisen's master thesis in 2011. The thesis will utilize Kittilsen's proposed methodology by using the machine learning tool 'support vector machine' in order to classify PDF-files as malicious or benign. This thesis will focus on online detection of PDF-files where as Kittilsen performed post-detection. One of the biggest problems with an online detection of PDF-files is the time frame from the PDF-file is detected until it has been classified as either malicious or benign. This master thesis seek to provide answers for the viability of an online detection system of PDF-files.

# Sammendrag

En PDF-fil kan bli brukt som et direkte angrep mot spesifikke mål som f.eks. regjeringen, militæret eller andre verdifulle mål. Slik angrep kan bli utført av organiserte kriminelle eller utenlandske etterretningstjenester fordi disse gruppene kan tjene mye på et suksessfullt angrep. Angrepene er ofte godt gjemt slik at sannsynligheten for at brukere uvitende kjører skadelig kode på deres PC-er er stor. En PDF-fil kan f.eks. inneholde en godt skrevet rapport med viktig informasjon som er relevant for brukeren [1], men PDF-filen kan også inneholde kode som kan gjøre det mulig for angriper å rekogniserer nettverket som brukeren befinner seg på.

Denne masteroppgaven er en videreutvikling basert på resultatene i Jarle Kittilsens masteroppgave fra 2011 [2]. Masteroppgaven vil bruke Kittilsens foreslåtte metode om å bruke maskinlærings verktøyet 'support vector machine' for å kunne klassifisere PDF-filer som godartet eller skadelig. Masteroppgaven vil fokusere på muligheten for et online deteksjonssystem av PDF-filer fordi Kittilsen fokuserte på deteksjon av PDF-filer i etterkant av at filene hadde kommet fram til mottakerne. Et av de største problemene til et online deteksjonssystem er tidsbruken fra en PDF-fil blir detektert til den har blitt klassifisert som godartet eller skadelig. Denne masteroppgaven ønsker å finne svar på hvorvidt et online deteksjonssystem for PDF-filer er en reell mulighet.

# Acknowledgements

I would like to thank my supervisor Prof. Katrin Franke for providing the master thesis topic. Franke provided guidance and assistance throughout the project, as well as insight and constructive criticism. I would also like to thank Jayson Mackie for technical support, ideas and tips in regards to the master thesis report.

A big thanks to my class mates at GUC, including both the graduation students and the first-year master students, for ideas and feedback regarding the master thesis.

Finally I would like to thank my family for motivation and support throughout my studies at Gøvik University College.

Knut Borg, Gjøvik 2013/06/02

# Contents

# List of Figures

# List of Tables

# Glossary

- BMH - Boyer-Moore-Horspool
- Naive - In this thesis the term is used about a brute force methodology to solve a problem
- I/O - Input/Output (information to/from keyboard, hard drives e.t.c)
- Ramdisk - A chunk RAM which can be used as a normal storage medium
- Online - Used about a system that works close to real time, but may have some delay.
- SSD - Solid State Drive
- Substring - Used when talking about specific word/text being searched for in a larger string of text/data.
- SVM - Support Vector Machine

# 1 Introduction

This chapter serves as an introduction for the author's master thesis topic. It will give a brief explanation about the topic itself, challenges, justifications as well as what kind of new knowledge the master thesis seeks to provide.

## 1.1 Problem Description

This master thesis extends on the ideas and the results from a previous master thesis written by Jarle Kittilsen [2]. Kittilsen presented an analysis methodology based on PDF-files that had traversed through the network during a given time frame. Kittilsen's idea consisted of having Snort logging any PDF-files passing by, then extract the PDF-file(s) from Snort's log file and analyse them. This was done by utilising parts of the Jsunpack-n-tool and writing different scripts in Python. For the analysing process Kittilsen ended up with a machine learning tool called "support vector machines" (SVM)[9] as the optimal classifier for PDF-files.

The author's objective will be to advance Kittilsen's idea to make it work in a online environment. Snort will be tasked with intercepting the PDF-files, log the file(s) to disk and then PDF-file will be classified as benign or malicious. This solution seems similar to what Kittilsen developed with his master thesis. However Kittilsen's solution was working in an offline environment where Kittilsen configured Snort to log all the PDFs to the hard drive and then Kittilsen came back after a time period in order to start the classification process. This solution can't prevent the PDF-file from being opened by a user since the classification process will happen after the user have received the PDF-file. An online solution can prevent the user from receiving the PDF-file, but that means the system has to be optimised for high efficiency because the user excepts to not experience any delay. The user will have to experience some delay and it is the author's task to minimize the delay to be as little as possible.

## 1.2 Keywords

Online detection system, PDF-file analysis, pattern matching algorithms, feature extraction, support vector machines, computer network security.

## 1.3 Justification, Motivation and Benefits

Awareness was raised in the late 90s and early 2000 about not opening e-mail attachments containing a .exe-/.bat-/.bin-files, but now these email-attacks have changed file format and started to attack a specific business or organisation [1]. Since the PDF-format is a very popular format and often used for benign purposes, people are less suspicious by default and may open the PDF-file and unintentionally run malicious code that was embedded in the PDF-file. These PDF-files can be sent as email attachments, but they can also be downloaded by a user through a web browser.

By improving the analysis methodology one could prevent these malicious PDF-files from reaching the user, which prevents them from unintentionally run malicious code. Not only will it prevent potential damage caused to the computer systems and databases, but it can also prevent the attacker in gathering non-/sensitive information from the business or organisation[1].

## 1.4 Research Questions

The motivation for this thesis is to reconstruct and improve Kittilsen's solution in order to be able to analyse PDF-files in real-time. The thesis seeks to provide answers to the following questions:

1. Is an online analysis of PDF-files viable and what kind of time delay may the user experience?
2. Will the programming language C perform the same task at a significantly higher speed than what Kittilsen could achieve with Python [2] and how significant is the difference?

## 1.5 Exclusion of JavaScript

The original idea was to increase the focus on JavaScript and how/if the SVM could distinguish between malicious and benign JavaScript, but after a discussion with the author's supervisor an agreement was made. It was was decided that the author's core contribution of his master thesis was "time". The author had only a limited amount of time to delegate to different parts of the master thesis and the author chose to prioritise time reduction elements rather than spend additional time on classifications of JavaScripts. It is also important to note that increased focus on JavaScript would also increase the computational complexity which would result in additional time delay before the user would receive the PDF-file.

## 1.6 Contributions

This master thesis seeks to provide more knowledge about online analysis of a PDF-file. Every element from Snort detecting and logging a PDF-file to the PDF-file being classified as benign or malicious will be explained. Achievements and problems will be explained, as well as pointing out possible counter measures to problems discovered. The focus of this thesis is aimed at the PDF-file format because the format have seen a high increase in exploits during the recent years [10]. While there have been done research in regards to analyse a PDF-file for malicious content, it has to the authors knowledge not been done any significant research towards online analysis of PDF-files captured directly from network traffic.

## 1.7 Thesis Outline

This section provides an overview over what each individual chapter contains.

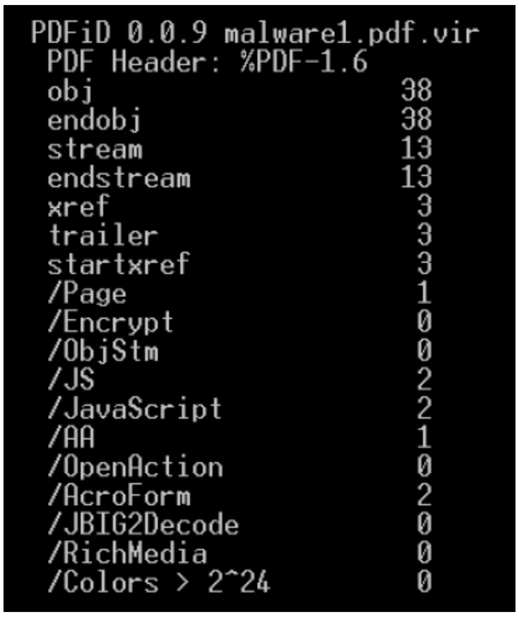
- Chapter 2 provides and overview of what kind of related research have previously been done. The chapter starts with giving an overview of general research towards the PDF-format. Machine learning tools are mentioned in the last section of the chapter.
- Chapter 3 explains why the author chose to use a certain kind of method in order to solve a problem.
- Chapter 4 explains how the author implemented the different parts of the detection system.
- Chapter 5 shows the different results the author got after performing different experiments.
- Chapter 6 Provides a summary of the project and what conclusion the author have drawn.

# 2 Related Work

This chapter seeks to provide information about relevant research in regards to the author's master thesis. The author believes he have made it quite clear that Jarle Kittilsen [2] has done quite a lot in regards to the related work department and the author feels it is unnecessary to repeat Kittilsen's contributions. The author wishes to focus on other researchers' contributions.

## 2.1 PDF Analysis

Didier Stevens is an IT security professional [11] who has done research and created handy tools for different use in the IT security world [3][12]. Several tools for PDF analysis are available from his blog and those are: a PDF parsing tool, a make-your-own-PDF-tool with JavaScript included and a simple python script called PDFiD for scanning a PDF-file for PDF-feature matches. Figure 1 shows an output example for string matches in a PDF-document [3].



```
PDFiD 0.0.9 malware1.pdf.vir
 PDF Header: %PDF-1.6
 obj                    38
 endobj                 38
 stream                 13
 endstream              13
 xref                    3
 trailer                 3
 startxref               3
 /Page                   1
 /Encrypt                0
 /ObjStm                 0
 /JS                     2
 /JavaScript             2
 /AA                     1
 /OpenAction             0
 /AcroForm               2
 /JBIG2Decode            0
 /RichMedia              0
 /Colors > 2^24          0
```

Figure 1: A screen shot of PDFid.py [3]

Stevens also presents a list of PDF-features and explains what they do and if those are a sign of malicious content. For instance the features '`/AA`' and '`/OpenAction`', which indicates an automatic action is to be performed, are very suspicious if the same PDF-file also contain JavaScript. JavaScript in a PDF-file are represented by either '`/JS`' or '`/JavaScript`'.

According to Stevens these are the most important features when detecting malicious PDF-files [12]:

- **/Page** - Number of pages inside a PDF-file. Most malicious PDF-files have only one page.
- **/JS**, **/JavaScript** - Indicate utilisation of JavaScript
- **/RichMedia** - Indicate utilisation of Flash.
- **/AA**, **/OpenAction** - Indicate an automatic action is to be performed. Often used to execute JavaScript.
- **/Acroform** - Number of embedded forms.
- **/JBIG2Decode**, **/Colors** (with a larger value of $2^{24}$) - Indicate utilisation of vulnerable filters.

A researcher named Paul Baccas has published his findings for analysing malicious PDFs [13]. For JavaScript he found that out of 64.616 PDF-files containing JavaScript, only 1093 of them were benign. 98% of the PDF-files containing JavaScript were malicious and this means that JavaScript is a good indication that PDF-file may be malicious. Baccas continued with looking for a mismatch between ’`obj`’ and ’`endobj`’ and a mismatch between ’`stream`’ and ’`endstream`’. Out of 10.321 PDF-files containing mismatched objects, there was 8.685 malicious PDF-files. This result shows that only 16% of the PDF-files containing mismatched objects are benign, which means that the occurrences of mismatching objects in PDF-files may serve as an indicator of the file being malicious. For mismatch between ’`stream`’ and ’`endstream`’ he found that 1.585 PDF-files were malicious out of 2.296 which gives a malicious rate of 69%. Almost 3 out of 4 PDF-files were malicious and this means a mismatch between ’`stream`’ and ’`endstream`’ can serve as an indicator of malicious presences.

There are also several tools available. These tools are mostly developed in spare time by people working in the IT security branch. Here is a brief overview of some tools available:

- PeePDF is a tool written in Python by Jose Miguel Esparza and it provides similar capabilities in analysing PDF-files like Steven’s PDFiD, however PeePDF can also create new PDFs or edit existing ones [14]. The purpose of the tool was to create a complete tool set instead of having to rely on three or four separate tools.
- PDFxray [15] is an analysis tool where one can upload your malicious PDF-files on PDFxray’s website, however the site is currently down for maintenance at the time of writing. As with the Jsunpack tool, where source code is hosted at Google [16], PDFxray can be compiled for private use from Github [17].
- PDF Scrutiniser [18] is a tool which uses static and dynamic detection mechanisms, i.e. statistical analysis and executing of malicious code. The tool also attempted emulate a PDF reader’s behaviour with success according to the authors.

## 2.2 Use of Machine Learning Tools

As mentioned in section 1.5, the author did exclude JavaScript from the scope of the master thesis. Even though JavaScript was excluded, the author did some research in regards to using support vector machine (SVM) to classify malicious JavaScript.

The paper named 'Static Detection of Malicious JavaScript-Bearing PDF Documents' [4] explains the authors', of said document, take on using SVM to analyse malicious JavaScript in PDF-files. The authors used a method called "One-Class Support Vector Machine" (OCSVM) which they claimed to be a very good option when trying to classify JavaScript being malicious or not. The idea is that the SVM only need examples from one class in order to build a classification model and therefore improve the classification performance. Figure 2 shows how the learning and classification is performed [4]. During the learning process, all samples of benign PDF-files are being mapped in a high-dimensional hypersphere. Then the OCSVM tries to find the center "c" and the radius "R". When the JavaScript is being classified, the OCSVM checks if the new data point's distance is longer or shorter than 'R'. If the distance is shorter than 'R', the data point will be treated as benign. However if the the distance is longer than 'R', then the data point will be treated as malicious.

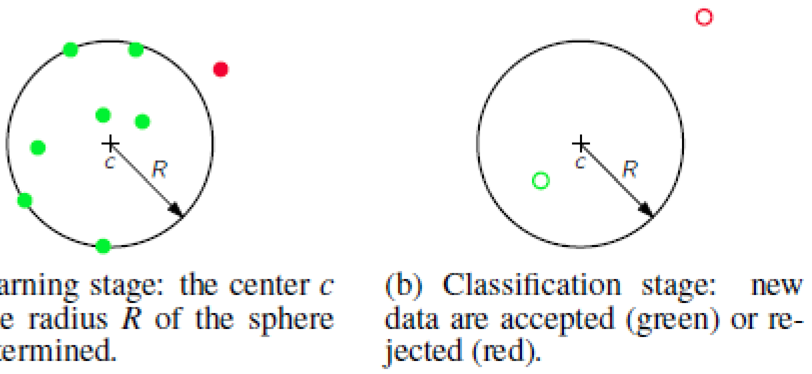
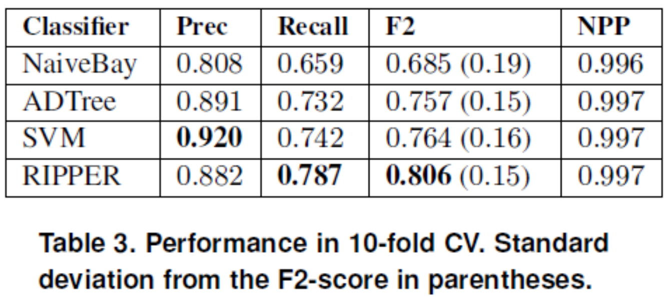

c
R
(a) Learning stage: the center c and the radius R of the sphere are determined.
c
R
(b) Classification stage: new data are accepted (green) or rejected (red).

Figure 2: OCSVM operation[4].

The paper 'Obfuscated Malicious Javascript Detection using Classification Techniques' [5] proposed methods to detect obfuscation in JavaScript, since obfuscation is often a sign of attackers trying to hide their malicious code. In the experiments the authors used Naive Bayes, ADTree, SVM and RIPPER. What they found was that the machine learning classifiers could produce highly accurate results. Figure 3 shows a table of results from their first experiment with classifying JavaScript. The first column shows precision which is the ratio of (malicious scripts labeled correctly)/(all scripts that are labeled as malicious). The second shows recall which is a ratio of (malicious scripts labeled correctly)/(all malicious scripts). The third column contains a "F2"-score which combines precision and recall, but valuing the recall value twice as much as precision. "F1"-score treats both equally and "F0,5" value precision twice as much as recall [19]. The last column is Negative Predictive Power (NPP) which is the ratio of (benign scripts labeled correctly)/(all benign scripts). As can be seen of figure 3, SVM have the best precision rate of 0.920 which is quite high. The authors were quite happy with the results of applying machine learning tools to analyse JavaScript. However the author of this document have to emphasis the fact that the paper in question analysed JavaScripts found in the wild and not necessarily attached to a PDF-file.



| **Classifier** | **Prec** | **Recall** | **F2** | **NPP** |
|---|---|---|---|---|
| NaiveBay | 0.808 | 0.659 | 0.685 (0.19) | 0.996 |
| ADTree | 0.891 | 0.732 | 0.757 (0.15) | 0.997 |
| SVM | **0.920** | 0.742 | 0.764 (0.16) | 0.997 |
| RIPPER | 0.882 | **0.787** | **0.806** (0.15) | 0.997 |

**Table 3. Performance in 10-fold CV. Standard deviation from the F2-score in parentheses.**

Figure 3: SVM have a precision on 0.920[5].

# 3 Choice of Methods

This chapter provides information about why the author chose to use a specific methodology to solve a specific problem.

## 3.1 Online Analysis

It is necessary to distinguish between the terms "online", "live" and "real time" analysis of PDF-files. Developing a live or real time system might prove to be to difficult because different people have a different idea of what the maximum response time should be. The maximum response time can often be broken by sending to much information through the system and the author does not have enough time to perform "extreme" optimisation or to be able to safely say that the system can not be exceeded. The author believes that using the term online, the author will have some leeway in regards to how stable and time efficient the system will be.

Online analysis is better than performing offline (i.e. post-detection) analysis because the PDF-files have already reached its destination. When the PDF-file is detected and classified as malicious by the offline detection system, the PDF-file may already have performed malicious actions. By performing online analysis one have a higher chance of preventing users to unintentionally open malicious PDF-files. However the biggest problem by performing online analysis is that the users expects an unnoticeable time delay from when they initiate a download to the PDF-file have been stored on their hard drive, with the exception of the time it takes to download the PDF-file. In order for online analysis to be a viable option in a real world environment, the time from when the author can analyse the PDF-file until the PDF-file is classified as malicious or benign have to be as low as possible. It is therefore important to utilise as many high efficiency options as possible. One of these options is to analyse the PDF-file directly in memory (RAM) instead of storing the PDF-file to the hard drive. Snort can store the network session of a PDF-file to a hard drive, but the author wants to be able to access it at the same time Snort is analysing the specific network session. This would allow the CPU to gain faster access to the PDF-file as well as decrease the wear on the hard drive. Storing files to a hard drive is a time consuming effort because the computer have to use I/O operations and the time it takes for the hard drive to seek after the stored data. Another option is to use the low level programming language C, which will be described in more detail in section 3.2.

The author is aware of the fact that it is very difficult to not cause any delay for the user receiving the PDF-file because the author need the entire PDF-file before the PDF-file can be classified properly. The question is if it is possible to reduce the time spent analysing a PDF-file to an amount which is satisfiable for the end-user? In any case an online system would provide better security than an offline version because awareness of malicious content would be raised faster.

## 3.2 Lower Level Programming Language

Kittilsen's solution [2] involved using a high level programming language called Python. The problem is that high level languages can be inefficient, in some cases, to use when one want a high performance system. One can say that low level programming provides efficient code, while high level programming makes the code less complex to read for the developer. However writing code in C is no guarantee that Python won't be more efficient as it is quite easy to write badly optimised code in C. Python can also be quite fast and efficient when including packages like Numpy [20]. There are also a lot of people who argue about the difference between the performance of Python and C [21][22][23] and it often comes down to what is the acceptable speed of execution compared to how much time it takes to develop said code or script. The author picked the low level programming language C in an attempt to improve the analysing process of the PDF-files.

## 3.3 Portable Document Format

The Portal Document Format was developed by Adobe during the early 90s and it is based on the PostScript format, a format which was to become the standard for digital printed papers. The specification was published in 1993, but the format remained proprietary until Adobe officially released it as an open standard ISO 32000-1 in July 2008. [24][8].

The PDF-file format have become a popular way of attacking computers due to its high popularity amongst users. Symantec states that the PDF-file was the most exploited format in 2011 and one vulnerability was exploited over one million times [10]. Kittilsen wanted to see if one could use machine learning tools in order to better determine if a PDF-file was malicious or benign. The basis for Kittilsen's thesis was to determine which measurable feature in a PDF-file would be a good indication of malicious content. This can for instance be the presence of JavaScript and automatic actions. Kittilsen developed an offline system and achieved successful results by choosing the support vector machine (SVM) as his classifier. The author is using Kittilsen's result for optimal classifier in order to develop an online detection system.

## 3.4 Snort

Snort is an Intrusion Detection System (IDS) developed and maintained by Sourcefire. It was released by Martin Roesch in 1998 and has grown in popularity by ca. 400.000 registered users and over 4 million downloads [25]. Snort's abilities ranges from protocol analysis, content matching to detect buffer overflow attacks and stealth port scans. Snort do also have the capability to perform real-time alerting and outputting the alarms to a user specified file, a WinPopup message for windows clients or UNIX sockets for UNIX/Linux distributions. Snort can also log the network session that was triggered by an alarm for analysis purposes.

Jarle Kittilsen used Snort in his detection system [2] and the author wanted to find out if Snort has the functionality available in order to be a part of an online detection system for PDF-files.

## 3.5 Extraction From Memory Locations

In a normal situation Snort will dump network packets to a hard drive location of the user's choice. Accessing these packets requires I/O operations to be performed as well as finding the packets on the hard drive (seek time) and extract the PDF-file from Tcpdump's log-file. The initial idea was to access Snort's memory in order to reassemble the PDF-file faster than by reading Tcpdump's log-file after the last network packet had passed Snort. However this would require a great deal of programming and it could prove to be a very difficult task because the author's program needs to know which memory address to check for data and in which order the data needs to be reassembled in. The idea of reading from specific memory locations was scrapped after finding information about ramdisks, which will be further explained in section 3.6. By utilising a ramdisk one could potentially achieve a very high speed on par with reading from Snort's memory locations. It is also a much safer and an easier approach in regards to reassembling a PDF-file. Extracting information from Snort's memory locations is quite a daunting task to reverse engineer and one could end up with a more time exhaustive solution than by logging the PDF-file to a regular hard drive. Even though if the author's software starts Snort, and gets access to Snort's memory by using shared memory, the potential time saved is not worth the time spent on trying to extract the data from Snort's memory.

## 3.6 Hard drive, SSD and Ramdisk

As explained in section 3.5, extracting information from memory locations of Snort was scrapped. This is because extracting information from memory locations could be quite difficult and not feasible to implement when compared to the time one could potentially save.

There are three different suitable storage devices one could use with Snort and these are regular hard drives, soild state drive (SSD) and ramdisk.

| | **Moving parts** | **Problem with fragmentation** | **Volatile** | **Max space** | **Access time** | **Read/Write** |
|---|---|---|---|---|---|---|
| **Hard drive** | Yes | Yes | No | 4TB | 8ms-12ms | 200MB/s +- |
| **SSD** | No | No | No | 1TB | 65-110μs | 500MB/s/250MB/s +- |
| **Ramdisk** | No | No | Yes | Available RAM | 11μ | 10GB/s+- |

Table 1: Information about different storage devices.

Table 1 shows specifications and features for the different storage devices. The speed, time and storage capacity information have been gathered from several different sources like PCWorld, HP and other people who have been doing different benchmark tests [26][27][28]. Note that the author have not taken the possibility of using a RAID setup into consideration when gathering information for hard drive information in tabel 1.

As can be seen in table 1 the ramdisk have the potential of incredible high speed, but the downside is that the ramdisk is a volatile storage medium. If the power shuts down, the information stored on the ramdisk will be gone. The size of the ramdisk is also limited to the amount of available RAM the computer have. The SSD is not a volatile storage device, have no issues with fragmentation and is a lot faster than a regular hard drive. The SSD is however limited in the amount of times it can write to the same space and it can be quite expensive in regards to the amount of storage it provides compared to a regular hard drive. The regular hard drive is slower than its counter parts (SSD and ramdisk), but the hard drive is a lot cheaper per gigabyte of storage capacity and have the highest possible storage capacity ranging up to four terabytes. Section 5.1 contain tests and results in regards to the different storage devices.

## 3.7 Extracting Features

This section will explain the theoretical aspect of extracting features and ideas of how to increase efficiency.

### 3.7.1 Finding Features

Jarle Kittilsen used Jsunpack's PDF-parser in order to scan for features [2]. Jsunpack is a PDF-parser written in Python and specially created for security in mind [16]. PDF-parsers written in C do exist, but they are targeted for more general use. This can for instance be extracting the normal text from the PDF-file, merge two PDF-files into one and converting a text-file to a PDF-file [29]. Didier Stevens states on his blog that:

> *Parsing a PDF document completely requires a very complex program, and hence it is bound to contain many (security) bugs*[3].

There isn't enough time develop a PDF-parser with security in mind, so the author decides that features will be extracted by performing a normal string matching of feature name patterns in the PDF-file. There are different ways to perform a string search and there have been written many papers on trying to achieve the optimal algorithm for single- and multi-pattern matching. Multi-string pattern matching is often used to check for plagiarism like the algorithm Rabin-Karp [30][31], while single string pattern matching is used to check the frequency of a string occurring.

There are different ways to measure the performance of a searching algorithm. One way is to measure the time difference between algorithms and another is to compare the amount of if()-sentences executed. An example of an if()-sentence is:
`if(string_block[i]==substring_block[i])`. The latter makes it easier to spot how the algorithm perform for specific PDF-features for different PDF-files. By counting how many if()-sentences are executed for each feature one can see how the algorithm scales to different sized PDF-files. Bottom line is to try to achieve as few if()-sentences executed as possible and as such decrease the amount of operations the CPU have to handle. This is important since one of the main goals of this master thesis is to decrease the time spent overall and because the feature extraction process itself was the most time consuming process Kittilsen had to handle [2].

