Detecting malicious PDF documents

Jarle Kittilsen

Masteroppgave
Master i informasjonssikkerhet
30 ECTS
Avdeling for informatikk og medieteknikk
Høgskolen i Gjøvik, 2011

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

Detecting malicious PDF documents

Jarle Kittilsen

01/12/2011

Abstract

As the internet has become the new playground for organized crime and foreign intelligence, the sophistication of internet attacks has increased. The traditional attacks targeting listening services on the target computer is no longer as viable as it used to, much thanks to firewalls, NAT and more secure administration of servers. This has forced the attackers to find new targets, which they have found in client applications, and in the users themselves. Client-side attacks are now the most used method of attack on the internet. A popular vector for conducting such attacks are malicious PDF documents. Traditional signature based network intrusion detection systems (IDS) have a hard time detecting such threats, and no good alternative solutions have been discovered.

In this thesis we seek the answer to the question ”How can malicious PDF-documents transferred in a network be detected? “ An anomaly based network IDS approach was chosen, several machine learning classifiers were investigated and Support Vector Machines gave the best accuracy and performance. Several features of PDFs are analyzed in order to retrieve those significant for the detection of malicious PDF documents. Experiments were performed to find the best combination of features and SVM configurations to maximize performance of the detection algorithm. A real world study was also performed by implementing the algorithm in a network belonging to the Norwegian Defence.

Sammendrag

Internett har med tiden blitt den nye tumleplassen for organisert kriminalitet og fremmed etterretning. Nettverksangrep over internett har stadig blitt flere og mer sofistikerte. Takket være brannmurer, NAT og bedre sikkerhetsbevissthet blant administratorer er tradisjonelle angrep mot lyttende tjenester på offerets maskin er ikke lenger en foretrukket metode. Dagens metode for angrep er klient-side angrep. En populær vektor for slike angrep er ondsinnede PDF dokumenter. Tradisjonelle signaturbaserte inntrengningsdeteksjonssystemer (IDS) har problemer med å detektere slike angrep, og det finnes ingen gode alternativer.

I denne masteroppgaven forsøker vi å besvare spørsmålet ”Hvordan kan ondsinnede PDF dokumenter detekteres i nettverket? “ En tilnærming med anomali-basert nettverks IDS ble valgt. Flere metoder fra maskin læring ble undersøkt, og Support Vector Machines gav best nøyaktighet og ytelse. Flere attributter i PDF formatet har blitt analysert for å finne frem til de som er signifikante for å kunne detektere ondsinnede PDF dokumenter. Eksperimenter har blitt gjennomført for å finne den beste kombinasjonen av attributter og SVM konfigurasjon for å maksimere ytelsen til deteksjonssystemet. En test har også blitt gjennomført i et virkelig scenario ved å implementere systemet i et nettverk tilhørende det norske Forsvaret.

Acknowledgements

I would first and foremost like to thank my supervisor, Prof. Katrin Franke, for great supervision and feedback at all times of the day, sharing her insight in the world of machine learning and pushing me on towards the goal.

Also, for great co-supervision and perpetual willingness to help, I would like to thank Hai Thanh Nguyen.

I would also like to thank my colleague Kjell Christian Nilsen for sharing his widespread network of contacts with me, without it this thesis would have been very difficult to realize. A thanks goes to the following for contributing to the PDF sample corpus:

- Didier Stevens
- Abuse.ch
- Shadowserver
- W
- E. M.
- Sourcefire
- Websense

Thanks also to my colleagues at the Norwegian Defence Center for Protection of Critical Infrastructure for picking up my slack during my leave of absence.

A big thanks to my opponent, Sjur Hartveit, for his valuable feedback.

And last, but not least, I would like to thank my wife who has allowed me to focus on the thesis, day in and day out, without any complaints.

Jarle Kittilsen, Lillehammer 01/12/2011

Contents

List of Figures

List of Tables

List of Abbreviations

AI - Artificial Intelligence

AUC - Area Under Curve

API - Application Programming Interface

CFS - Correlation Feature Selection

CPU - Central Processing Unit

CV - Cross-Validation

CVE - Common Vulnerabilities and Exposures

CSV - Comma Separated Vector

DM - Data Mining

FN - False Negative

FNR - False Negative Rate

FP - False Positive

FPR - False Positive Rate

GeFS - Generic Feature Selection

GUI - Graphical User Interface

HiG - Høgskolen i Gjøvik (Gjøvik University-College)

HIDS - Host Intrusion Detection System

HTTP - HyperText Transfer Protocol

ICT - Information and Communication Technologies

IDA - Intelligent Data Analysis

IDS - Intrusion Detection System

KDD - Knowledge Discovery in Databases

MD5 - Message Digest 5

mRMR - minimal-Redundancy Maximal-Relevance

MSSE - Microsoft Security Essentials

NAT - Network Address Translation

NIDS - Network Intrusion Detection System

OS - Operating System

PDF - Portable Document Format

RBF - Radial Basis Function

ROC - Receiver Operating Curve

SVM - Support Vector Machine

TN - True Negative

TNR - True Negative Rate

TP - True Positive

TPR - True Positive Rate

XOR - eXclusive OR

1 Introduction

In this chapter the topic of this thesis will be presented, and insight into the challenges we wish to solve is provided. We also give some motivations towards why it is important to solve these challenges, and how this will be a benefit for the community. Finally the research questions are presented, along with the the planned contributions of the project.

1.1 Topic Covered by the Project

Over the past 10-15 years, corporations, government and other organizations, and even the individual person have become ever more dependent on information and communication tecnology (ICT) for effectively solving every day tasks. A disruption in the ICT systems can be catastrophic for an organization heavily dependent on it. A study performed by Ponemon Institute [7] reports a median annual cyber attack cost of $3.8 million for the 45 companies that were surveyed. The annual costs ranged from $1 million to $52 million, such figures could without doubt lead to bankruptcy.

For some organizations, the threat from foreign intelligence and industrial espionage is a big concern. And both organizations and the private citizen have to deal with organized cybercriminals constantly feeding the internet with malware, like viruses, worms and trojans, which are designed to steal our banking information, steal our online identity, create huge botnets, fill our inboxes with spam, and so on [8, 9, 10].

This thesis will focus on a modern and widely used class of attacks on ICT systems, namely client-side attacks. Client-side attacks, as opposed to server-side attacks, are aimed at the client applications running on a users computer, and are often enhanced by exploiting the users lack of information security knowledge.

Client-side attacks have surpassed server-side attacks as the attackers method of choice[11], some of the reasons for this will be discussed later on.

Popular applications to target are the ones that are found on “all” computers; internet browsers like Internet Explorer, document viewers like Adobe Reader, runtime environments like Java and media file viewers like Flash Player. Exploitation of the users lack of knowledge is often done by social engineering, e.g. phishing or scareware.

This project will focus on one of the most widely used attack vectors, namely malicious PDF documents. The goal is to create a new and efficient approach for a network intrusion detection system (NIDS), capable of detecting malicious PDF documents that are transferred over a network. Due to the shortcomings of traditional signature based NIDS, the proposed NIDS will be anomaly based and perform classification using a machine learning classifier.

The author of this paper has been working in the information security field for several years, with intrusion detection and incident handling as one of his main tasks. This will be very valuable in conducting this project, as it will require a deep and practical understanding of intrusion

detection. Also the project will require deep and practical understandings of machine learning techniques, this will have to be acquired by the author, and is available through the project supervisor who has extensive knowledge and experience in the area.

1.2 Keywords

Computer network security, Intrusion detection, Artificial intelligence, Pattern recognition, Feature extraction

1.3 Problem Description

Client-side attacks have become the preferred method of network attacks. Organized crime regularly launch huge campaigns on the internet where the goal is to fool the regular users into opening content exploiting common applications found on most personal computers [12]. This gives the attacker a plethora of vulnerabilities to exploit in all kinds of client applications, as well as exploiting the users lack of security knowledge.

In [13] Provos et al. explains how lately this attack strategy has become prevalent.

> The proliferation of technologies such as Network Address Translators (NATs) and firewalls make it difficult to remotely connect and exploit services running on users' computers. This filtering of incoming connections forced attackers to discover other avenues of exploitation. Since applications that run locally are allowed to establish connections with servers on the Internet, attackers try to lure users to connect to malicious servers. Such attacks fall into the category of "client-side attacks" and have been on the rise for the past couple of years."

Provos et al. goes on to explain another reason for this paradigm shift in network attacks:

> Contrary to the small set of applications running in the tightly managed and frequently updated commercial servers, a personal client computer contains a large number of applications that are usually neither managed nor updated. To make things worse, discovering older, vulnerable versions of popular applications is an easy task: a single visit to a compromised web site is sufficient for an attacker to detect and exploit a browser vulnerability.

One group of such client applications are PDF readers. A PDF reader is found on most computers, whether it is Adobe Reader, Foxit Reader or some other brand. By fooling the user into opening a malicious PDF document or rendering such a document in the browser, an attacker can perform a client-side attack. In fact, using PDFs for this purpose has been prevalent for the last couple of years and is still on the rise [1] as can be seen in figure 1.

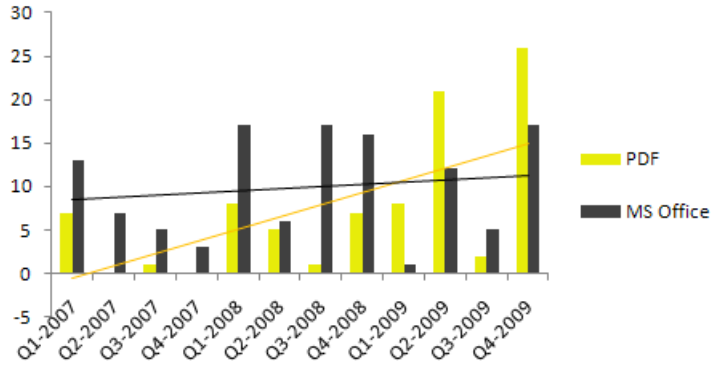

Figure 1: Number of CVEs reported on MS Office and PDF file formats [1]

In [14] Toralv Dirro, a security strategist at McAfee Labs, argues that the percentage of exploitative malware targeting PDF vulnerabilities is skyrocketing. In 2007 and 2008, only 2% of all malware that included a vulnerability exploit leveraged an Adobe Reader or Acrobat bug. That figure jumped to 17% in 2009 and to 28% during the first quarter of 2010.

Protection against such attacks today relies, to a great extent, on signature based IDS and anti virus software. Such tools work great for detecting known attacks that have already been analyzed, and remain static in their signature. However, adversaries are putting great effort into obfuscation of the exploit code, making their attacks dynamic in its signatures. The nature of PDF and it's support for embedded JavaScript gives the attacker great opportunities in making dynamic exploit code, as they allow for techniques such as embedded objects, dynamic and advanced code, encoding and encryption.

Also, the nature of client-side attacks make them ideal for targeted attacks against specific persons or organizations. Such targeted attacks are often carried out by highly resourceful entities, like governments or organized crime. Such entities must be expected to be in possession of zero-day exploits. When such exploits are used, signature based approaches are rendered next to useless.

In the authors own experience from several years as a network security analyst, signature based protection measures have very low detection rate for client-side attacks. The problems of signature based approaches will be bypassed by looking at anomalies in the PDFs, rather than specific signatures.

There exists tools already that will help with the analysis of malicious PDF and JavaScript based on anomalies, e.g. PDFiD [15] or PDF X-ray [16]. However, these tools are tools for manual anaysis that require a lot of user interaction. To the best efforts of the author, no existing network based IDS tools that attempt anomaly detection of malicious PDF have been found.

1.4 Justification, Motivation and Benefits

Client-side attacks have become a huge problem in todays internetworked world. Such attacks are widely used in online organized crime such as identity theft, fake anti-virus and creation of botnets. Also such attacks are often used in targeted attacks, such as the 2011 attack on the Norwegian Armed Forces where several hundred employees received a malicious PDF by e-mail [17], or the 2010 attack on the leader of the Nobel Peace prize committee in Norway [18].

To the authors best knowledge, there is at present no good tools for detecting malicious PDFs in network traffic, as the existing signature based alternatives do not provide good enough detection of these threats.

When the project is solved it will enable detection of modern client-side attacks using PDFs as an attack vector. The method and tool will also be extensible to other attack vectors. The IDS tool will make it harder for organized crime to conduct their large scale campaigns against internet users and it will make targeted attacks more difficult to perform. When solved the entire IDS research community will be beneficiaries and be able to use the new knowledge in applied IDS research. The research will also provide some ideas leading the community towards commercially viable IDS products based on machine learning. A successful project will be of great importance to organizations, and the general population, who have to deal with highly motivated and resourceful threats every day.

1.5 Research Questions

From the previous sections the following research questions are formed, and should lead to contributions relevant to solving the identified challenges:

Main research question:
How can malicious PDF-documents transferred in a network be detected?

Sub research questions:

- Which features are significant for detecting malicious PDF documents?
- Which classifier design and configuration yields optimal performance in malicious PDF detection?
- How can a real-world IDS be implemented based upon our findings?

1.6 Methodology

The main research question of the project is "*How can malicious PDF-documents transferred in a network be detected?*". To solve this question we will develop a prototype IDS that can be used in experiments. This way we can identfy all functions needed for such an IDS, and we can get empirical results to support our findings. The IDS will be based on a machine learning classifier that is trained using two different feature vector datasets.

The first feature vector is based on what is perceived to be the "expert knowledge" in the field for detecting malicious PDF, i.e. state-of-the-art or best practice. It contains features that other researchers believe to be significant. A literature study, combined with personal knowledge, will

be the main sources for creating this feature vector. This vector will only serve as a starting point, as it will be subjected to feature selection and performance testing, in order to find the most relevant features for detecting malicious PDF.

The second type of feature vector is one based on n-grams. The use of n-grams in the detection of malicious network traffic has been successful in other areas, like the detection of malicious HTTP headers [19]. The idea of using a classifier trained with n-grams for detection of malicious PDFs is novel and will be compared to the performance of the expert knowledge based feature vector to determine its feasibility for use in a working real-world IDS application. In the experiments different sizes of n will be chosen to see which one gives the optimal performance, mainly based on the success rate of detection, but also on computational performance.

The classifier will first be tested offline, on a dataset containing both malicious and benign PDF documents. Several experiments will be conducted to determine the optimal configuration of the classifier. Based on these experiments the classifier with the best performance will be selected and implemented into a prototype IDS application. This IDS application will then be tested in a real-world setting.

To summarize the methodology:

1. Collect extensive dataset for experimentation.
2. Perform literature study to reveal the state-of-the-art and "expert knowledge" on detecting/-analyzing malicious PDF.
3. Extract an expert knowledge feature vector.
4. Perform feature selection on the feature vector.
5. Extract an n-gram feature vector.
6. Train, test and optimize a machine learning classifier for detecting malicious PDF.
7. Implement and evaluate real-world PDF analysis.

In all experiments quantitative measures will be used to evaluate performance of the classifier. Where applicable qualitative measures will also be used.

1.7 Thesis Contributions

The master thesis project will have the following contributions:

- A new and efficient approach for an anomaly based NIDS for detecting malicious PDFs.
- A detection method that can be extended to other types of network attacks.
- Knowledge on significant features for detecting malicious PDFs.
- A sizable dataset available for further research.

1.8 Thesis Outline

The following chapters are organized to provide the reader with a top-down approach to the problem at hand. First the several disciplines involved in making an IDS based on machine learning is presented, before we dive into the experiments and the results that are obtained.

- Chapter 2-4 introduces the disciplines and the theoretical background needed to follow and understand the rest of the thesis. It also provides related work performed by others within the different areas.
- Chapter 5 presents the proposed IDS solution and the experimental design that will be used in order to obtain the needed knowledge to create the IDS.
- Chapter 6 presents the results of the experiments, along with intermediate discussions and conclusions.
- Chapter 7 presents the overall discussion of the thesis.
- Chapter 8 concludes the thesis.
- Chapter 9 presents suggestions for future work.

2 Network Intrusion Detection

In this chapter we will introduce the reader to network attacks, and how they are detected, what types of intrusion detection systems exist and how their performance is measured. The chapter will also present related work performed by other researchers on the topics.

2.1 Client-side Attacks

We often divide information security into the fields of computer security and network security. By computer security we mean measures taken to protect the individual computer from attacks. When we speak about network security we mean measures taken to protect the entire network, including connected systems and devices, from attacks.

In a similar manner, attacks on our computer systems may be conducted locally at the host, or remotely over the network. We often refer to the latter as network attacks.

Traditionally network attacks have been targeted at the services running on the computers and servers, in so called server-side attacks. These attacks rely on the target running services on open ports, and exploits vulnerabilities in these services. Server-side attacks are generally conducted in five phases, described by Ed Skoudis in [20].

- Reconnaissance
- Scanning
- Exploitation
- Keeping Access
- Removing tracks

However, with more and more computers protected from such attacks by firewalls set up to block new incoming traffic by default, attackers have changed tactics. Firewalls typically block new inbound connection attempts, but allow users behind the firewall to create outbound connections. This allows both parties of an established connection to communicate freely in both direction over that channel [21]. This fact is exploited by attackers in what we call client-side attacks.

Client-side attacks exploit vulnerabilities in client software, such as web browsers, e-mail applications, media players, runtime environments, and last but not least document viewers.

As we have already seen in figure 1, the exploitation of PDF document viewers has been significant for the last couple of years, and seems to still be on a rise. PDF document viewers are popular targets for several reasons.

First of all we have all got one, and PDF is the de-facto standard for document exchange. Hence, we are all able to open a PDF document, and expect to receive PDF documents from all kinds of sources.

Second PDF is an old format, but at the same time extremely versatile. This allows the attackers to use the versatility to exploit pieces of code in ways that the authors never could have imagined. Chapter 3 will provide a deeper look at the PDF format and how it is exploited.

Third there is a huge user population out there who don't care enough to update their PDF viewer software. A recent study performed by anti-virus vendor Avast [2] shows that 6 out of 10 users run a vulnerable version of Adobe Reader. Figure 2 shows the patch level of the Adobe Reader user population in greater detail.