The string search process is potentially an easy task and one could have picked different algorithms from C-libraries, but due to the nature of the PDF-file it is better to implement the chosen search algorithm from scratch. This gives the author more control of where the algorithm is in the PDF-file at any given time and makes it easier to double check if the string match found is actually a valid feature. An example of this is the feature '`/Page`'. One '`/Page`' is one "physical" page in the PDF-file, however '`/Page`**s**' is a root node containing several '`/Page`'s [8]. More information about finding features, '`/Page`' and '`/Pages`' can be found in 4.3.1.

These are the features Kittilsen proposed to use in order to achieve optimal results:

- **/Page**: Indicator of one page in the document. Malicious documents tends to contain only one page.
- **/AcroForm**: Number of embedded forms.
- **/OpenAction** and **/AA**: Malicious files tends to have automatic actions to be performed without user interaction.
- **/RichMedia**: Embedding of flash-based content.
- **/Launch**: Amount of launch actions.
- **/JavaScript** and **/JS**: Indicator of a JavaScript.
- **startxref**: Presences of a startxref statement.
- **trailer**: Presence of a trailer.
- Mismatch between **obj** and **endobj**: Malicious documents may have a mismatch of obj and endobj.

The author notes that when talking about matching a feature in a PDF-file, the author will use the word "substring" about the feature name and "string" about the entire PDF-file.

### 3.7.2 Naive String Search

The naive string search algorithm is the brute force method. The worst case of a regular naive string search is O(mn) [32] where *m* is the length of the pattern and *n* is the the length of the text. Generally every character in text *T* have to be compared several times with the naive method.

The naive string search is the algorithm that is the easiest to implement, but like any brute force method it isn't necessary very efficient [33]. As mentioned in section 3.7.1, the feature '`/Pages`' can be confused with the feature '`/Page`' and the author decided to start with developing a naive algorithm first. However by implementing the naive algorithm first, the author would be sure that all potential features had been matched and that it was only down to validating the feature match in order to find the correct occurrence of a specific feature. In order to double check that the algorithm worked correctly, the author used Didier Steven's PDF-tool called 'pdfid.py' [3] and Jarle Kittilsen's feature extraction script on the same PDF-file [2].

Kittilisen proposed 12 features to extract from a PDF-file to get the optimal classification detection rate. Because 12 features have to be extracted, there will be two possible ways to develop the naive searching algorithm. One method is to search through the PDF-file once for each individual feature, which allow for scanning of multiple features at the same time and record the efficiency of each scanned feature. The other method is to check one byte in the PDF-file against the first byte in every feature to be searched for until a substring match is found. This means that a PDF-file on 2MB would have a less amount of executed if()-sentences compared to if the algorithm had to search through the same PDF-file 12 times in order to count each individual feature (12x2MB). The downside with comparing one byte in the PDF-file against all of the 12 features is that one can't search for multiple features at the same time and therefore excluding the potential use of multi-threading. It is also impossible to control how many times a single feature have triggered an if()-sentence because it relies on which feature one have matched one byte against first. However for single core systems only, this version of a naive string search could prove to be a lot faster than by searching through the same PDF-file 12 times.

### 3.7.3 Improved String Search

This subsection will provide information about possible algorithms which will be an improvement over the naive algorithm.

One method is the KMP algorithm developed by Donald E. Knuth, James H. Morris and Vaugan R. Pratt [34]. The algorithm is developed in such a way that only one comparison happens of one character in text *T*. KMP have pre-compute complexity on O($m^2$)[33]. The substring is compared to the start of the string and then shifted to the right until a substring match occur.

Another method is the Boyer-Moore algorithm. The Boyer-Moore algorithm changed the way of how string searching was previously performed by comparing the last character in the substring (right to left), while scanning the text from left to right. This allowed the algorithm to skip characters and comparisons by jumping *x* number of blocks to the right if no character match was found. This algorithm requires two tables with information. One of the table is called "jump" and have information about the jump length when a mismatch occurred. The second table is called "right" where it contains the rightmost index of the string where character *t* appears[33]. Horspool improved the BM algorithm in 1980 by only require one table compared to the original two [32]. The table the Boyer-Moore-Horspool (BMH) required was the jump length for each letter. If the letter compared did not exist in the substring (i.e. not in the "jump" table), then the jump-length would be the size of the substring. If a match with a character within the substring was found (i.e. it existed in the "jump" table), then the jump length would be a predetermined value depending on where the matched character existed in the substring.

The author chose to start with implementing BMH because of its effectiveness and easy to comprehend algorithm. Other scientific research papers have proposed algorithms that have increased efficiency for about 10% increase speed [35] compared to the BMH algorithm, but the author feels that the gain in speed is not significant enough to warrant implementing a more complex algorithm at this stage. This depends on the average sized PDF-file that needs to be classified. The result of the feature extractions can be found in section 5.2.

The reason why only the naive and the BMH algorithm was implemented in this master thesis is because the feature extraction process is not just about finding a match. One must also have to make sure that the match found is a valid feature. The author had to cross check the results with other feature extraction tools, Didier Stevens' tool and Jarle Kittilsen's python script [3][2], and the debugging process could be quite exhaustive. The author had to figure out where the problem occured, either an algorithmic problem or a "fault" with the PDF-file where a substring is confused to be a valid feature, and it took quite a lot of time to properly ensure that the algorithm would only count valid features.

### 3.7.4 Multi-threading

Utilising multi-threading allows the feature extraction process to assign one thread to each feature to be scanned for. This only works if the computer have multiple CPU-cores available or utilising the GPU-threads to do the computations instead. An application programming interface that supports multi-threading is OpenMP [36].

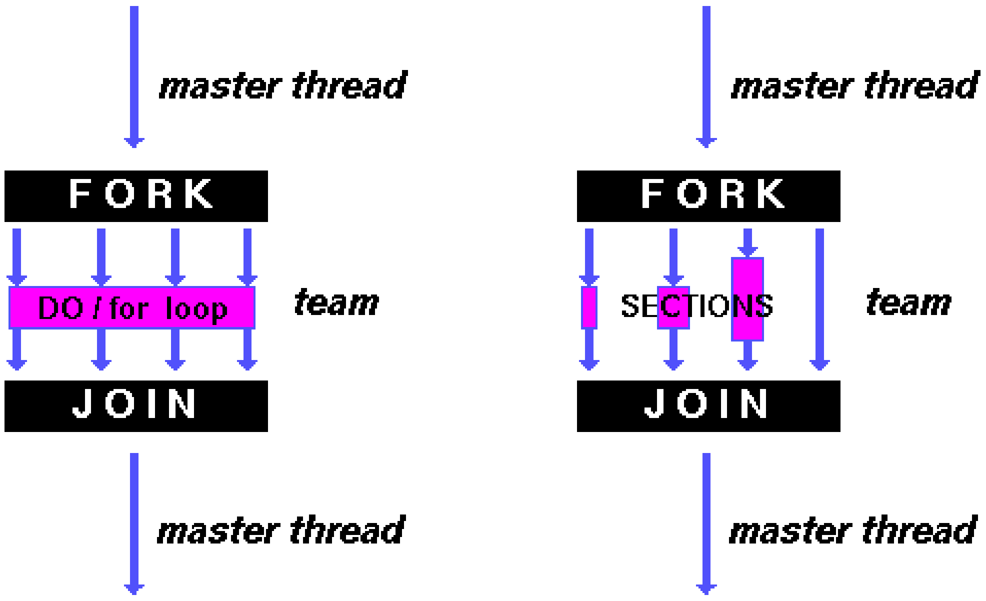


Figure 4: Different ways of multi-threading as shown on the OpenMP's website [6]

Figure 4 shows that multi-threading can be implemented in different ways [6]. One way is create a for-loop that will loop through all the features and then OpenMP will fork one subthread for each iteration of the for-loop resulting in one thread for each feature. An example can be seen in code-list 1:

```
//code-list 1
#pragma omp parallel /*(+ Specific parameters based on your program) */
{
  #pragma omp for /*(+ Specific parameters based on your program) */
  for(i=1; i<13;i++)                         //Each feature
  {
    f_c[i] = feature_check(f, P_C, i, size, f[i].type);    //Number of matches.
  }
}
```

An other way of performing multi-threading is to perform different sections, where it is possible to do different processes for different threads. In this scenario the same function call is used for all of the features, which means the for-loop implementation will suffice. However if one wanted to use different algorithms or perform other actions simultaneously, then "omp section" is the better choice. An example can be seen in code-list 2:

```
//code-list 2
#pragma omp parallel /*(+ Specific parameters based on your program) */
{
  #pragma omp sections /*(+ Specific parameters based on your program) */
  {
    #pragma omp section
    f_c[1] = feature_check(f, P_C, 1, size, f[1].type);    //Number of matches.
    #pragma omp section
    f_c[2] = feature_check(f, P_C, 2, size, f[2].type);    //Number of matches.
    #pragma omp section
    f_c[3] = feature_check(f, P_C, 3, size, f[3].type);    //Number of matches.
  }
}
```

The code example of parameters are the following: *f_c* is an array containing features, *f* is the struct where the features are stored, *P_C* is the PDF-file, *i* is the number of which a feature is stored (code-list 1 only), *size* is of the PDF-file and *type* tells the algorithm if the feature name contains '/' or not since additional checking have to be performed on features without '/' at the beginning of the feature name like '`obj`' and '`startxref`'. More information about the additional checks can be found in section 4.3.1. For feature extraction, if all of the features had a similar time cost, the total amount of time spent on feature extraction could have been divided on 12 or the number of features to be extracted. The time saved also depends on how many threads the CPU have available. A normal quad-core i-7 CPU have 4 threads and 4 virtual threads, which is not enough when every feature should be searched for simultaneously. An option could be to utilise the GPU to perform this task. This way the CPU can focus on Snort and logging network packets, while the GPU takes care of feature extraction process. Kristian Nordhaug wrote a master thesis about "GPU Accelerated NIDS Search" with Snort in mind with Cuda-technology [37]. By utilising the GPU instead of CPU, one will suddenly have access to a large amount of threads. This means one could also use multi-threading inside the feature_check()-function. The PDF-file could be divided into several smaller sized chunks and then be scanned for one feature individually. If a PDF-file is split into four chunks, then one would need a maximum of 48 available threads in order to scan every feature simultaneously. This is not a problem if the feature extraction process is delegated to the GPU.

While multi-threading could have proved some increased efficiency over single-thread-processing on the author's computer (CPU), the real efficiency increase would come from using multi-threading and running a search for all of the features simultaneously with a GPU implementation. However the efficiency might decrease somewhat due to scheduling operations. Due to time constraints, multi-threading was not implemented and the author could not experiment with Nvidia's CUDA technology [38].

## 3.8 Support Vector Machine

Jarle Kittilsen experimented to classify PDF-files by using different machine learning tools. Among these tools were BayesNet, C4.5, Multilayer Perceptron, RBF Network and Support Vector Machines (SVM) [2]. When Kittilsen used SVM on his collection of malicious and benign PDF-files, he achieved very good results with a success rate at 0.9949 and the area under the ROC curve was 0.9967. This is one of the reasons why SVM is popular, namely because of:

> *(...) its high accuracy, ability to deal with high-dimensional data such as gene expression, and flexibility in modelling diverse sources of data.* [7]

The author chose to use Jarle Kittilsen's recommendation by utilising Support Vector Machine[2]. There are some C-code implementations of SVM and these are:

- LibSVM [39]
- LibSVM-light [40]
- Shark [41]

Jarle Kittilisen used PyML which is compatible with the input data structures for LibSVM and LibSVM-light[42]. The author chose to utilize LibSVM as a starting point. Depending on how LibSVM performed classification wise, and how much time it consumed, the author would decide to stick with LibSVM or try out other implementations.

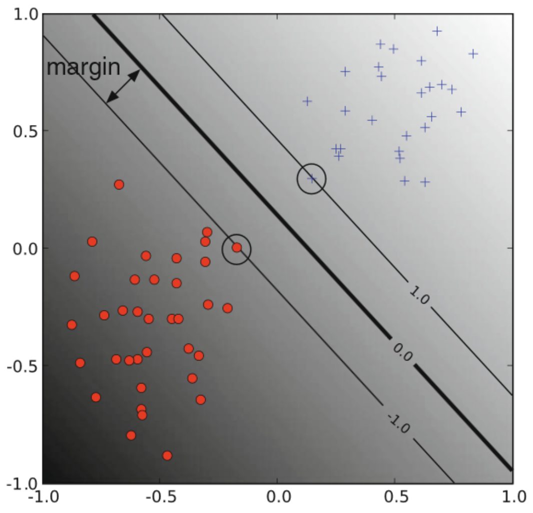


Figure 5: *A linear SVM. The circled data points are the support vectors - the samples that are closest to the decision boundary. They determine the margin with which the two classes are separated* [7].

Figure 5 shows an example of how SVM handles a two-class learning problem. One of the classes is often noted as '1' (positive) and the other class as '−1' (negative). The dots in the figure shows the different data points where the red circles belong to one class while the blue crosses belongs to the other class. It is important to find the optimal decision boundary in order to get the best classification rate. Kittilsen chose to use the Gaussian kernel which have two important values one can tweak and these are the "inverse-width" parameter $\gamma$ and the "penalty value" C [2]. Kittilsen's optimal values are 0,1$\gamma$ and C=100. The author decides to use Kittilsen's proposed optimal classifier so more time can be allocated to develop a system suitable for online implementation.

Figure 6 shows an example of how different values for gamma have an effect on the decision boundary for the Gaussian kernel.

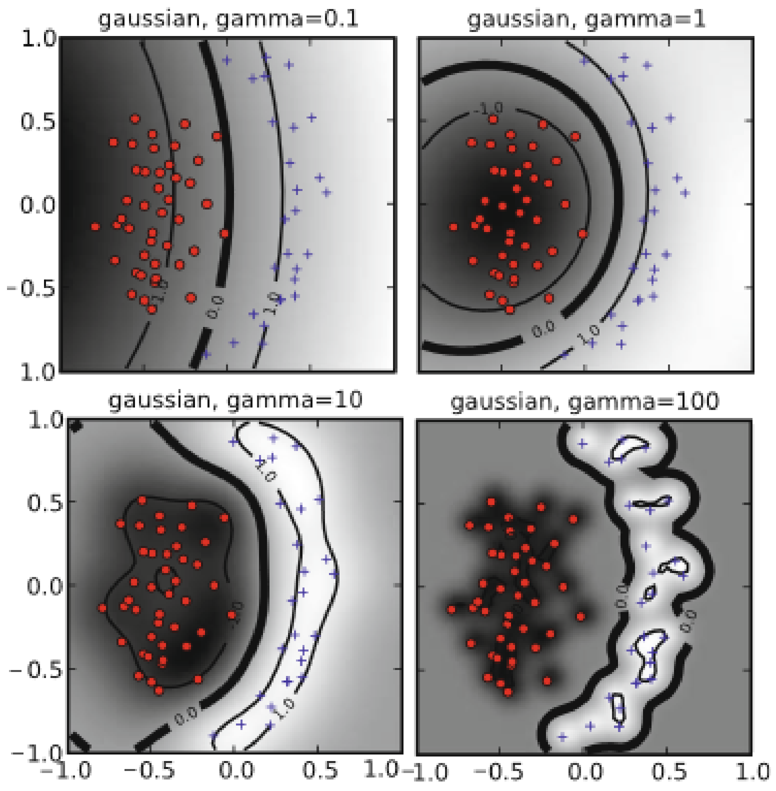


Figure 6: An example of the Gaussian kernel[7].

# 4 Implementation

This chapter will explain the developing process from capturing network traffic to the classification of the PDF-file.

## 4.1 Snort and Extraction of PDF-file

There are two ways of installing Snort on the computer. For Ubuntu one can use the command in the terminal window: `sudo apt-get install snort`
This is a great solution for normal use, but for developing purposes it would be beneficial to download the source code from Sourcefire instead [43]. This allows you to tweak the source code by for instance adding '`printf()`' commands in order to get a better understanding of how and/or when Snort calls different functions. This was quite useful when the author wanted to know how Snort and Tcpdump dumped network packets to disk.

A side note is that Kittilsen used Snort's own logging format *unified2* instead of logging with the pcap-format. Since Snort can allow Tcpdump to log network packets in the pcap-format, it means that the author will save time by not having to convert the unified2-log file to the pcap-format.

In order to capture the PDF-files with Snort, the following rules were created:

- `pdf tcp any any <> any any (msg:"PDF detected"; content:"%PDF-"; fast_pattern;tag:session,0,packets,60,seconds; sid:2000011;)`
- `pdf tcp any any <> any any (msg:"EOF detected"; content:"%%EOF"; fast_pattern;sid:2000012;)`

The rule id 2000011, which some of the content was gathered from Kittilsen's detection rule [2], detects the PDF-files by looking for the PDF-header in both outgoing and ingoing network traffic. The rule id 2000012 detects the end of a PDF-file. This alarm is used to alert the author's software in order to start the extraction of the PDF-file from Tcpdump's log-file. The two PDF-detection rules can be placed in a new rule file called 'pdf.rules' and be registered in the Snort config-file as: `include $RULE_PATH/pdf.rules`
An important note is that the first rule states that Snort will log the following packets in the specific network session for 60 seconds. This may cause problems in regards to network bandwidth and is discussed in section 6.1.

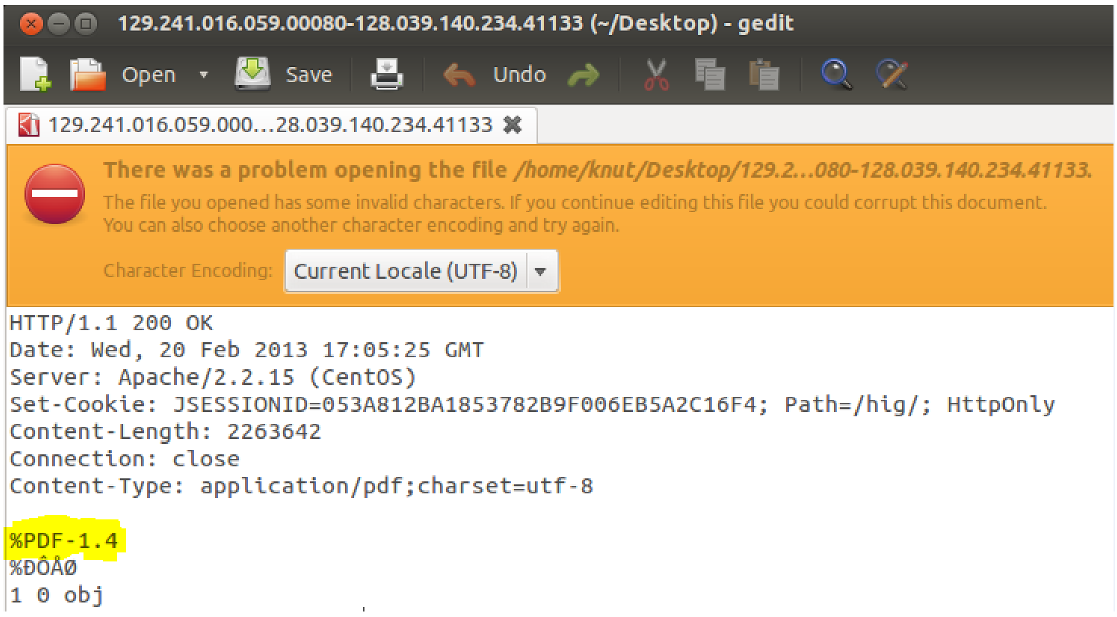
The following declares a new rule type based on the form of `alert`:

```
ruletype pdf
{
    type alert
    output alert_fast: pdf.alert
    output alert_unixsock
}
```

The new rule type *pdf* utilise the warning type *alert* , the log of alarms triggered contain only basic information and outputs this information to a file called '`pdf.alert`'. An alert will also be sent by using an Unix socket by the name '`snort_alert`'. An Unix socket allows two or more programs to communicate with each other by either establishing a connection between sender and receiver by using '`SOCK_STREAM` '(TCP like) or just pushing packets with '`SOCK_DGRAM`' (UDP like)[44]. The author chose to use the '`DGRAM`' option because that is what Snort is using to push out alerts with. To start Snort the following will have to be entered:
`snort -c snort.conf`
One can add additional parameters to the starting command, like forcing Snort to use '`alert_fast`' and log with Tcpdump, however the author have already enabled these things in Snort's config-file. As mentioned in section 3.4, the author is forcing Snort to log with Tcpdump in the the pcap-format, unlike Kittilsen who used Snort's own logging format called unified2 [2]. This allows one to skip the process of converting the unified2 log file over to the pcap-format in order to be able to extract the PDF-file.

```
//———Source code for Unix_socket alarm———
//Waiting for the PDF alarm to trigger
//[0] is the first block in the snort alarm message output.
do
{
  printf("Waiting for alert.\n");
  recv= recvfrom(sock, (void *)&alert, sizeof (alert), 0, (struct sockaddr *) &temp , &
      len);
  printf ("[%s] [%d]\n", alert.alertmsg, alert.event.event_id);
}while(alert.alertmsg[0] == 'E');
          //Waiting for the entire TCP session to be captured (%%EOF hit)
recv = recvfrom(sock, (void *)&alert, sizeof (alert), 0, (struct sockaddr *) &temp , &
    len);
}
```

The listed source code shows how the author's program loops until a PDF-header (`%PDF-`) is detected. When the PDF-footer ('`%%EOF`') is detected, the author's program will tell Tcpflow to start extracting the PDF-file from Tcpdump's log-file. The name of the log-file is "`*.log.*`" meaning either "snort.log.1234" or "tcpdump.log.1234". The author's program starts Tcpflow with "`tcpflow -r [name of log-file]`" and then Tcpflow will output a file on the form of "`*.*.*.*.*-*.*.*.*.*`". The file is named "[Ip-adress].[port-number]-[Ip-adress].[port-number]". The author finds the extracted file by using the '`popen()`'-function with the input "`find *.*.*.*.*-*.*.*.*.*`". Tcpflow's output file contain the HTTP-header information as can be seen in figure 7, though the rest of the PDF-file is however intact. The author discovered that even though the HTTP information was in front of the PDF-header, the file was defined as a PDF-file and could be opened by a PDF-reader. According to Symantec [45], this is a potential attack methodology and the author decided it could be interesting to see if any of the PDF-files in Kittilsen's benign dataset contained suspicious content in front of the PDF-header. More information about this can be found in section 5.4.


129.241.016.059.00080-128.039.140.234.41133 (~/Desktop) - gedit
Open
Save
Undo
129.241.016.059.000...28.039.140.234.41133

There was a problem opening the file /home/knut/Desktop/129.2...080-128.039.140.234.41133.
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

Character Encoding: Current Locale (UTF-8)

```
HTTP/1.1 200 OK
Date: Wed, 20 Feb 2013 17:05:25 GMT
Server: Apache/2.2.15 (CentOS)
Set-Cookie: JSESSIONID=053A812BA1853782B9F006EB5A2C16F4; Path=/hig/; HttpOnly
Content-Length: 2263642
Connection: close
Content-Type: application/pdf;charset=utf-8

%PDF-1.4
%ÐÔÅØ
1 0 obj
```

Figure 7: HTTP information in front of a PDF-file

A small side effect of the PDF-file specification not being picky regarding the PDF-header can be seen in figure 8. The file shown is the file containing rules for Snort, but it is identified as a PDF-file because the file begin with a Snort rule looking like this:

```
pdf tcp any any <> any any (msg:"PDF detected"; content:"%PDF-";
fast_pattern;tag:session,0,packets,60,seconds; sid:2000011;)
```

I.e. it is missing '`%`' in front of '`pdf`'.

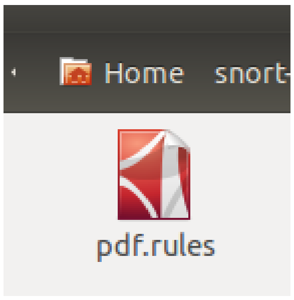


Figure 8: A rule file used by Snort is identified as a PDF-file

After the features have been extracted and the PDF-file have been classified, the Tcpflow's output file is removed, however the Tcpdump log-file remains. The Tcpdump log-file proves to be a problem as more PDF-files are being logged by Snort. Every time a new PDF-file is detected by Snort, the Tcpdump log-file have to be extracted of its content. After *n* PDF-files have been logged, *n* PDF-files will have to be extracted from the Tcpdump log-file at the same time and the process becomes a resource hog as new PDF-files are detected. The author has tried several methods in order to counter this problem:

- Deleting the Tcpdump log-file. The problem is that Snort/Tcpdump won't recreate the file. The file is only created on start-up of Snort.
- Deleting the file and create a new one with the same name. The problem is that Snort/Tcpdump won't log to the new file.
- Using the '`write()`'-function to recreate the file. The good thing is that the file is empty, but the problem is that it will no longer receive logged packets.
- Using the '`write()`'-function to delete the file's content in order to start over again. The good thing is that the file is empty. The problem is that when the author logged the same PDF-file, the Tcpdump log-file would suddenly be twice the size. The author believes that Snort/Tcpdump has a reference point in the Tcpdump log-file and it is therefore very difficult to manipulate where in the log-file Snort/Tcpdump should start with dumping network packets.

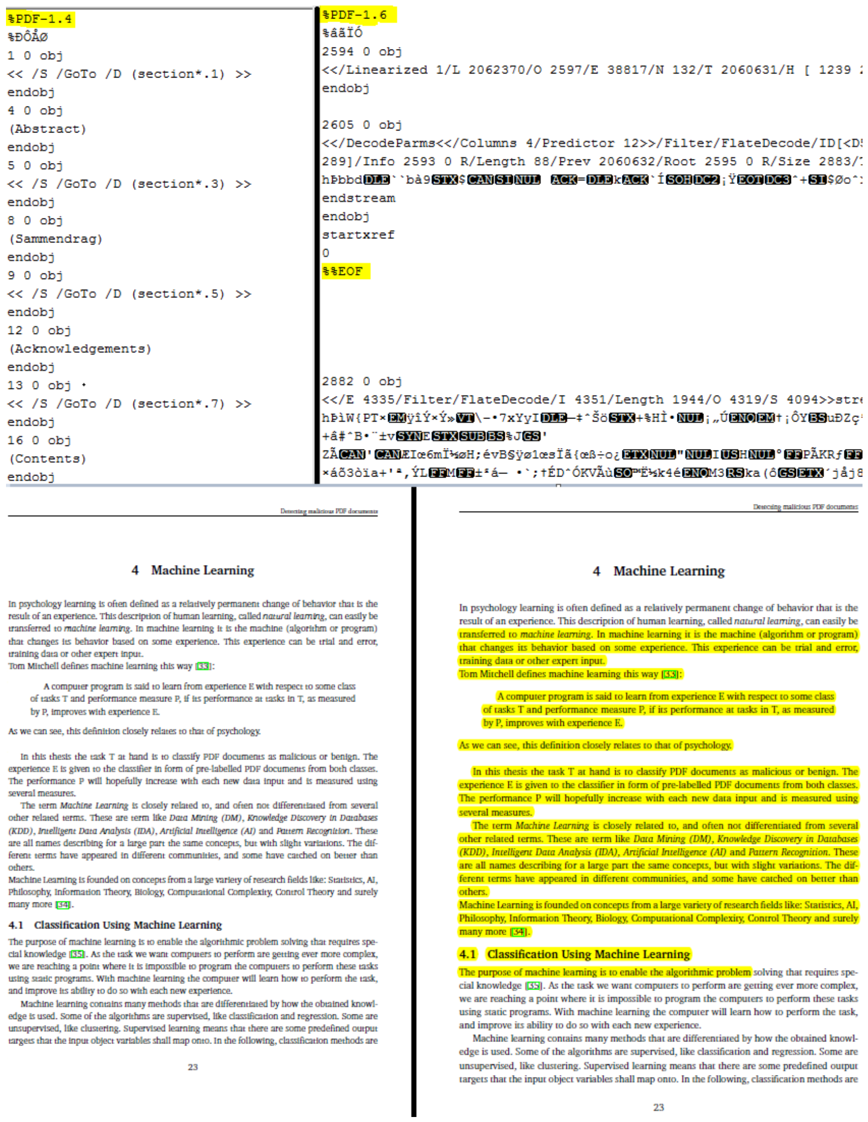
The result of the Tcpdump log-file problem can be summarized as follows:

- The extraction of PDF-files becomes heavier (time/space) as additional files are logged.
- Finding the correct PDF-file may prove to become more difficult as new PDF-files are logged, seeing as each PDF-file will be named with IP-addresses and ports. Duplicates may occur.

The easiest way to counter the Tcpdump log-file problem, and maybe the only way, is to force Snort to quit after a PDF-file have been logged. Then the used Tcpdump log-file is removed and Snort can start running again. A new Tcpdump log-file will be created and the system is ready to receive a new PDF-file. However one can't implement a system where the IDS itself (Snort) has to be shut down every time a PDF-file is logged to disk and the author will therefore not implement the system, in its current form, in an online environment. Another problem caused by the Snort and the Tcpdump log-file is that Snort can't receive two or more PDF-files at the same time because of the following reasons:

- A large PDF-file is being logged by Snort, and during this time a smaller PDF-file is also being logged. The small PDF-file finishes first and the author's program have no way of knowing how to distinguish between the two PDF-files. The author's program do neither know which PDF-file have finished being logged to disk.
- A benign PDF-file may have two, or more, PDF-footers (`'%%EOF'`) which will cause problems because the author's program utilize '`%%EOF`' in order to indicate that the PDF-file has been completely been written to disk. An example can be seen in figure 9.

In regards to checking when the entire PDF-file has been logged to disk, Snort does not have the ability to control if the PDF-file have finished transferring. There is however a possibility to check the file size with the HTTP header that is being sent (if available), but Snort doesn't give you access to count packets or measure the size of the packets. Checking the HTTP-header is therefore an unreliable method and can not be utilized. The only way Snort is allowed to split the Tcpdump log-file is when a given size limit have been set. Since PDF-files can vary in size from 100KB to several megabytes, there is no point in using this functionality.



```
%PDF-1.4
%ÐÔÅØ
1 0 obj
<< /S /GoTo /D (section*.1) >>
endobj
4 0 obj
(Abstract)
endobj
5 0 obj
<< /S /GoTo /D (section*.3) >>
endobj
8 0 obj
(Sammendrag)
endobj
9 0 obj
<< /S /GoTo /D (section*.5) >>
endobj
12 0 obj
(Acknowledgements)
endobj
13 0 obj •
<< /S /GoTo /D (section*.7) >>
endobj
16 0 obj
(Contents)
endobj
```

```
%PDF-1.6
%âãÏÓ
2594 0 obj
<</Linearized 1/L 2062370/O 2597/E 38817/N 132/T 2060631/H [ 1239
endobj

2605 0 obj
<</DecodeParms<</Columns 4/Predictor 12>>/Filter/FlateDecode/ID[<D
289]/Info 2593 0 R/Length 88/Prev 2060632/Root 2595 0 R/Size 2883/
hÞbbdDLE``bà9STX$CANSINUL ACK=DLEkACK`ÍSOHDC2¡ŸEOTDC3^+SI$Øo^
endstream
endobj
startxref
0
%%EOF




2882 0 obj
<</E 4335/Filter/FlateDecode/I 4351/Length 1944/O 4319/S 4094>>str
hÞìW{PT×EMÿîÝ×Ý»VT\-•7xYyIDLE—‡^ŠöSTX+%HÌ•NUL¡„ÚENQEM†¡ÔYBSuÐZç
+â‡^B•¨±vSYNESTXSUBBS%JGS'
ZÃCAN'CANÆIœ6mÏ¼øH;évB§ÿø1œsÏã{œß÷o¿ETXNUL"NULIUSHNUL°FFPÃKRƒFF
×áõ3òïa+'ª,ÝLFFMFF±²á– •`;†ÉD^ÓKVÃùSO™Ë¼k4éENQM3RSka(ôGSETX´jåj
```

Detecting malicious PDF documents

## 4 Machine Learning

In psychology learning is often defined as a relatively permanent change of behavior that is the result of an experience. This description of human learning, called *natural learning*, can easily be transferred to *machine learning*. In machine learning it is the machine (algorithm or program) that changes its behavior based on some experience. This experience can be trial and error, training data or other expert input.
Tom Mitchell defines machine learning this way [33]:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

As we can see, this definition closely relates to that of psychology.

In this thesis the task T at hand is to classify PDF documents as malicious or benign. The experience E is given to the classifier in form of pre-labelled PDF documents from both classes. The performance P will hopefully increase with each new data input and is measured using several measures.

The term *Machine Learning* is closely related to, and often not differentiated from several other related terms. These are term like *Data Mining (DM)*, *Knowledge Discovery in Databases (KDD)*, *Intelligent Data Analysis (IDA)*, *Artificial Intelligence (AI)* and *Pattern Recognition*. These are all names describing for a large part the same concepts, but with slight variations. The different terms have appeared in different communities, and some have catched on better than others.
Machine Learning is founded on concepts from a large variety of research fields like: Statistics, AI, Philosophy, Information Theory, Biology, Computational Complexity, Control Theory and surely many more [34].

### 4.1 Classification Using Machine Learning

The purpose of machine learning is to enable the algorithmic problem solving that requires special knowledge [35]. As the task we want computers to perform are getting ever more complex, we are reaching a point where it is impossible to program the computers to perform these tasks using static programs. With machine learning the computer will learn how to perform the task, and improve its ability to do so with each new experience.

Machine learning contains many methods that are differentiated by how the obtained knowledge is used. Some of the algorithms are supervised, like classification and regression. Some are unsupervised, like clustering. Supervised learning means that there are some predefined output targets that the input object variables shall map onto. In the following, classification methods are

23

Detecting malicious PDF documents

4 Machine Learning

In psychology learning is often defined as a relatively permanent change of behavior that is the result of an experience. This description of human learning, called *natural learning*, can easily be transferred to *machine learning*. In machine learning it is the machine (algorithm or program) that changes its behavior based on some experience. This experience can be trial and error, training data or other expert input.
Tom Mitchell defines machine learning this way [33]:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

As we can see, this definition closely relates to that of psychology.

In this thesis the task T at hand is to classify PDF documents as malicious or benign. The experience E is given to the classifier in form of pre-labelled PDF documents from both classes. The performance P will hopefully increase with each new data input and is measured using several measures.

The term *Machine Learning* is closely related to, and often not differentiated from several other related terms. These are term like *Data Mining (DM)*, *Knowledge Discovery in Databases (KDD)*, *Intelligent Data Analysis (IDA)*, *Artificial Intelligence (AI)* and *Pattern Recognition*. These are all names describing for a large part the same concepts, but with slight variations. The different terms have appeared in different communities, and some have catched on better than others.
Machine Learning is founded on concepts from a large variety of research fields like: Statistics, AI, Philosophy, Information Theory, Biology, Computational Complexity, Control Theory and surely many more [34].

4.1 Classification Using Machine Learning

The purpose of machine learning is to enable the algorithmic problem solving that requires special knowledge [35]. As the task we want computers to perform are getting ever more complex, we are reaching a point where it is impossible to program the computers to perform these tasks using static programs. With machine learning the computer will learn how to perform the task, and improve its ability to do so with each new experience.

Machine learning contains many methods that are differentiated by how the obtained knowledge is used. Some of the algorithms are supervised, like classification and regression. Some are unsupervised, like clustering. Supervised learning means that there are some predefined output targets that the input object variables shall map onto. In the following, classification methods are

23

Figure 9: Left side: Kittilsen's original report. Right side: Kittilsen's report with marked content by the author. Marking in the PDF-file resulted in a new PDF-file and '`%%EOF`' right under the PDF-header.

## 4.2 Hard drive, SSD and RamDisk

Hard drives and SSDs are quite easy to use for logging purposes. Both the regular hard drive and the SSD will show up as individual storage devices. In order to force Snort to log to a different directory (i.e. storage device) one have to enter the following command:

```
snort -c snort.conf -ld [folder_structure]/[snort_log_folder]/
```

In order to utilize a ramdisk one can start by creating a directory of the user's choice. The author chose to use Kate Pauls' example [46] and create a folder which would be suitable for a ramdisk. The author created a folder in the '`/tmp/`'-folder called "ram". The author entered the following command:

```
sudo mount -t tmpfs -o size=512M tmpfs /tmp/ram/
```

This command will mount the ramdisk and make it available for use. The author notes that even though the ramdisk will disappear when shutting down the computer, it is possible for the system to recreate the ramdisk at start-up [26]. Now that the ramdisk is ready to use, the author notes that by following the given configuration one will only have a storage device with a capacity of 512MB. The storage capacity can be increased, but one should take note of how much physical RAM the computer have available and how much RAM the other running processes consume.

## 4.3 Feature Extraction

The following sections describes the implementation of the feature extraction process. The author notes that the Feature Extraction chapter will talk about a PDF-file as it is stored in a long string and one byte is stored in one block (i.e. '`/`' exists in one block = `[/]`). Instead of talking about a "forward slash", the author will use '`/`' as it would be easier for the reader to comprehend what is going on and the fact that the feature extraction basically consist mostly of block comparisons between the "PDF-string" and the substring which contains the feature name. It is also important to note that because the author does not parse the document and only performs a string search, there is no way for the author to determine if a feature is placed in a logical place. For instance a '`/Page`' feature could be placed at the end of the document, though it should not be there, the author's program will only count the feature match and not take into account where it was found.

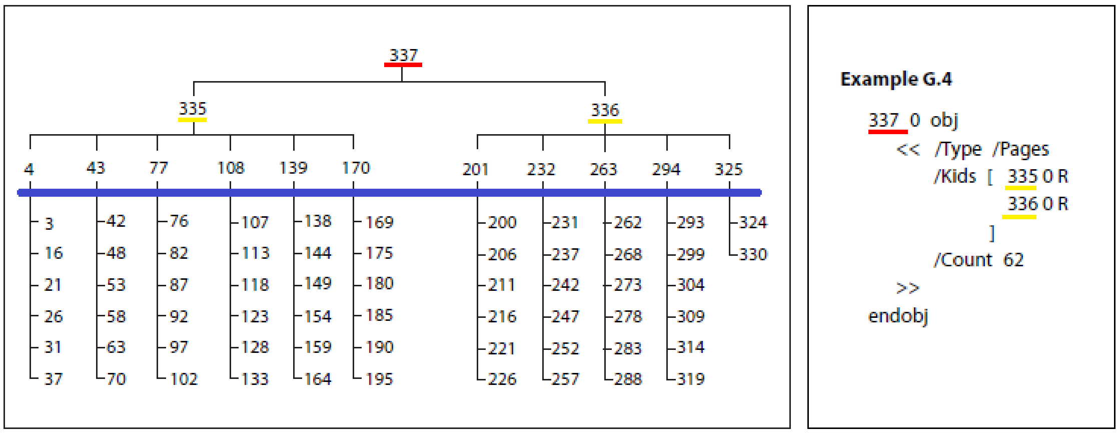
### 4.3.1 Finding Features

The author used Jarle Kittilisen's master thesis report as a starting point for feature extraction [2]. In order to confirm that the correct amount of features had been extracted, the author chose to validate the results with both Kittilisen's Python script and Didier Steven's pdf-tool [3]. For information about the Adobe PDF-file format the author looked at Adobe's own specification paper [8].

There are two "kinds" of features the author is looking for in a PDF-file. One kind of feature starts its feature name with a '`/`' and the other kind does not. The author knows that '`/`' is a definitive start of a feature name, however the other features like 'obj' and 'endobj' may be a part of a longer word and as such require additional validation in order to confirm if the substring match is a legit feature or not. The author is going to explain the procedure for feature names containing '`/`' first.

### /Feature Name

An example of a feature name starting with '/' is '`/Page`'. Looking for '`/Page`' in Kittilsen's report gives a feature count of 162, however there are only 132 pages in the PDF-file. The problem occurring is that there is also a feature name called '`/Pages`', which describe the structure of the "page-tree". Figure 10 shows an example of this and the example is gathered from Adobe's specification paper of the PDF-file [8].



**Example G.4**

```
337 0 obj
    << /Type /Pages
       /Kids [ 335 0 R
               336 0 R
             ]
       /Count 62
    >>
endobj
```

Figure 10: '/Pages' is the root node. Everything below the blue line are objects and each object contains one '/Page'. The picture shows a PDF-file with 62 pages [8].

In order to prevent '`/Pages`' to be counted along with '`/Page`', one will have to check the next block after the last character in the feature name. In this case this would be the fifth block after '/'. If the block contains the ASCII value of either end-of-line, a space or '/' we will know that the match is most likely a valid feature. The author use the phrase "most likely" because one can't be a 100% sure if the match is valid or not. This will be explained in section 6.1 and more examples will be presented in section 4.3.1.

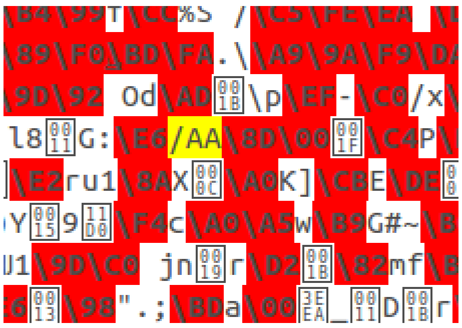
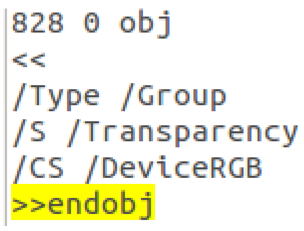


Figure 11: /AA with what Gedit presents as hex values in the block in front and after the feature name.

The author found another PDF-file [47] and tried to extract features from it. A cross check with Didier Stevens' tool [3] showed that the feature '`/AA`' was missing. As can be seen in figure 11, the feature name had a "hex value" in the block after the feature name. The author improved



```
828 0 obj
<<
/Type /Group
/S /Transparency
/CS /DeviceRGB
>>endobj
```

Figure 12: '>' to close to 'endobj

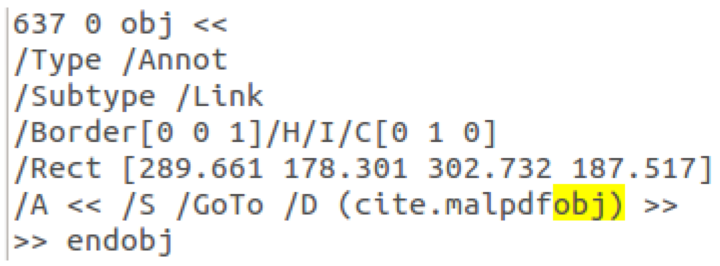
the validation rule by including the fact as long as the block after the feature name did not contain an ASCII value of a-z or A-Z, then the chance of it being a valid feature is high.
In order to check feature names with '/', one will have to atleast check the following bytes: `[/][F][e][a][t][u][r][e]`**<u>`[?]`</u>**

**Normal Feature Name**

These features do not start with '/' which makes it slightly harder to perform matching validation. An example are the two features named '`obj`' and '`endobj`'. Kittilsen used a mismatch between the two as a pointer to if the PDF-file is malicious or not. This is because '`obj`' marks the beginning of an object while '`endobj`' marks the end of an object and therefore a mismatch between the two could mean malicious content. While still using Kittilsen's report as a point of reference for feature extraction, a mismatch of '`obj`' and '`endobj`' occurred as can be seen in figure 12. This was because the author started checking feature names by only counting features which had a block in front of them with an ASCII value for 'end-of-line' or 'space'. Having '>' next to '`endobj`' is most likely a result of a bug with LaTeX and the creation of the PDF-file, since the PDF-file states that it was created by LaTeX. However it does not change the fact that the author's algorithm missed an important feature.

Out of 2719 '`obj`' and '`endobj`',as one feature name had the ASCII value of '>' in front of the name. It is nonetheless important to take into account such kind of possibilities. In order to check if it could be a potential bug with Snort and Tcpdump, the md5 hash-sum was compared with the PDF-file downloaded through a normal web browser and the PDF-file extracted with Tcpflow. The HTTP-header was removed and a hashing of the document was performed. The documents where identical. At first the author thought of ignoring checking the block in front of the feature name, but that led to a new problem as can be seen in figure 13. While the block after '`obj`' contains a '`(`' one can clearly see that substring match is not a valid feature, but just a part of Kittilsen's keyword for a citation in the PDF-document. The example in figure 13 also shows that a false positive would be counted if Kittilsen used a citation keyword named '`/Page`' because the ASCII value of '`.`' is not accounted for. The validation check do also need to take into account of the numbers between 0 and 9. As can be discovered in figure 12 and figure 12, the feature '`obj`' is quite close to '`0`'. Since '>' is next to '`endobj`', one could argue that at some point there will be a number right next to a feature name. In this case, a number would indicate a valid match, but one can't be completely sure.

In order to check feature names without '/', one will have to check the following bytes: **<u>`[?]`</u>**`[F][e][a][t][u][r][e]`**<u>`[?]`</u>**



```
637 0 obj <<
/Type /Annot
/Subtype /Link
/Border[0 0 1]/H/I/C[0 1 0]
/Rect [289.661 178.301 302.732 187.517]
/A << /S /GoTo /D (cite.malpdfobj) >>
>> endobj
```

Figure 13: Kittilsen's keyword used for citation in his master thesis was found.

**Summarizing Feature Matching**

This subsection will summarize the rules of validation the author concluded would have to be performed in order to make sure that the substring match is a legit feature.
These values need to be checked for at the end of the feature name:

- 'End-of-line': End-of-line after matching a substring indicates end of a word and a valid match.
- 'Space': A space after matching a substring indicates end of a word and a valid match.
- '/': May indicate the beginning of a new feature name and hence ending the feature matching the substring.
- 'A-Z, a-z': Normal readable characters right after the feature name indicates that the match is a part of something else.
- '0-9': Normal numbers after the matching substring indicate the match is valid. Might need to be doubled checked later(!).

These values need to be checked for in the block in front of the feature name:

- '.': Citation keywords will have a '.' in front of the name.
- 'A-Z, a-z': Normal readable characters in front of the name indicate the match being a part of something else.
- '0-9': Normal numbers in front of the name indicate there is a valid match. Might need to be doubled checked later(!).
- '.': As seen in figure 13, a '.' in front of the feature name may indicate a citation.

The author notes that the '.' discovered in regards with citations can also be a problem if someone creates a PDF-document with LaTeX and use the following string "`\cite{/Page}`". However it is the author's belief that these cases are extremely rare and it is up to the people who want to implement the system if they want to take this into account.

### 4.3.2 Naive String Search With Validation

There are two ways to create a naive string match algorithm. One method is to check one byte in the PDF-file against all the features and the other is to check one byte in the feature against one byte in the PDF-file. The naive algorithm was first developed as a "check one byte in the pdf file against the first byte in each feature name or until a full match is found". This is a good way of decreasing the amount of block comparisons, but it makes it impossible to utilize multi-threading for each feature as explained in section 3.7.2. In order to make the feature extraction process multi-threading friendly, the code was rewritten to fit into a for-loop meaning that one byte in one feature will be matched against the PDF-file and not the other way around. Each value of the for-loop-variable represents one feature and it easier to track the performance of each feature as well as implementing multi-threading. The code example in code-list 3 shows the pseudo-code the naive search. The first pseudo-code sample shows how the algorithm is called and that it is easy to measure the time for each feature.

```
//Code–list 3
//Pseudo code for how the feature extraction process is initiated
for(each feature)
  start_time_feature_x
  for(each byte in PDF)
    feature_check(from current byte to feature length)
  end_time_feature_x
```

The next pseudo-code sample, in code-list 4, shows how the naive algorithm is developed. As can be seen of the pseudo code, the naive algorithm is quite simplistic. This will actually have an effect on the performance as will be discussed in section 5.2.

```
//Code–list 4
//Pseudo code for how the feature extraction process is performed

for(bytes in feature length)
  if(feature_name[]!=PDF[])
    return 0;

if(feature don't have '/')
  if(block in front have a–z, A–Z, 0–9, '.')
    return 0;

if(block after have a–z, A–Z, 0–9)
    return 0;
return 1;
```

### 4.3.3 Boyer-Moore-Horspool String Search With Validation

There is "only" one way to perform this algorithm compared to the naive method. Due to the way the BMH algorithm works it is impossible to check one byte in the PDF-file against one byte in each feature. This is because of the algorithm, which skips bytes if a mismatch between the PDF-string and the substring occurs, and it is self-evident that it won't work when different features have different lengths. Even with features of the same length, it would cause trouble because the algorithm may jump past important bytes that would have matched the other feature name. As can be seen in the pseudo-code example in section 4.3.2, the set-up, in code-list 5, is quite similar and it is easy to measure the performance of each individual feature.

```
//Code-list 5
//Pseudo code for how the feature extraction process is initiated
for(each feature)
  start_time_feature_x
    feature_check(return frequency of one feature in the PDF-file)
  end_time_feature_x
```

The second pseudo-code example, in code-list 6, shows how the BMH algorithm have been roughly implemented. If one compare the naive algorithm in section 4.3.2 with the example of BMH one can quickly discover that the implementation of BMH is far more complex than the naive counter part. A problem that may, or may not, occur is that the BMH have to account for a lot of different if()-sentences and jumping back and forth in the code. The longer (and more often) the program have to jump between code, the less efficient it will get. However one does not know if this will have an effect on a PDF-file of x size and at which size a naive algorithm will be equally efficient as the BMH algorithm. The efficiency question in regards to the size goes for both the size of the PDF-file and the length of the feature being searched for.

```
//Code-list 6
//Pseudo code for how the feature extraction process is performed
  for(every byte in PDF-file)
    if(last byte does not match)
      for(all unique characters in substring)
        if(match)
          jump x bytes according to table
          exit for()
      if(no match)
        jump size of substring
    else (last byte do match)
      for(every byte in substring, backwards)
        if(match)
          move backwards one byte
        else (no match)
          for(unique characters in substring)
            if(match)
              jump x bytes according to table
              exit for()
          if(no unique match)
            jump size of substring
        if(verified match)
          check byte in front and after substring match
          if(all good)
            total count of feature +1
          else (not the substring I was looking for)
            jump size of feature length
return the total count
```

As one can see of the pseudo-code of the BMH-algorithm, the author predict that the last byte in the substring does not match a given byte in the PDF-string. This is an optimization question where one want to organise the code where the content likely to be executed the most close to the if()-statement, while the content going to be executed the least would be placed at the bottom. The author notes that this isn't a normal string search through a normal book or text a database. The raw PDF-file contains bytes with a wider range of ASCII values than the normal alphabet with the addition of the numbers zero to nine. Therefore the likelihood of the last byte in the substring matching a byte in the PDF-string is smaller. If a match of the last ASCII value occurs, the algorithm will go to second last character in the substring and perform a comparison. It will continue that way unless a mismatch occurs. If a mismatch doesn't occur, the algorithm will jump to the validation checks. If a valid feature is found, the algorithm will count the feature match and continue further into the PDF-file. However if a mismatch occur, the algorithm will check if the character in the PDF-file matches a character in the substring. If a match is found, then the algorithm will jump a predefined length. If the character does not exist in the substring, the algorithm will jump forward the same length as the size of the substring.

## 4.4 Support Vector Machine

This section will explain how the author utilised support vector machine with LibSVM.

### 4.4.1 SVM Training

Before one can classify PDF-files, one needs a SVM-model to base the decisions on. This operation is done prior to the detection taking place, so the time spent building the model will not be an issue. However what will be an issue is what kind of foundation one is basing the SVM-model on. The best case scenario would be to have only benign PDF-files in the benign class and malicious PDF-files in the other. The malicious PDF-files have been determined to be malicious by Jarle Kittilsen or Kittilsen's sources, but the "benign" ones have been gathered through wget [2].

> *Wget is a free utility for non-interactive download of files from the Web* [48].

As can be seen in appendix D and noted in section 5.4, there have been malicious PDF-files in the benign set of PDF-files Kittilsen built his SVM model on. The author does not have the time to manually check all of the "benign" PDF-files to make sure that they are actually benign, but it is important to note that the foundation of which one build the SVM-model should be clearly separable i.e. no malicious files in the benign PDF-set. It is also important to note that while JavaScript may be a strong indication of malicious PDF-file [13], it won't do any good if none of the malicious PDF-files contain any JavaScript. Add to the fact that half of the benign dataset could contain JavaScript, which would cause the SVM model to become skewed.