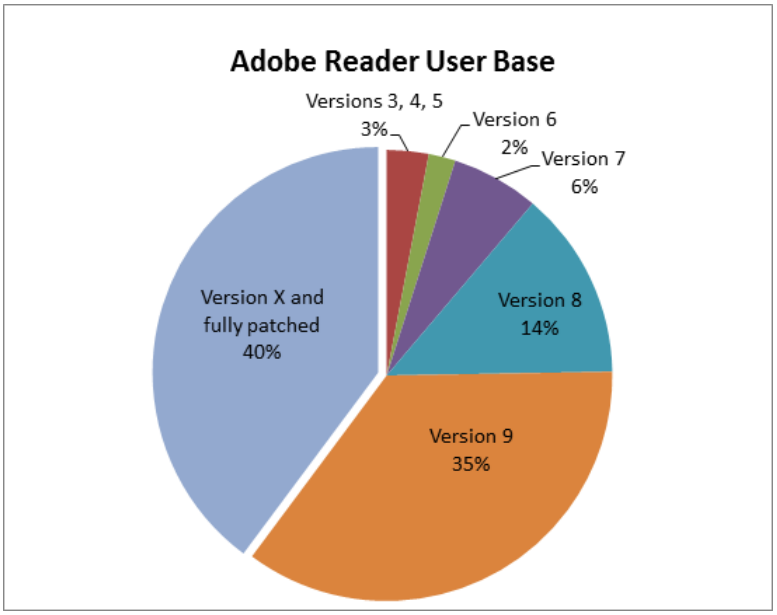

Figure 2: Patch level of Adobe user base [2]

2.2 Intrusion Detection

In security, and also information security, we often speak of three general principles; Protection, Detection and Reaction.

- *Protection* is all measures that we can put in place before an incident has occurred. In information security this includes measures such as firewalls, passwords and policies.
- *Detection* is the measures that we can put in place to be alerted when an incident has occurred, or is occurring, on our systems. In information security this includes measures such as anti-virus and IDS.
- *Reaction* is all measures that we can put in place after an incident has occurred. In information security this includes measures such as digital forensics, backup and insurance.

In this thesis we are only concerned with *detection*, as we are trying to build a system capable of detecting malicious PDF documents being transferred over a network. In other words we are

attempting to create a narrow-scope network intrusion detection system.

2.2.1 Classification of IDS

Generally an intrusion detection system can be classified in two different ways; by scope of protection or by detection model [22].

In the scope of protection an IDS may be classified as host based or network based. A host based IDS (HIDS) is installed on the individual computer, and will protect only this computer. It collects it's data from sources internal to the computer and monitors file system activities and application executions. A network based IDS (NIDS) is deployed at a strategic point in the network. It monitors the network traffic for suspicious activities.

In the scope of detection model an IDS may be classified as signature (misuse) based or anomaly based. A signature based IDS, like Snort [23] relies on comparing certain patterns (signatures) indicating maliciousness, to the packets in the network traffic. This approach is similar to the way of detecting viruses in many antivirus applications. To create such a signature the malicious activity needs to be observed and analyzed. Hence, signature based IDS will never be able to detect novel attacks that have never been seen before (zero-day). An anomaly based IDS, like IDES [24], establishes a profile of what is normal, and alerts when deviations from this profile is detected. Anomaly based IDS can overcome the shortcoming of the misuse detection systems and has the potential to detect novel attacks. However, it is not easy to define normal behaviors and these systems have a high risk of generating false positives.

In this thesis we wich to overcome the limitations of the exsiting signature based approaches by creating an anomaly based NIDS capable of detecting novel malicious PDF documents.

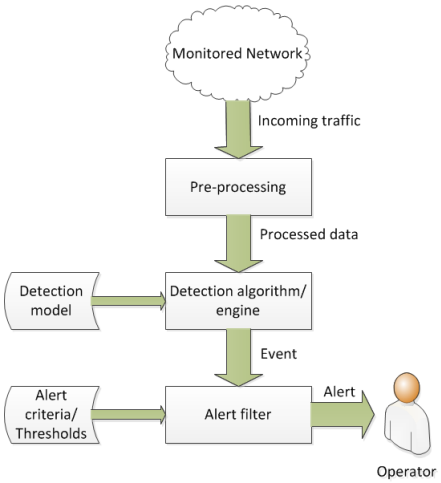

Figure 3: Generic network IDS

Figure 3 shows how an IDS generally functions. In the figure we see that traffic is first taken from the monitored network. This traffic is sent into a pre-processing step, where it is prepared for the detection algorithm. This preparation may include parsing of protocol headers and data,

decoding of data, assembly of fragmented data, and much more.

The processed data is then passed on to the detection algorithm. In a signature based IDS this algorithm is basically a pattern matcher searching for special signatures in the data. In the case of anomaly based IDS the algorithm looks for activity outside (or within) an established baseline. The anomaly based algorithm can be as simple as just a set of thresholds. However, in this master thesis will will implement this as a machine learning algorithm.

When the detection algorithm has detected an event, this event is sent to the alert filter. This filter is tuned by the operator who decides which events will cause an alert. Such tuning of the filter may include setting IP addresses, ports or session states that should not generate an alert, and also setting thresholds that say that several events of the same type must be observed before generating an alert. The filter is mostly used to filter out false positives.

If alert criteria are met, an alert is generated and presented to the operator, who will respond with the proper action.

This thesis is about creating an anomaly based network IDS, where a baseline for what is considered normal and what is considered abnormal will be established through training of a machine learning classifier on a large labeled dataset.

In [25], Sommer and Paxson look into the basic challenges of making an IDS based on machine learning techniques. The challenges they point out are: Difficulties with outlier detection, high cost of errors, semantic gap for the operators, diversity of network traffic, difficulties with evaluation and operation in adversarial environment.

The master thesis project proposed in this paper will have to take these challenges seriously to avoid the same pitfalls as other researchers in the area has fallen into. Sommer and Paxson go on to provide some advice for future research in the field:

- Understand the threat model
- Keep the scope narrow
- Reduce the costs
- Realistic evaluation

Following the provided advise will raise the quality of the master thesis project, therefore they will be taken into consideration when making the experimental design.

2.2.2 IDS Performance

Before discussing IDS performance we need to introduce some terminology. When an IDS makes a decision it has four possible outcomes.

- *True positive (TP)* - The IDS gives an alert and there has actually been an incident.
- *True negative (TN)* - The IDS does NOT give an alert, and there has actually NOT been an incident.
- *False positive (FP)* - The IDS gives an alert, but there has actually NOT been an incident.
- *False negative (FN)* - The IDS does NOT give an alert, but there has actually been an incident.

From these we can define some measures of performance.
True positive rate (TPR) is the number of true positives over the total number of incidents. TPR is also referred to as *sensitivity* in machine learning, and is discussed in greater detail in section 4.1.2.

$$TPR = \frac{TP}{TP + FN} \tag{2.1}$$

True negative rate (TNR) is the number of true negatives over the total number of non-incidents. TNR is also referred to as *specificity* in machine learning, and is discussed in greater detail in section 4.1.2.

$$TNR = \frac{TN}{TN + FP} \tag{2.2}$$

False positive rate (FPR) is the number of false positives over the total number of non-incidents.

$$FPR = \frac{FP}{TN + FP} \tag{2.3}$$

False negative rate (FNR) is the number for false negatives over the total number of incidents.

$$FNR = \frac{FN}{FN + TP} \tag{2.4}$$

Base rate is the total number of incidents over the total number of decisions that are made.

$$Baserate = \frac{TP + FN}{N} \tag{2.5}$$

To be able to use these measures one needs a dataset where every packet is labeled with the correct output class, e.g. benign or malicious. Finding or creating such a dataset is a big challenge for IDS research. A large part of the IDS research community relies on the outdated KDD Cup '99 dataset[1]. This dataset has, in addition to being hopelessly old, been criticized on several aspects in papers such as [26, 27, 28]. The critiques include the age of the dataset, the lack of documentation on how it was created, the existence of artifacts that makes classification trivial and bad statistics.

In [25] the authors promote real world data as the "gold standard" for IDS evaluation. This advice has been followed in papers such as [19], and will also be followed in this master thesis project.

In this thesis we create a new task specific dataset from thousands of recently collected benign and malicious PDF files. More on the dataset and the collection of this is presented in section 6. A real world test will also be performed according to this advice.

[1] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

3 The Portable Document Format (PDF)

The Portable Document Format (PDF) was created by Adobe Systems in 1993 as an open standard for representing documents independent of software, hardware and operating system. In 2008 the PDF format was officially released as an open standard by the International Organization for Standardization as ISO 32000-1. [5]

Since its release PDF has become the de facto industry standard for document exchange. Much of its success lies in the flexibility the format offers. In addition to containing text and images, the format also supports the embedding of Javascript and Flash, and has the ability to open external resources from the local computer or the Internet [29]. These features however are also the features used by attackers to exploit vulnerabilities in the PDF document viewers.

3.1 PDF File Structure

A PDF file consist of four main parts; the header, the main body, the cross-reference table and the trailer.[5]

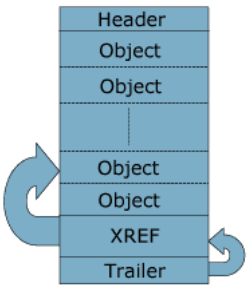

Figure 4: The structure of a PDF document [3].

The header should always be the single first line of the PDF file. It should be of the format `%PDF-M.N`, where M.N denotes the version of the PDF standard used when producing the document. The latest version per time of writing is 1.7.

The main body is where we find the contents of the PDF document represented as objects or object streams. Objects will be discussed in section 3.2.

The cross-reference table, aka the xref table, is a table specifying the byte offset of every object specified in the main body. The xref table is used for direct access to a specified object without searching.

The trailer giving the location of the cross-reference table and of certain special objects within the body of the file, like the root/starting (/root) object. A single line fixed marker of the format `%%EOF`, marks the end of the trailer, and thus the document. The trailer should be located directly after the xref table. A PDF viewer conforming to the standard should read the file from the end and the trailer therefore serves as a reference for locating the xref table.

3.2 Objects

The main part of a PDF file consists of objects. There are eight different types of objects defined in the PDF standard, which will be briefly introduced here.

- **Boolean values** - Can either be *True* or *False*
- **Numeric objects** - PDF provides the well-known concepts of integers and real numbers. Integers are represented with decimal digits, optionally preceded by a sign (+/-). E.g. 42 -23 Real numbers are represented with decimal digits, with a leading, trailing or embedded decimal point (period), and optionally preceded by a sign (+/-). E.g. 42.2 -23.2123 .0101
- **Strings** - A string object can be either in literal or hexadecimal form. Literal strings are enclosed by parenthesis, e.g. (Hello World!). Hexadecimal strings are enclosed in angle brackets, e.g. <48656c6c6f20576f726c64210d0a>. Strings size is limited to 32kB in a 32-bit environment.
- **Names** - Names are used to name other objects. A name is defined by putting a forward-slash in front of the sequence of characters making the name, e.g. /a_name. The name must be unique.
- **Arrays** - Arrays are one-dimensional sequences of other objects. The array may consist of any type of objects. Arrays are enclosed in square-brackets, e.g. [(Hello World!) 42 True]. Arrays may also contain other arrays.
- **Dictionaries** - Dictionaries are tables of key-value pairs. The keys are names and the values can be any kind of object. Dictionary objects are the main building blocks of a PDF document. They are commonly used to collect and tie together the attributes of a complex object, such as a font or a page of the document.
- **Streams** - A stream object is, like a string, a sequence of bytes. The stream objects is however not limited in size. Objects with large data sizes are therefor represented as steams, e.g. images, scripts and pages. The data of a stream is placed between the starting keyword *stream* and the ending keyword *endstream*. All streams must have a *Length* entry that indicated the size of the stream in bytes. Streams may also be subjected to filters that compress or encode the data within them. More on this in section 3.3
- **The null object** - The null object is simply NULL. It is returned when referencing no-existing objects or empty dictionary entries.

Objects may be labeled with a unique object identifier so that they can be referenced by another object. A labeled object is called an indirect object. The object identifier consists of two parts; the object number and the generation number.

The object number is a positive integer assigned to the object.

The generation number is a positive integer used only to show what revision number of the object. In a newly created PDF file all generation numbers will be set to zero.

Together the object number and generation number uniquely identifies an indirect object. Defining an indirect object is done in this manner:

```
7 0 obj
 (Hello World)
endobj
```

This indirect object can now be referred to from any other object using the reference `7 0 R`. Like in the simplified example shown below where the object 4 0, which is a page, refers to the content in object 7 0.

```
4 0 obj
<<
  /Type /Page
  /Parent 3 0 R
  /MediaBox [0 0 612 792]
  /Contents 7 0 R
  ...
>>
endobj

7 0 obj
    ...
    (Hello World)
    ...
endobj
```

3.3 Filters

As mentioned in the section above, streams may be subject to filters. Filters serve the purpose of compressing the data, i.e. reducing its size, or encoding the data to a portable format. The use of a filter must be included in the streams dictionary. More than one filter may also be applied to the stream in cascade, the example below show a clear text stream object before and after being encoded using both the /FlateDecode and /AsciiToHexDecode filters. A PDF reader will apply the filters as specified from left to right in order to retrieve the original text.

```
6 0 obj
<</Length 44>>
stream
  BT /F1 24 Tf 175 720 Td (Hello World!)Tj ET
endstream
endobj
```

```
6 0 obj
<<
/Length 44
/Filter /AsciiHexDecode /FlateDecode
>>
stream
  789cd3ad020000d600a8
endstream
endobj
```

A multitude of filters are part of the PDF standard, both for compression and encoding. Some filters have special purposes, e.g. some are specially created for images. Among the filters that perform compression there are both lossless and lossy filters, i.e. when using lossy filters some data/quality is lost due to the compression.

Table 1 gives an overview of the standard filters available for PDF documents, where the first five are most relevant in the context of this paper.

Name	*Description*
ASCIIHexDecode	Decodes data encoded in an ASCII hexadecimal representation, reproducing the original binary data.
ASCII85Decode	Decodes data encoded in an ASCII base-85 representation, reproducing the original binary data.
LZWDecode	Decompresses data encoded using the LZW (Lempel-Ziv-Welch) adaptive compression method, reproducing the original text or binary data.
FlateDecode	Decompresses data encoded using the zlib/deflate compression method, reproducing the original text or binary data.
RunLengthDecode	Decompresses data encoded using a byte-oriented run-length encoding algorithm, reproducing the original text or binary data (typically monochrome image data,or any data that contains frequent long runs of a single byte value).
CCITTFaxDecode	Decompresses data encoded using the CCITT facsimile standard, reproducing the original data (typically monochrome image data at 1 bit per pixel).
JBIG2Decode	Decompresses data encoded using the JBIG2 standard, reproducing the original monochrome (1 bit per pixel) image data (or an approximation of that data).
DCTDecode	Decompresses data encoded using a DCT (discrete cosine transform) technique based on the JPEG standard, reproducing image sample data that approximates the original data.
JPXDecode	Decompresses data encoded using the wavelet-based JPEG2000 standard, reproducing the original image data.
Crypt	Decrypts data encrypted by a security handler, reproducing the data as it was before encryption.

Table 1: Standard PDF stream filters [5]

3.4 Malicious Use of PDF Documents

As already stated, PDF is the de-facto standard for document exchange. Virtually every personal computer has a PDF reader installed, or the owner will simply have a hard time to participate in society. Added with the fact that PDF is a highly powerful and flexible format, it is almost a dream come true for an attacker.

In the following the most common methods of malicious PDF distribution will be discussed. Following this will be a superficial look at some ways the exploits are actually implemented.

3.4.1 Distribution of Malicious PDF

According to [1] there are three main channels for distributing malicious PDF documents. These are mass mailing, drive-by downloads and targeted attacks. These are all client-side attack methods and are discussed in general in section 2.1 ”Client-side attacks“.

Mass mailing is well suited for malicious PDF distribution since most people are accustomed to receiving PDF attachments in e-mails. In a mass mailing scheme large spam campaigns are set up to deliver e-mails containing malicious PDFs to a huge number of users. Social engineering tricks are used to entice the receiver into opening the attached document. Often the contents of such an e-mail shows an ingress style paragraph to a recent news event, with promise of the full

story and exciting details inside the attached PDF document. Popular subjects include:

- E-mail from the government or a big company.
- Politics.
- Recent incident (Accidents, disasters, war).
- Controversial/sexual subjects.

Malicious PDFs sent in mass mailing campaigns will often contain embedded executable payloads, which is extracted and executed when the PDF is opened in a vulnerable reader.

Due to the embedded feature of automatically opening PDF documents in most browsers, *drive-by downloads* is also a popular channel for malicious PDF distribution. A user may not even notice that a PDF has been opened on his computer when falling victim to a drive-by download.

As opposed to PDFs sent through mass mailing, a web-hosted PDF will usually be small and not contain any embedded executables. Instead they contain small pieces of code that, after successful exploitation, will download and execute malicious executables from the internet. Such a scheme gives the attacker great flexibility, as he will be able to update the malicious code that is downloaded at any time.

Targeted attacks are however where malicious PDFs fulfill its true potential. A targeted malicious PDF is targeting an individual or an organization, and is specially crafted to be successful against this target. The chance of success is boosted by carefully researching the target and planning the attack. By gathering information on the target the social engineering content of the attack can be made in such a way that the target is going to have high trust in the received PDF document. Also the exploit can be chosen in such a way that it has a high probability of being successful on the target system.

With targeted attacks there is often a highly motivated and resourceful organization responsible. Such threats may have access to zero-day exploits which will greatly increase the probability of success. The number of targeted attacks is relatively low, and due to its sophistication and stealth many are probably never reported as the victim is unaware of the compromise.

Figure 5: Cyber attack on Norwegian Defence after Libya decision.

Figure 5 shows an example of a targeted attack against an organization. In this case the attacker sent e-mails containing a malicious PDF to over 300 people working in the Norwegian Defence. The contents of the e-mail was an ingress-style paragraph concerning extreme Islam, and a promise of a detailed report in the attached PDF. The e-mail was signed by a fictive employee of a Norwegian government directorate. The malicious purpose of the malicious PDF is not disclosed to public.

3.4.2 Exploit Implementation

New vulnerabilities in PDF readers and associated plugins and libraries emerges all the time, and with most new vulnerabilities follows an exploit. Different kinds of exploits can be used in a malicious PDF, and one single PDF may contain several exploits grouped together. In [1] PDF exploits are grouped into two distinct classes; JavaScript based and non-JavaScript based exploits.

Javascript based exploits are made possible through the JavaScript support in the PDF specification. Attackers know to appreciate the power a scripting language like JavaScript brings to the format. JavaScript is used to exploit vulnerabilities in the PDF JavaScript API and to fill the PDF readers memory with malicious code, using a technique called heap spray[1]. A complete exploit often consists of code that first heap sprays the readers memory with shellcode, and then call a vulnerable function. This may result in the shellcode being executed.

According to [1] the majority of malicious PDFs today use JavaScript in one form or another.

Non-javascript based exploits are far more rare than the javascript based exploits. An alternative to JavaScript is to use PDFs ability to embed Flash content. Such content may be used to exploit vulnerabilities in the Flash engine, or to put shellcode in the heap of the reader. There also exist a

[1]Heap spray means to put a wanted sequence of bytes at a predetermined location in the memory of a target process by having it allocate (large) blocks on the process' heap memory and fill the bytes in these blocks with the wanted values.

vulnerability in the way TIFF-images are handled, which can lead to code execution even without a heap spray.

In addition to such very specific vulnerabilities, the Portable Document Format has many nice features which can help the attacker in building a malicious PDF without the use of any vulnerability per se. At Black Hat Europe 2008 Eric Filiol et al. presented a paper [30] on this subject. The features includes functionality to open other documents, open hyperlinks, change document hierarchy placement, access resources outside the active document, execute applications, open files, print documents, access remote resources, submit resources to remote and import data from user. A deeper look into some of these features and their possible misuse follow:

- OpenAction - The OpenAction function lets the creator of a PDF document define actions to trigger when the document is opened. The function does not do much by itself, but when functions such as Launch or ActionClass area given as input to it things can get ugly. The function is maliciously used to run exploits as soon as the PDF document is opened, giving the victim no chance to stop it.
- AA - The AdditionalAction function works in a similar way as the OpenAction function. However, instead of triggering on the document being opened, it triggers on specific actions set by the PDF creator. Such actions included triggering when a certain page is opened or closed, when clicking certain areas, when mouse is over certain areas, when printing and so on.
- Action class - The Action class contains several functions that can be put inside an OpenAction or AddidtionalAction function. There are Action Class functions for executing files, activating hyperlinks, sending form data and much more. All these functions may aide the attacker in creating a working malicious PDF. Today these threats are mitigated by most PDF readers by showing the victim a confirmation box whenever an action is triggered. However, through social engineering and the limited awareness of the general public this threat is still one to reckon with.

  Launch is a function from the Action class. It allows for execution of any file on the target system, with optional arguments. Before confirmation boxes were implemented into PDF readers this was the single most critical vulnerability in PDF.

  SubmitForm is normally used for electronic forms. It allows the creator of the PDF to send data from the form elements to a specified URL. This provides an excellent data channel for an attacker to retrieve data from the target host.

  ImportData lets a PDF import data into forms from an external file. This allows an attacker to steal information from the victims computer. Used along with the SubmitForm function is provides all the functionality needed to build a spyware PDF.

Other functions may also pose a similar threat, for a more detailed discussion please refer to [30].

3.5 Obfuscation of PDF Documents

Attackers use several methods to hide, or obfuscate, the malicious contents of a PDF document. Such methods are possible due to the powerful and flexible nature of the PDF format. Especially the javascript often found in a malicious PDF document has vast amounts of obfuscation methods.

Obfuscation is meant to throw off the analyst trying to analyze the malicious content, and also to evade detection by signature intrusion detection systems or anti-virus solutions. The techniques may work on their own, but often a combination is used to provide an effective countermeasure against both detection mechanisms and human analysts. In [31], Leif Arne Sand presents some common techniques which will be briefly discussed in the following.

3.5.1 Separating Malicious Code over Multiple Objects

As in any coding language also the malicious code of a PDF document can be spread among several objects. The code is then made in such a way that it during execution reassembles itself into a complete code performing the desired actions [1]. This technique can be used both by taking advantage of the ability to refer to indirect objects in PDF or by using custom code spreading methods in a javascript embedded in the document. Spreading the code over multiple objects will make the work of an analyst much harder and will throw of signature based IDS and antivirus.

3.5.2 Applying Filters

By applying filters the author is able to encode and compress the streams of a PDF document. Attackers can use this feature in order to evade detection by security software. If the software does not support the filters used it will never even see the malicious code. Applying filters will not evade a knowledgeable human analyst, but it will most certainly make his job more time consuming.

3.5.3 White Space Randomization

Since JavaScript ignores whitespace at run-time, it is possible to insert as arbitrary amounts of whitespace characters in the code [32]. While this will not fool the human analyst, signature based detection mechanism may easily be thrown off. Any detection mechanism relying on the hash sum of the JavaScript will also be fooled, as this will change when whitespace characters are inserted.

3.5.4 Comment Randomization

Comments are also ignored by the JavaScript parser at run-time. This means that an attacker may insert or edit comments in the source code to change it's hash sum. This will however only affect detection mechanisms relying on the hash sum. The method has no effect on the human analyst.

3.5.5 Variable Name Randomization

Since one can give variables almost any name one would want, it is possible for the attacker to change variable names. This may fool signature based detection mechanisms looking for specific variable names, but will have little effect on the human analyst.

3.5.6 String Obfuscation

The goal of string obfuscation is to change strings so that they seem meaningless and unreadable to the human analyst. This can be achieved in several ways. The attacker may split the string into several substrings, which are concatenated at run-time. Also the strings may be encoded using schemes such as hexadecimal, unicode, base64 and so on. Finally the attacker may obfuscate the string using an arbitrary function on it, like XOR or substitutions. A deobfuscation function would then be executed at run-time, revealing the true string just before it is used. This method has a huge effect on the human analyst, which will have to spend a lot of time revealing the true content of the strings. The method is also effective for hiding for example shellcode from signature based IDS.

3.5.7 Function Name Obfuscation

This method is applied to hide the use of standard functions, such as the often used unescape() and eval(). Making pointers for such functions, using arbitrary names, will make a human analyst job harder and will bypass signature based detection mechanisms looking for specific functions.

3.5.8 Integer Obfuscation

Integer obfuscation aims at representing numbers in a set of different ways. For instance if the malicious code uses a suspicious memory address e.g. 0x08000000, detection mechanisms may check for this address in the code. Using integer obfuscation the attacker may instead represent 0x08000000 as 16777216*8.

3.5.9 Block Randomization

Block randomization involves changing the structure of the JavaScript in such a way that it functions in the same way, but has a different syntax. The example below shows three different ways of writing a loop that performs exactly the same function [32].

```
for (i = 0; i < 100; i++) { ...do this... }
while (i < 100) {i++; ...do this...}
do { i++; ...do this... } while (i < 100)
```

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

briefly discussed.

In classification the goal of the machine learning algorithm is to assign an object, described with a number of features, into the correct class. The features are independent observable variables, discrete or continuous, like number of legs, number of eyes, length and temperature. While the classes are dependent unobservable discrete variables, like human, animal, insect or alien. Properly trained the machine will be able to put the object with two legs, two eyes, 180cm length and temperature of 37,5 degrees into the human class. While the object with three legs, five eyes, 30 cm length and temperature of 10 degrees is correctly put in the alien class. The algorithm used to classify the objects is called the classifier. There are several well known classifiers:

- Decision trees and rules
- Bayesian classifier
- Nearest neighbor classifiers
- Discriminant functions
- Support Vector Machines
- Artificial Neural Networks
- Hybrid algorithms

In this thesis we will be using a classifier in order to classify PDF documents as benign or malicious. The next section will go on to explain the process of classification.

4.1.1 The Classification Process

We can describe a general high level process that is followed for all classification problems. The process includes a training phase and a testing/operational phase, as depicted in figure 6.

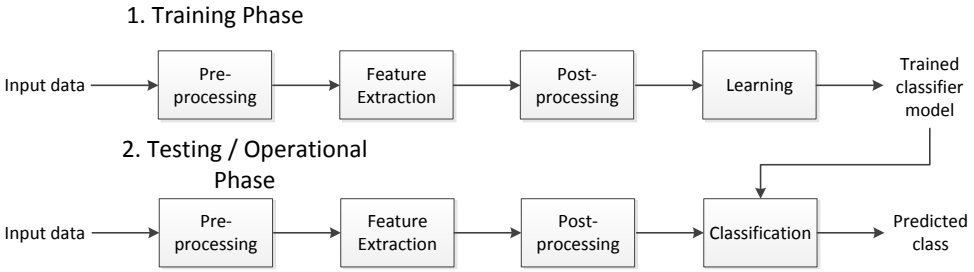

Figure 6: The classification process

The Training Phase

The objective of the training phase is to train a classifier, i.e. give it experience E, so that it can perform its classification task T with a good performance P. As input to this phase we have the training input data. This input data may be of many forms. In the case of this project it is in the form of PDF documents, other examples may be images, comma separated vector files, audio, or just about anything that you can digitize and feed into a computer. For a classifiers training phase the input data will need to be labeled with the correct output class.

The raw input data may be in a form which is not suitable to extract features from directly, therefore the first step in the training phase is *pre-processing*. In the pre-processing step the goal is to clean up the data, and get it into a form which is suitable for feature extraction. Examples of pre-processing tasks include noise filtering, decoding, decompression, removal of excess data and so on.

The next step is the *feature extraction*. In this step the input data is searched for a given set of features, and when such a feature is found it is either counted or measured. For example the existence of the word ”Hello“ in a text may be counted or the distance between two objects in an image may be measured. The output of the feature extraction step is a feature vector, a list of features with its corresponding values.

The third step is the *post-processing* step. This is where such tasks as feature selection and normalization is performed. Such tasks are performed on the input data in order to make it better suited for the learning algorithm (normalization) and to make sure that only the relevant data is fed into it (feature selection).

Finally the data is fed into the *learning algorithm*, where for every new piece of data the classifier gains a little experience and hopefully performs better at the classification. In this step we can say that the classifier learns what kind of input belongs to class A and what type of input belongs to class B. As an example the task of the classifier in this project is to classify the input PDF documents as either benign or malicious. For every PDF it processes in the training phase it learns more about what a benign PDF looks like and what a malicious PDF looks like, and thus will improve its performance as a classifier in the next phase.

The output of the learning step is a trained classifier that will be used in the testing/operational phase.

The Testing/Operational Phase

The objective of the testing phase is to classify every new piece of input data in the correct output class. The input data needs to be labeled and in the same form as in the training phase, however the input labels will no longer be provided to the classifiers, as it is the task of the classifier to predict the class of the new data.

Just as in the training phase the raw input data needs to be pre-processed, the same features need to be extracted and the same post-processing must be performed. If the same steps are not performed in the operational phase as in the training phase the classifier will not perform as intended or may not work at all.

In the classification step the trained classifier processes the input feature vector and based on its experience from the training predicts its output class label. The predicted class label is

then compared to the actual labels of the input data. Several performance measures may then be calculated to assess the performance of the classifier. Performance measures are discussed in section 4.1.2.

If the performance of the classifier is satisfactory it can be put to operational use. The operational phase is exactly the same as the testing phase, except that there no longer exists any labels. The classifier now has to perform its true task of classifying data that has never been classified before.

4.1.2 Measuring Performance

When a classifier is trained for a task its performance at that task needs to be measures in order to determine if it can be of any use. In this section the measurement strategy called cross-validation and some of the more frequently used performance measures as described in [35] will be discussed.

Cross-validation

As we have seen, when we want to test a classifiers performance we need to have one labeled dataset for training and one labeled dataset for testing. The naïve way of measuring performance would then be to split our complete dataset in half, and use one half for training and the other for testing.

However, labeled data samples is a scarce resource and we really would want to use all our samples for both training and testing. Then again, training and testing using the same dataset is not a valid solution and will provide overly good performance results.

To solve this dilemma we use a well known scientific method called cross-validation, or more precisely k-fold cross-validation. In k-fold cross-validation we split our complete dataset into k subsets. Training is then performed on k-1 subsets, and testing is performed on the remaining subset. The training and testing is re-iterated for k iterations, each time testing with a different subset. For each iteration performance measures are calculated and after all k iterations they are averaged to yield the overall value for the performance measures.[35]

Typically k=10 is chosen, which means that the dataset is split into 10 subsets. For each iteration 9 subsets (90% of the samples) are used for training and the last one (10% of the samples) is used for testing.

Performance Measures

Sensitivity is a performance measure for two-class problems only. It is defined as the relative frequency of correctly classified positive samples. In information security this is often referred to as the true positive rate.

$$Sens = \frac{TP}{TP + FN} \tag{4.1}$$

Specificity is also a performance measure for two-class problems only. It is defined as the relative frequency of correctly classified negative samples. In information security this is often referred to as the true negative rate.

$$Spec = \frac{TN}{TN + FP} \tag{4.2}$$

Accuracy is also called the *Success Rate*. It is defined as the number of correctly classified samples (n_{corr}) over the total number of samples (n).

$$Acc = \frac{n_{corr}}{n} \tag{4.3}$$

For a two-class problem the accuracy may also be written:

$$Acc = \frac{n_{TP+TN}}{n} \tag{4.4}$$

Balanced accuracy or *balanced success rate* takes into account any differences in the number of samples belonging to each class in the test data (unbalanced dataset). It is defined as the number of correctly classified samples from class 1 ($n_{corr_{c1}}$) over the total number of samples in class 1 (n_{c1}), plus the number of correctly classified samples from class 2 ($n_{corr_{c2}}$) over the total number of samples i class 2 (n_{c2}), divided by 2.

$$BalAcc = (\frac{n_{corr_{c1}}}{n_{c1}} + \frac{n_{corr_{c2}}}{n_{c2}})/2 \tag{4.5}$$

Confusion matrix is a matrix showing the numbers of TP,TN,FP and FN in a very informative way. From the confusion matrix it is easy to read how well the classifier has done, and what it struggles with.

		Given labels	
		Benign	Malicious
Predicted labels	Benign	TN	FN
	Malicious	FP	TP

Table 2: Confusion matrix

Receiver Operating Characteristic (ROC) curves show the relationship between the classifiers sensitivity and specificity. On the horizontal axis the false positive rate (1-specificity) is represented, on the vertical axis the true positive rate (sensitivity) is represented. A classifier is then tuned by varying the thresholds for the values on the axes. The ideal classifier is the one where the true positive rate is 1 for all values of the false positive rate, although this is never achieved for non-trivial problems. Instead the curve is optimized for the problem at hand, depending on what costs more of a false positive or a false negative.

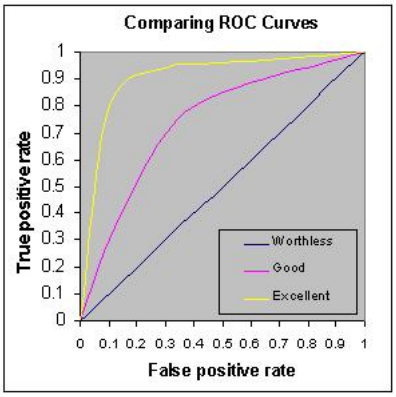

Figure 7: ROC curves

The quality of the classifier is reflected in the *Area under the ROC curve (AUC)*. The quality can roughly be divided into:

- 0.9-1 - Excellent
- 0.8-0.9 - Good
- 0.7-0.8 - Fair
- 0.6-0.7 - Poor
- 0.5 - 0.6 - Useless

Stability of a classification algorithm is the degree to which it generates repeatable results, given different batches of data from the same process [36]. In [36] Peter Turney proposes using m x 2-fold cross-validation for testing stability, observing the agreement of the learned models. Agreement is defined as:

> The agreement of f_1 and f_2 is defined to be $P_{DA}(f_1(a) = f_2(a))$, the probability that f_1 and f_2 assign the random variable to the same class.

We test the agreement of f_1 and f_2 by observing that samples are predicted to be in the same classes (accuracy) over m re-iterations of the 2-fold cross-validation.

4.1.3 No Free Lunch Theorem

In [37] Wolpert and Macready states that:

> For any algorithm, any elevated performance over one class of problems is offset by performance over another class.

Essentially this means that even though one classifier proves to be best for one learning problem, it is most surely not best for all learning problems. That is, there is no ”best“ classifier in a general sense. For us it means that we have to test the performance over several classifiers, since good performance of one classifier using selected features, and target, might not mean good performance for all classifiers.

4.2 Features and Feature Selection

Machine learning algorithms take what we call ”feature vectors“ as input. A feature vector x contains a number of features x_i. It can be represented in several forms, such as the “vector form” of the “feature-value” form. In vector form feature vectors describing fruits may look something like this:

```
round,orange,156,orange
round,green,85,apple
```

While in feature-value form the vectors may look something like this:

```
shape:round,color:orange,weight:156,class:orange
shape:round,color:green,weight:85,class:apple
```

The exact syntax for the vectors will vary for each implementation of a machine learning method.

The values of the features may be of three types [38]:

- *Real-valued numbers.* E.g. weight, height, length, etc.
- *Discrete numbers.* E.g. count, frequency, ranks, etc.
- *Categorical values.* E.g. color:{red,white,blue,green}, shape:{round, square, oval}, etc.

In the examples above we have also included the class label (apple/orange) for the feature vector. This is known as the output target and is used for training the machine learning algorithm and testing its performance.

4.2.1 Feature Extraction

To create the feature vectors one first needs to decide what features are relevant to the learning task and then do the required measurements to determine the values of the features on a sufficient number of samples.

By sample we mean the objects that we want the algorithm to predict something about. For instance the fruits of the above example, or the PDF files that we will be working with in this thesis. The sufficient number of samples will vary from one case to the next, and estimating the performance gain of adding more samples is a difficult task. One way is to study classification accuracy as a function of training set size is by building empirical scaling models called learning curves [39]. Learning curves estimate the empirical error rate as a function of training set size for a given classifier and dataset. Generally we can say that a larger set of samples is better, as long as it does not have severe adverse effects on the computational resources needed to train the algorithm. Ideally the samples should also be evenly distributed over all the output classes, however this is not always achievable.

What features are relevant is often based on “expert knowledge”, that is the knowledge of the human experts working in the field and performing the task at hand. This expert input may serve as a starting point before doing further analysis on the performance of the features and automatic feature selection (see next section).

To measure the values of the features any appropriate tool can be used, from manual measurements using a ruler or a tape measure, to sophisticated and automated measurements using computers with specialized applications.

4.2.2 Feature Selection

Feature selection simply means to select the relevant features for the learning task at hand. Feature selection is done manually and the quality of selected features depends strongly on expert knowledge. However, humans may fail and introduce features that have no importance, or even has adverse effects, on the learning task. By removing irrelevant and redundant features, we can improve classification performance and reduce the computational complexity. For automatic feature selection, the wrapper and the filter models from machine learning are frequently applied. The wrapper model assesses the selected features by learning algorithm's performance. Therefore, the wrapper method requires a lot of time and computational resources to find the best feature subsets. The filter model considers statistical characteristics of a data set directly without involving any learning algorithm [40].