The following features are used to build the SVM model.

- /Page
- /AcroForm
- /OpenAction and /AA
- /RichMedia:
- /Launch:
- /JavaScript and /JS
- startxref
- trailer
- The mismatch between 'obj' and 'endobj'

The SVM-model itself is built by using Jarle Kittilsen's settings for the most successful experiment [2]. However LibSVM is utilised for its C-code implementation instead of using the Python counterpart called PyML [49]. The parameters for LibSVM are c-svc (which is also the default option in LibSVM), the kernel type is 'radial basis function' (Gaussian kernel), $\gamma$ is 0.1 (gamma) and the cost is 100. The author utilised Weka [50] in order to perform the model experiments and the results are shown in table 2 and table 3 . Appendix F shows the Weka GUI after performing a 10-fold cross-validation and by using the entire dataset for classification. The SVM-model itself, which the author used for the classification of PDF-files, was built by LibSVM with no involvement from Weka.

| Classified as ————> | **Benign** | **Malicious** |
|---|---|---|
| **Benign** | 7418 | 36 |
| **Malicious** | 94 | 16186 |

Table 2: Weka and LibSVM result by using cross-validation (10-fold)

- Correctly Classified Instances: 99,45%
- Precision: 0,998
- Recall: 0,994
- F-Measure: 0,996
- ROC Area: 0,995

| Classified as ————> | **Benign** | **Malicious** |
|---|---|---|
| **Benign** | 7441 | 13 |
| **Malicious** | 69 | 16211 |

Table 3: Weka and LibSVM result by using the entire training set

- Correctly Classified Instances: 99,65%
- Precision: 0,999
- Recall: 0,996
- F-Measure: 0,997
- ROC Area: 0,997

### 4.4.2 SVM Classification

After the SVM-model has been created it is time to start classifying PDF-files. LibSVM work by sending one file containing the features of the PDF-file one wants to be classified and the SVM-model itself. In order for LibSVM to classify the PDF-file, one needs to "classify" the PDF-file first

by adding '`1`' or '`-1`' in front of the list of features in the input file. The author has defined benign files as '`1`' while malicious files are classed as '`-1`'. If one "guesses" correctly, the LibSVM will output: `Accuracy = 100% (1/1) (classification)`
This means that if the author assumes all files captured are benign, then a malicious file will be outputted as: `Accuracy = 0% (0/1) (classification)`
This allows the author's program to distinguish between feedback and log the captured files in the correct folder (benign/malicious) for post-analysis. LibSVM is suited for post-classification because LibSVM needs to read and interpreted the SVM-model for each classification. The best scenario would be to have LibSVM running and send a feature file from new PDF-files as they are logged by Snort.

## 4.5 Overview of the Author's Current System

This section will explain how the author's detection system is working in its current form. Section 6.3 shows how the author's intended system was supposed to be.

### 4.5.1 Limited by Scope of the Master Thesis

Due to the time limitations, the author's original intended detection system could not be installed. An overview of the author's intended detection system can be found in section 6.3. Figure 14 shows how the system currently works as of the time the master thesis report is written.

The grey square defines the computer on which Snort and the author's software is installed and the network traffic in and out of the "computer" displays two network cards. Figure 14 contain numbers displayed near the arrow lines and these are used to document the flow and processes of the author's system.

The system works as follows:

1. Network traffic entering the computer is being monitored by Snort. Snort does not modify the network traffic.
2. When Snort detects a PDF-file in a network session, an alarm is sent to an Unix socket.
3. While the network packets containing the PDF-file is passing through Snort, Snort logs all of the network packets for the specific network session to storage medium of the user's choice.
4. When an '`%%EOF`' passes through Snort, Snort sends an alarm to the Unix socket stating that then of the PDF-file have been found. Program 1 is detecting this alarm.
5. Program 1, which is dedicated to listening for alarms, just detected Snort's alarm and tells Tcpflow to extract the PDF-file from Tcpdump's log-file.
6. When the Tcpflow-process have finished, Program 1 will call Program 2 which will start the feature extraction and classification process.
7. Program 2 reads the PDF-file from the storage medium, extracts features and starts the classification process.
8. After the PDF-file have been classified, the PDF-file will be stored in either a 'benign' or 'malicious' folder as well as important data is written in a log file for an operator to use when examining the PDF-file later.

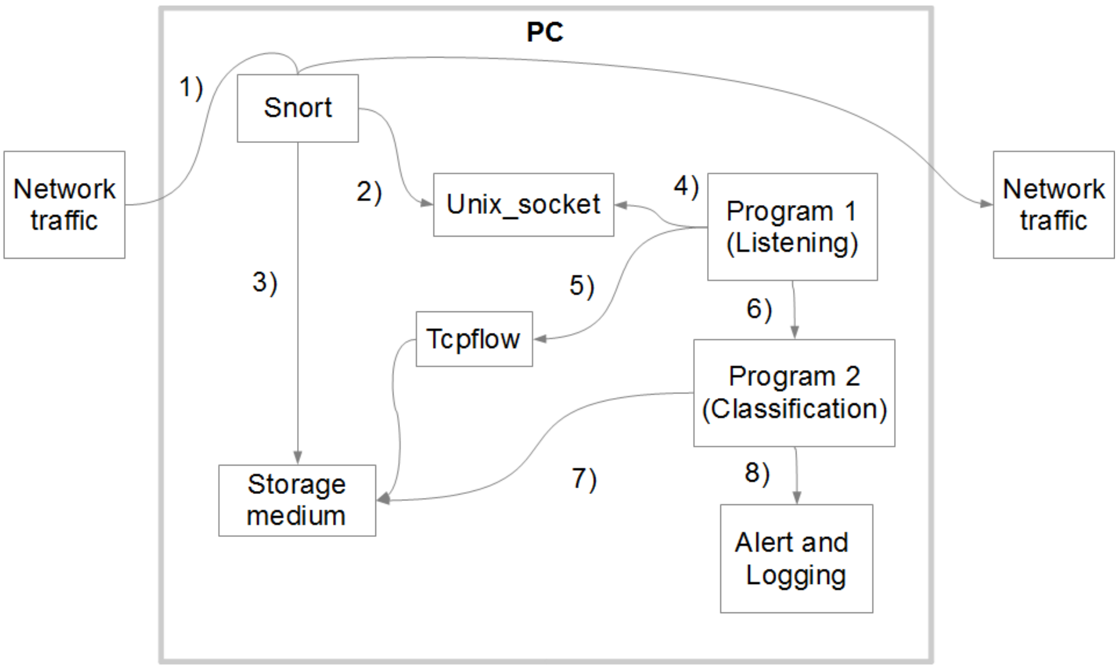


Figure 14: Brief documentation of how the author’s detection system works

# 5 Testing and Results

This chapter contains result of the different tests performed ranging from extracting the PDF-file, extracting features from a PDF-file, classification using SVM and scanning suspicious files found during the master thesis period.

The development and testing was mainly performed on this computer set-up:

- Ubuntu 12.10 (32-bit).
- VmWare workstation 9.0.0 build-812388.
- Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz
- 2x4GB DDR3 RAM
- Seagate 750GB, Model number: ST9750420AS

A test computer provided by NISLab [51], without VmWare, had the following set-up:

- Ubuntu 12.10 (32-bit).
- Intel core: Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz
- 2x2GB DDR3 RAM
- Seagate 250GB, Model number ST3250310AS

This virtual computer was created in order to scan specific PDF-files with different anti-virus software.

- MS Windows 7 Pro(32-bit).
- VmWare workstation 9.0.0 build-812388.
- Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz
- 2x4GB DDR3 RAM
- Seagate 750GB, Model number: ST9750420AS

The following third party applications were used:

- Snort v.2.9.3 [43]
- LibSVM v.3.16 [39]
- Weka v.3.6.9 [50]

The testing of the feature extraction process were performed on a large dataset of PDF-files and the author have been using Jarle Kittilsen's PDF-corpus of 7454 benign PDF-files and 16280 malicious PDF-files [2].

## 5.1 Snort, Preparation of the PDF-file and Storage Devices

This section will explain the experiments performed with Snort and the extraction of the PDF-file from the log-file of Tcpdump. The tests are performed in regards to the speed of the different storage devices.

### 5.1.1 Setup and Preparation

The experiments have to be executed on the same storage device several times in order to get an accurate result. The author repeated the experiments ten times and the calculated the average time spent. In order to get Snort to log network sessions to the hard drive, the default logging location was chosen. The default logging location for Snort on Ubuntu is '`/var/log/snort/`'. In order to get Snort to log to a ramdisk one can either tell Snort to log to '`/dev/shm/`' [52] or one can create a custom ramdisk by entering the following commands [46]:

1. `mkdir -p /tmp/ram`
2. `sudo mount -t tmpfs -o size=512M tmpfs /tmp/ram/`

The time is measured from when Snort detects the PDF-footer until Tcpflow have finished extracting the PDF-file. The tests will show if the difference between utilising a regular hard drive and a ramdisk is significant enough. In order to get correct time measurements, the computer was restarted to ensure only a minimum of processes running at the same time. The time was recorded with the '`gettimeofday()`'-function.

### 5.1.2 Results

Table 4 and 5 shows the time from the first network packet arrives, containing the PDF-footer, to when Tcpflow have finished extracting the PDF-file. Table 4 shows the time spent for what the author thought was the physical hard drive, while table 5 shows the time spent for a ramdisk. The different PDF-files are:

- PDF1: adobe_supplement_iso32000_1.pdf (324KB) [53]
- PDF2: jarle_kittilisen.pdf (2,3MB) [2]
- PDF3: PDF32000_2008.pdf (9MB)[54]

| | **PDF1** | **PDF2** | **PDF3** |
|---|---|---|---|
| **Size** | 324KB | 2,3MB | 9MB |
| **Time (ms)** | 17 | 17 | 19 |

Table 4: Time measurements with VmWare's hard drive

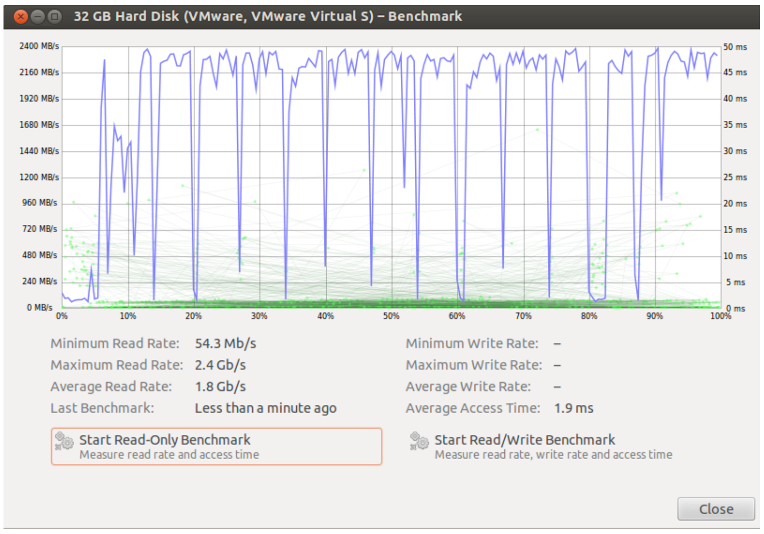
The time is quite similar with both the ramdisk and the "hard drive". The author first thought this was because of the network packets getting stored in the hard drive's disk cache. A disk

| | PDF1 | PDF2 | PDF3 |
|---|---|---|---|
| **Size** | 324 KB | 2,3MB | 9MB |
| **Time (ms)** | 16 | 16 | 19 |

Table 5: Time measurements with VmWare's ramdisk

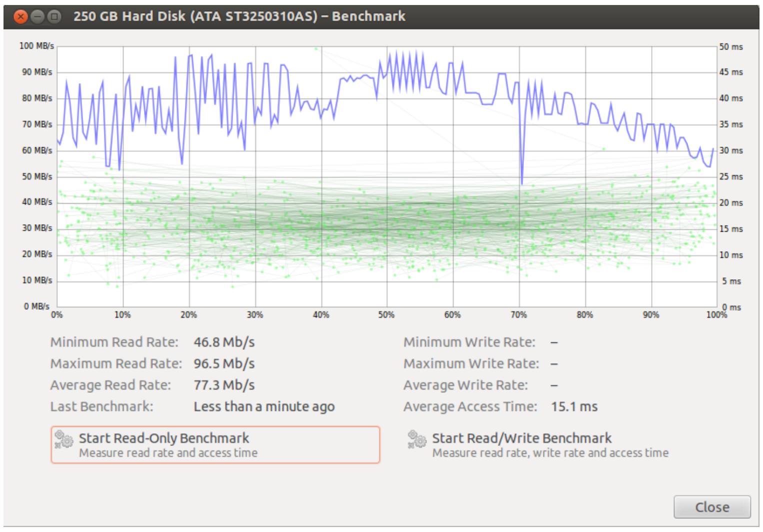
cache is a part of a hard drive where data is stored temporarily for faster access time [55]. This is quite useful for smaller pieces of data that is often used by the computer. However the disk cache will have no effect when streaming large video files etc. Because the PDF-files are so small and the Tcpflow-process' effectiveness is dictated by the CPU, the difference in speed gain between the disk buffer and the ramdisk could be negligible. The speed measured for both tests ranged from 12 seconds up to 25 seconds at most. Overall the time spikes were the same for each file regardless of storage device.

The author didn't believe that same results would be achieved for both a hard drive and a ramdisk. In order to make sure that the hard drive results were valid, a reading speed benchmark of the hard drive was performed as can bee seen in figure 15. The benchmark provided an interesting result as one can see that the maximum read speed have been measured to 2,4GB/s! The speed is not consistent as it varies between 54Mb/s and 2,4Gb/s. The author believes this is caused by VMware preparing the section of the hard drive the benchmark test is about to measure and that section gets placed in RAM. This means that the benchmark test doesn't measure the speed against the physical hard drive, but rather the section of the hard drive that is placed in RAM, which in turn have a much higher read speed. Another hard drive benchmark were performed on the computer provided by NISLab and the result can be seen in figure 16. Figure 16 shows a result one would expect from a physical drive with its maximum read speed on 96,5Mb/s and an average access time on 15,1ms.


32 GB Hard Disk (VMware, VMware Virtual S) – Benchmark
2400 MB/s
2160 MB/s
1920 MB/s
1680 MB/s
1440 MB/s
1200 MB/s
960 MB/s
720 MB/s
480 MB/s
240 MB/s
0 MB/s
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
50 ms
45 ms
40 ms
35 ms
30 ms
25 ms
20 ms
15 ms
10 ms
5 ms
0 ms
Start Read-Only Benchmark
Measure read rate and access time
Start Read/Write Benchmark
Measure read rate, write rate and access time
Close

Minimum Read Rate: 54.3 Mb/s
Maximum Read Rate: 2.4 Gb/s
Average Read Rate: 1.8 Gb/s
Last Benchmark: Less than a minute ago
Minimum Write Rate: –
Maximum Write Rate: –
Average Write Rate: –
Average Access Time: 1.9 ms

Figure 15: Hard drive benchmark of VMware disk


250 GB Hard Disk (ATA ST3250310AS) – Benchmark
100 MB/s
90 MB/s
80 MB/s
70 MB/s
60 MB/s
50 MB/s
40 MB/s
30 MB/s
20 MB/s
10 MB/s
0 MB/s
50 ms
45 ms
40 ms
35 ms
30 ms
25 ms
20 ms
15 ms
10 ms
5 ms
0 ms
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Start Read-Only Benchmark
Measure read rate and access time
Start Read/Write Benchmark
Measure read rate, write rate and access time
Close

Minimum Read Rate: 46.8 Mb/s
Maximum Read Rate: 96.5 Mb/s
Average Read Rate: 77.3 Mb/s
Last Benchmark: Less than a minute ago
Minimum Write Rate: –
Maximum Write Rate: –
Average Write Rate: –
Average Access Time: 15.1 ms

Figure 16: Benchmark of a regular hard drive, without VMware

The author decided to perform a new test on the computer that NISLab had provided in order to get correct measurements of a physical drive as well as the SSD performance as the author's laptop only had one normal hard drive at its disposal. However problems occurred when the author tried to implement parts of the PDF-detection system on NISLab's lab computer. The problem is related to how Snort sends alerts to the Unix socket. The author used the same detection rules as was developed in the virtual machine. However while Snort manages to detect both the PDF-header and PDF-footer in the virtual machine, Snort was only able to detect the PDF-footer in the PDF-file. The author made sure that Snort was compiled from the same source code that was used for the virtual machine and that the additional tools like DAQ [56], libpcap etc. had the same version number. Yet Snort refused to detect the PDF-header. The author tried to tweak the detection rules, which led to Snort being able to detect the PDF-header. However while Snort was able to detect the PDF-header, Snort would not be able to log the network session that contained the PDF-file. One could either get Snort to detect the PDF-header and not log the network session or one could keep the rule about Snort logging the network session and end up with Snort not "detecting" the PDF-header at all. The author don't know exactly what is causing this problem, but decided to leave the problem be in order to focus on more important tasks. The author's supervisor can however attest that the PDF-detection system worked on the virtual machine.

## 5.2 Experiments With Feature Extraction

This section will explain how the performance of the feature extraction process was performed as well as the result from both the naive- and the Boyer-Moore-Horspool algorithm.

### 5.2.1 Setup and Preparation

There are two ways of measuring the performance of the different searching algorithms and one of those are measuring the amount of if()-sentences the algorithm have to execute in order to extract all the features from the PDF-file while the other is the time it takes for the algorithm from to finish. Note that the author is aware of profiling tools for compiling C-code like Gprof [57][58], but the author wants to make sure that only wanted information is recorded (i.e. specific if()-sentences, time measurements across $x$ lines of code etc.). In order to count the number of if()-sentences the author had to add a variable in the source code to count each time an if()-sentence was executed. The author had to make sure to add *else* statements to every if()-sentence because the author wanted to know how many if()-sentences were executed in total and not just how many if()-sentences were executed when a an if()-statement was true. An example of the pseudo-code can be seen in code-list 7:

```
 1 //Code-list 7
 2 //if()-sentence counting
 3 if(argument==OK)
 4 {
 5   Do something...
 6   if_count++
 7 }
 8 else
 9 {
10   if_count++
11 }
12 //could have had if_count++ here instead.
```

The other reason why the author chose to count if()-comparisons by adding additional *else*-statements and not at not after the code executed by the if()-sentences is because the algorithm often jumps back and forth, which is especially the case with the BMH algorithm. Instead of counting the if()-sentences at line 12 in the pseudo-code sample, the author wanted to make sure that the algorithm counted all of the if()-sentences correctly.

The time was measured for each individual feature and then the mean was calculated based on how many iterations the feature was searched for. The author decided to let the algorithm iterate the search for each individual feature ten times. The time is displayed in milliseconds and is recorded by the '`gettimeofday()`'-function. The reason for choosing the '`gettimeofday()`'-function instead of the '`clock()`'-function is because the author wants to see how much time the feature extraction takes with normal processes running and not the time isolated to one process. In order to make sure no unnecessary processes were running during the experiments, the author restarted Ubuntu before the experiments were executed. It is also important to note that the author performed experiments on the algorithms when they counted if()-sentences executed and measured the time separately.

The PDF-files used in the result section are the following:

- adobe_supplement_iso32000_1.pdf [53]
- jarle_kittilisen.pdf [2]
- PDF32000_2008.pdf [54]
- pdf_reference_1-7.pdf [8]
- adobe_supplement_iso32000.pdf [59]
- ScientificMethodology.pdf

The ScientificMethodology.pdf-file was given to the author by a student at GUC for experimental purposes. The PDF-file consisted of images and had a size of 171MB.

### 5.2.2 Results

The author have picked several different sized PDF-files for testing purposes and the results are shown in the table 6, 7 and 8. The tables showcases some of the features that have been extracted from the PDF-file. The "total" count at the end of each table is for the total amount of features and not only the features shown in the table. The author chose to show one of each kind of feature with different attributes. The features have been chosen by the following metrics:

- With or without '`/`' in front of the feature name, meaning '`/AA`' and '`obj`'.
- The length of the feature name ranging from short, medium and long length. Shortest length being '`/AA`' or '`obj`', medium lenght being '`/Page`' or '`trailer`' and the long length being '`/AcroForm`'.

The if()-column shows how many times if()-sentences have been executed during the extraction of one specific feature and the entire result from the naive- and BMH algorithm can found in appendix A and B.

The first PDF-file the author want to perform the feature extraction is Jarle Kittilsen's master thesis report[2] and the results are shown in table 6. The different graphs displaying the results can be found in figure 17 and figure 18.

Kittilsen's report have a file size of 2.263.642 bytes.

| | **Naive if()** | **Naive time** | **BMH if()** | **BMH time** |
|---|---|---|---|---|
| **/AcroForm** | 2.303.326 | 5 | 520.515 | 5 |
| **/Page** | 2.303.711 | 5 | 938.606 | 7 |
| **/AA** | 2.303.326 | 5 | 1.524.714 | 8 |
| **trailer** | 6.805.628 | 18 | 672.720 | 7 |
| **obj** | 6.823.848 | 19 | 1.547.819 | 9 |
| **Total** | 45.686.110 | 113 | 10.066.067 | 83 |

Table 6: Time and if()-sentences executed for the PDF-file: jarle_kittilsen.pdf

A notable difference that can be spotted in table 6 is the difference between the two kinds of features which have been extracted with the naive algorithm. Features starting with '/' have a

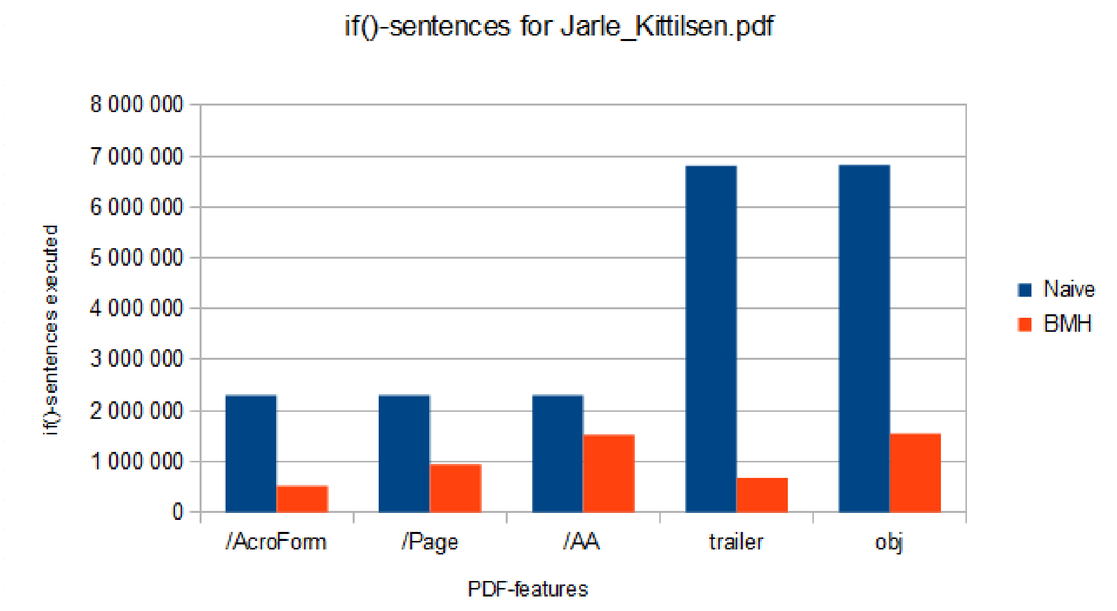


Figure 17: Graph showing difference between if()-sentences executed by the naive- and BMH algorithm for the PDF-file: jarle_kittilsen.pdf 6

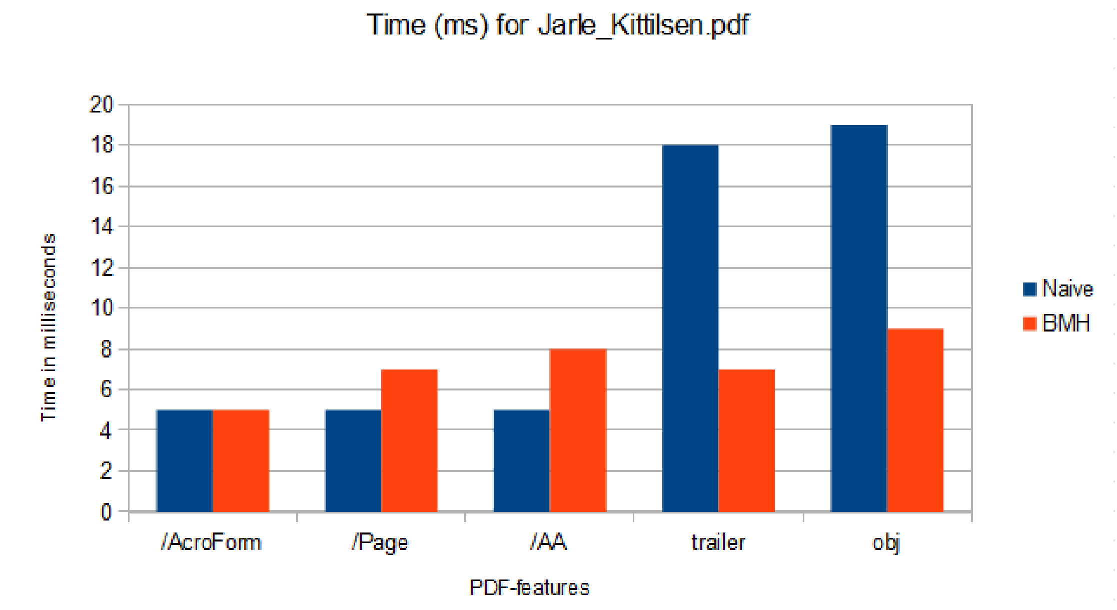


Figure 18: Graph showing the time difference between the naive- and BMH algorithm for the PDF-file: jarle_kittilsen.pdf 6

similar if()-sentence execution to the size of the PDF-file in question, while the none '/' features have a huge increase in the amount of if()-sentence executed and the time measured. The difference is about three times which is quite a significant difference compared to the Boyer-Moore-Horspool(BMH) algorithm as the results are quite similar across the board. The author believes the reason for a such significant difference is because the '/' is quite rare in the PDF-file and that characters like '`o`' and '`t`' occurs more frequently. However the results within the same kind of feature is quite similar, meaning '`/Page`', '`/AcroForm`' and '`/AA`' have roughly the same amount of if()-sentences executed. The length of the feature name in general seems to not cause any significant difference when performing a naive search. The author believe this is because the naive algorithm is matching one and one byte from left to right, where as the BMH algorithm benefits from the longer the substring to be matched is. Though the BMH algorithm have a higher variance for if()-sentences executed, in return it also have executed a significant lesser amount of if()-sentences than its naive counterpart. The BMH time for each feature is similar across the board, while the naive algorithm experience a huge spike for '`obj`', '`trailer`' and similar named features by about three times then features starting with a '/'. The reason why the naive algorithm spends a total of 113 milliseconds while BMH only spends 83 milliseconds is because the author did not list all of the features in the table. The author notes that there are an additional two non '/' features which is causing the naive algorithm to be a lot slower.

The next PDF-file is "PDF32000_2008.pdf" from Adobe's PDF specification page [54] and the results are shown in table 7. The different graphs showing the results can be found in figure 19 and figure 20. The file size of the PDF-file is 8.995.189 bytes.

| | **Naive if()** | **Naive time** | **BMH if()** | **BMH time** |
|---|---:|---:|---:|---:|
| **/Acroform** | 9.218.579 | 21 | 2.102.873 | 22 |
| **/Page** | 9.218.587 | 20 | 3.766.206 | 32 |
| **/AA** | 9.218.945 | 20 | 6.138.309 | 34 |
| **trailer** | 27.074.868 | 71 | 2.709.435 | 29 |
| **obj** | 27.078.928 | 72 | 6.121.530 | 34 |
| **Total** | 182.052.133 | 448 | 40.335.141 | 349 |

Table 7: Time and if()-sentences executed for the PDF-file: PDF32000_2008.pdf

As shown in table 7 the result quite look similar with the previous PDF-file in table 6. The amount of if()-sentences between features which start with and without '/' are about three times when performing a naive search and the total amount of if()-sentences executed between the naive and the BMH algorithm is about five times. This is roughly the same result as can be seen in table 6. An interesting thing to point out is that some of the features are more time exhaustive when searched by the BMH algorithm than if the naive-algorithm is used. This is the case for features like '`/Page`', '`/AA`' and '`/AcroForm`'. This is strange because BMH clearly executes fewer if()-sentences than the naive algorithm. The author believe the reason why the naive algorithm performs better in some cases is because the BMH algorithm is quite complex. At one point the BMH algorithm have a "tree" of six for-loops/if()-sentences as can be seen in the pseudo-code in section 4.3.3. When an algorithm contains a lot of for-loops and if-sentences (i.e. becomes more complex) like the BMH implementation, it causes the program to jump a lot between code-lines. This is one of the reasons for why one should always organise the if()-segments like the pseudo-code sample "Tips to organise if()-statements":

```
//Tips to organise if()—statements
if(argument match)
  {This content will be executed the most and is therefore placed here.}
else if(argument match)
  {This content is not executed that often and should therefore be placed here.}
else
  {This content is executed the least and should therefore be placed here.}
```

The author notes that the amount of code to be executed if an if()-statement match can also be taken into consideration when organising the order of the if()-statements.

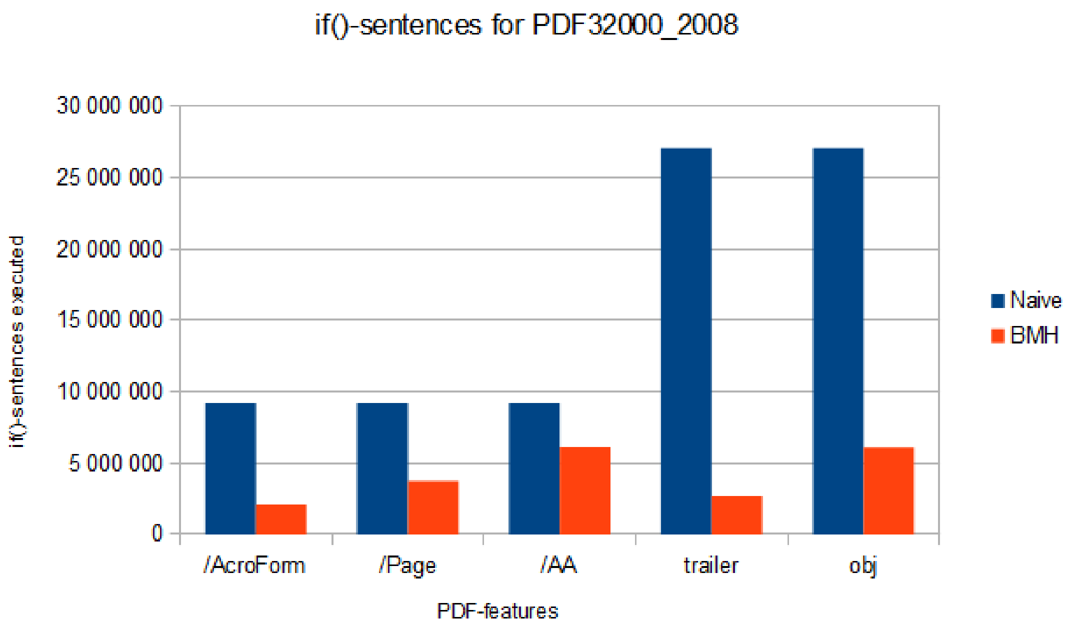


Figure 19: Graph showing difference between if()-sentences executed by the naive- and BMH algorithm for the PDF-file: PDF32000_2008.pdf 7

The complexity of the implemented algorithm explains why the BMH algorithm may be more time consuming process than the naive algorithm. However because some of the features do not contain a '/' in the beginning of the feature name, the BMH algorithm will still be more efficient overall as the total time difference between the two algorithms are 99 milliseconds in table 7. This is backed up by the fact that for both the total result of if-sentence executed in table 6 and table 7, the difference between BMH and the naive algorithm are about 4,5 times. Yet the naive algorithm only spends about 1,2 times the time then what the BMH algorithm does.

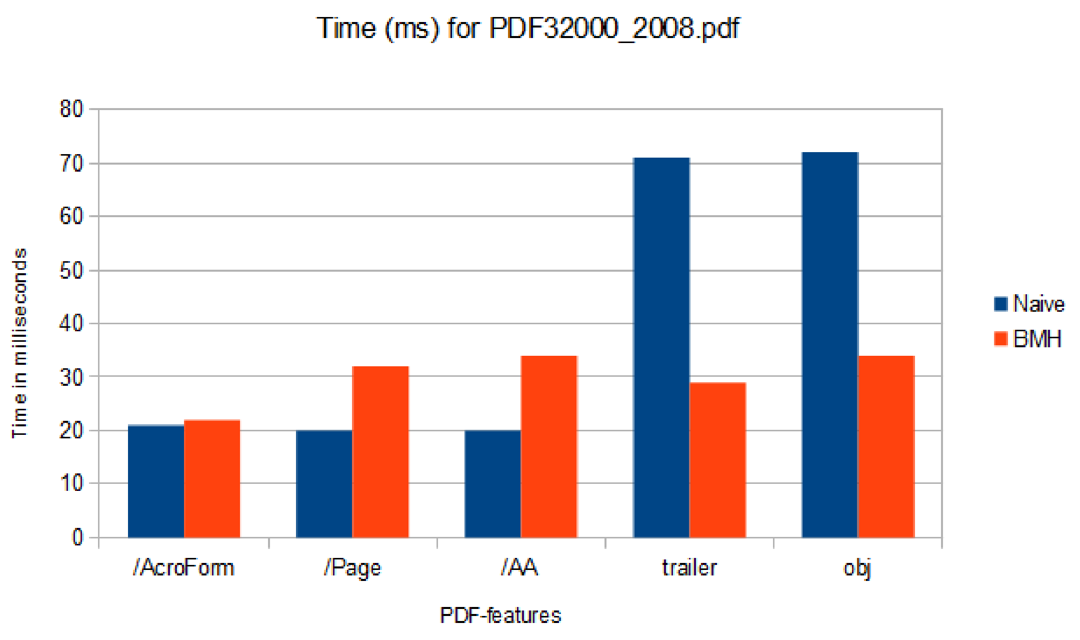


Figure 20: Graph showing the time difference between the naive- and BMH algorithm for the PDF-file: PDF32000_2008.pdf 7

The last PDF-file is the Adobe's reference guide for the PDF-format, named "pdf_reference_1-7.pdf", and contain 32.472.771 bytes [8]. The results can be found in table 8. The different graphs displaying the results can be found in figure 21 and figure 22.

| | **Naive if()** | **Naive time** | **BMH if()** | **BMH time** |
|---|---:|---:|---:|---:|
| **/Acroform** | 33.854.245 | 76 | 7.478.539 | 80 |
| **/Page** | 33.798.997 | 74 | 13.727.950 | 109 |
| **/AA** | 33.854.643 | 74 | 22.043.295 | 130 |
| **trailer** | 97.805.684 | 259 | 9.853.221 | 103 |
| **obj** | 98.752.653 | 263 | 22.848.465 | 125 |
| **Total** | 663.313.317 | 1.642 | 148.378.208 | 1.237 |

Table 8: Time and if()-sentences executed for the PDF-file: pdf_reference_1-7.pdf

The results in table 8 reflects the tendency from the previous PDF-files:

- Total if()-sentences executed for the naive algorithm is about 4,5 times as many as the BMH algorithm and yet the naive algorithm is only 1,2 times slower than BMH.
- The amount of if()-sentences executed are stable throughout the naive feature search, while BMH have a greater variance which is affected by the length of the feature name.
- The final result of the naive-algorithm is affected by how many features start with '/' and non '/', while the BMH algorithm is only affected by the length of the feature name in general.
- For feature names with '/', the naive algorithm is currently more efficient than the BMH algorithm.

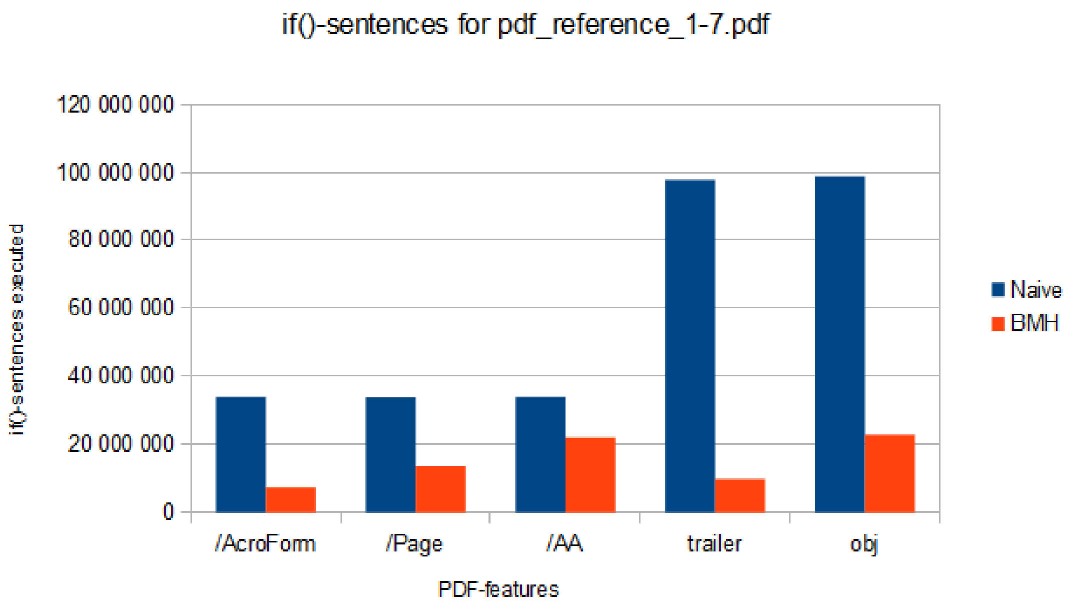


Figure 21: Graph showing difference between if()-sentences executed by the naive- and BMH algorithm for the PDF-file: pdf_reference_1-7.pdf 8

The author is using the term "efficiency" about how much time an algorithm is using from the start until every feature has been counted successfully. One could argue that the naive algorithm is more efficient than the BMH algorithm because the naive algorithm can process more if-sentences per time unit than the BMH algorithm. This is backed up by the fact that if it weren't for the features without '/', the naive algorithm would potentially have been the best algorithm of the two. The author notes that the even though the naive algorithm have beaten the BMH algorithm in some cases, the author would recommend trying other searching algorithms like KMP in order to see if the naive algorithm is still the better option due to its simplicity. Scientific papers regarding new and improved searching algorithms are often used to scan through normal text and/or databases. The PDF-file is a different scenario because it contains extra conditions like:

- Validation checks to validate a feature.
- '/' in the beginning of the feature name.
- Bytes with a larger range of values than the standard a-z, A-Z and 0-9 (i.e. a larger distribution of ASCII values which means normal characters like 's' and 'e' occur less frequently than in a text file with the same size of the PDF-file.

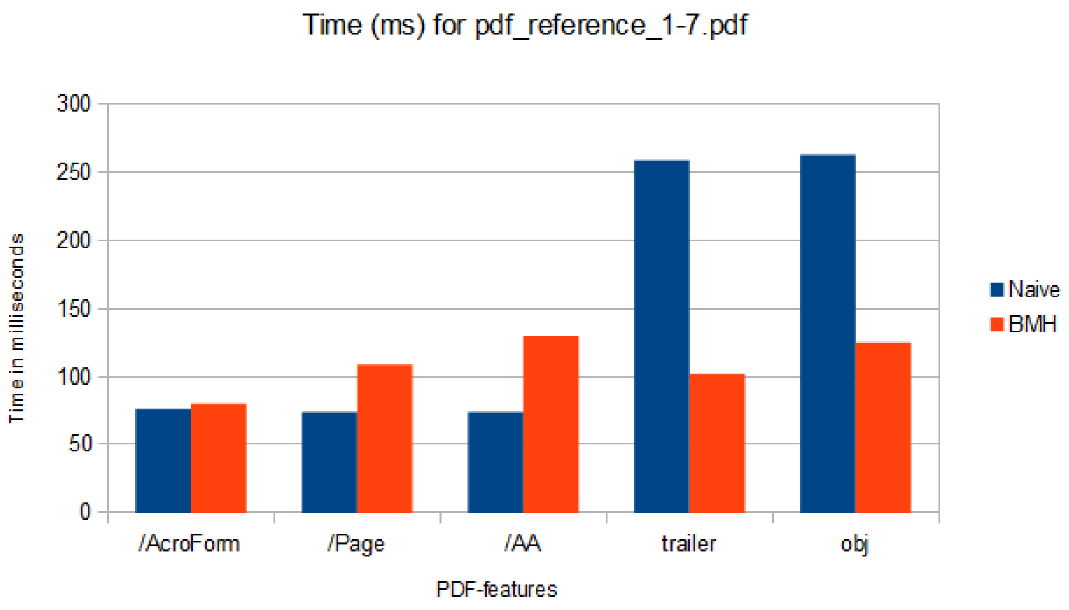


Figure 22: Graph showing the time difference between the naive- and BMH algorithm for the PDF-file: pdf_reference_1-7.pdf 8

The last table, which is table 9, shows the difference between the total amount of time spent for the naive algorithm, the BMH algorithm and Jarle Kittilsen's python script. Kittilsen's result is gathered from appendix C. The difference is quite significant were Kittilisen's script uses roughly 73 seconds in order to extract features from his own thesis report. There is however a good reason as to why Kittilsen's Python script is so slow. Kittilsen is parsing the PDF-document and then counting the feature matches, while the author is only performing a normal string search. Another question that arise is why a 2,3MB PDF-file uses significantly more time to process than a PDF-file of 171MB as can be seen in table 9. The 171MB PDF-file was given to the author by student at Gjøvik University College and it is a PDF-file consisting of scanned papers with notes and other things. The PDF-file basically only consist of scanned images which makes sense performance wise. The reason being that Kittilsen's master thesis report have a more complex structure than the PDF-file consisting of images only. The master thesis report contains citations, bibliography, table/figure references etc. It makes sense that a PDF-file with a complex document structure is more time exhaustive to parse than parsing a file only containing images. The time spent where only a normal string search is performed, are roughly linear in regards to the size of the PDF-file. The author notes that the large variation with Kittilsen's Python script is not a result of having other CPU intensive processes running in the background. The processing times have also been consistent over ten times.

| PDF-files \Time | Naive time | BMH time | Python script |
|---|---|---|---|
| **jarle_kittilsen.pdf** (2,3MB) | 113 | 80 | 73.359 |
| **pdf_reference_1-7.pdf** (32,5MB) | 1.642 | 1.212 | 36.409 |
| **ScientificMethodology.pdf** (171MB) | 8.138 | 6.263 | 51.380 |
| **adobe_supplement_iso32000.pdf** (1,3MB) | 68 | 49 | 2.681 |
| **PDF32000_2008.pdf** (9MB) | 448 | 329 | 18.164 |
| **adobe_supplement_iso32000_1.pdf** (0,3MB) | 20 | 12 | 3.976 |

Table 9: The time, in milliseconds, spent to extract features with the naive-, BMH algorithm and Kittilsen's Python script

## 5.3 SVM Classification

This section will explain how the experiments with LibSVM was performed and the results of the classification process.

### 5.3.1 Setup and Preparation

The author have implemented the classification process in two programs. One of the programs is the feature extraction program based on the BMH algorithm (for testing purposes) and the other program is a part of the detection system. The time was recorded with the '`gettimeofday()`'-function and is started right before the classification command is initiated and ends after the classification process have finished. The classification process do not require any large files which means the reward by using an SSD or ramdisk decreases. In order to get a correct measurement of the time the process spends, the author rebooted the computer in order to close any unnecessary processes.

### 5.3.2 Results

This procedure have been tested on both the set of benign PDF-files as well as the malicious ones. The time spent from starting the classifying process have been consistent on both sets. As can be seen in table 10, the time for classifying a PDF-file is 6 milliseconds. The classifying time have been consistent for all of the 23.734 PDF-files, which is logical because the time spent on classifying a PDF-file is not dependent on the size of the PDF-file or how fast the feature extraction process is. The data LibSVM have to process is quite similar every time it performs a classification, where the variance is only a about five to ten bytes depending on how many digits the different features have to represented by. If one wanted to save additional time during the classification process, LibSVM would have to pre-process the SVM-model and then go in a "loop" to accept the PDF-feature file and then perform the classification. However a constant time on 6 milliseconds is not bad either because there are other and more time exhaustive processes that needs more attention which is the feature extraction process.

| PDF-files | Classification time |
|---|---|
| **PDF32000_2008.pdf** (9MB) | 6 |
| **adobe_supplement_iso32000_1.pdf** (0,3MB) | 6 |
| **jarle_kittilsen.pdf** (2,3MB) | 6 |
| **ScientificMethodology.pdf** (171MB) | 6 |
| **adobe_supplement_iso32000.pdf** (1,3MB) | 6 |
| **pdf_reference_1-7.pdf** (32,5MB) | 6 |

Table 10: Classification time for different sized PDF-files.

## 5.4 Anti-Virus Scan of Suspicious Files

During the the capturing the PDF-files with Snort and the development of the feature extraction algorithms, the author discovered four PDF-files within Kittilsen's benign data set that had a strong indication of being malicious. These files had suspicious content before the PDF-header and this is a potential form of attack according to Symantec [45]. The Adobe specification states:

> *Acrobat viewers require only that the header appear somewhere within the first 1024 bytes of the file* [8].

Screen shots of the PDF-files with suspicious content can be found in appendix D. Since Kittilsen scanned these files with different anti-virus solutions in 2011, the author decided to scan the four PDF-files again with updated virus definitions (1/5-13). The author used the same three anti-virus solutions Kittilsen used in 2011, with the addition of two new ones. The following anti-virus solutions were used:

- MS security essentials [60]
- Trend Micro [61]
- AVG [62]
- Norton Internet Security [63]
- Norman [64]

| PDF-files/Anti-virus | Norton | Trend Micro | Norman | AVG | MS Security Essentials |
|---|---|---|---|---|---|
| 31f1d83ae496b204f72b9a638f307273 | No | No | No | No | No |
| 49cebd3c7c3c268984c7ca7b2de94463 | No | No | No | No | No |
| afdf0e2247db263978dc3bfffbc0ba7b | No | No | No | No | No |
| cf3b09c09c7ecf95bd86ed1fb282da40 | No | No | No | No | No |

Table 11: Anti-virus scan of suspicious PDF-files. No= Benign, Yes=Malicious.

The result of the scan is shown in table 11. The author got quite surprised by the result, but understand that anti-virus software have to catch several other types of files and attacks which means some malicious files of a given file type will slip through the cracks. As the white paper from Symantec explained [45], anti-virus solution may experience performance issues if they only scan the first four bytes in order to find a file-header. The author can however not with certainty state that this is the case with these four PDF-files. The fact remains that the Adobe specification do allow for related or unrelated bytes to be in front of the PDF-header as can clearly be seen in figure 7 in section 4.1 where the author was able to open the PDF-file in a PDF-reader despite the HTTP-header information.

The author notes these four PDF-files were also uploaded to Jsunpack's website [65]. No malicious PDF-files were detected and the screen shots of the results from Jsunpack's website can found in appendix E.

# 6 General Discussion

This chapter will discuss both theoretical implications, practical considerations, provide a conclusion and discuss possibilities for further work.

## 6.1 Theoretical Implications

One of the main problems is the time factor which relates to what kind of time frame is acceptable for the user to wait? The author intended to develop a method that forces the PDF-file to be buffered while the classification takes place, which means the entire PDF-file have to be "downloaded" in order for the classification to begin. The author's intended idea is explained in section 6.3.

Depending on the bandwidth available, the overall time spent may vary greatly. If time isn't an issue, meaning the time frame is small enough, there is still a problem with buffering network packets which the author will explain in section 6.2.

The theoretical problems are listed as follow:

- Bandwidth from the internet/network source. The PDF-file can't be analysed before the entire PDF-file have been logged by Snort. A three minute download results in atleast three minute processing time.
- The time to extract features varies depending on which searching algorithm is used. The naive- and the BMH algorithm shows a roughly linear increase in time spent when the size of the PDF-file increases. For a 171MB file it will take about 6 seconds with the BMH algorithm, which is shown in table 9 in section 5.2.2.
- The classification process is executed quite fast and the effectiveness is not dependent on the attributes of the PDF-files. The process is only dependent on the CPU installed and the time have been consistently been measured to 6 milliseconds.
- The SVM classification is based on a model which is based on features selected by Kittilsen [2], however the SVM-model is only as useful as the dataset the it is built upon. This means that while JavaScript might be a good indication of maliciousness, it won't help if the benign data set have lots of JavaScripts present while it is absent in the malicious data set.
- As mentioned in section 5.1, there can be a problem in regards to Snort and the use of an Unix socket. The author does not currently know if the problem is hardware related or not.

## 6.2 Practical Implications and Considerations

The author has come across several practical problems which need to be solved before a implementation can be performed.

- The system in its current state needs to restart Snort for every PDF-file logged due to how Snort only creates one Tcpdump log-file when Snort stops.
- The system in its current state can't process more than one PDF-file at the time with some additional time in-between two PDF-files. In order to process several PDF-files simultaneously one might need to develop a new packet sniffer to give full control over each network session or tweak Snort to create a new Tcpdump log-file for each new PDF-file header detected.
- The system the author has developed has no option to give the user any kind of feedback of the classification progress. In order to provide proper feed back, the system needs more control over the PDF-file being transferred. If the system needs more control over the PDF-file being transferred, the system needs to be narrowed down to specific task like for instance downloading PDF-files from a web browser or when a user is uploading a PDF-file to his local storage area on a server within the company or organisation.
- Text-files with the string '`%PDF-`' will trigger the system and the file will be treated as if it was a normal PDF-file. There is no way to counter this problem in the system the author developed.
- Snort is currently only running rules in regards to detect a PDF-file. Due to the way the author's system is developed, the author is dependent on the fact that no other network sessions are logged a long side the network session containing the PDF-file. This is because Snort does not give enough control to the author's software in order to distinguish between different network sessions and because Snort insist to only log to one Tcpdump log-file. This is a problem because one would need to have two Snort sensors running instead of one. One can not simply merge the author's system with an already running Snort sensor.
- The inclusion of using the Unix-socket to communicate between Snort and the author's software makes it unusable with Windows based systems. One would have to figure out another way to overcome this problem if one is going to port the system to a windows-based computer.
- The classification process is executed quite fast and the effectiveness is not dependent on the attributes of the PDF-files. The process is only dependent on the CPU installed and the time have been consistently been measured to 6 milliseconds.
- One PDF-file might has several '`%%EOF`'-footers within the same PDF-file. Sometimes one of these footers are located within the first 1000 bytes of the PDF-file with a size of several megabytes. This means that the author's software might start Tcpflow to extract the PDF-file before the entire PDF-file have been logged to a storage device. One PDF-file even have four instances of '`%%EOF`' which makes it quite hard for the author's system to determine when the entire PDF-file have been logged by Snort.

## 6.3 Overview of the Intended System

This section will provide an overview of how the author originally intended the system to look like, but lacked the time in order to develop it.

### 6.3.1 Potential Implementation in a online Environment

Figure 23 shows the system as the author originally intended it to be. One of the main differences between the planned system and the current one is the lack of a proxy server. This server was supposed to buffer the network traffic for a limited amount of time. This would give enough time for Snort to detect a PDF-file, send alarm to an Unix socket and let the author's program tell the proxy server to permanently hold the specific network session until told otherwise.