Several algorithms exist for automatically selecting features, in this thesis we will be using two filter model algorithms: Golub-score and Generic Feature Selection

Golub-score Filter Method

The *Golub-score* algorithm uses the statistical measure "golub-score" to rank the features by their relevance to the learning task. The Golub-score can be expressed as:

$$F(x_i) = \left| \frac{\mu_i^+ - \mu_i^-}{\sigma_i^+ + \sigma_i^-} \right| \tag{4.6}$$

Where x_i denotes the i'th feature. μ_i^+ and σ_i^+ denotes the mean and standard deviation for the i'th feature over all samples belonging to the class +. And μ_i^- and σ_i^- denotes the mean and standard deviation for the i'th feature over all samples belonging to the class -.

$F(x_i)$ will give the highest score to those features whose expression levels differ most on average in the two classes and also favors those with small deviations in scores in the respective class. We then chose to keep the features with the highest score [41].

Generic Feature selection

Generic feature selection (GeFS) is a fusion of the correlation feature-selection (CFS) measure and the minimal-redundancy-maximal-relevance (mRMR) measure as presented by Nguyen et al. in [40].

A generic feature-selection measure used in the filter model is a function $GeFS(x)$, which has the following form with $x = (x_1, \ldots, x_n)$:

$$GeFS(x) = \frac{a_0 + \sum_{i=1}^{n} A_i(x)x_i}{b_0 + \sum_{i=1}^{n} B_i(x)x_i}, x \in \{0,1\}^n \tag{4.7}$$

In this definition, binary values of the variable x_i indicate the appearance ($x_i = 1$) or the absence ($x_i = 0$) of the feature f_i; a_0, b_0 are constants; $A_i(x)$, $B_i(x)$ are linear functions of variables $x_1, \ldots, x_n$.

The feature selection problem is to find $x \in \{0,1\}^n$ that maximizes the function $GeFS(x)$.

$$\max_{x \in \{0,1\}^n} GeFS(x) = \frac{a_0 + \sum_{i=1}^{n} A_i(x)x_i}{b_0 + \sum_{i=1}^{n} B_i(x)x_i} \tag{4.8}$$

There are several feature selection measures, which can be represented by equation 4.7, such as the correlation-feature-selection (CFS) measure, the minimal-redundancy-maximal-relevance (mRMR) measure and the Mahalanobis distance.

Solving equation 4.8 is based on solving a mixed 0-1 linear programming problem (M01LP), the solution process is very complex and is out of the scope of this paper.

4.2.3 The Ugly Duckling Theorem

The "Ugly Duckling Theorem" states[42]:

> Given that we use a finite set of predicates that enables us to distinguish any two patterns under consideration, the number of predicates shared by any two such patterns is constant and independent of the choice of those patterns. Furthermore, if pattern similarity is based on the total number of predicates shared by two patterns, then any two patterns are "equally similar."

To put it more simply: *There is no problem-independent, privileged or "best" set of features, or feature attributes.*

4.3 Classifiers Used in this Thesis

In this thesis we will be testing several different classifiers, both in order to select the optimal classifier for the task at hand and to deal with the ”No free lunch“ theorem. In this section the classifiers BayesNet, C4.5, Multilayer Perceptron and RBFNetwork will be briefly presented, while Support Vector Machines will be presented in greater detail as this is the classifier that was found to be best for the classification task of this thesis study, and is used throughout the paper. All classifiers described here are implemented in the Weka machine learning framework [43], which will be used in testing later on, and some of the classifier descriptions have been found in the Weka documentation.

4.3.1 BayesNet

Bayesian networks are directed acyclic graphs whose nodes represent random variables in the Bayesian sense: they may be observable quantities, latent variables, unknown parameters or hypotheses. Edges represent conditional dependencies; nodes which are not connected represent variables which are conditionally independent of each other. Each node is associated with a probability function that takes as input a particular set of values for the node’s parent variables and gives the probability of the variable represented by the node.[44]

4.3.2 C4.5

J48 [45] implements Ross Quinlan’s C4.5 algorithm [46] for generating a pruned or unpruned C4.5 decision tree. J48 builds decision trees from a set of labeled training data using the concept of information entropy. It uses the fact that each feature of the data can be used to make a decision by splitting the data into smaller subsets. J48 examines the normalized information gain (difference in entropy) that results from choosing a feature for splitting the data. To make the decision, the feature with the highest normalized information gain is used. Then the algorithm recurs on the smaller subsets. The splitting procedure stops if all instances in a subset belong to the same class. Then a leaf node is created in the decision tree telling to choose that class. But it can also happen that none of the features give any information gain. In this case J48 creates a decision node higher up in the tree using the expected value of the class.

4.3.3 Multilayer Perceptron

The Multilayer Perceptron is an example of an artificial neural network that is used extensively for the solution of a number of different problems, including pattern recognition and interpolation. A Multilayer Perceptron consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. Multilayer Perceptron utilizes a supervised learning technique called back propagation for training the network.[47]

4.3.4 RBF Network

A radial basis function network is an artificial neural network that uses radial basis functions as activation functions. It is a linear combination of radial basis functions. Radial basis function (RBF) networks typically have three layers: an input layer, a hidden layer with a non-linear RBF activation function and a linear output layer.[48]

4.3.5 Support Vector Machines

The Support Vector Machine (SVM) is a state-of-the-art classifier introduced in 1992 by Boser, Guyon, and Vapnik [49]. The SVM classifier is very popular due to its high accuracy and ability to handle datasets of very high dimensionality. As SVM is a kernel method, it only relies on dot-products of the input data which are in turn replaced by *kernel functions*. The advantage of such functions is that it allows non-linear decision boundaries using methods designed for linear classifiers, and gives the ability to handle data with no fixed dimensionality limit. [4]

SVMs can be used to solve two-class learning problems. That means it is used to decide which of two classes an input sample belongs to. The classes are generally denominated 1 (positive) and -1 (negative). The goal of the SVM classifier is to find the optimal boundary between regions classified as positive and negative, this boundary is called the decision boundary of the classifier. Figure 8 shows a trivial learning problem for separating the red and blue class in a two-dimensional space. Many decision boundaries separating the two classes exist, represented as red lines. In three-dimensional space the decision boundary becomes a plane, and in all dimensions above this it becomes a hyperplane. Figure 8 show multiple possible decision boundaries for the problem, however only one is the optimal decision boundary. The optimal decision is the one that is equally distant from the support vectors of both classes.

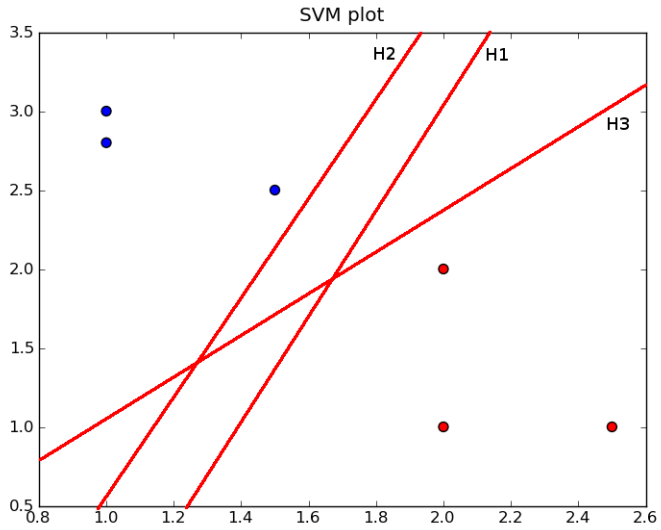

Figure 8: SVM decision boundaries

The support vectors are the vectors (data points) that are nearest to the decision boundary, they are called support vectors because if any of them are removed the decision boundary will change. Removing any of the other learning examples will not influence the decision boundary. The distance from the decision boundary to the support vectors is called the margin. Hence, finding the optimal decision boundary is a question of maximizing the margin. The decision

boundary, support vectors and margin are illustrated in figure 9.

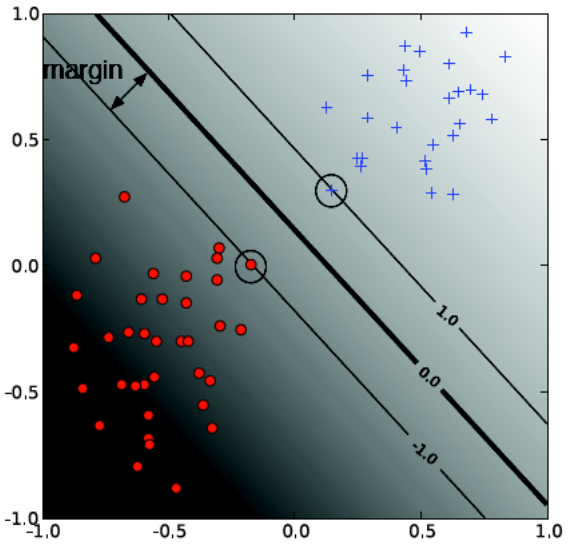

Figure 9: SVM concepts [4]

In this master thesis we use a Gaussian kernel with our SVM. The Gaussian kernel has two values that need to be tweaked for optimal performance, the "inverse-width parameter γ and "the penalty value" or "soft-margin constant" C.

For a large value of C a large penalty is assigned to errors/margin errors. This is seen in the left panel of figure 10, where the two points closest to the hyperplane affect its orientation, resulting in a hyperplane that comes close to several other data points. When C is decreased, those points become margin errors; the hyperplane's orientation is changed, providing a much larger margin for the rest of the data [4].

As seen from the examples in figure 11 the γ parameter of the Gaussian kernel determines the flexibility of the resulting SVM in fitting the data. If this complexity parameter is too large, overfitting will occur (as seen in the bottom panels in the figure) [4].

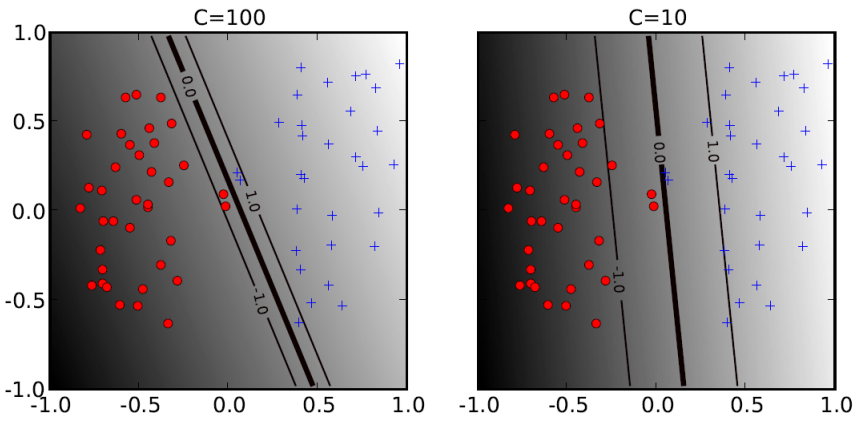

Figure 10: SVM kernel soft-margin [4]

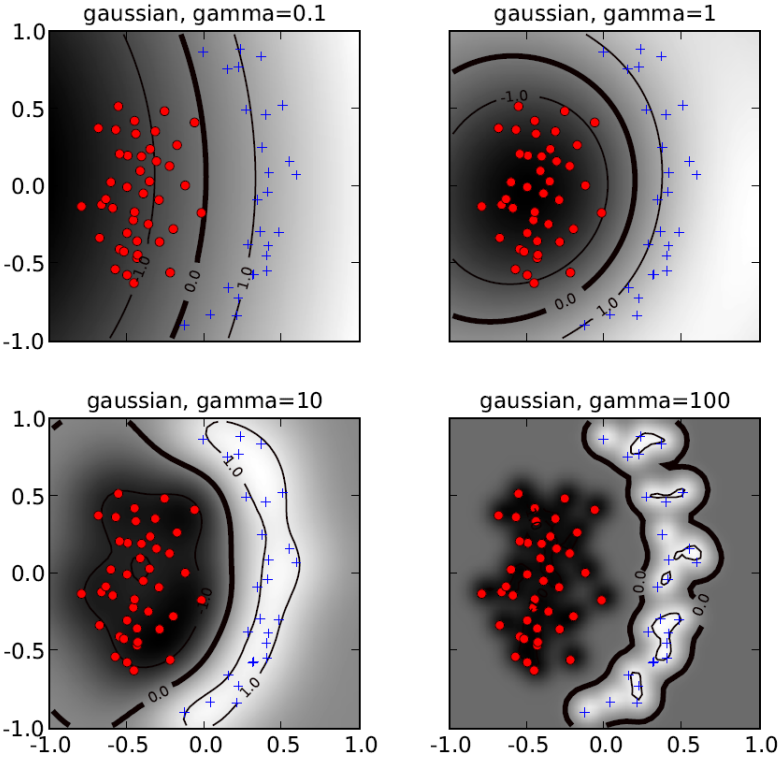

Figure 11: SVM kernel inverse-width [4]

5 Proposed Method and Experimental Design

In this chapter we will first present the proposed IDS solution and all its building blocks. Here we also propose the experiments needed to gain the required knowledge to build the IDS. We then look at some related work to the experiments we propose. Then we move on to the experimental design for the proposed experiments. All experiments will be described in detail, and evaluation criteria will be set.

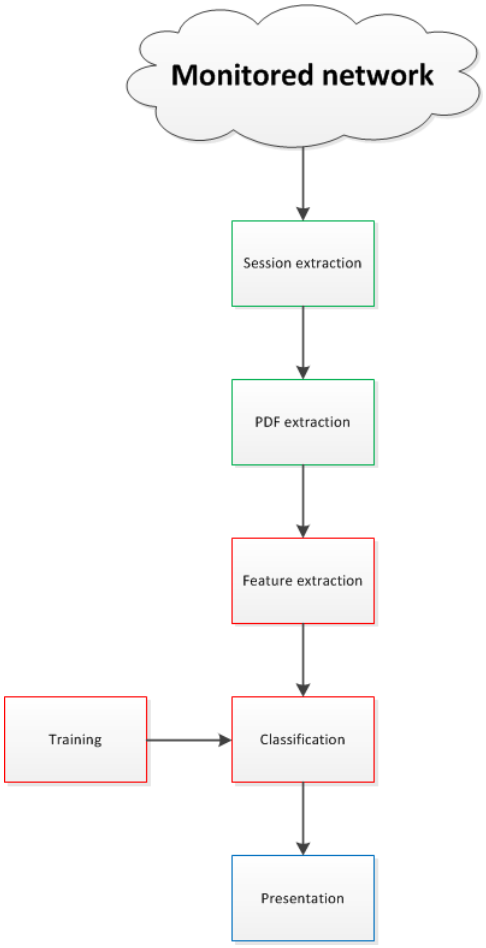

Figure 12: The proposed IDS solution

5.1 Proposed Method

This thesis proposes an IDS capable of detecting malicious PDFs that are transferred over a network. Figure 12 shows a conceptual overview of the proposed solution. In this thesis we will focus on gaining the knowledge needed to build such an IDS. That means that the end result only will serve as a prototype, and not a solution ready for operational use. To create a fully operational IDS will require a lot of engineering, which is not a goal for this master thesis, and will thus be part of the future work.

In figure 12 the main tasks of this thesis is indicated with red boxes. This is where knowledge has to be gained in order to create the system, and includes feature extraction, training and classification. The green boxes indicate engineering tasks that need to be solved in order to do the research. These tasks are session extraction and PDF extraction. The blue box, presentation, indicates an engineering task and feature that will not be implemented in this thesis.

In the following we will take a closer look at each of the building blocks and explain how they are solved or what research we need to do in order to solve them.

5.1.1 The Monitored Network

The monitored network could be any network, from a local office network to the internet connection of a large business. Network traffic can be captured using a tap-box, a monitor port on a switch or by placing a sensor in-line. The important part is to do the capture at a location that all the traffic that needs to be inspected will pass through, typically on a core switch or at the internet gateway.

5.1.2 Session Extraction

Session extraction is performed on every session where a PDF file is downloaded. This task is performed using SNORT, u2boat [23] and tcpflow [50]. The task of SNORT is to capture the entire session where a PDF file has been transferred and to store alert metadata for each stored session. In our setup SNORT v.2.9.1 was used and configured to log output in the unified2 format. To instruct SNORT to capture all sessions with PDF transfer, a new rule was created:

```
pdf tcp any any <> any any (msg:"PDF detected"; flow:from_server,established; \
file_data; content:"%PDF-"; fast_pattern; tag:session,0,packets,60,seconds; \
sid:2000010;)
```

- *pdf* tells SNORT to perform the custom action ”pdf“ whenever this rule is triggered.
- *tcp* tells SNORT to only check this rule against tcp sessions.
- *any any <> any any* tells snort to check this rule against all source and destination addresses in any direction to or from any port.
- *msg:"PDF detected";* gives a description of the rule for the human operator.
- *flow:from_server,established;* tells SNORT to only check established connections, i.e. TCP-handshake completed.

- *file_data;* will in this case tell SNORT to only look in the unchunked and uncompressed payload of HTTP or SMTP sessions.
- *content:"%PDF-";* tells SNORT to look for the string ”%PDF-“ in the payload, which we saw in section 3 marks the start of a PDF document.
- *fast_pattern;* tells SNORT to use its fast pattern matching algorithm.
- *tag:session,0,packets,60,seconds;* tells SNORT to record the session for 60 seconds, so that we get the entire PDF document.
- *sid:2000010;* assigns a signature identifier to the rule.

To create the custom action type ”pdf“ some lines where added to the SNORT configuration:

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

This makes sure that every time a PDF is detected an alert is logged in a text file along with some metadata in this manner:

```
09/27-09:51:43.894986  [**] [1:2000010:0] PDF detected [**] \
[Priority: 0] {TCP} 10.0.0.2:80 -> 10.0.0.4:33897
```

Now we use ”u2boat“ to convert the unified2 log file with the correct timestamp into a pcap file. From the pcap file we extract the correct session using tcpflow and the involved addresses and ports. E.g. ”*tcpflow -r out.pcap 'src host 10.0.0.2 and dst port 33897*“. This yields exactly the session we are looking for and only the traffic going in the direction that the PDF document is sent. However, the application layer headers are still remaining in the file, and we therefore still need to extract only the PDF itself.