1. The network traffic is entering the computer and is buffered by the proxy server.
2. The same network traffic is monitored by Snort.
3. When Snort detects a PDF-file within a network session, Snort will send an alarm to an Unix socket.
4. While the network packets containing the PDF-file is passing through Snort, Snort logs all of the network packets to the specific network session to storage medium of the user's choice.
5. When an '`%%EOF`' passes through Snort, Snort sends an alarm to the Unix socket stating that the end of the PDF-file have been found. Program 1 is detecting this alarm.
6. Program 1, which is dedicated to listening for alarms, just detected Snort's alarm and tells Tcpflow to extract the PDF-file from Tcpdump's log-file.
7. When the Tcpflow-process have finished, Program 1 will call Program 2 which will start the feature extraction and classification process.
8. Program 2 reads the PDF-file from the storage medium, extracts features and starts the classification process.
9. When Program 2 gets confirmation on if the PDF-file is malicious or not, it will send the appropriate response to an Unix socket.
10. The proxy server will listen to the Unix socket and take the appropriate response depending on the message received. I.e. either the network session containing the PDF-file will be dropped or the network session will be released to the user expecting it.
11. After the PDF-file have been classified and an alert to the proxy server has been sent, the PDF-file will be stored in either a 'benign' or 'malicious' folder as well as important data is written in a log file for an operator to use when examining the PDF-file later.

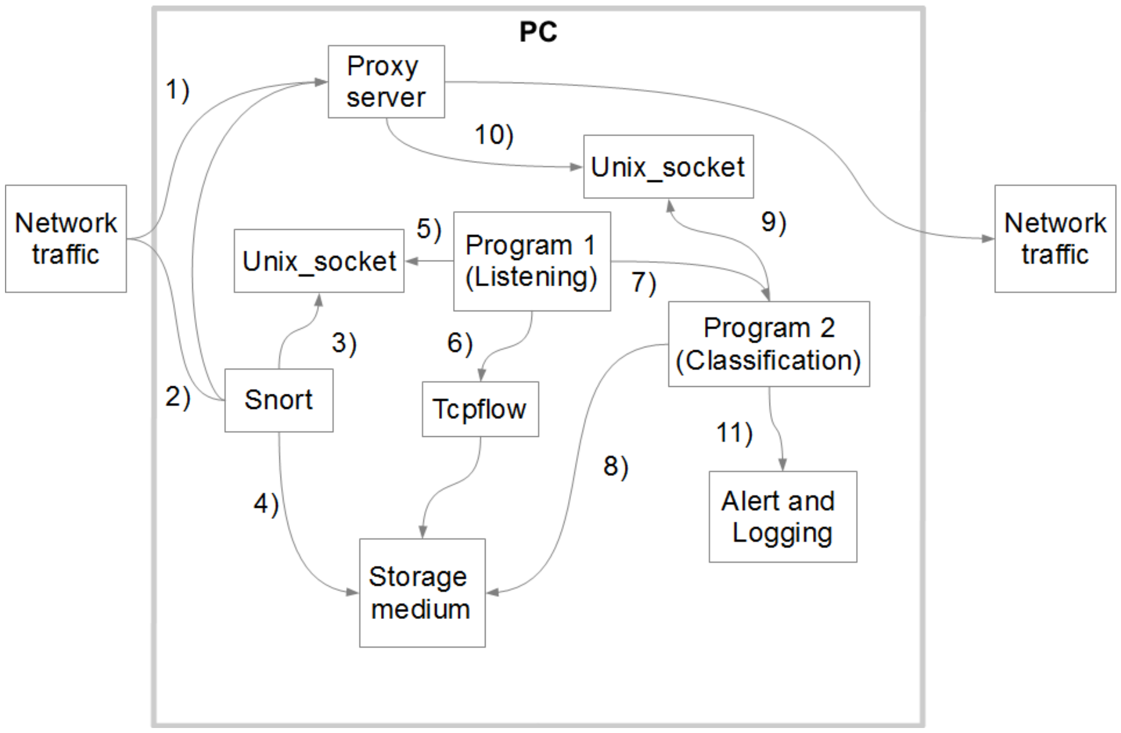


Figure 23: Brief documentation of how the author intended the detection system to work

### 6.3.2 Problems with the Intended Solution

The downside with this solution is that the proxy server would delay all of the traffic passing through Snort. This is not viable when people use IP-phones and have video-conferences. One possible solution could to implement a plugin in Firefox that communicated with a server. The server would process the PDF-download while giving time estimates back to the user. That way one can ensure that the entire PDF-file is processed and no race conditions occur like Tcpflow starting to extract the PDF-file before the entire network traffic session have been logged to disk. This is however a specific approach targeting "user downloading PDF-files with a web browser" and do not account for PDF-files being downloaded/transferred by other means. The author feels that a "one-size-fits-all" approach is less likely to be viable than a number of smaller specialized systems. More details can be found in the conclusion of this thesis, which can be found in section 6.5.

## 6.4 Difference Between Kittilsen and the Author's Solution

This section contains table 12 which explains the main differences between Kittlisen's and the author's solution.

| | **Kittilsen** | **Borg** |
|---|---|---|
| **Scope** | Offline detection | Online detection |
| **Code** | Python | C |
| **Time** | Worst case: Size/complexity dependent | Worst case: Size dependent |
| **Focus** | Detection accuracy | Speed efficiency |
| **Testing** | Different machine learning algorithms | Storage mediums and string search algorithms. |

Table 12: The main differences between Kittilsen's master thesis [2] and the author's master thesis.

The differences are presented as follows:

- Code: C-code is a low level programming language which may be more efficient in some cases than using high level languages like Python.
- Scope: Online implementation allows the system to detect and analyse PDF-file in real time which can prevent malicious PDF-files to be opened by a user.
- Time: Kittilsen is parsing the PDF-file . The author is only performing a string search due to a security concern of existing PDF-parsers written in C. However while one occurrence of a feature might not be detected by a normal string search, the string search is way more efficient to use.
- Optimisation: Kittilsen focused on testing out different kinds of machine learning algorithms and tweaking the input parameters in order to achieve the best result regarding detection accuracy. When optimising for time efficiency one have to focus on the most time consuming elements. In this case it was the process of performing feature extractions that was the most time consuming operation. Since the feature extraction process is performed by doing a string search, the time consumed will depend on which string matching algorithm is chosen, how the algorithm(s) is(are) implemented and the size of the PDF-file.
- Testing: Kittilsen had to test different machine learning algorithm to find the best one to suit his criteria. In the author's thesis the focus is on performing tests regarding algorithmic performances (time and bytes comparisons) as well as the time spent reading/writing to a storage medium (i.e. hard drive/SSD and ramdisk).

## 6.5 Conclusion

An online implementation looks feasible and possible to develop, but can be quite hard to implement perfectly with the current tools utilised in this thesis. One of the main issues that makes it difficult to implement the system is that the there is no reliable way to know exactly when every network packet of the PDF-file have been logged. The author tried to use '`%%EOF`' as an indicator when reaching the end of the PDF-file, but problems occur when the PDF-file have several '`%%EOF`'. One of the them might even be placed within the first 1000 bytes of the PDF-file which means that the author's software may force Tcpflow to extract a PDF-file that are being logged by Snort. There are possibilities to circumvent this issue, but that requires the system to be divided into several subsystem/processes taking care of one specific PDF-transfer event. A couple of examples can for instance be a user who is uploading a PDF-file to his local storage space on a server or a user downloading a PDF-file from the internet with a web browser.

This problem leads to the author's first research question which is the question about if a online PDF-file detection system is viable? The answer is that the detection system in its current form should not to be implemented in a real environment. There are to many faults present which include the limitations of the control Snort have, the way the PDF-file is constructed by having an end-of-file marker almost "anywhere" in the document and that buffering all of the traffic in a network is a bad idea as some of the traffic can't be delayed. Countering these issues requires narrowing the area of usage to several specialised systems and the analysis of these systems are outside the scope of this master thesis. In regards to the time spent on analysing a file one can look at the results in table 9 in section 5.2.2. These results show that by using the Boyer-Moore-Horspool algorithm one can achieve a linear time "usage" in regards to the size of the PDF-file where the longest measured time was 6,7 seconds for the largest PDF-file on 171MB. For a file of 171MB in size, spending only 6,7 seconds to extract all of the features on a single process thread is not bad. While other scientific papers claim to have a 10% increased efficiency algorithm [35], that would only account for roughly 670 milliseconds of a total of 6,7 seconds.

The second research question asked if there was any significant difference in time between Jarle Kittilsen's feature extraction written in Python and the authors version written in C. There was a significant difference in time usage as can be seen in table 9 in section 5.2.2. However since Kittilisen's solution is parsing the PDF-document instead of performing a string search, complex documents like master thesis reports require more processing time than simplistic documents. As described in section 5.2.2, the time spent parsing the document is not linear in regards to the size of the document. An example is the comparison between the time spent on a 171MB PDF-file containing images only (51 seconds) versus a complex master thesis report on only 2,3MB (73 seconds).

The author believes it is quite difficult to develop a system that can analyse PDF-files and prevent the files from reaching the user by only listening to network traffic. Other tools needs to be in place in order to make sure that all of the PDF-file's bytes are available for analysis. To be able to add helpful tools, one will have to narrow down the system's scope in order to focus on a specific way of how PDF-files are transported through the network by for instance focusing on people downloading PDF-files through a web browser.

## 6.6 Further Work

This section will briefly explain potential topic areas that can be further researched in order to find out if a online PDF-file detection system really is viable.

### 6.6.1 Specialised System

A network based detection system have a problem in that the user receiving the PDF-file does not get any information. If one narrow the scope down to users using a web browser, a plug-in could be created for that specific web browser. The plug-in would send a request to a server to download the PDF-file from a specific url address. The server would download the PDF-file and then the file could be analysed and classified. Progress information could be sent back to the plug-in for the user to monitor the progress. This would solve a lot of problems that the author encountered during the development period. It would also eliminate the need to develop a new packet sniffer as suggested in section 6.2. However this solution would only work with web browsers and not with users uploading PDF-files to their local storage area on a server.

### 6.6.2 GPU Utilisation

Due to time constraints there was no time for the author to experiment with multi-threading and GPU utilisation. The author refers to a master thesis written in 2012 by Kristian Nordhaug and points out that it should be very viable to push all of the analysis tasks to the GPU in order offload the CPU [37]. This way one can ensure that less packets could potentially be dropped by Snort due to high amounts of network traffic and ensure that the time spent on the feature extraction process is limited to the most time exhaustive feature. One could also go even further and split the PDF-file in for instance four equal sized chunks and scan through each chunk individually. By splitting the PDF-file into four chunks, one could potentially scan for a single feature up to four times as fast than previously.

### 6.6.3 Develop a New Network Sniffer

If one wants a detection system that can catch everything by "only" logging network packets, one has to know that Snort does not give enough control over individual network sessions in order to distinguish between different PDF-files. By being able to distinguish between $x$ number of PDF-files which are being logged at the same time and log those PDF-files into their own separate Tcpdump log-file, the system would be suitable for testing and experimenting in a online environment. Developing a packet sniffer from scratch will secure a version optimised and tweaked to fit the author's/developer's system.

# Bibliography

[1] Johnsrud, I., Ege, R. T., Henden, H., & Johnsen, A. B. Her er virus-e-posten som angrep forsvaret. `http://www.vg.no/nyheter/innenriks/artikkel.php?artid=10093859`. (Last Visitied 21/12-12).

[2] Kittilsen, J. Detecting malicious pdf documents. `http://brage.bibsys.no/hig/retrieve/2128/Jarle%20Kittilsen.pdf`.

[3] Stevens, D. Pdf tools. `http://blog.didierstevens.com/programs/pdf-tools/`. (Last Visitied 3/3-13).

[4] Laskov, P. & Šrndić, N. 2011. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, 373–382. ACM.

[5] Likarish, P., Jung, E., & Jo, I. 2009. Obfuscated malicious javascript detection using classification techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, 47–54.

[6] Multi-threading openmp tutorial. `https://computing.llnl.gov/tutorials/openMP`. (Last Visitied 3/5-13).

[7] Ben-Hur, A. & Weston, J. 2010. A user's guide to support vector machines. In *Data Mining Techniques for the Life Sciences*, Carugo, O. & Eisenhaber, F., eds, volume 609 of *Methods in Molecular Biology*, 223–239. Humana Press.

[8] Adobe. 2006. Pdf reference, sixth edition, v1.7. `http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf`. (Last Visitied 21/12-12).

[9] Statsoft. Support vector machines. `http://www.statsoft.com/textbook/support-vector-machines`. (Last Visitied 21/12-12).

[10] Symantec. 2012. Internet security threat report 2011 trends. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf`. (Last Visitied 21/12-12).

[11] Stevens, D. About. `http://blog.didierstevens.com/about/`. (Last Visitied 21/12-12).

[12] Stevens, D. Malicious-pdf-analysis-ebook.pdf. `http://didierstevens.com/files/data/malicious-pdf-analysis-ebook.zip`. (Last Visitied 21/12-12).

[13] Baccas, P. 2010. Finding rules for heuristic detection of malicious pdfs: With analysis of embedded exploit code. (Last Visitied 21/12-12).

[14] Esparza, J. M. Peepdf. `http://eternal-todo.com/tools/peepdf-pdf-analysis-tool`. Last Visitied 21/12-12).

[15] Dixon, B. Pdfxray. `https://www.pdfxray.com`. (Last Visitied 21/12-12).

[16] Hartstein, B. Jsunpack-n. `https://code.google.com/p/jsunpack-n/`. (Last Visitied 21/12-12).

[17] Dixon, B. Pdfxray. `https://github.com/cvandeplas/pdfxray`. (Last Visitied 21/12-12).

[18] Schmitt, F., Gassen, J., & Gerhards-Padilla, E. 2012. Pdf scrutinizer: Detecting javascript-based attacks in pdf documents. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, 104–111.

[19] Wikipedia.org. F1-score. `http://en.wikipedia.org/wiki/F1_score`. (Last Visitied 21/12-12).

[20] Numpy. `http://www.numpy.org/`. (Last visitied 27/5-13).

[21] Python speed. `http://wiki.python.org/moin/PythonSpeed/`. (Last visitied 20/5-13).

[22] Is python faster and lighter than c? `http://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c`. (Last visitied 20/5-13).

[23] performance-differences-between-python-and-c? `http://stackoverflow.com/questions/3533759/performance-differences-between-python-and-c`. (Last visitied 20/5-13).

[24] ISO.org. Iso 32000-1:2008. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502`. (Last visitied 10/1-13).

[25] Sourcefire. About snort. `http://www.snort.org/snort`. (Last Visitied 1/5-13).

[26] Chacos, B. 2012. How to supercharge your pc with a ram disk. `http://www.pcworld.com/article/260918/how_to_supercharge_your_pc_with_a_ram_disk.html`. (Last Visitied 1/5-13).

[27] Hp solid state drives (ssd) for workstations. `http://h18000.www1.hp.com/products/quickspecs/13379_na/13379_na.HTML`. (Last Visitied 1/5-13).

[28] El_Capitan. 2012. Ramdisk benchmarks. `http://computerhardwareupgrades.blogspot.no/2012/10/ramdisk-benchmarks.html`. (Last Visitied 1/5-13).

[29] A pdf-parsing tool. `http://podofo.sourceforge.net/about.html`. (Last Visitied 27/4-13).

[30] Karp, R. M. & Rabin, M. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2), 249–260.

[31] Kustanto, C. & Liem, I. 2009. Automatic source code plagiarism detection. In *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09. 10th ACIS International Conference on*, 481–486.

[32] Horspool, R. N. 1980. Practical fast searching in strings. *Software Practice and Experience*, 10, 501–506.

[33] Xiong, Z. 2010. A composite boyer-moore algorithm for the string matching problem. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*, 492–496.

[34] Knuth, D. E., Morris, J. H., & Pratt, V. R. 1974. Fast pattern matching in strings.

[35] Sunday, D. M. August 1990. A very fast substring search algorithm. *Commun. ACM*, 33(8), 132–142.

[36] Multi-threading openmp. `http://openmp.org/wp/`. (Last visitied 29/4-13).

[37] Nordhaug, K. 2012. Gpu accelerated nids search. `http://brage.bibsys.no/hig/handle/URN:NBN:no-bibsys_brage_34366`.

[38] Cuda parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`. (Last visitied 1/5-13).

[39] Chang, C.-C. & Lin, C.-J. Libsvm – a library for support vector machines. `www.csie.ntu.edu.tw/~cjlin/libsvm/`. (Last visitied 3/5-13).

[40] Joachims, T. Svmlight support vector machine. `http://svmlight.joachims.org`. (Last visitied 3/5-13).

[41] The_Shark_developer_team. Shark machine learning library. `http://image.diku.dk/shark/sphinx_pages/build/html/index.html`. (Last visitied 3/5-13).

[42] Ben-Hur, A. Pyml tutorial. `http://pyml.sourceforge.net/tutorial.html`. (Last Visitied 21/12-12).

[43] Snort download options. `http://www.snort.org/snort-downloadsl`. (Last Visitied 1/5-13).

[44] unix(7) - linux man page. `http://linux.die.net/man/7/unix`. (Last visitied 5/5-13).

[45] Itabashi, K. 2010. Portable document format malware. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/portable_document_format_malware.pdf`. (Last Visitied 21/12-12).

[46] Pauls, K. Ubuntu ramdisk: An easy way. http://www.ubuntuka.com/ubuntu-ramdisk-ramdrive-easy-way/. (Last visitied 5/5-13).

[47] Kittilsen, J. C yberkrig og cyberforsvaret. https://it.uib.no/ithjelp/images/3/34/Cyberkrig_og_cyberforsvar_Kittelsen_IT-forum_2012.pdf. (Last Visitied 21/12-12).

[48] Gnu.org. Gnu wget 1.13.4 manual. http://www.gnu.org/software/wget/manual/wget.html. (Last Visitied 21/12-12).

[49] Ben-Hur, A. Pyml at sourceforge. http://sourceforge.net/projects/pyml/. (Last Visitied 21/12-12).

[50] Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/~ml/weka/. (Last visitied 20/5-13).

[51] Nislab - norwegian information security laboratory. http://nislab.no. (Last visitied 1/3-13).

[52] Shm_overview - overview of posix shared memory. http://linux.die.net/man/7/shm_overview. (Last visitied 17/5-13).

[53] Adobe supplement to iso 32000-1 baseversion: 1.7 extensionlevel: 5. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/adobe_supplement_iso32000_1.pdf. (Last visitied 4/5-13).

[54] Document management — portable document format — part 1: Pdf 1.7. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf. (Last visitied 4/5-13).

[55] The file system with linux. http://tldp.org/LDP/tlk/fs/filesystem.html. (Last Visitied 1/5-13).

[56] Data acquisition (daq). http://www.snort.org/snort-downloads. (Last Visitied 1/5-13).

[57] Gnu gprof. http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html. (Last visitied 20/5-13).

[58] Gprof tutorial – how to use linux gnu gcc profiling tool. http://www.thegeekstuff.com/2012/08/gprof-tutorial/1. (Last visitied 20/5-13).

[59] Adobe supplement to the iso 32000 baseversion: 1.7 extensionlevel: 3. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/adobe_supplement_iso32000.pdf. (Last visitied 4/5-13).

[60] Microsoft security essentials. http://windows.microsoft.com/en-us/windows/security-essentials-download. (Last visitied 1/5-13).

[61] Trend micro. `http://www.trendmicro.no/`. (Last visitied 1/5-13).

[62] Avg. `http://free.avg.com/eu-en/homepage`. (Last visitied 1/5-13).

[63] Norton anti-virus. `http://www.symantec.com/index.jsp`. (Last visitied 1/5-13).

[64] Noman. `http://www.norman.com/index/`. (Last visitied 1/5-13).

[65] Hartstein, B. Jsunpack-n. `http://jsunpack.jeek.org/`. (Last Visitied 29/5-12).

# A Data Results for Boyer-Moore-Horspool Algorithm

Boyer-Moore-Horspool results (A): [Feature name]: ’[Feature count]’,’[If()-count]’,’[Time in ms]’

```

//————————If()—sentences counted, disregard the time in this section————————
Filename:jarle.pdf
  /Page: '132', '938606','13'        /AcroForm: '0', '520515', '7'
  /OpenAction: '1', '445718', '7' /AA: '0','1524714', '9'
  /JS: '0', '1527404','9'            /JavaScript: '0', '434150','7'
  /RichMedia: '0', '472652','7'      /Launch: '0', '668205','8'
  startxref:'1', '521398','6'        trailer: '1','672720','8'
  obj/end: '2719', '2719'|'0',| '1547819', '792166' | '9', '8'
Total if_c:10066067, Time classify:6, Total time feature (ms):98,
Total time file:1676, Size:2263642

Filename:pdf_reference_1—7.pdf
  /Page: '1310', '13727950','122'     /AcroForm: '1', '7478539', '88'
  /OpenAction: '0', '6533122', '107' /AA: '17','22043295', '142'
  /JS: '0', '22139243','137'          /JavaScript: '0', '6516163','112'
  /RichMedia: '0', '6861256','96'     /Launch: '0', '9753739','113'
  startxref:'2', '7723735','89'       trailer: '5','9853221','114'
  obj/end: '110776', '110776'|'0',| '22848465', '12899480' | '141', '118'
Total if_c:148378208, Time classify:8, Total time feature (ms):1379,
Total time file:13952, Size:32472771

Filename:ScientificMethodology.pdf
  /Page: '299', '69712712','634'      /AcroForm: '0', '38843714', '462'
  /OpenAction: '0', '32069000', '546' /AA: '40','114936637', '730'
  /JS: '1', '115412876','730'         /JavaScript: '0', '32255577','583'
  /RichMedia: '0', '35028347','504'   /Launch: '0', '49753430','582'
  startxref:'2', '38903149','459'     trailer: '0','50118106','591'
  obj/end: '1522', '1522'|'0',| '115099151', '58011860' | '725', '615'
Total if_c:750144559, Time classify:6, Total time feature (ms):7161,
Total time file:72943, Size:171372579

Filename:adobe_supplement_iso32000.pdf
  /Page: '140', '591232','5'          /AcroForm: '0', '319315', '4'
  /OpenAction: '2', '285789', '4'    /AA: '0','932707', '6'
  /JS: '0', '933205','6'              /JavaScript: '0', '275093','4'
  /RichMedia: '0', '294972','4'       /Launch: '0', '414449','5'
  startxref:'4', '322725','4'         trailer: '3','419462','5'
  obj/end: '5573', '5573'|'0',| '979231', '515233' | '6', '5'
Total if_c:6283413, Time classify:6, Total time feature (ms):58,
Total time file:617, Size:1373256

Filename:PDF32000_2008.pdf
  /Page: '756', '3766206','34'        /AcroForm: '0', '2102873', '25'
  /OpenAction: '1', '1762296', '29'   /AA: '17','6138309', '39'
  /JS: '0', '6103556','38'            /JavaScript: '0', '1783583','30'
  /RichMedia: '0', '1915836','27'     /Launch: '0', '2689016','36'
  startxref:'2', '2113368','25'       trailer: '0','2709435','31'
  obj/end: '3325', '3325'|'0',| '6121530', '3129133' | '38', '32'
Total if_c:40335141, Time classify:6, Total time feature (ms):384,
Total time file:3952, Size:8995189

```

```
Filename:adobe_supplement_iso32000_1.pdf
  /Page: '8', '148992','1'          /AcroForm: '0', '76442', '1'
  /OpenAction: '1', '79128', '1'  /AA: '0','220969', '1'
  /JS: '0', '219360','1'            /JavaScript: '0', '68869','1'
  /RichMedia: '0', '73775','1'     /Launch: '0', '98012','1'
  startxref:'2', '83505','1'        trailer: '0','101790','1'
  obj/end: '69', '69'|'0',| '219049', '112905' | '1', '1'
Total if_c:1502796, Time classify:6, Total time feature (ms):12,
Total time file:148, Size:323997

//—————————————Time measured without if()−sentences counted————————————

Filename:jarle.pdf
  /Page: '132', '0','7'         /AcroForm: '0', '0', '5'
  /OpenAction: '1', '0', '7'   /AA: '0','0', '8'
  /JS: '0', '0','8'             /JavaScript: '0', '0','6'
  /RichMedia: '0', '0','6'      /Launch: '0', '0','7'
  startxref:'1', '0','6'        trailer: '1','0','7'
  obj/end: '2719', '2719'|'0',| '0', '0' | '9', '7'
Total if_c:0, Time classify:6, Total time feature (ms):83,
Total time file:872, Size:2263642

Filename:pdf_reference_1−7.pdf
  /Page: '1310', '0','109'       /AcroForm: '1', '0', '80'
  /OpenAction: '0', '0', '97'   /AA: '17','0', '130'
  /JS: '0', '0','122'             /JavaScript: '0', '0','93'
  /RichMedia: '0', '0','88'      /Launch: '0', '0','102'
  startxref:'2', '0','81'        trailer: '5','0','103'
  obj/end: '110776', '110776'|'0',| '0', '0' | '125', '107'
Total if_c:0, Time classify:7, Total time feature (ms):1237,
Total time file:12477, Size:32472771

Filename:ScientificMethodology.pdf
  /Page: '299', '0','568'        /AcroForm: '0', '0', '424'
  /OpenAction: '0', '0', '506'  /AA: '40','0', '634'
  /JS: '1', '0','643'             /JavaScript: '0', '0','483'
  /RichMedia: '0', '0','466'     /Launch: '0', '0','535'
  startxref:'2', '0','418'       trailer: '0','0','543'
  obj/end: '1522', '1522'|'0',| '0', '0' | '636', '552'
Total if_c:0, Time classify:14, Total time feature (ms):6408,
Total time file:64494, Size:171372579

Filename:adobe_supplement_iso32000.pdf
  /Page: '140', '0','5'          /AcroForm: '0', '0', '3'
  /OpenAction: '2', '0', '4'    /AA: '0','0', '5'
  /JS: '0', '0','5'               /JavaScript: '0', '0','4'
  /RichMedia: '0', '0','4'       /Launch: '0', '0','4'
  startxref:'4', '0','3'         trailer: '3','0','4'
  obj/end: '5573', '5573'|'0',| '0', '0' | '5', '4'
Total if_c:0, Time classify:6, Total time feature (ms):50,
Total time file:540, Size:1373256

Filename:PDF32000_2008.pdf
  /Page: '756', '0','32'         /AcroForm: '0', '0', '22'
  /OpenAction: '1', '0', '27'   /AA: '17','0', '34'
  /JS: '0', '0','34'              /JavaScript: '0', '0','25'
  /RichMedia: '0', '0','25'      /Launch: '0', '0','34'
  startxref:'2', '0','23'        trailer: '0','0','29'
  obj/end: '3325', '3325'|'0',| '0', '0' | '34', '30'
Total if_c:0, Time classify:7, Total time feature (ms):349,
Total time file:3558, Size:8995189