5.1.3 PDF Extraction

To be able to process the PDFs we need to remove all header data that is not part of the document format. This is done using our own flow2pdf.py-script[1]. It simply reads the output from tcpflow, searches for the ”%PDF-“ string marking the start of the PDF document, and removes everything before that marker. This yields the PDF document in its original form, ready to be processed.

5.1.4 Feature Extraction

To extract the needed features from the PDF document we have created the pdfextract.py-script[2]. This script uses the PDF-parser found in the ”jsunpack-n“-tool [51] to parse and decode/decompress the PDF documents. Details on feature extraction are found in section 5.3.2.

Research is needed to find out which features are significant and relevant to extract for classifying PDF documents, experiments on feature selection will thus need to be conducted to gain

[1] See appendix A.8
[2] See appendix A.3

this knowledge. This is the motivation for ”Experiment 1 - Feature and classifier selection“, found in section 5.3.3.

5.1.5 Classification

Classification will be done using a machine learning classification algorithm. Several pieces of knowledge need to be researched to complete this task. The best suited classifier needs to be found, however it is not feasible to test all. Therefore a small selection is tested in ”Experiment 1: Feature and classifier selection“, found in section 5.3.3. The selected classifier then needs to be optimized for the best possible performance. This is done in ”Experiment 2.1: Classifier Optimalization“, found in section 5.3.4. Next we need to test the classifier for stability. This is performed in ”Experiment 2.2: Classifier Stability“, found in section 5.3.4. We also design an experiment to test the generalization abilities of the classifier, i.e. its ability to classify samples that are not part of the training dataset. This is done in ”Experiment 2.3: Classifier Generalization“, found in section 5.3.4. And finally we wish to test the classifier in a real-world setting, to gain hopefully an indication of how well the proposed solution will function in operational use. We therefore set up “Experiment 3: Real-world test”, found in section 5.3.5.

5.1.6 Training

Any classifier is dependent on a good dataset for training. In order to conduct our research we need to collect such a dataset. We set up this as “Task 1: Collect dataset” and present it in section 5.3.1.

5.1.7 Presentation

To be a useful tool the solution will need to alert when a malicious PDF is detected and present the alert along with relevant metadata to the IDS operator. Preferably this should be done in a graphical user interface. The detection process itself must be fully automatic and require no operator interaction. Other facilities that can ease the burden of the operator should also be in place:

- Severity indication to help the operator prioritize alerts when time is limited.
- Facilities for linking notes to an event for other operators.
- Automatic checks locally and online to see it this file has been seen and analyzed earlier, in order to prevent duplicate work.

All of the above are engineering tasks that produces no new knowledge and takes very much time to implement. It is therefore deemed to be out of the scope of this thesis.

5.2 Related Work

This section provides insight to some of the work which we are basing the proposed method and the experimental design on.

5.2.1 Proposed Method

In [52] Alex Kirk of Sourcefire presents a plugin for SNORT IDS called Razorback that works in a very similar manner as the proposed IDS. The concept of Razorback is to extract data from the network to perform heavier offline evaluation of the content. However, the project is poorly documented and the detection method it self seems only in the starting pit.

5.2.2 Expert Knowledge

Notable recent work in the field of malicious PDF analysis has been performed by Didier Stevens[3]. Stevens has made some tools for analyzing PDF files, most relevant is the PDFiD tool[15]. This tool looks for and enumerates 18 different keywords, or features, found in the PDF document. These features should be familiar from the PDF standard presented in Chapter 3:

- obj
- endobj
- stream
- endstream
- xref
- trailer
- startxref
- /Page
- /Encrypt
- /ObjStm
- /JS
- /JavaScript
- /AA
- /OpenAction
- /AcroForm
- /JBIG2Decode
- /RichMedia
- /Colors with a value larger than 2^24

According to Stevens paper [53] the most important features for detecting malicious PDFs are:

/Page - which gives the number of pages in the PDF.
Most malicious PDFs have only one page.

/JS, /JavaScript - which indicates the use of javascript in the PDF.

[3] http://blog.didierstevens.com/

Most malicious PDFs use javascript to exploit Java vulnerabilities or to create heap sprays.

/RichMedia - which indicates the use of Flash.
Many malicious PDFs use Flash to exploit Flash vulnerabilities.

/AA, /OpenAction, /AcroForm - which indicates that an automatic action is to be performed. Often used to execute javascript without user interaction.

/JBIG2Decode,/Colors with a value larger than 2^{24} - which indicates the use of vulnerable filters.

In [6], Sophos researcher Paul Baccas provides findings that support the use of several of the features used in PDFiD. Experiments performed on large datasets of benign and malicious PDFs reveal these interesting statistics:

Javascript

Out of 64.616 PDFs that contained javascript, 63.523 were malicious. Only 1.093 were benign. This means that approximately 98% of the PDFs containing javascript were malicious. This means that the presence of javascript in a PDF document is a good indicator of maliciousness. However, malicious PDFs can be made without the use of javascript, and PDFs with javascript can certainly be benign.

Object and Stream Mismatch

Out of 10.321 PDFs that had a mismatch between the number of obj and endobj statements, 8.685 were malicious. Only 1.636 were benign. This means that approximately 84% of the PDFs containing such a mismatch were malicious. Thus object mismatch in PDFs can serve as an indicator of maliciousness.

Out of 2.296 PDFs that had a mismatch between the number of stream and endstream statements, 1.585 were malicious. Only 711 were benign. This means that approximately 69% of the PDFs containing such a mismatch were malicious. Thus stream mismatch in PDFs can also serve as an indicator of maliciousness.

Cross-reference Tables

Out of 5.506 PDFs that had no startxref statement, 5.373 were malicious. Only 133 where benign. This means that approximately 97% of the PDFs lacking a startxref table were malicious. This means that the lack of a startxref table in PDFs is a good indicator of maliciousness.

It is also shown that invalid cross reference tables could be a good indicator, however checking the validity of reference tables is beyond the scope of this paper and would be to time consuming to implement and would take up a lot of computational resources as well. Thus it is not implemented.

Filters

The statistics for the use of filters is somewhat indecisive, but shows that some encodings are more prevalent in malicious PDFs than others. Out of 178 PDFs using the RichMedia filter all 178 were malicious. Out of 689 PDFs using the JBIG2Decode filter, 525 were malicious. The use of such filters could thus be an indicator of maliciousness. Also the ASCIIHexDecode filter shows

interesting statistics. Out of 6.451 PDFs using this filter, 6.404 were malicious.

5.2.3 N-gram feature vector

In [54] the ANAGRAM IDS using n-grams is proposed. In the training phase ANAGRAM collects n-grams from the application layer of incoming data packets and maps this to a Bloom filter. In operation ANAGRAM maps the n-grams from incoming packets to the Bloom filter and reports an anomaly if a bit is 0 in one of the bit-positions of the filter that the n-gram maps to. The paper also suggest how to avoid training the system with malicious traffic and also methods to avoid evasion through mimicry attacks.

A novel PhD thesis written by Konrad Rieck [19] holds promising results using n-grams in intrusion detection. The paper gives several important contributions. First, the paper proposes a method for extracting application layer network traffic payload into feature vectors using n-grams. Second the paper explores the use of kernels to enable efficient learning in high-dimensional data. Third, the paper proposes learning methods based on one-class SVM and neighborhoods. A prototype system, SANDY, is built and evaluated. The system is trained and evaluated using real network traffic and injected network attacks. SANDY shows very promising results considering true-positive and false-negative rates.

Both these papers will provide some input on how to extract and manipulate the n-gram feature vector, as well as many other relevant techniques.

5.2.4 Other tools for PDF analysis

In [55] the tool peepdf is presented. It has similar capabilities as the PDFiD tool for analyzing, decoding and looking for suspicious elements. In addition it has functionalities for creating your own (malicious) PDF files.

PDF X-Ray[4] is a online scanning tool for PDFs. It employs several different techniques to analyze uploaded PDF documents. It compares various hashes of the uploaded PDF to a database of malicious PDF hashes, it checks if the PDF is related to any exploit packs, it scans the PDF file using anti virus products, it looks for suspicious objects and it looks for particularity large objects (>650bytes).

[4] https://www.pdfxray.com/

5.3 Experimental Design

We now go on to describe the design of the experiments proposed in section 5.1.

5.3.1 Task 1: Collect Dataset

For training and testing the classifier a relatively large amount of malicious and benign PDFs needs to be collected. The aim was to collect at least 5000 benign and 5000 malicious PDF documents.

The benign PDFs was collected from the web using a webcrawler. Diversity in the data was ensured by downloading from different locations with different areas of interest.

Several approaches were tested, including the open-source webcrawlers Heretrix[5] and HTTrack[6]. However these approaches proved to require heavy configuration and was abandoned. In stead a simple approach using the standard Linux ”wget“ command was used. Wget provides limited webcrawling functionality, satisfactory for the use in this project. The Alexa top 1 million CSV-file[7] was given as a seed, and wget was instructed to only download files that matched the filter ”*.pdf“. This filter makes sure that only PDF files are downloaded. By using the Alexa top 1 million sites it is assumed that the diversity in sites will also yield a good diversity in the downloaded files.

Wget was executed using the following command:

```
wget -r -N -l 5 ---no-remove-listing -i top-1m_url -A *.pdf
-w 3 --waitretry=14 --random-wait --referer="www.google.com"
--user-agent="Mozilla/5.0 (compatible;)" --limit-rate=80k
```

The crawl took several weeks, and in the end a couple of thousand PDFs were collected through this approach. However, a set of 6052 benign PDF files were also provided by Websense[8] an thus took us beyond the goal of 5000 benign PDFs.

When collecting data from third party webpages there is of course no guarantee that some data may be malicious. The existence of malicious data in the benign dataset will have adverse effects on the planned experiments. To reduce the risk of this happening the following steps have been taken.

All files in the benign corpus was scanned using Trend Micro, MS Security Essentials and AVG Free. This was done using the following setup.

- A virtual machine was created in VMware for each of the three antivirus products.
- A copy of the benign corpus was then put into each VM and scanned.
- All files reported as malicious was deleted from the corpus.

  From all benign files collected none was reported as malicious.

The collection of malicious PDFs was mainly done through personal connections and helpful members of the information security community. Especially Websense and abuse.ch made

[5] http://crawler.archive.org/
[6] http://www.httrack.com/
[7] http://s3.amazonaws.com/alexa-static/top-1m.csv.zip
[8] http://www.websense.com/content/home.aspx

significant contributions.

The nature of how the malicious files have been submitted to the contributors gives great confidence in a very good variety in the dataset. Although the details are not disclosed it can be said that the files malicious PDFs have been detected in a variety of ways and submitted by hundreds of different people from all over the globe, and are almost guaranteed to be malicious.

Thanks to the people and organizations of the information security community the total malicious corpus consists of over 20000 malicious PDF-files.

The collected files were then saved in different directories based on whether they were benign or malicious. All files were renamed to the format: ”MD5sum.<pdf>.<bad|good>“, using our own ”renamer.py“-script[9], like in the following example:
9e107d9d372bb6826bd81d3542a419d6_PDF.bad
This uniquely identifies all files, give a good indication of its contents, and prevent accidental opening/execution of the files.

Then all duplicates, based on the MD5-sum, were deleted using the ”uniqify.py“-script[10] we have created. After this step the data corpus consisted of 7454 unique benign and 16280 unique malicious PDF-files.

Too get a feel for which exploits were present in the malicious PDF file corpus, all the malicious files were scanned by the Microsoft Security Essentials (MSSE) anti virus solution. A script[11] was written to analyze the logs.

- Total files scanned: 16280
- Number of unique threats detected: 373
- Number of threats detected: 27231
- Number of files falsely reported as benign: 950

The numbers mean that MSSE has detected 373 threats with different signatures. However, each signature may include more than one exploit. The high number of threats detected means that several of the PDF documents contain more than one threat that triggers a signature. The complete list and count of threats detected in the dataset can be found in appendix B.

As many as 950 PDF documents where reported as benign, however a manual analysis of a random sample of these 950 PDF documents show that they are indeed malicious. Some use very clever obfuscation techniques to hide its malicious content. We assume that the 950 undetected malicious PDF files are indeed malicious, and that MSSE has no signatures to detect these.

[9] See Appendix A.2
[10] See Appendix A.1
[11] See appendix A.10

5.3.2 Task 2: Feature Extraction

In experiment 1 the goal is to find features that are relevant and significant for the classification of PDF documents as benign or malicious. To get a starting point expert knowledge was sought in the literature. Two different approaches for creating a feature vector is explored. First, an ”expert knowledge“ feature vector was built from the features that other researchers have been using to detect and analyze malicious PDF. Second, a n-gram based feature vector is created.

Expert Knowledge Feature Vector

Based on the work of Stevens and Baccas presented in section 5.2.2 it is decided to create a feature vector with 18 features to serve as a starting point for further research. These features are:

- obj_mis - The mismatch between obj and endobj statements.
- stream_mis - The mismatch between stream and endstream statements.
- xref - The presence of a cross-reference table.
- trailer - The presence of a trailer.
- startxref - The presence of a startxref statement.
- /Page - The number of pages in the document.
- /Encrypt - Is the document encrypted or not?
- /ObjStm - The number of object streams.
- /JS - The number of javascript launched.
- /JavaScript - The number of embedded javascript.
- /AA - The number of additional actions.
- /OpenAction - The presence of an open action.
- /AcroForm - The number of embedded forms.
- /JBIG2Decode - The use of JBIG2Decode filters.
- /RichMedia - The use of RichMedia.
- /Colors - The use of /Colors with a value larger than 2^24.
- /AsciiHexDecode - The use of the asciihexdecode filter.
- /Launch - The use of the /Launch statement.

Our contribution in this sense is to merge the work of Stevens and Baccas, and later on we will use automatic feature selection on these features to empirically determine which features are really significant. Also, these features have never been used for automated detection of malicious PDF documents, which we will attempt later on.

The features are extracted from the PDF document using our own pdfextract.py-script[12]. This script uses the PDF-parser found in the ”jsunpack-n“-tool [51] to parse and decode/decompress the PDF documents. Then the frequencies of all features are counted and the feature vector for each PDF document is created. In addition the script will dump all embedded javascript to separate files for further analysis. The script takes a directory of PDF-files as it’s input and outputs a file with the feature vectors of all the files in the directory.

After running the extraction tool on our entire dataset we end up with a feature vector file containing 23734 feature vectors that we can use in our experiments.

[12] See appendix A.3

As an interesting small experiment we decided to see if the findings of Sophos [6] presented in 5.2.2 coincide with our own dataset. Table 3 shows a comparison of the findings. The percentages are computed by taking the number of malicious files with the specific attribute, divided by the total number of files with the specific attribute.

	Javascript	**obj_mismatch**	**stream_mismatch**	**xref_missing**
Sophos	98%	84%	69%	97%
Our data	97%	90%	21%	63%
	RichMedia	**JBIG2Decode**	**ASCIIHexDecode**	
Sophos	100%	84%	69%	
Our data	89%	25%	99%	

Table 3: Comparison of findings in [6] and this thesis.

As we can see from the table the findings coincide for the javascript feature, the mismatch between obj and endobj statements, the use of RichMedia and somewhat for the lack of a cross-reference table. Why the others do not match is hard to tell, but it may have something to do with how the datasets are collected and the trends at the moment of collection.

N-gram Feature Vector

An n-gram is basically just a contiguous sequence of n items from a sequence of items, like a string. An n-gram with n=1 is reffed to as an ”unigram“, n=2 is a ”bigram“, n=3 is a ”trigram“, and so on.

The use of n-grams are popular in situations where the data has no strict structure, like in language analysis. Unlike IDS handling well defined lower layer protocols like TCP, the proposed IDS will handle the unstructured payload of the application layer. This makes n-grams an interesting approach also for PDF classification.

Based on the work done by other researchers, presented in section 5.2.3, we wished to find out whether such an n-gram feature vector could work for classifying PDF documents as benign or malicious. We therefore extracted five n-gram feature vectors with n=1,2,3,4,5. These n-gram feature vectors where extracted from the PDF documents using the ngramextractor.py-script [13] we have created. The script lets you specify n and then extracts all n-grams from the PDF document. It then counts the frequencies of each unique n-gram. The script takes a directory of PDF-files as it's input and outputs a file with the feature vectors of all the files in the directory. The output feature vector for each PDF sample has a structure like this:

```
id,label ngram1:frequency,ngram2:frequency,ngram3:frequency,...,ngram<j>:frequency
```

The extraction of n-grams soon proved very computationally resource intensive for such large input data as PDF documents represent. After spending days extracting all the feature vector datasets it was decided to do a quick test to determine if the n-gram feature vectors were at all suited for classification. The 3-gram dataset, was selected for the test. It was fed into an SVM with all default settings and 10-fold cross-validation was performed. SVMs are known to

[13] See appendix A.4

handle large dimensional datasets quite well. However the test showed that the computational complexity was to great. The test was ended after the SVM had not finished computation for two whole days.

Using n-grams for classification of PDF documents was thus abandoned due to the computational complexity.

Therefore the ”expert knowledge“ feature vector described in section 5.3.2 is used for further experiments.

5.3.3 Experiment 1: Feature and Classifier Selection

The expert knowledge feature vector we have created relied heavily on human experiance and assumptions, even though it is to a great extent supported by the findings in [6]. In this experiment we wish to use statistical methods in order to analyze the features in an objective manner. Using the two algorithms for automatic feature selection presented in 4.2.2, Golub-score filter and Generic feature selection (GeFS), we will find out what features are really significant for classifying the documents in our dataset. Hopefully the results will be applicable to PDF documents in general.

A great number of feature selection algorithms could have been chosen, however we have to limit the number to two for this master thesis. The Golub-score method is selected for it's simplicity, which makes it easy to understand what it actually does. It has achieved good results in other research, like in [41]. Also, the Golub-score method is readily available in the PyML Python-library, that saves us a lot of time on implementation. The Generic Feature selection-method is a state of the art feature selection method, and has been tested with very good results in IDS context [40]. GeFS is mathematically advanced, and it is therefore not so easy to understand exactly what it does.

The Golub-score is realized using the PyML Python library. GeFS is performed by fellow researcher Hai Thanh Nguyen at HiG who has developed the algorithm.

After running the two algorithms on the dataset we may have two different feature vectors. We then need to decide which one yields the best performance in classification. According to the ”no free lunch“ theorem testing with only one classifier will not tell us what the performance may be with other classifiers. Therefore a small selection of classifiers will be used in the performance test. As a side effect we will at the same time gain knowledge of which classifier is best suited for our classification task.

The experiment will be performed as follows:

1. Automatic feature selection
   a. Perform feature automatic feature selection on expert knowledge dataset using Golub-filter. Save new feature vector.
   b. Perform feature automatic feature selection on expert knowledge dataset using GeFS. Save new feature vector.
2. Evaluate classification performance

a. Perform classification using 10-fold cross-validation on both the original dataset and the new feature vector datasets with BayesNet classifier. Save performance measures.
b. Perform classification using 10-fold cross-validation on both the original dataset and the new feature vector datasets with SVM classifier. Save performance measures.
c. Perform classification using 10-fold cross-validation on both new the original dataset and the feature vector datasets with C4.5 classifier. Save performance measures.
d. Perform classification using 10-fold cross-validation on both new the original dataset and the feature vector datasets with RBFNetwork classifier. Save performance measures.
e. Perform classification using 10-fold cross-validation on both new the original dataset and the feature vector datasets with MulitlayerPerceptron classifier. Save performance measures.

There exists hundreds of classifiers that we could have tested, but we are simply not able to test them all. These five are commonly used in the machine learning community and our results can therefore with greater ease be compared to results of other research.

The performance measures used will be balanced successrate, area under ROC curve and the confusion matrix. Analysis of the results will be performed to establish which feature vector and which classifier achieves the best performance. The results and implications will be discussed. Conclusion and recommendations for future experiments will be provided.

5.3.4 Experiment 2: Classifier Optimalization and Testing

Experiment 2 is actually a series of smaller experiments to optimize the performance of the classifier and to test it thoroughly. In experiment 2.1 we optimize the classifier performance by finding the best configuration, and we try normalization of the dataset. In experiment 2.2 we test the classifier stability. And in experiment 2.3 we test the classifiers generalization ability, meaning it's ability to correctly classify samples it has never seen before.

Experiment 2.1: Optimal Configuration and Normalization

Once we have found the significant features and the best suited classifier in experiment 1, we need to configure the selected classifier for optimal performance. A classifier will often have different variables that can be fine tuned and also we can try to manipulate the dataset through normalization.

To find the optimal configuration of our classifier we design the following experiment:

1. Normalization
   a. Perform normalization on the selected feature vector dataset and classify it using 10-fold cross-validation with the selected classifier. Save performance measures.
2. Search for optimal configuration of variables
   a. Select a range with 5 test values for each variable that needs to be testes.
   b. Perform a grid-search by testing all combinations of values for each variable.
   c. Save and analyze performance measures.

The performance measures used will be balanced successrate, area under ROC curve and the

confusion matrix. Analysis of the results will be performed to establish which if normalization should be used and the optimal values for each variable. The results and implications will be discussed. Conclusion and recommendations for future experiments will be provided.

Experiment 2.2: Classifier Stability

When doing cross-validation it is possible that one does ”lucky draws“ from the samples, so that the reported performance is not representative for the classifier we put into operational use trained on the whole dataset. To remedy this we do a stability experiment by doing five iterations of training and testing using 2-fold cross-validation, according to the stability measure presented in 4.1.2. If no significant changes is shown in the performance for all five iterations we assume that the classifier is stable and reliable.
To test the stability of the classifier we design the following experiment:

1. Do 5 iterations of training and testing the classifier using 2-fold cross-validation.
2. Save and analyze performance results.

The performance measures used will be balanced successrate, area under ROC curve and the confusion matrix. Analysis of the results will be performed to establish if the classifier is stable. The results and implications will be discussed. Conclusion and recommendations for future experiments will be provided.

Experiment 2.3: Classifier Generalization

Running the malicious PDF corups through MSSE shoes that there are a large number of unique expolits present, but that there are also a large number of samples employing the same exploits. At the same time obfuscation and subtle changes in the PDF documents will mean that even though the MD5 hashes are all unique, the underlying exploit may indeed be the same. The large number of samples takes us some way in mitigating the risk of having a very homogeneous dataset. However, we wish to test the generalization abilities of our trained classifier, i.e. make sure that the classifier also works on PDF exploits which with a high certainty is not in our training dataset.

To do this we design the following experiment:

1. Download as many PDF exploits as possible exploiting vulnerabilities discovered after the creation date of the training dataset (January 2011).
2. Train classifier using entire dataset.
3. Classify the new malicious PDF files.
4. Analyze results.

Analysis of the results will be performed to establish if the trained classifier is able to generalize. The results and implications will be discussed. Conclusion and recommendations for future experiments will be provided.

5.3.5 Experiment 3: Real-world Test

To evaluate how our IDS solution will work in a real world setting it will be installed on a sensor owned by the Norwegian Defence. This sensor will provide data from the internet usage of about 300 people. The goal of this experiment is simply to evaluate to what degree the solution will be a helpful tool for security analysts and if the computational complexity is within the limits of a sensor placed in an enterprise size network. The experiment will be performed in the following manner:

1. Train IDS using all full dataset.
2. Leave the IDS operational for 7 or more days.
3. Report qualitatively on results and compare to other measures that are in place.
4. Report on performance (network, CPU, memory) by using the SNORT performance monitor
5. Do manual analysis of any PDF documents classified as malicious to decide if the classification was correct.

Analysis of the results will be performed to establish if the IDS solution is applicable for real-world use. The results and implications will be discussed. Conclusion and recommendations for future experiments will be provided.

5.3.6 Experiment 4: A Closer Look at Embedded Javascript

After completing all of the above experiments we realized that a big number of the malicious PDF files used javascript to perform malicious actions. This discovery is supported by the research of Paul Baccas in [6] where he found that 94.1% of the samples containing javascript in his corpus of 64616 PDF samples where indeed malicious. In our own dataset we found that as many as 14556 of 16296 (90%) malicious samples contained javascript.

This gave us the motivation to look closer at embedded javascript. We decided to create a small experiment to create a classifier capable of classifying embedded javascript as benign or malicious.

1. Collect malicious and benign dataset.
   a. Extract all embedded javascript from malicious PDF-file corpus.
   b. Collect benign javascript corpus by using a webcrawler.
2. Perform feature selection and extraction.
   a. Do a literature study to find expert knowledge on what features may be relevant for classifying javascript.
   b. Add features based on own experience.
   c. Create script for feature extraction.
3. Train and test SVM classifier using 10-fold cross-validation.
4. Analyze results.

The experiments are done by re-using code and gained knowledge from experiment 1 to 3 and

can therefore be executed without too much extra work. This is important since this experiment can be considered a ”bonus experiment“ that was not part of the original plan.

The goal of the experiment is to explore the possibilities of classifying javascript embedded in PDF documents, and serve as a starting point for future research. The performance measures used will be balanced successrate, area under ROC curve and the confusion matrix. Analysis of the results will be performed to establish whether this form of javascript classification is plausible. The results and implications will be discussed. Conclusion and recommendations for future experiments will be provided.

6 Experiment Execution and Results

In this chapter the experiments presented in section 5 are executed. It will begin with a short presentation of the experiment and environment setup, before it continues with the individual experiments conducted. For each experiment the results and their implications are discussed.

The following experiments were conducted in order to build a system capable of detecting malicious PDF:

1. Feature and classifier selection
2. Classifier Optimalization and Testing
3. Real-world Test
4. A Closer Look at Embedded Javascript

6.1 Experiment and Environment Setup

The experiments were executed on a stand-alone computer running Ubuntu Linux. It was decided to not use any virtualization, in order to get the best performance possible. As the malware will never be executed there was no need for the extra layer of protection virtualization would offer.

The computer on which most of the experiments were conducted had the following configuration:

- CPU: 2,4GHz Intel Dual Core
- Memory: 4GB
- Harddrive: 280GB
- OS: Ubuntu Desktop 10.10

The following software and library versions have been used in the experiments:

- Python v2.6.6 [56]
- numpy v1.5.1 [57]
- PyML v0.7.7 [58]
- tcpflow v0.21 [50]
- SNORT v2.9.1 [23]
- MS Security Essentials v2.1.1116.0 Virus definitions updated on day of use. [59]
- Trend Micro Titanium v3.00 Virus definitions updated on day of use. [60]
- AVG Free v10 Virus definitions updated on day of use. [61]

The custom source code we have created for this thesis can be found in appendix A. These include:

- uniqify.py - Generates MD5 hash for sample files and stores them in a dictionary. When duplicates are detected they are deleted to make sure that our dataset has only unique samples.
- renamer.py - Renames all samples to the appropriate filename, defined in section 5.3.1. *MD5sum.<pdf>.<bad|good>*
- pdfextract.py - Extracts the ”expert knowledge“ feature vector from all PDF files in the provided directory and labels the vectors with the provided label.
- ngramextractor.py - Extracts the n-gram feature vector from all PDF files in the provided directory and labels the vectors with the provided label.
- pdfidsvm_grid.py - Performs grid search for optimal SVM values of C and γ based on the provided dataset.
- pdfidsvm_norm.py - Tests SVM performance with optimal C and γ, using normalization on the dataset.
- pdfidsvm.py - Tests SVM performance with optimal C and γ.
- flow2pdf.py - Removes HTTP/SMTP headers from flows extracted by the tcpflow tool.
- classify_new.py - Loads a previously trained SVM model and dataset, and classifies all new PDF samples in the provided directory.
- parse_msse_v2.py - Parses the log output from MSSE and lists out all the threats detected with frequency for each threat.

6.2 Experiment 1: Feature and classifier selection

This experiment was conducted as described in section 5.3.3. The goal of the experiment is to find the significant features for classifying PDF documents as benign or malicious, and to find the best classifier for this task.

First the complete feature vector dataset with all 18 features was run through the Golub-score filter and the Generic feature selection algorithm. This yielded the following results:

Original Feature Vector

`AA, RichMedia, xref, Encrypt, JBIG2Decode, Launch, JavaScript, OpenAction, Colors, JS, obj_mis, startxref, AsciiHexDecode, ObjStm, AcroForm, stream_mis, Page, trailer`

Golub-score Feature Vector

`JavaScript, OpenAction, JS, obj_mis, AcroForm, Page, trailer`

GeFS Feature Vector

`JavaScript, JS, startxref, Page, trailer`

We the tested how well the 5 GeFS features and the 7 Golub-score features performed at the classification task, compared to the 18 original features.

We tested the performance of five different classifiers using all three datasets, as described in section 5.3.3. The results can be seen in table 4.

	BayesNet			**C4.5/J48**			**RBFNet**		
	18	**7**	**5**	**18**	**7**	**5**	**18**	**7**	**5**
Bal succ	0.973	0.94	0.976	0.995	0.995	0.975	0.718	0.797	0.874
AUC	0.996	0.995	0.996	0.997	0.998	0.994	0.879	0.922	0.926
	MLP			**SVM**					
	18	**7**	**5**	**18**	**7**	**5**			
Bal succ	0.96	0.966	0.920	0.995	0.995	0.977			
AUC	0.985	0.987	0.978	0.995	0.996	0.974			

Table 4: Results matrix experiment 1

Figure 13 shows a plot of the balanced successrates of all the classifiers for each of the three datasets.

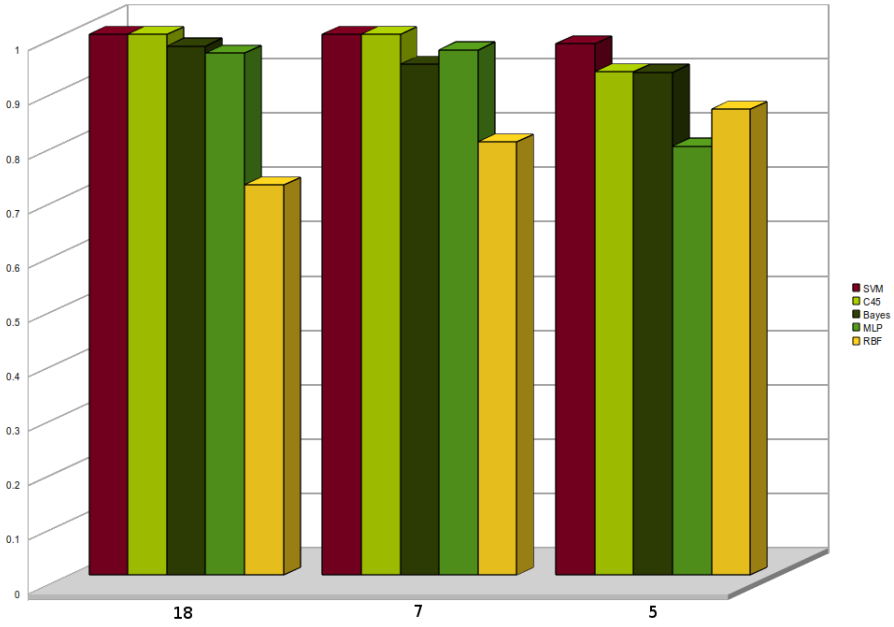

Figure 13: Balanced successrate for all classifiers

As we can see the Golub-score method selected 7 features to be significant, while the GeFS method only selected 5. Using 5 features instead of 7 will have a positive effect on the performance of the classifier. However, when we inspect the numbers of table 4 and look at the plot in figure 13 we see that using the 5 features from the GeFS methods has adverse effects on our classifiers performance in all cases except for RBFNetwork and BayesNet. However, the RBFNetwork classifier performs very bad no matter what dataset is selected, and the BayesNet

classifier only gives the third best performance. On all other trails the 7 feature Golub-score vector outperformed the 5 feature GeFS vector.

Looking further into the results we see that the reduction from 18 to 7 features as virtually no impact on the classifier performance, with regards to accuracy. However, when testing the computational performance in an SVM there are some benefits, as can be seen i table 5.

	18	**7**	**5**
Training time, full dataset	12.59s	6.77s (-50%)	9.12s (-27%)
Testing time, 100 samples	25.8ms	14.4ms (-50%)	26.3ms (+2%)
Testing time, 1 sample	0.25ms	0.14ms	0.26ms

Table 5: Computational Complexity - Experiment 1

We observe that the reduction from 18 features to 7 features offers almost a 50% improvement in both training and testing time. Surprisingly the reduction to 5 features does not perform as well, and only improves training time by 27% and actually increased the testing time by 2%. This has to mean that the five GeFS features form a more complicated function for the decision boundary of the SVM.

Time measures are always complicated, and we have only used the standard time() function in Python to make our measurements. Therefore variations will occur in the measurements, especially at the millisecond level. Different samples will also give different results. Therefore we have calculated the testing time for 100 samples, and extrapolated the testing time for 1 sample simply by dividing by 100.

From this experiment we also learn that C4.5 and SVM classifiers have the best accuracy performance, with marginal and insignificant difference.

Experiment 1 shows us that we should keep the Golub-score feature vector containing seven features. It performed just as well as the 18 features of the original vector, and it outperformed the 5 features of the GeFS vector. The C4.5 and SVM classifiers are both excellent candidates to use as classifier for the rest of the experiments. Based on the availability of good Python libraries we choose to move on using SVM.

The feature selection process gives us good objective recommendations as to which features are most significant for correct classification. However, it is the authors belief, based on expert knowledge, that some more features should be kept in order to cover some well-known exploitation methods. We therefore propose the following vector as an mixed expert knowledge enhanced version of the Golub-score and GeFS feature vector:

```
AA, RichMedia, Launch, JavaScript, OpenAction, JS, obj_mis, startxref, \
AcroForm, Page, trailer
```

The enhanced vector gives the following results when training and testing with SVM:

Confusion matrix:

		Given labels	
		Benign	Malicious
Predicted labels	Benign	7436	103
	Malicious	18	16177

Table 6: Confusion matrix Enhanced feature vector

Balanced successrate: 0.9956

Area under ROC curve: 0.9918

Training time, full dataset, 11 features: 3.93 seconds

Testing time, 100 samples: 16,7 milliseconds

As we can see the enhanced feature vector gives slightly better successrate than any of the other vectors and we therefore decide to keep this vector for use in the further experiments.

6.3 Experiment 2: Classifier Optimalization and Testing

Experiment 2 consists of a series of small experiments designed to find the optimal configuration for our classifier and to test some of its properties. Experiment 2.1 seeks optimal performance through optimizing configuration values, and normalization. Experiment 2.2 investigates the classifiers stability. Experiment 2.3 tests the classifiers generalization abilities.

In [62] Hsu et al. gives a very practical guide for using and configuring an SVM classifier. A simple procedure is proposed on how to get the best SVM performance. The procedure includes data preprocessing, kernel selection and model selection using parameter search.

6.3.1 Experiment 2.1: Optimal Configuration and Normalization

This experiment was conducted as described in section 5.3.4. The goal of this experiment was to find the optimal configuration for the SVM classifier.

First we searched for the optimal values of the penalty value C and the inverse-width parameter γ for the Gaussian kernel we use with our SVM.

The 11 feature dataset was given as input to the pdfidsvm_grid.py-script[1]. This script loads the dataset and then performs a grid-search using cross-validation to optimize the SVM values of C and gamma. The search space for C=0.1, 1, 10, 100, 1000 and gamma=0.01, 0.1, 1, 10. When the optimal values are found 10-fold cross-validation is performed on the entire dataset using the optimal values.

The results of this experiment was:

Optimal C: 100

Optimal γ: 0.1

Sensitivity: 0.9949

Successrate: 0.9956

Balanced successrate: 0.9960

Area under ROC curve: 0.9967

Confusion matrix:

		Given labels	
		Benign	Malicious
Predicted labels	Benign	7433	83
	Malicious	21	16197

Table 7: Confusion matrix Experiment 2.1 - Configuration

[1] See Appendix A.5

Next we tested whether normalizing the feature vector dataset would improve classifier performance. The dataset was normalized using cosine-normalization in kernel space. The 11 feature dataset was input to the pdfidsvm_norm.py-script[2]. It attaches the Gaussian kernel, using optimal parameters, to the dataset and performs *cosine normalization* in kernel space. After normalization 10-fold cross-validation is performed to see if performance has improved:

Sensitivity: 0.9950

Successrate: 0.9955

Balanced successrate: 0.9959

Area under ROC curve: 0.9980

Confusion matrix:

		Given labels	
		Benign	Malicious
Predicted labels	Benign	7429	80
	Malicious	25	16200

Table 8: Confusion matrix Experiment 2.1 - Normalization

From the first experiment we learned that the optimal kernel parameters are C=100 and γ=0.1. Looking at the performance for the classifier using these optimal parameters, compared to the results from experiment 1 we notice that there is a slight gain from this optimalization. The false negatives are reduced by 20%, and the false positives increase by 15%. The reason that the gain was so small may be due to the fact that the optimal values are close to the default values (C=100 γ=1) used in experiment 1.