```

```
Filename:adobe_supplement_iso32000_1.pdf
  /Page: '8', '0','1'         /AcroForm: '0', '0', '1'
  /OpenAction: '1', '0', '1'  /AA: '0','0', '1'
  /JS: '0', '0','1'           /JavaScript: '0', '0','1'
  /RichMedia: '0', '0','1'    /Launch: '0', '0','1'
  startxref:'2', '0','1'      trailer: '0','0','1'
  obj/end: '69', '69'|'0',| '0', '0' | '1', '1'
Total if_c:0, Time classify:6, Total time feature (ms):12,
Total time file:144, Size:323997
```

# B Data Results for Naive Algorithm

Date results for Naive algorithm: [Feature name]: '[Feature count]','[If()-count]','[Time in ms]'

```
//————————If()−sentences counted, disregard the time in this section————————
Filename:jarle.pdf
  /Page: '132', '2303711', '7'  /AcroForm: '0', '2303326', '7'
  /OpenAction: '1', '2302567', '7'  /AA: '0','2303326', '7'
  /JS: '0', '2302544', '7'    /JavaScript: '0', '2302544', '7'
  /RichMedia: '0', '2302986', '7'   /Launch: '0', '2303497', '7'
  startxref:'1', '6804369', '20'  trailer: '1','6805628', '21'
  obj/end: '2719', '2719'|'0'|, '6823848', '6827764'| '20', '21',
Total if_c:45686110, Total time feature (ms):138, Size:2263642

Filename:pdf_reference_1−7.pdf
  /Page: '1310', '33798997', '100'  /AcroForm: '1', '33854245', '98'
  /OpenAction: '0', '33789632', '100'   /AA: '17','33854643', '100'
  /JS: '0', '33787440', '100'     /JavaScript: '0', '33787444', '100'
  /RichMedia: '0', '33822249', '100'    /Launch: '0', '33825607', '100'
  startxref:'2', '97548474', '301'  trailer: '7','97805684', '296'
  obj/end: '110776', '110776'|'0'|, '98752653', '98686249'| '303', '303',
Total if_c:663313317, Total time feature (ms):2001, Size:32472771

Filename:ScientificMethodology.pdf
  /Page: '299', '172774128', '505'  /AcroForm: '0', '172773931', '503'
  /OpenAction: '0', '172772782', '506'  /AA: '58','172774022', '508'
  /JS: '1', '172771412', '518'    /JavaScript: '0', '172771415', '534'
  /RichMedia: '0', '172773203', '512'   /Launch: '0', '172772649', '506'
  startxref:'2', '514704628', '1546'  trailer: '7','514770616', '1556'
  obj/end: '110776', '110776'|'0'|, '515103548', '514676773'| '1560', '1554',
Total if_c:3441439107, Total time feature (ms):10308, Size:171372579

Filename:adobe_supplement_iso32000.pdf
  /Page: '140', '1429782', '4'  /AcroForm: '0', '1429508', '4'
  /OpenAction: '2', '1428100', '5'  /AA: '0','1429516', '4'
  /JS: '0', '1428049', '4'    /JavaScript: '0', '1428050', '5'
  /RichMedia: '0', '1429521', '4'   /Launch: '0', '1429654', '4'
  startxref:'4', '4129906', '12'  trailer: '4','4135991', '12'
  obj/end: '5573', '5573'|'0'|, '4185523', '4184209'| '13', '13',
Total if_c:28067809, Total time feature (ms):84, Size:1373256

Filename:PDF32000_2008.pdf
  /Page: '756', '9218587', '28'   /AcroForm: '0', '9218579', '28'
  /OpenAction: '1', '9215120', '28'   /AA: '17','9218945', '27'
  /JS: '0', '9213927', '30'     /JavaScript: '0', '9213928', '28'
  /RichMedia: '0', '9215423', '28'    /Launch: '0', '9216686', '27'
  startxref:'2', '27055651', '82'   trailer: '0','27074868', '82'
  obj/end: '3325', '3325'|'0'|, '27078928', '27111491'| '82', '82',
Total if_c:182052133, Total time feature (ms):552, Size:8995189
```

```
Filename:adobe_supplement_iso32000_1.pdf
  /Page: '8', '334334', '1'   /AcroForm: '0', '334421', '1'
  /OpenAction: '1', '334303', '1'   /AA: '0','334437', '1'
  /JS: '0', '334277', '1'     /JavaScript: '0', '334277', '1'
  /RichMedia: '0', '334293', '1'    /Launch: '0', '334360', '1'
  startxref:'2', '984124', '3'  trailer: '0','983060', '3'
  obj/end: '69', '69'|'0'|, '976456', '981789'| '3', '3',
Total if_c:6600131, Total time feature (ms):20, Size:323997


//--------------------Time measured without if()-sentences counted-----------------

Filename:jarle.pdf
  /Page: '132', '0', '5'  /AcroForm: '0', '0', '5'
  /OpenAction: '1', '0', '5'  /AA: '0','0', '5'
  /JS: '0', '0', '5'    /JavaScript: '0', '0', '5'
  /RichMedia: '0', '0', '5'   /Launch: '0', '0', '5'
  startxref:'1', '0', '18'  trailer: '1','0', '18'
  obj/end: '2719', '2719'|'0'|, '0', '0'| '19', '18',
Total if_c:0, Total time feature (ms):113, Size:2263642

Filename:pdf_reference_1-7.pdf
  /Page: '1310', '0', '74'  /AcroForm: '1', '0', '76'
  /OpenAction: '0', '0', '74'   /AA: '17','0', '74'
  /JS: '0', '0', '75'     /JavaScript: '0', '0', '75'
  /RichMedia: '0', '0', '75'    /Launch: '0', '0', '74'
  startxref:'2', '0', '256'   trailer: '7','0', '259'
  obj/end: '110776', '110776'|'0'|, '0', '0'| '263', '267',
Total if_c:0, Total time feature (ms):1642, Size:32472771

Filename:ScientificMethodology.pdf
  /Page: '299', '0', '353'  /AcroForm: '0', '0', '352'
  /OpenAction: '0', '0', '354'  /AA: '58','0', '354'
  /JS: '1', '0', '355'    /JavaScript: '0', '0', '351'
  /RichMedia: '0', '0', '354'   /Launch: '0', '0', '352'
  startxref:'2', '0', '1340'  trailer: '0','0', '1316'
  obj/end: '1522', '1522'|'0'|, '0', '0'| '1329', '1328',
Total if_c:0, Total time feature (ms):8138, Size:171372579

Filename:adobe_supplement_iso32000.pdf
  /Page: '140', '0', '3'  /AcroForm: '0', '0', '3'
  /OpenAction: '2', '0', '3'  /AA: '0','0', '3'
  /JS: '0', '0', '3'    /JavaScript: '0', '0', '3'
  /RichMedia: '0', '0', '3'   /Launch: '0', '0', '3'
  startxref:'4', '0', '11'  trailer: '4','0', '11'
  obj/end: '5573', '5573'|'0'|, '0', '0'| '11', '11',
Total if_c:0, Total time feature (ms):68, Size:1373256

Filename:PDF32000_2008.pdf
  /Page: '756', '0', '20'   /AcroForm: '0', '0', '21'
  /OpenAction: '1', '0', '20'   /AA: '17','0', '20'
  /JS: '0', '0', '20'     /JavaScript: '0', '0', '20'
  /RichMedia: '0', '0', '20'    /Launch: '0', '0', '20'
  startxref:'2', '0', '72'  trailer: '0','0', '71'
  obj/end: '3325', '3325'|'0'|, '0', '0'| '72', '72',
Total if_c:0, Total time feature (ms):448, Size:8995189
```

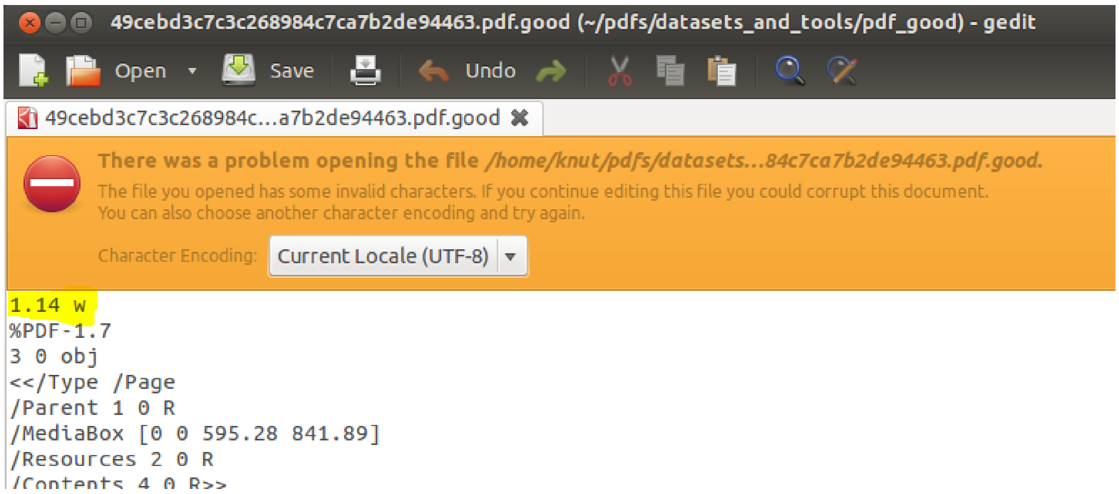
```
Filename:adobe_supplement_iso32000_1.pdf
  /Page: '8', '0', '1'  /AcroForm: '0', '0', '1'
  /OpenAction: '1', '0', '1'  /AA: '0','0', '1'
  /JS: '0', '0', '1'    /JavaScript: '0', '0', '1'
  /RichMedia: '0', '0', '1'   /Launch: '0', '0', '1'
  startxref:'2', '0', '3'   trailer: '0','0', '3'
  obj/end: '69', '69'|'0'|, '0', '0'| '3', '3',
Total if_c:0, Total time feature (ms):20, Size:323997
```

# C Data Results for Jarle Kittilsen's Feature Extraction

Jarle Kittilsen's results with extracting features with Python:

```
Parsing PDF32000_2008.pdf
Extracting features...
JavaScript seems to be present, but no script dumped
Writing to vector file...
Work done in 18.1646280289 seconds.
----
Parsing adobe_supplement_iso32000_1.pdf
Extracting features...
JavaScript seems to be present, but no script dumped
Writing to vector file...
Work done in 3.97685909271 seconds.
----
Parsing jarle_kittilsen.pdf
Extracting features...
JavaScript seems to be present, but no script dumped
Writing to vector file...
Work done in 73.3590581417 seconds.
----
Parsing ScientificMethodology.pdf
Extracting features...
JavaScript seems to be present, but no script dumped
Writing to vector file...
Work done in 51.3802189827 seconds.
----
Parsing adobe_supplement_iso32000.pdf
Extracting features...
JavaScript seems to be present, but no script dumped
Writing to vector file...
Work done in 2.6815559864 seconds.
----
Parsing pdf_reference_1-7.pdf
Extracting features...
JavaScript seems to be present, but no script dumped
Writing to vector file...
Work done in 36.4098320007 seconds.
```

## D PDF-Files with Content Before File-Header

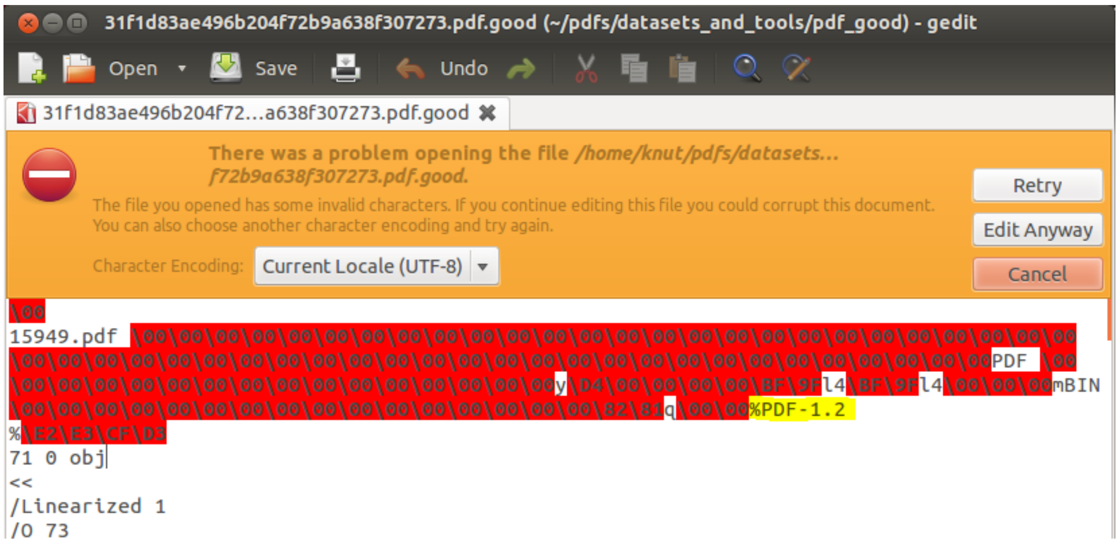

49cebd3c7c3c268984c7ca7b2de94463.pdf.good (~/pdfs/datasets_and_tools/pdf_good) - gedit
Open
Save
Undo
49cebd3c7c3c268984c...a7b2de94463.pdf.good

There was a problem opening the file /home/knut/pdfs/datasets...84c7ca7b2de94463.pdf.good.
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

Character Encoding: Current Locale (UTF-8)

```
1.14 w
%PDF-1.7
3 0 obj
<</Type /Page
/Parent 1 0 R
/MediaBox [0 0 595.28 841.89]
/Resources 2 0 R
/Contents 4 0 R>>
```

Figure 24: File: [49cebd3c7c3c268984c7ca7b2de94463.pdf.good] - '%PDF-' starting at block [7]

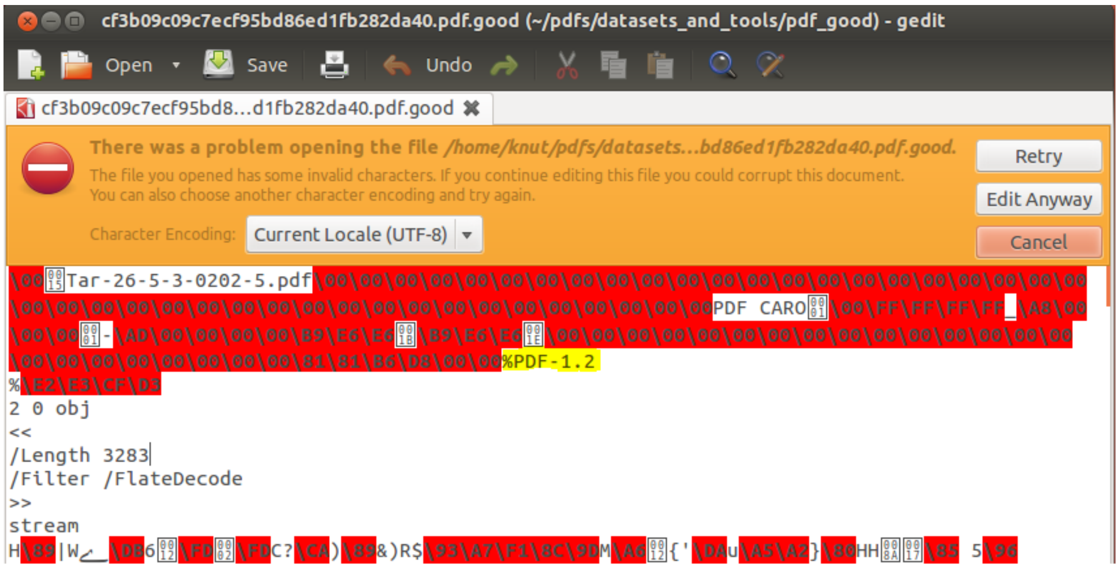

31f1d83ae496b204f72b9a638f307273.pdf.good (~/pdfs/datasets_and_tools/pdf_good) - gedit
Open
Save
Undo
31f1d83ae496b204f72...a638f307273.pdf.good
Retry
Edit Anyway
Cancel

There was a problem opening the file /home/knut/pdfs/datasets... f72b9a638f307273.pdf.good.
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

Character Encoding: Current Locale (UTF-8)

```
\00
15949.pdf \00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00PDF \00
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00y\D4\00\00\00\00\BF\9Fl4\BF\9Fl4\00\00\00mBIN
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\82\81q\00\00%PDF-1.2
%\E2\E3\CF\D3
71 0 obj
<<
/Linearized 1
/O 73
```

Figure 25: File: [31f1d83ae496b204f72b9a638f307273.pdf.good] - '%PDF-' starting at block [128]

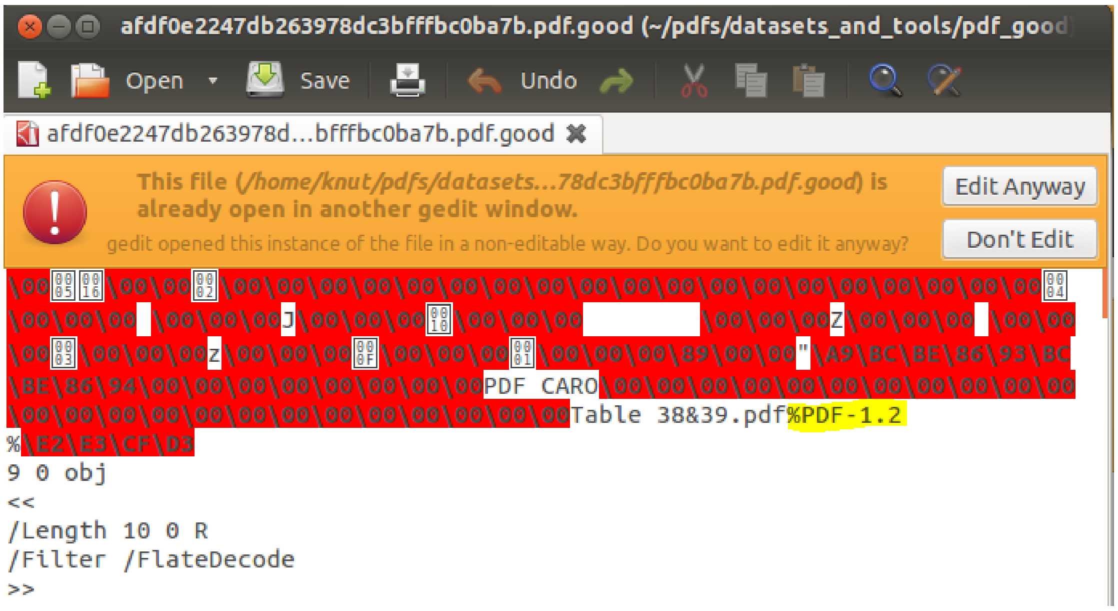

cf3b09c09c7ecf95bd86ed1fb282da40.pdf.good (~/pdfs/datasets_and_tools/pdf_good) - gedit
Open
Save
Undo
cf3b09c09c7ecf95bd8...d1fb282da40.pdf.good
Retry
Edit Anyway
Cancel

There was a problem opening the file /home/knut/pdfs/datasets...bd86ed1fb282da40.pdf.good.
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

Character Encoding: Current Locale (UTF-8)

```
\00Tar-26-5-3-0202-5.pdf\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00PDF CARO\00\FF\FF\FF\FF_\A8\00
\00\00-\AD\00\00\00\00\B9\E6\E6\B9\E6\E6\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\00\00\00\81\81\B6\D8\00\00%PDF-1.2
%\E2\E3\CF\D3
2 0 obj
<<
/Length 3283
/Filter /FlateDecode
>>
stream
H\89|W_\DB6\FD\FDC?\CA)\89&)R$\93\A7\F1\8C\9DM\A6{'\DAu\A5\A2}\80HH\85 5\96
```

Figure 26: File: [cf3b09c09c7ecf95bd86ed1fb282da40.pdf.good] - '%PDF-' starting at block [128]

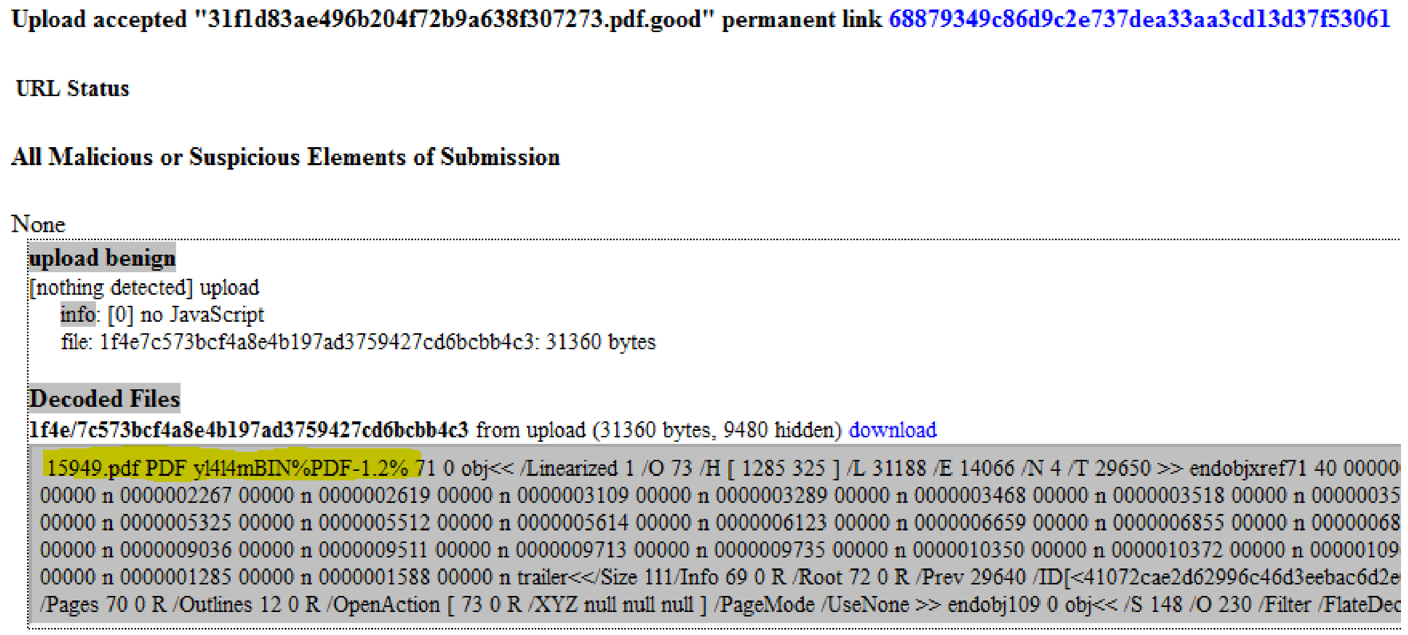

afdf0e2247db263978dc3bfffbc0ba7b.pdf.good (~/pdfs/datasets_and_tools/pdf_good
Open
Save
Undo
afdf0e2247db263978d...bfffbc0ba7b.pdf.good
Edit Anyway
Don't Edit

This file (/home/knut/pdfs/datasets...78dc3bfffbc0ba7b.pdf.good) is already open in another gedit window.
gedit opened this instance of the file in a non-editable way. Do you want to edit it anyway?

```
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00 \00\00\00J\00\00\00\00\00\00          \00\00\00Z\00\00\00 \00\00
\00\00\00\00z\00\00\00\00\00\00\00\00\00\00\89\00\00"\A9\BC\BE\86\93\BC
\BE\86\94\00\00\00\00\00\00\00PDF CARO\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\00\00\00\00\00\00\00\00Table 38&39.pdf%PDF-1.2
%\E2\E3\CF\D3
9 0 obj
<<
/Length 10 0 R
/Filter /FlateDecode
>>
```

Figure 27: File: [afdf0e2247db263978dc3bfffbc0ba7b.pdf.good] - '%PDF-' starting at block [137]