From the normalization experiment we learned that normalization had no significant effect on the classifier performance. The small difference in performance results is more likely to be a consequence of cross-validation sampling, rather than the normalization.

Even though finding the optimal kernel parameters did not improve classifier performance we keep the parameters as there is no adverse effect of doing this. All future experiments of this thesis will use kernel parameters C=100 and γ=0.1.

As normalization did not improve classifier performance future experiments of this thesis will not be using normalization. This is due to the fact that normalization is a processing step that increases the computational complexity of the classification process, and thus will make it less efficient.

[2]See Appendix A.6

6.3.2 Experiment 2.2: Classifier Stability

This experiment was conducted as described in section 5.3.4. The goal of this experiment was to test the stability of the classifier over multiple iterations of training and testing.

We did five iterations of 2-fold cross-validation over the 11 feature dataset. The results are shown here:

	Bal. succ.	AUC
1	0.9948	0.9983
2	0.9948	0.9985
3	0.9952	0.9977
4	0.9945	0.9984
5	0.9952	0.9985

Table 9: Results Experiment 2.2

As we can see from this experiment the classifier performance is stable over multiple iterations. Also the number of number of false positives and false negatives stay very stable. We evaluate the classifier as stable and reliable, and keep it for future experiments.

6.3.3 Experiment 2.3: Classifier Generalization

This experiment was conducted as described in section 5.3.4. The goal of this experiment was to test whether the classifier is able to correctly classify PDF samples guaranteed to not be a part of the training dataset.

In order to do this experiment we first needed to collect as many PDFs as possible that uses and exploit with a newer date than our dataset creation time (January 2011). By doing a search on the Offensive Security Exploit Database [63] we were able to find the malicious PDFs listed in table 10.

#	MD5	Date	Vulnerability
1	86d2018e9ecca17cb848d62dec910f46	16 Dec 2010	Adobe Embedded Executable
2	78e13cb81b84f72a8b5a6675cad25a75	16 Dec 2010	Adobe Embedded Executable (No JS)
3	b039e93c03db20980c5b5f742a67500d	12 May 2011	SlimPDF DoS
4	7fd06913ff0144a188f29833b79b75f5	14 Mar 2011	Foxit Reader filewrite
5	1ea237047d96770aef5bed54132e0c8c	16 Dec 2010	Foxit Reader title BO
6	8849500941046136c3697cb363e7c569	28 Feb 2011	NitroPDF Heap Corruption
7	6857fa199d3e2224f80424282c5a48b6	08 Jan 2011	Nuance PDF reader Launch BO
8	84387f6929fbbb57e5cb4ffaebefbc70	07 Mar 2011	Adobe Reader X CVE-2011-0611
9	72011f96c98e3f9c78288d2e81c1db2b	11 Dec 2010	Active PDF Buffer Overflow

Table 10: Novel malicious PDF

We also include 10 new benign samples, to also test if these are classified correctly.

The experiment resulted in 15 out of the 19 PDF being correctly classified. However, four of the malicious samples were incorrectly classified as benign:

1ea237047d96770aef5bed54132e0c8c Foxit Reader title BO

8849500941046136c3697cb363e7c569 NitroPDF Heap Corruption

72011f96c98e3f9c78288d2e81c1db2b Active PDF Buffer Overflow

b039e93c03db20980c5b5f742a67500d SlimPDF DoS

After a lot of testing it seems that the ”/AcroForm” feature in our feature vector is the culprit of confusing our classification. Removing this from the feature vector gives us 18 out of 19 correctly classified samples. The only sample incorrectly classified is now:
8849500941046136c3697cb363e7c569 NitroPDF Heap Corruption

Analysis of this PDF shows that all it does is to open a mediabox with a specially crafted content to crash the Nitro PDF Viewer, without any other specific malicious actions. The PDF therefore looks and behaves very much like a benign PDF and will not be detected by our algorithm.

Removing the "/AcroForm" feature from our feature vector has adverse effects on the total performance of our classifier. When testing using 10-fold cross-validation of the entire dataset we now get the following results:

Sensitivity: 0.9921

Successrate: 0.9842

Balanced successrate: 0.9794

Area under ROC curve: 0.9955

Confusion matrix:

		Given labels	
		Benign	Malicious
Predicted labels	Benign	7207	129
	Malicious	247	16151

Table 11: Confusion matrix Experiment 2.3

As we see from the confusion matrix removing the "/AcroForm" vector especially increases our false positive rate. This is not good for the operators that shall use the classification model in an IDS.

To make a comparison with a top signature based anti virus product, the novel corpus of 19 PDF documents were scanned by MS Security Essentials. As it turns out MSSE is only able to detect 5 out of the total of 9 malicious PDFs, even though the novel samples are getting close to one year old at the time of writing this.
1ea237047d96770aef5bed54132e0c8c Foxit Reader title BO

86d2018e9ecca17cb848d62dec910f46 Adobe Embedded Executable

72011f96c98e3f9c78288d2e81c1db2b Active PDF Buffer Overflow

84387f6929fbbb57e5cb4ffaebefbc70 Adobe Reader X CVE-2011-0611

6857fa199d3e2224f80424282c5a48b6 Nuance PDF reader Launch BO

As we can see, MSSE was not able to detect the **8849500941046136c3697cb363e7c569 NitroPDF Heap Corruption** sample that we were not able to detect, as well as four other samples. Out of the four samples our classifier got incorrect in the first run, two coincide with the four samples MSSE did not detect.

This further reinforces our belief in the generalization ability of our classifier. Having only

9 novel malicious samples for testing is very little to draw any conclusions. And the benefit of having "/AcroForm" as a feature by far outweighs the disadvantages from what we can see from available data. Until we can get more new samples for testing we will need to have "AcroForm" in our feature vector. This experiment showed that the method will be able to detect also samples that are not part of the training dataset, even though it will not be able to catch all.

6.4 Experiment 3: Real-world Test

This experiment was conducted as described in section 5.3.5. The goal of this experiment was to test how the classifier, or IDS prototype, would perform in a real-world setting.

An IDS prototype was built as described in section 5.1 ”Proposed Solution”, and implemented at a network sensor owned by the Norwegian Defence Center for Protection of Critical Infrastructure. This sensor monitors all internet traffic going to and from a Norwegian Defence site with approximately 300 employees. The number of employees using the internet connection on a daily basis is probably much lower. The experiment was started on Sunday 30th of October and ended on Tuesday 9th of November, that is approximately 10 days of run-time. During this time the prototype reported a total of 625 PDF downloads. It was able to extract 556 PDFs from the logged data. The discrepancy between the reported number of downloads and the number of extracted PDFs might be that the “%PDF” tag was also found in non-PDF traffic. Out of the 556 PDFs extracted, 150 were corrupted. There may be several reasons for this:

- PDF downloads where stopped before completion.
- The SNORT packet logging dropped some of the packages.
- Transfers were split over several TCP sessions for some reason (and SNORT will only follow the initial session).

All of the 556 PDFs where classified, and the results from this showed that:

- 546 samples where classified as benign.
- 10 samples where classified as malicious.

The samples classified as malicious were manually analyzed by the author and found to actually be benign. The reasons for the mis-classifications being:

- Corrupted data causing object mismatches. Combined with...
- ...benign use of OpenAction to present dialogue boxes upon viewing. And...
- ...the benign use of AcroForm.

There is no way to manually analyze all the 546 samples classified as benign to determine is they are really benign. However, as a small measure the samples where run through the MS Security Essentials anti-virus solution to look for any obvious maliciousness. Zero malicious samples were reported.

Also there was no malicious incidents with PDF documents reported from for the particular site during the period.

Even though this test revealed no malicious PDF documents being transferred across this particular network during the test period, we can say that the test was a success.

The test revealed that the proposed IDS solution will work as assumed. The following shows a breakdown of the timing for all involved processes from an alert until the PDF is classified:

- Converting SNORT unified2 log format to pcap: 0,5 - 3 seconds depending on logfile size.
- Extracting correct flow from pcap: 5 - 700 milliseconds depending on pcap size and flow location in file.
- Removing headers from flow to get PDF: 0,5 - 100 milliseconds
- Extracting features from PDF: 0,2 - 4 seconds depending on PDF file size.
- Classifying a PDF file: 0,3 - 0,5 milliseconds
- **Worst case total:** 8 seconds

These measurements show that the proposed IDS can provide close to real-time classification of PDF, but inline checking of the PDFs with capability to permit or deny PDFs will introduce a delay of approximately 8 seconds. It is the authors opinion that a delay of 8 seconds will be OK for non-interactive applications, such as e-mail. However, introducing a 8 second delay for all PDF downloads may be considered too much for some users. It should however be noted that a huge increase in performance probably can be achieved by implementing the method in a compiled language, e.g. C/C++.

What remains to make the prototype viable for operational use is a portion of engineering work to eliminate the over-reporting of malicious downloads and corrupted data. As mentioned this may have something to do with packet drop, but unfortunately the computational performance statistics from SNORT where not available due to technical limitations.

The number of false positives are very good. To handle 10 false positives out of 556 extracted PDFs over a period of 10 days is no problem for an IDS operator and is well within what is acceptable for an IDS.

6.5 Experiment 4: A Closer Look at Embedded Javascript

This experiment was conducted as described in section 5.3.6. The goal of this experiment was to explore the possibilities of classifying javascript embedded in PDF documents, and serve as a starting point for future research.

Through a quick analysis of the embedded javascript found in our malicious PDF corpus we decided to extract the following features to form the javascript feature vector:

function - Counts the number of new functions that are defined.
Malicious scripts often employ their own obfuscation functions.

eval_length - The length of the longest string passed to eval().
Malicious scripts tend to use eval() to execute dynamic code.

max_string - The length of the longest string defined.
Strings for shellcode tend to be very long, compared to strings that are normally used in benign javascript.

stringcount - The number of strings that are defined.
Malicious code writers like to split strings into very many small strings to obfuscate the script.

replace - Counts the uses of the javascript replace() function.
This function is much used in javascript obfuscation.

substring - Counts the uses of the javascript substring() function.
This function is much used in javascript obfuscation.

eval - Counts the uses of the javascript eval() function.
Malicious scripts tend to use eval() to execute dynamic code.

fromCharCode - Counts the uses of the javascript fromCharCode() function.
fromCharCode() is used to decode Unicode encoded strings. Often used for obfuscation.

We then created a javascript corpus with malicious javascript extracted from the malicious PDF corpus, and benign javascript extracted from the benign PDF corpus and supplemented by a webcrawl. This resulted in 1020 unique benign samples and 4795 unique malicious samples.

The features were extracted and the feature vectors were fed into the SVM classifier. Training and testing was performed using 10-fold cross-validation as it was in the earlier experiments.

The results, using all 8 features, was:

Confusion matrix:

		Given labels	
		Benign	Malicious
Predicted labels	Benign	849	375
	Malicious	171	4420

Table 12: Confusion matrix Experiment 4

Balanced successrate: 0.908

Area under ROC curve: 0.954

The results show that our feature vector will be a good starting point for future research into malicious javascript. Adding classification of embedded javascript will further improve the performance of the PDF classifier.

7 Discussion

In this chapter follows a discussion of theoretical and practical considerations from the master thesis.

Choice of Method

The method chosen in this master thesis was largely based on creating all the building blocks for a working IDS capable of detecting malicious PDF documents in network traffic. The positive side to this approach is that it gives a very practical insight into what is really needed to build such a system. It also lets us present empirical data for all parts of the project.

The disadvantage of this approach is that it requires a lot of engineering, i.e. system design and programming. It can be argued that this takes away focus from the pure research. However we feel that making a practical solution makes the research more applicable and more interresting. To put it in the words of John Dewel: *Theory without practice is empty; practice without theory is blind.*

An alternative could be to take a more theoretical approach and device some assumptions on how PDF should be detected by studying the literature and interviewing experts in the field. However, it is our belief that this approach would produce more theoretical and uncertain answers.

In the thesis we chose to only test five classifiers and two feature selection methods. The ideal situation, according to the No Free Lunch and Ugly Duckling Theorems, would of course be that we tested even more. However, as the time frame for a master thesis is rather short we had to limit our experimentation in this aspect.

The Dataset

The main caveat of this thesis, as often with IDS research, is the dataset. Getting up-to-date and relevant datasets for IDS testing is a challenge that many have struggled with. Building datasets and ensuring that all factors are under control is a task suited for a master thesis or PhD research itself.

For this master thesis we created a brand new dataset, consisting of over 20.000 PDF samples. It is simply not feasible to subject this entire dataset to manual analysis, to consider questions like:

- Are all samples in the benign dataset really benign?
- Are all samples in the malicious dataset really malicious?
- Is there a good variety of samples and exploits in the dataset?
- What exploits are present in the dataset?

Some measures have been taken to provide some answers to these questions. To check if all samples in the benign dataset really is benign we set up the antivirus test in section 5.3.1. This

test at least makes sure that there is no obvious maliciousness in the benign corpus that triggers the signature detection of the antivirus products.

To test whether all samples in the malicious dataset really are malicious a similar test was conducted, where the malicious PDF corpus was run through MS Security Essentials. Not surprisingly most samples were detected as malicious, however 950 (5.8%) of the dataset was not detected. Some of the PDFs not detected where selected for manual analysis, and where indeed malicious and used some very clever obfuscation techniques. We therefore assumed that all 16280 malicious samples indeed were malicious, as there is no proof otherwise.

From this we also learned that there are at least 373 unique exploits kinds of exploits in the dataset, which tells us that there is a good variety of exploits in the dataset. To further ensure the variety of samples and exploits in the dataset is very difficult. By collecting malicious samples from sources who have received samples from all over the world for a large period of time, we also trust to a certain degree that the corpus has a good variety of samples. Another measure we have taken here is to make sure that all samples are unique, using MD5 hashes. However, this measure can of course be easily fooled by obfuscation, as even a change to only one bit will make the MD5 hash totally different. A very interesting tool to remedy this situation would be fuzzy hashing, that is hash algorithms that produce similar hashes for similar samples. Using fuzzy hashing for such an application is still at the research stage [64] and must be left for future work.

From the anti virus test we know to a certain degree what exploits are present in the malicious PDF corpus. However, we know nothing about the 950 samples that was not detected. So therefore this question will remain only partly answered.

Ideally the dataset should also be more balanced. For this thesis we did not want to cut out any of the malicious samples to make this happen, as it would have adverse effects on the performance. For future work more benign samples should be collected in order to obtain a balanced dataset.

Active Content

As we can see from the feature vector we use in our detection scheme, a lot of the features are related to “active content” in PDF documents. This includes javascript, flash media, forms and actions. So, a reasonable question to ask is “Does the method really detect maliciousness or just active content?”. This is a valid question. As stated both in [6] and our own findings in section 5.3.6 over 90% of all PDF documents containing javascript are malicious. And while only 393 (5.2%) of our benign PDF documents contained javascript, 14556 (90%) of our malicious PDF documents did.

However, basing the detection solely on active content will not yield as good results as we have seen in this thesis with only 25 (0.3%) of the benign corpus wrongfully classified as malicious. This goes to show that there is more to it than just looking for active content, like javascript.

To further address the active content we created a starting point for classifying the javascript themselves in experiment 4, where the basis for a feature vector for classifying javascript as malicious or benign is formed.

Robustness

When discussing robustness in this context we mean the ability of the classifier/detection system to withstand evasion attempts by an active adversary. Such an adversary may craft malicious PDF documents specifically to evade/fool our classifier. Defending against such an adversary is hard. But hopefully the selected features used in our detection algorithm is significant enough to reduce the adversaries ability to manipulate several and still maintain a working malicious PDF document. We have however chosen not to design any specific experiments to test the classifiers robustness, due to the limited time available for the completion of the master thesis. This will therefore be a part of future work.

Need for Maintenance

The classifier and the detection method as a whole will need to be followed up over time. The attacks, the malicious code and techniques used will inevitably change over time. So the proposed method is not a one-shot - work forever method.

This maintenance can be done either by selecting one point in the future where a new dataset is found, and the experiments are re-done to find the optimal classifier setup and features once more. Another interesting approach is "online learning". With online learning the classifier is constantly trained with all the new samples it observes in it's operational use. Such learning schemes are currently undergoing research, and will therefore need to be explored in future work.

Computational Complexity

In experiment 1 and 3 we presented several timing results to measure the computational performance of the proposed method. These results show that computational performance is well within what is acceptable for intrusion detection. Considering the real-world test approximately 600 PDF documents where transferred in the network over 10 days. That is an average of 60 samples per day. The worst case processing time for each samples is calculated to be 8 seconds. Any up-to-scale sensor monitoring a network should be able to handle such a load.

Using the proposed method in an intrusion prevention scheme, however, could prove more difficult at the moment. Such a scheme would imply putting the sensor in an inline-mode, that is the sensor will receive all traffic, process it, and only let benign traffic pass after inspection. Our proposed solution will thus induce an 8 second delay for all PDF downloads, which will be unacceptable for the users. However, for some applications, like e-mail, a 8 second delay will probably be accepted.

Keeping in mind that the prototype IDS solution is programmed in Python, there is also a huge potential for performance increase. By employing compiled programming languages like C, performance should be increased significantly.

Ethical Considerations

As always with any product monitoring the network activity of human users, there are some ethical issues. Deploying an IDS really means that you are deploying surveillance equipment, and this means that there are some rules that needs to be followed. There is no clear legislation on network monitoring in Norway. But any organization implementing such measures need to consider the Privacy Act (Personopplysningsloven), and to find their pursuant for implementing.

Often such pursuant can be found in legislation or rules governing the specific work area of the organization, or it can be found in the employers right of access.

In any case local policies for network monitoring needs to be established. The users of the information system under surveillance need to be informed that the system is indeed monitored, and give their informed consent to this.

Such ethical considerations was considered for the real-world experiment of this thesis. As the experiment was performed on an already established sensor most of the work was however already done. The pursuant was already in place, local policies for network monitoring exists and all the users of the information system need to sign a document where it is explained that the system is monitored before gaining access.

In addition we implemented some restrictions on our own. First we decided to only monitor web traffic (HTTP) and not e-mail (SMTP). This is because we assess that there is a bigger chance of personal information being transferred over SMTP. We also made sure that extracted PDF documents were only analyzed by software up until it was actually classified as malicious. This hindered that benign documents, with a higher risk of private or sensitive data, was read by any human analyst.