## E PDF-Files Analysed by Jsunpack (Malware Before Header)



**Upload accepted "31f1d83ae496b204f72b9a638f307273.pdf.good" permanent link 68879349c86d9c2e737dea33aa3cd13d37f53061**

**URL Status**

**All Malicious or Suspicious Elements of Submission**

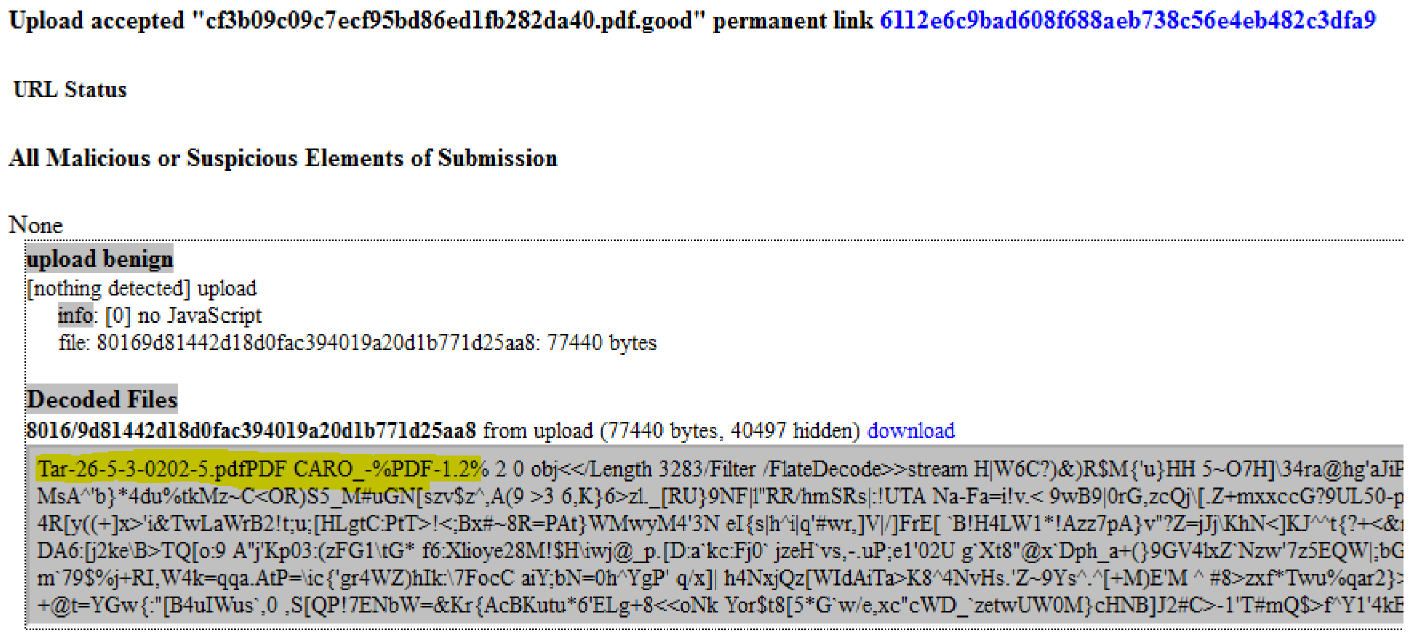
None

**upload benign**
[nothing detected] upload
info: [0] no JavaScript
file: 1f4e7c573bcf4a8e4b197ad3759427cd6bcbb4c3: 31360 bytes

**Decoded Files**
**1f4e/7c573bcf4a8e4b197ad3759427cd6bcbb4c3** from upload (31360 bytes, 9480 hidden) download

```
15949.pdf PDF yl4l4mBIN%PDF-1.2% 71 0 obj<< /Linearized 1 /O 73 /H [ 1285 325 ] /L 31188 /E 14066 /N 4 /T 29650 >> endobjxref71 40 00000
00000 n 0000002267 00000 n 0000002619 00000 n 0000003109 00000 n 0000003289 00000 n 0000003468 00000 n 0000003518 00000 n 00000035
00000 n 0000005325 00000 n 0000005512 00000 n 0000005614 00000 n 0000006123 00000 n 0000006659 00000 n 0000006855 00000 n 00000068
00000 n 0000009036 00000 n 0000009511 00000 n 0000009713 00000 n 0000009735 00000 n 0000010350 00000 n 0000010372 00000 n 00000109
00000 n 0000001285 00000 n 0000001588 00000 n trailer<</Size 111/Info 69 0 R /Root 72 0 R /Prev 29640 /ID[<41072cae2d62996c46d3eebac6d2e
/Pages 70 0 R /Outlines 12 0 R /OpenAction [ 73 0 R /XYZ null null null ] /PageMode /UseNone >> endobj109 0 obj<< /S 148 /O 230 /Filter /FlateDec
```

Figure 28: File: [31f1d83ae496b204f72b9a638f307273.pdf.good] - Jsunpack



**Upload accepted "cf3b09c09c7ecf95bd86ed1fb282da40.pdf.good" permanent link 6112e6c9bad608f688aeb738c56e4eb482c3dfa9**

**URL Status**

**All Malicious or Suspicious Elements of Submission**

None

**upload benign**

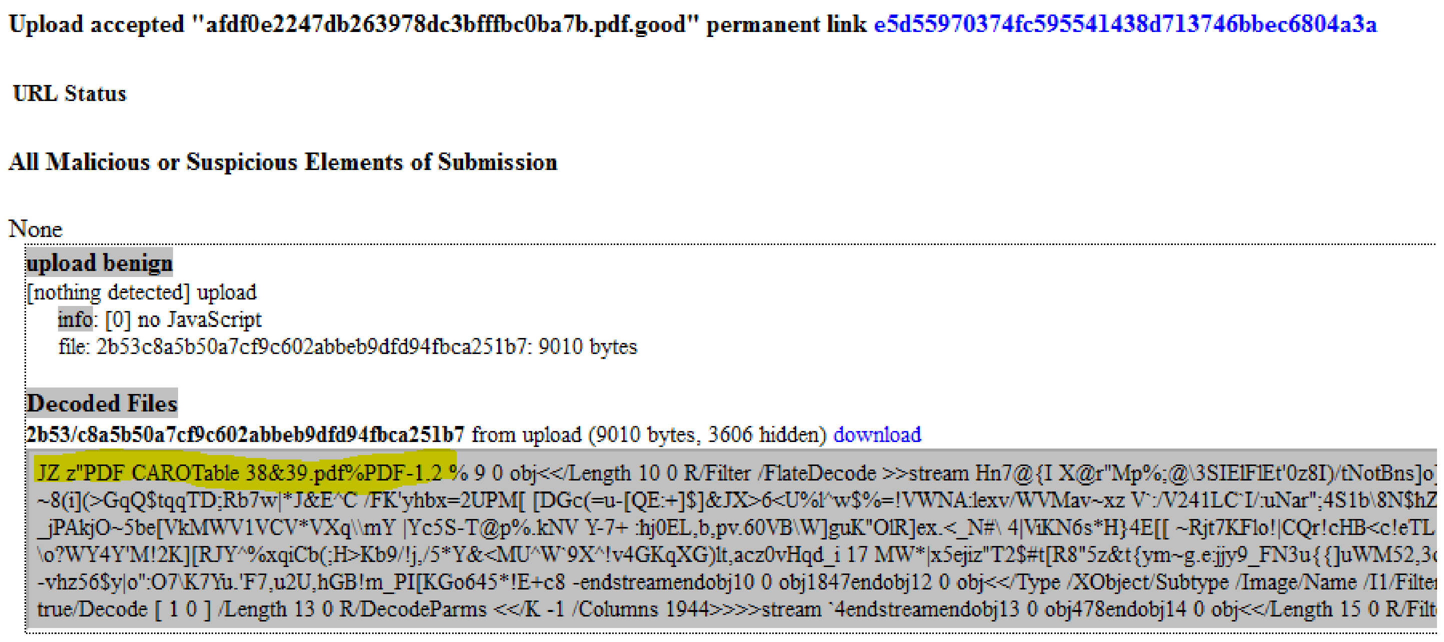
```
[nothing detected] upload
    info: [0] no JavaScript
    file: 80169d81442d18d0fac394019a20d1b771d25aa8: 77440 bytes
```

**Decoded Files**

**8016/9d81442d18d0fac394019a20d1b771d25aa8** from upload (77440 bytes, 40497 hidden) download

```
Tar-26-5-3-0202-5.pdfPDF CARO_-%PDF-1.2% 2 0 obj<</Length 3283/Filter /FlateDecode>>stream H|W6C?)&)R$M{'u}HH 5~O7H]\34ra@hg'aJiP
MsA^'b}*4du%tkMz~C<OR)S5_M#uGN[szv$z^,A(9 >3 6,K}6>zl._[RU}9NF|l"RR/hmSRs|:!UTA Na-Fa=i!v.< 9wB9|0rG,zcQj\[.Z+mxxccG?9UL50-p
4R[y((+]x>'i&TwLaWrB2!t;u;[HLgtC:PtT>!<;Bx#~8R=PAt}WMwyM4'3N eI{s|h^i|q'#wr,]V|/]FrE[ `B!H4LW1*!Azz7pA}v"?Z=jJj\KhN<]KJ^^t{?+<&r
DA6:[j2ke\B>TQ[o:9 A"j'Kp03:(zFG1\tG* f6:Xlioye28M!$H\iwj@_p.[D:a`kc:Fj0` jzeH`vs,-.uP;e1'02U g`Xt8"@x`Dph_a+()9GV4lxZ`Nzw'7z5EQW|;bG
m`79$%j+RI,W4k=qqa.AtP=\ic{'gr4WZ)hIk:\7FocC aiY;bN=0h^YgP' q/x]| h4NxjQz[WIdAiTa>K8^4NvHs.'Z~9Ys^.^[+M)E'M ^ #8>zxf*Twu%qar2}>
+@t=YGw{:"[B4uIWus`,0 ,S[QP!7ENbW=&Kr{AcBKutu*6'ELg+8<<oNk Yor$t8[5*G`w/e,xc"cWD_`zetwUW0M}cHNB]J2#C>-1'T#mQ$>f^Y1'4kE
```

Figure 29: File: [cf3b09c09c7ecf95bd86ed1fb282da40.pdf.good] - Jsunpack



**Upload accepted "afdf0e2247db263978dc3bfffbc0ba7b.pdf.good" permanent link** e5d55970374fc595541438d713746bbec6804a3a

**URL Status**

**All Malicious or Suspicious Elements of Submission**

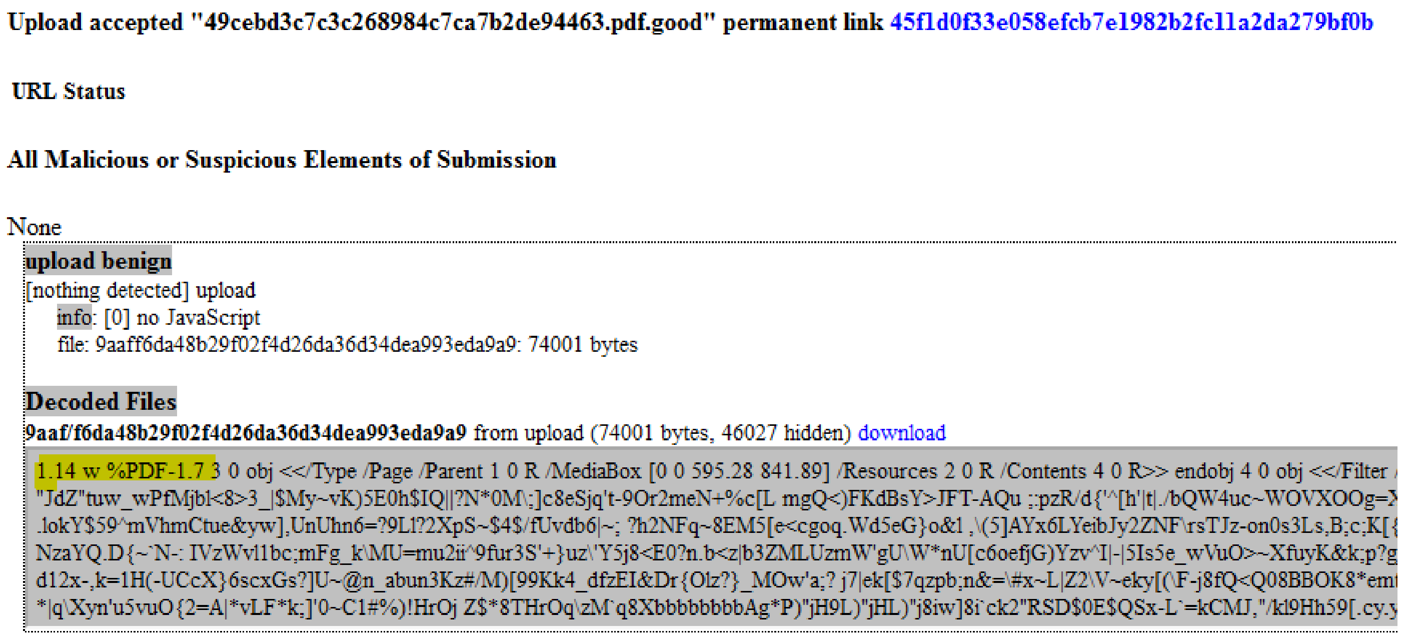
None

**upload benign**
[nothing detected] upload
info: [0] no JavaScript
file: 2b53c8a5b50a7cf9c602abbeb9dfd94fbca251b7: 9010 bytes

**Decoded Files**
**2b53/c8a5b50a7cf9c602abbeb9dfd94fbca251b7** from upload (9010 bytes, 3606 hidden) download

```
JZ z"PDF CAROTable 38&39.pdf%PDF-1.2 % 9 0 obj<</Length 10 0 R/Filter /FlateDecode >>stream Hn7@{I X@r"Mp%;@\3SIElFlEt'0z8I)/tNotBns]o]
~8(i](>GqQ$tqqTD;Rb7w|*J&E^C /FK'yhbx=2UPM[ [DGc(=u-[QE:+]$]&JX>6<U%l^w$%=!VWNA:lexv/WVMav~xz V`:/V241LC`I/:uNar";4S1b\8N$hZ
_jPAkjO~5be[VkMWV1VCV*VXq\\mY |Yc5S-T@p%.kNV Y-7+ :hj0EL,b,pv.60VB\W]guK"OlR]ex.<_N#\ 4|ViKN6s*H}4E[[ ~Rjt7KFlo!|CQr!cHB<c!eTL
\o?WY4Y'M!2K][RJY^%xqiCb(;H>Kb9/!j,/5*Y&<MU^W`9X^!v4GKqXG)lt,acz0vHqd_i 17 MW*|x5ejiz"T2$#t[R8"5z&t{ym~g.e:jjy9_FN3u{{]uWM52,3c
-vhz56$y|o":O7\K7Yu.'F7,u2U,hGB!m_PI[KGo645*!E+c8 -endstreamendobj10 0 obj1847endobj12 0 obj<</Type /XObject/Subtype /Image/Name /I1/Filter
true/Decode [ 1 0 ] /Length 13 0 R/DecodeParms <</K -1 /Columns 1944>>>>stream `4endstreamendobj13 0 obj478endobj14 0 obj<</Length 15 0 R/Filt
```

Figure 30: File: [afdf0e2247db263978dc3bfffbc0ba7b.pdf.good] - Jsunpack



**Upload accepted "49cebd3c7c3c268984c7ca7b2de94463.pdf.good" permanent link 45f1d0f33e058efcb7e1982b2fc11a2da279bf0b**

**URL Status**

**All Malicious or Suspicious Elements of Submission**

None

**upload benign**

[nothing detected] upload

info: [0] no JavaScript

file: 9aaff6da48b29f02f4d26da36d34dea993eda9a9: 74001 bytes

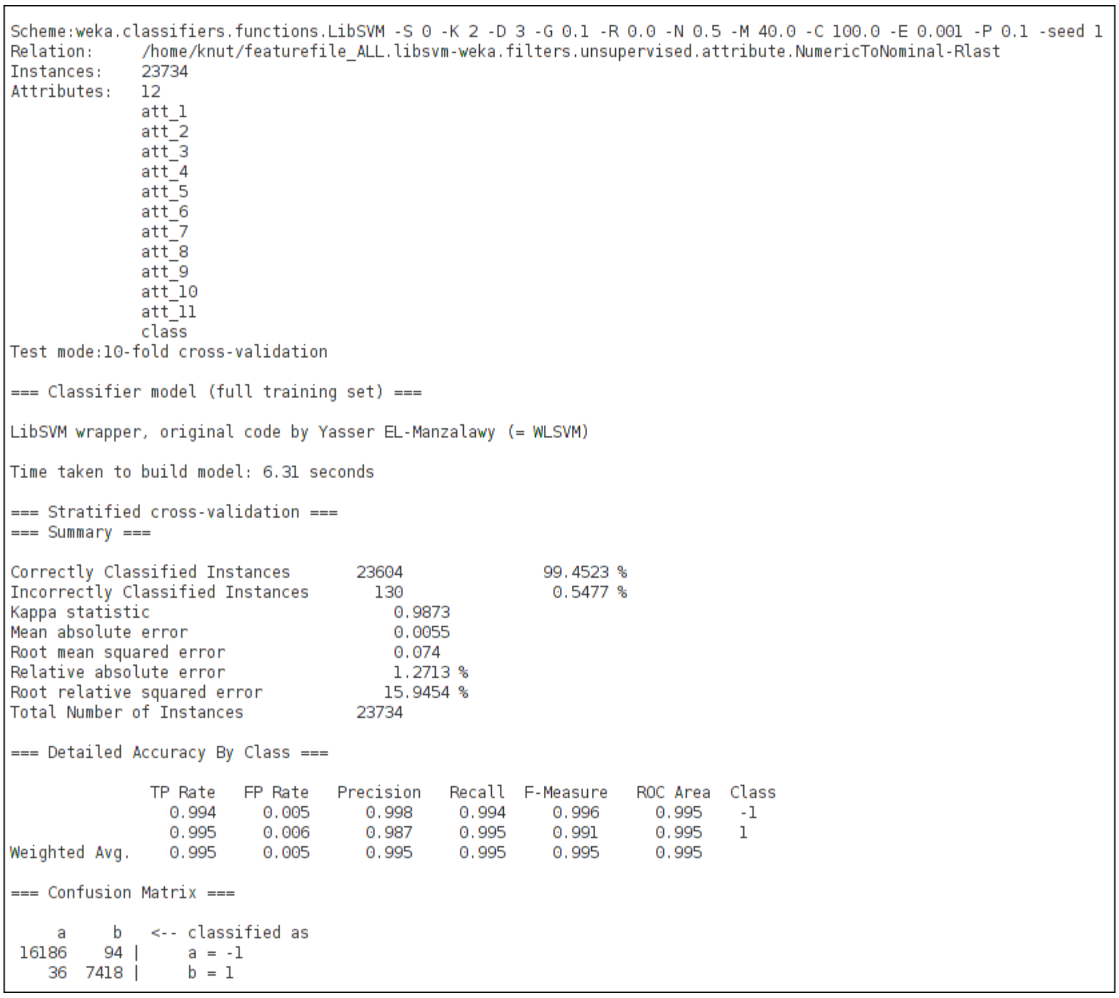
**Decoded Files**

**9aaf/f6da48b29f02f4d26da36d34dea993eda9a9** from upload (74001 bytes, 46027 hidden) download

```
1.14 w %PDF-1.7 3 0 obj <</Type /Page /Parent 1 0 R /MediaBox [0 0 595.28 841.89] /Resources 2 0 R /Contents 4 0 R>> endobj 4 0 obj <</Filter
"JdZ"tuw_wPfMjbl<8>3_|$My~vK)5E0h$IQ||?N*0M\;]c8eSjq't-9Or2meN+%c[L mgQ<)FKdBsY>JFT-AQu ;:pzR/d{'^[h'|t|./bQW4uc~WOVXOOg=X
.lokY$59^mVhmCtue&yw],UnUhn6=?9Ll?2XpS~$4$/fUvdb6|~; ?h2NFq~8EM5[e<cgoq.Wd5eG}o&l ,\(5]AYx6LYeibJy2ZNF\rsTJz-on0s3Ls,B;c;K[{
NzaYQ.D{~`N-: IVzWvl1bc;mFg_k\MU=mu2ii^9fur3S'+}uz\'Y5j8<E0?n.b<z|b3ZMLUzmW'gU\W*nU[c6oefjG)Yzv^I|-|5Is5e_wVuO>~XfuyK&k;p?g
d12x-,k=1H(-UCcX}6scxGs?]U~@n_abun3Kz#/M)[99Kk4_dfzEI&Dr{Olz?}_MOw'a;? j7|ek[$7qzpb;n&=\#x~L|Z2\V~eky[(\F-j8fQ<Q08BBOK8*em
*|q\Xyn'u5vuO{2=A|*vLF*k;]'0~C1#%)!HrOj Z$*8THrOq\zM`q8XbbbbbbbbAg*P)"jH9L)"jHL)"j8iw]8i`ck2"RSD$0E$QSx-L`=kCMJ,"/kl9Hh59[.cy.y
```

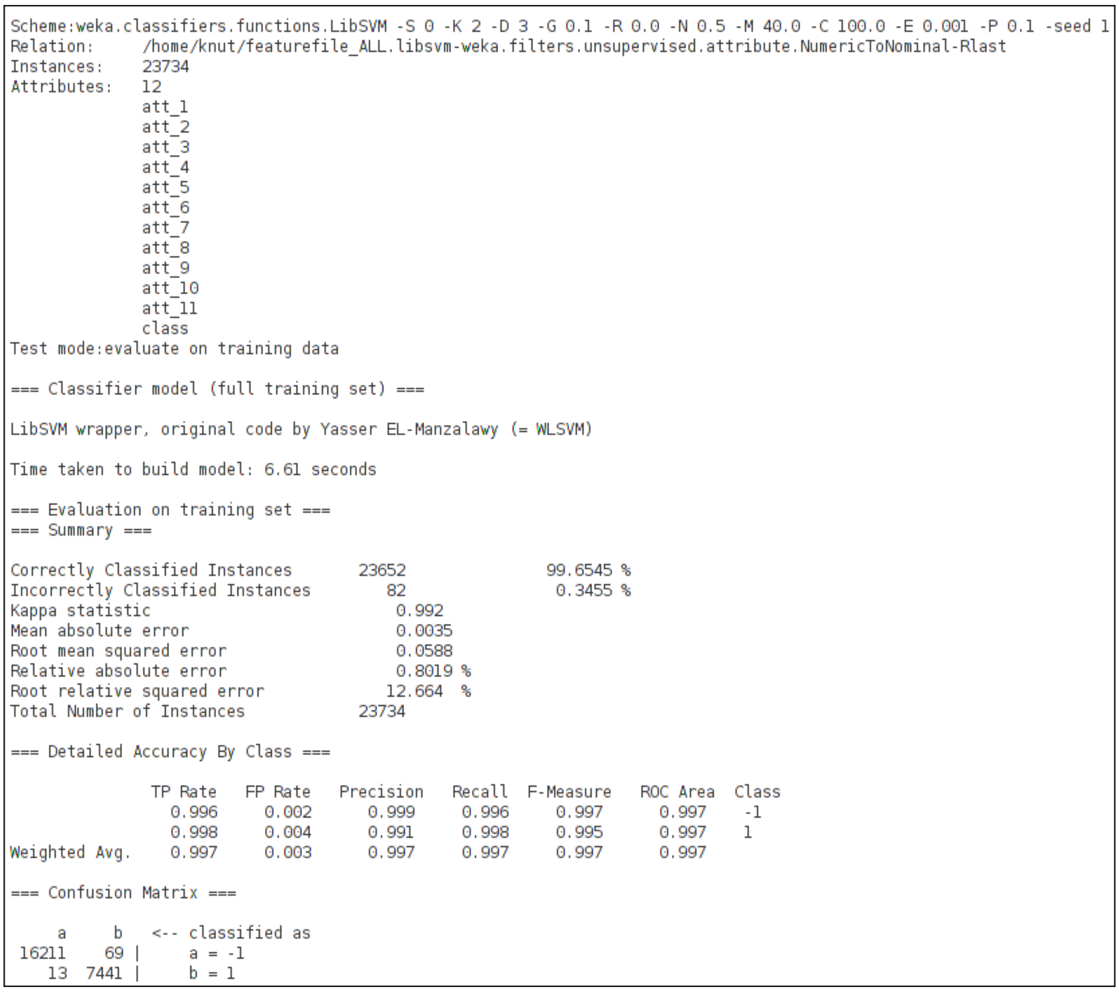
Figure 31: File: [49cebd3c7c3c268984c7ca7b2de94463.pdf.good] - Jsunpack

# F Using Weka for Presentation of SVM-Statistics



```
Scheme:weka.classifiers.functions.LibSVM -S 0 -K 2 -D 3 -G 0.1 -R 0.0 -N 0.5 -M 40.0 -C 100.0 -E 0.001 -P 0.1 -seed 1
Relation:     /home/knut/featurefile_ALL.libsvm-weka.filters.unsupervised.attribute.NumericToNominal-Rlast
Instances:    23734
Attributes:   12
              att_1
              att_2
              att_3
              att_4
              att_5
              att_6
              att_7
              att_8
              att_9
              att_10
              att_11
              class
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

LibSVM wrapper, original code by Yasser EL-Manzalawy (= WLSVM)

Time taken to build model: 6.31 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances       23604               99.4523 %
Incorrectly Classified Instances       130                0.5477 %
Kappa statistic                          0.9873
Mean absolute error                      0.0055
Root mean squared error                  0.074
Relative absolute error                  1.2713 %
Root relative squared error             15.9454 %
Total Number of Instances            23734

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.994     0.005      0.998     0.994     0.996      0.995    -1
                 0.995     0.006      0.987     0.995     0.991      0.995    1
Weighted Avg.    0.995     0.005      0.995     0.995     0.995      0.995

=== Confusion Matrix ===

     a     b   <-- classified as
 16186    94 |     a = -1
    36  7418 |     b = 1
```

Figure 32: Cross-validation with Weka and LibSVM



```
Scheme:weka.classifiers.functions.LibSVM -S 0 -K 2 -D 3 -G 0.1 -R 0.0 -N 0.5 -M 40.0 -C 100.0 -E 0.001 -P 0.1 -seed 1
Relation:     /home/knut/featurefile_ALL.libsvm-weka.filters.unsupervised.attribute.NumericToNominal-Rlast
Instances:    23734
Attributes:   12
              att_1
              att_2
              att_3
              att_4
              att_5
              att_6
              att_7
              att_8
              att_9
              att_10
              att_11
              class
Test mode:evaluate on training data

=== Classifier model (full training set) ===

LibSVM wrapper, original code by Yasser EL-Manzalawy (= WLSVM)

Time taken to build model: 6.61 seconds

=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances       23652               99.6545 %
Incorrectly Classified Instances        82                0.3455 %
Kappa statistic                          0.992
Mean absolute error                      0.0035
Root mean squared error                  0.0588
Relative absolute error                  0.8019 %
Root relative squared error             12.664  %
Total Number of Instances            23734

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.996     0.002     0.999      0.996     0.997      0.997    -1
                 0.998     0.004     0.991      0.998     0.995      0.997    1
Weighted Avg.    0.997     0.003     0.997      0.997     0.997      0.997

=== Confusion Matrix ===

     a     b   <-- classified as
 16211    69 |     a = -1
    13  7441 |     b = 1
```

Figure 33: Entire training set with Weka and LibSVM