All this considered, we believe that the ethical issues of network monitoring has been well taken care of in the thesis.

8 Conclusion

The main research question of this master thesis was “How can malicious PDF-documents transferred in a network be detected?”. The first thing we knew that we needed was a comprehensive corpus of benign and malicious PDF documents. We obtained a huge corpus, containing over 16000 unique malicious samples and over 7000 unique benign samples.

The main research question was broken down into three subquestions. The first subquestion was “Which features are significant for detecting malicious PDF documents?”. To answer this question we performed a literature study seeking expert knowledge on the topic. From the expert knowledge we derived what we called the “expert knowledge feature vector”, which was extracted from our PDF document corpus and served as our starting dataset for further research.

We then employed statistical methods to objectively select only the most relevant features based on our dataset. Two feature selection methods were employed, and gave us two different resulting feature vectors. An “enhanced feature vector” was then created based on the output of the feature selection and expert knowledge. This new feature vector outperformed all the other feature vectors, and provided the answer to our first research subquestion.

The second subquestion was “Which classifier design and configuration yields optimal performance in malicious PDF detection?”. To answer this question we first tested the performance of five different classifier designs. The results showed that SVM and C4.5 gave us the best results. SVM was therefore selected as the classifier design to be used. This answered the first part of the question. Next we needed to optimize the performance of the SVM classifier. This was achieved through finding the optimal configuration values for the SVM and the Gaussian kernel of the SVM. The optimal values were found and offered a slight increase in performance. Normalization was also tested, but offered no performance increase. Thus we answered the second part of this subquestion.

The third subquestion was “How can a real-world IDS be implemented based upon our findings?”. To answer this question we created a prototype IDS and deployed it to one of the sensors owned by the Norwegian Defence Center for Protection of Critical Infrastructure. During the 10 day test period no malicious PDF documents were detected, but the probability is high that there really was no malicious PDFs being transferred. The test was anyways successful as it proved that the prototype functioned as planned. For detection of malicious PDFs we have to rely on our previous results. The test showed that our proposed implementation of our findings worked.

As a step further on the practical implementation of an IDS we conducted an experiment where the goal was to classify embedded javascript, as javascript are found in the majority of malicious PDF documents. This experiment showed that javascript classification using similar methods to what we have used in the PDF experiments is plausible, and can serve as a good starting point for future research into embedded javascript classification.

To summarize, the acquired knowledge from this master thesis is:

- A comprehensive PDF dataset for future research (16280 malicious / 7454 benign).
- Increased knowledge on significant features for PDF classification, based on empirical findings.
- A proven method for automated detection of malicious PDF in network traffic.
- A starting point for future research on malicious embedded javascript detection.

9 Future Work

Hopefully this master thesis provides a step forward in malicious PDF detection, however we also realize that there is more that should be done to further improve and validate the method and to make it ready for operational use. Many of these items have already been presented in the discussion, and will only be presented briefly in this chapter.

Further Validation of the Dataset

To further validate the obtained results the dataset should be further validated. One idea here is to employ fuzzy hashing to review the similarity between samples in the PDF document corpus. Furthermore additional benign PDF documents should be collected to obtain a balanced dataset.

Further Research on Embedded Javascript

During the work on the thesis, it was discovered that a very large part (90%) of all malicious PDF documents contained embedded javascript that performs parts or all of the malicious actions. This encouraged Experiment 4, where classification of such embedded javascript was investigated. The experiment showed results that make classification of embedded javascript plausible. Further research should be performed to improve classification of embedded javascript, as they are an important factor in detecting malicious PDF.

Testing Robustness

It is possible that evading detection could be easily done by an active adversary by manipulating malicious documents is such a way that they appear benign to the classifier. An experiment should be designed and executed to test the robustness against such evasion.

Online Learning

As already discussed, there is a need for re-learning of the classifier after a period of time to adapt to the evolving threat landscape. Such re-learning could be performed manually, but it is our belief that manual re-learning is a too troublesome process for commercial use. Online learning provides the solution to this, but is not fully researched yet. With online learning the system would adapt to new threats automatically, without requiring time-consuming manual processes.

Operator Interface

To be of any real practical use the proposed IDS needs to have a user interface. Preferably a graphical user interface (GUI). The GUI should present alerts to the operator in a simple fashion, and should let the operator process the alerts with ease.

To improve the GUI and the operator performance a GUI should also include some measure of certainty or severity, a database of previously analyzed PDFs, MD5 checking with online databases. The certainty measure can possibly be implemented by using some certainty measures from the SVM, like the distance from the margin. However, this will need to be further researched. Severity is hard to say anything about, since the proposed solution does not report

any specifics about the exploit used like a signature based solution would. One way of dealing with this could be to make a hybrid of the proposed IDS and a signature based IDS, so that severity could at least be reported for well known attacks.

More features

During the work with this thesis, some new candidates to serve as features in the feature vector was discovered. These include /win, which indicates the use of a Windows command, and /EmbeddedFile, which indicates an embedded file within the PDF. Sadly, there was no time to include these in the feature vector an run all experiments once more at the time of this discovery. Therefore these features, and probably even more, should be tested in the future.

Bibliography

[1] Selvaraj, K. & Gutierrez, N. F. The rise of pdf malware. Technical report, Symantec, 2010.

[2] 2011. Study: 6 out of 10 users run vulnerable adobe reader. http://www.zdnet.com/blog/security/study-6-out-of-every-10-users-run-vulnerable-adobe-reader/9014. Last visited: 28. Sept 2011.

[3] Pdf physical structure. http://www.planetpdf.com/. Last accessed: 14. nov 2011.

[4] Ben-Hur, A. & Weston, J. 2008. A user's guide to support vector machines. *Data Mining Techniques for the Life Sciences*, 609.

[5] Inc., A. S. *ISO Standard 32000-1:Document management - Protable Doument Format - Part 1: PDF 1.7*. International Organization for Standardization, 2008.

[6] Baccas, P. 2010. Finding rules for heuristic detection of malicious pdfs with analysis of embedded exploit code. In *Virus Bulletin Conference*.

[7] Reseachdept. First annual cost of cyber crime study benchmark study of u.s. companies. Technical report, Ponemon Institute, 2010.

[8] Websense. Security predictions for 2012 from websense security labs. Technical report, Websence Security Labs, 2011.

[9] McAfee. Mcafee threats report: First quarter 2011. Technical report, McAfee Labs, 2011.

[10] PandaLabs. The cyber-crime black market: Uncovered. Technical report, Panda Security - Panda Labs, 2011.

[11] SANS. Sans top cyber security risks - vulnerability exploitation trends. http://www.sans.org/top-cyber-security-risks/trends.php. Last accessed: 19. dec 2010.

[12] VeriSign. Malware security report: Protecting your business, customers, and the bottom line. Technical report, VeriSign Inc., 2010.

[13] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., & Modadugu, N. 2007. The ghost in the browser: Analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*.

[14] Pdf exploits explode, continue climb in 2010. http://www.computerworld.com/s/article/9176117/PDF-exploits-explode-continue-climb-in-2010. Last visited: 28. Sept 2011.

[15] Stevens, D. Pdfid tool. http://blog.didierstevens.com/programs/pdf-tools/. Last visited: 27. Sept 2011.

[16] malpdfobj tool. http://blog.9bplus.com/tag/pdfxray. Last visited: 27. Sept 2011.

[17] Vg nett - forsvaret angrepet etter libya-beslutning. http://www.vg.no/nyheter/utenriks/libya/artikkel.php?a Last visited: 27. Sept 2011.

[18] Aftenposten. Nov 2010. Hackere prøvde å ta seg inn på nobel-direktørens pc. http://www.aftenposten.no/nyheter/iriks/article3898053.ece.

[19] Rieck, K. *Machine Learning for Application-Layer Intrusion Detection*. PhD thesis, Technischen Universität Berlin, 2009.

[20] Skoudis, E. & Liston, T. 2005. *Counter hack reloaded, second edition: a step-by-step guide to computer attacks and effective defenses*. Prentice Hall Press, Upper Saddle River, NJ, USA.

[21] Harris, S., Harper, A., Eagle, C., & Ness, J. 2007. *Gray hat hacking: The ethical hackers handbook*. McGraw-Hill, 2 edition.

[22] Petrovic, S. 2009. Ids/ips definition and classification. Lecture from IMT4741 Intrusion Detection and Prevention at Gjøvik University College.

[23] Roesch, M. 1999. Snort - lightweight intrusion detection for networks.

[24] Lunt, T. 1993. Detecting intruders in computer systems. In *In Proceedings of the 1993 Conference on Auditing and Computer Technology*.

[25] Sommer, R. & Paxson, V. 2010. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*.

[26] McHugh, J. 2000. Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security*, 3(4).

[27] Brown, C., Cowperthwaite, A., Hijazi, A., & Somayaji, A. 2009. Analysis of the 1999 darpa/lincoln laboratory ids evaluation data with netadhict. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Security and Defence Applications*.

[28] Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A. A. 2009. A detailed analysis of the kdd cup 99 data set. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Security and Defence Applications*.

[29] ACM. *Combining Static and Dynamic Analysis for the Detection of Malicious Documents*, 2011.

[30] Blonce, A., Filiol, E., & Frayssignes, L. Portable document format (pdf) security analysis and malware threats. Technical report, French Army Signals Academy, 2008.

[31] Sand, L. A. The portable document format - a study of common distribution, exploit, evasion, mitigation and detection techniques. Student paper in Information Security at Gjøvik University College, 2011.

[32] Feinstein, B. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. Technical report, SecureWorks Inc., 2007.

[33] Mitchell, T. 1997. *Machine Learning*. McGraw Hill.

[34] Franke, K. Lecture: Machine learning introduction. Lecture from IMT4631 Machine LEarning and Data Mining at Gjøvik University College, 2009.

[35] Kononenko, I. & Kukar, M. 2007. *Machine Learning and Data Mining*. Horwood.

[36] Turney, P. Technical note: Bias and the quantification of stability. Technical report, NRC-CNRC, 1995.

[37] Wolpert, D. H. & Mavready, W. G. 1997. No free lunch theorems for optimization. In *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, volume 1.

[38] Nilsson, N. J. 1996. Introduction to machine learning: An early draft of a proposed textbook. http://robotics.stanford.edu/people/nilsson/mlbook.html.

[39] Cortes, C., Jackel, L. D., Solla, S. A., Vapnik, V., & Denker, J. S. 1993. Learning curves: Asymptotic values and rate of convergence. In *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA]*, 327–334.

[40] Nguyen, H. T., Franke, K., & Petrovic, S. 2010. Towards a generic feature-selection measure for intrusion detection. In *20th International Conference on Pattern Recognition (ICPR), 2010*, 1529–1532. IEEE.

[41] Furey, T. S., Cristianini, N., Duffy, N., Bednerski, D. W., Schummer, M., & Haussler, D. 2000. Support vector machine classification and validation of cancer tissue samples using microarray expression data.

[42] Duda, R. O., Hart, P. E., & Stork, D. G. 2000. *Pattern Classification*. Wiley Interscience.

[43] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. 2009. The weka data mining software: An update. In *SIGKDD Explorations*, volume 11.

[44] Bayesian network - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Bayesian_network. Last accessed: 1. nov 2011.

[45] Quinlan, R. J. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.

[46] Quinlan, R. J. 1992. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*.

[47] Multilayer perceptron - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Multilayer_perceptron. Last accessed: 1. nov 2011.

[48] Radial basis function network - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Radial_basis_function_network. Last accessed: 1. nov 2011.

[49] Boser, B. E., Guyon, I. M., & Vapnik, V. N. 1992. A training algorithm for optimal margin classifiers. In *5th Annual ACM Workshop on COLT*, Haussler, D., ed, 144–152. ACM Press.

[50] tcpflow – tcp flow recorder. http://www.circlemud.org/jelson/software/tcpflow/. Last visited: 5. oct 2011.

[51] jsunpack-n tool. http://code.google.com/p/jsunpack-n/. Last visited: 27. Sept 2011.

[52] Kirk, A. Detecting obfuscated malicious javascript with snort and razorback. http://labs.snort.org/papers/DetectingMaliciousJSwithSnortRazorback.pdf Last accessed Dec 18th 2010, 2010.

[53] Stevens, D. Malicious pdf analysis ebook. Unpublished bookchapter. Found at http://didierstevens.com/files/data/malicious-pdf-analysis-ebook.zip, Sept 2010.

[54] Wang, K., Parekh, J. J., & Stolfo, S. J. 2006. Anagram: A content anomaly detector resistant to mimicry attack*. *Lecture Notes in Computer Science*.

[55] peepdf v0.1 released: a tool to analyse/modify malicious pdf files. http://eternal-todo.com/blog/peepdf-pdf-analysis-tool. Last accessed: 14. nov 2011.

[56] Python programming language - official website. http://www.python.org/. Last visited: 16. aug 2011.

[57] Scientific computing tools for python - numpy. http://numpy.scipy.org/. Last visited: 16. aug 2011.

[58] Pyml - machine learning in python. http://pyml.sourceforge.net/. Last visited: 16. aug 2011.

[59] Microsoft security essentials - free antivirus for windows. http://windows.microsoft.com/en-US/windows/products/security-essentials. Last accessed: 14. nov 2011.

[60] Virusbeskyttelse & internett-innholdssikkerhet - trend micro norge. Virus definitions updated on day of use. Last accessed: 14. nov 2011.

[61] Avg free. http://free.avg.com/ww-en/homepage. Last accessed: 14. nov 2011.

[62] Hsu, C.-W., Chang, C.-C., & Lin, C.-J. 2003. A practical guide to support vector classification. *Bioinformatics*, 1.

[63] Offensive security exploit database. http://www.exploit-db.com/. Last visited: 12. oct 2011.

[64] Baier, H. & Breitlinger, F. 2011. Security aspects of fuzzy hashing. http://hdl.handle.net/2003/28934. Last accessed: 14. nov 2011.

A Custom Code

Listing A.1: uniqify.py

```
#!/usr/bin/env python


import os,sys
import hashlib

#md5 function for files in Python by Thejaswi Raya
def md5(fileName, excludeLine="", includeLine=""):
    """Compute md5 hash of the specified file"""
    m = hashlib.md5()
    try:
        fd = open(fileName,"rb")
    except IOError:
        print "Unable to open the file in readmode:", fileName
        return
    content = fd.readlines()
    fd.close()
    for eachLine in content:
        if excludeLine and eachLine.startswith(excludeLine):
            continue
        m.update(eachLine)
    m.update(includeLine)
    return m.hexdigest()




#Getting file list:
path= sys.argv[1]
list = os.listdir(path)
md5_list = []

for f in list:
    md5sum = md5(path+f)
    #print 'md5 for ' +f+ ' is ' +md5sum
    if md5sum in md5_list:
        os.remove(path+f)
        print 'Deleting ' +f+ ' as it is not unique...'
    else:
        md5_list.append(md5sum)
```

Listing A.2: renamer.py

```
#!/usr/bin/env python


import os,sys
import hashlib

#md5 function for files in Python by Thejaswi Raya
def md5(fileName, excludeLine="", includeLine=""):
    """Compute md5 hash of the specified file"""
    m = hashlib.md5()
    try:
        fd = open(fileName,"rb")
    except IOError:
        print "Unable to open the file in readmode:", fileName
        return
    content = fd.readlines()
    fd.close()
    for eachLine in content:
        if excludeLine and eachLine.startswith(excludeLine):
            continue
        m.update(eachLine)
    m.update(includeLine)
    return m.hexdigest()




#Getting file list:
path= sys.argv[1]
extension = sys.argv[2]
list = os.listdir(path)

#Calculate md5 for all files
for f in list:
    md5sum = md5(path+f)
    filename = path+f
    new_filename = path+md5sum+extension
    os.rename(filename, new_filename)
    print 'Renamed ' + filename + ' to ' + new_filename
```

Listing A.3: pdfextract.py

```
#!/usr/bin/python
import pdf #Imports a part of the jsunpack-n tool by Blake Hartstein
import re, os, sys, time


def count_xref(input):          #Counts occurences of cross-reference tables
        rg = re.compile(r'\bxref\b')
        matches = rg.findall(input)
        return len(matches)

def count_startxref(input):     #Counts occurences of "startxref" keywords
        rg = re.compile(r'\bstartxref\b')
        matches = rg.findall(input)
        return len(matches)

def count_obj(input):           #Counts obj keywords
        rg = re.compile(r'\bobj\b')
        matches = rg.findall(input)
        return len(matches)

def count_endobj(input):        #Counts endobj keywords
        rg = re.compile(r'\bendobj\b')
        matches = rg.findall(input)
        return len(matches)

def count_stream(input):        #Counts stream keywords
        rg = re.compile(r'\bstream\b')
        matches = rg.findall(input)
        return len(matches)

def count_endstream(input):     #Counts endstream keywords
        rg = re.compile(r'\bendstream\b')
        matches = rg.findall(input)
        return len(matches)

def count_trailer(input):       #Counts trailer keywords
        rg = re.compile(r'\btrailer\b')
        matches = rg.findall(input)
        return len(matches)

def getFeatures(data,mypdf):    #Collects all keyword frequencies and stores in a dictionary
        features = {}

        features['xref'] = count_xref(data)
        features['startxref'] = count_startxref(data)
```

```
46          features['trailer'] = count_trailer(data)
47          features['Page'] = 0
48          features['Encrypt'] = 0
49          features['ObjStm'] = 0
50          features['JS'] = 0
51          features['JavaScript'] = 0
52          features['AA'] = 0
53          features['OpenAction'] = 0
54          features['AcroForm'] = 0
55          features['JBIG2Decode'] = 0
56          features['RichMedia'] = 0
57          features['Launch'] = 0
58          features['AsciiHexDecode'] = 0
59          features['Colors'] = 0
60          if (count_obj(data) - count_endobj(data)) != 0:  #Mismatch
                between obj and endobj
61                  features['obj_mis'] = 1
62          else:
63                  features['obj_mis'] = 0
64          if (count_stream(data) - count_endstream(data)) != 0:    #
                Mismatch between stream and endstream
65                  features['stream_mis'] = 1
66          else:
67                  features['stream_mis'] = 0
68
69          #Parse PDF to decompress and decode streams
70          if mypdf.is_valid():
71                  mypdf.parse()
72
73
74          #Search parsed PDF for features
75          for k in mypdf.objects:
76                  for i in mypdf.objects[k].tags:
77                          if 'Page' in i:
78                                  features['Page'] += 1
79                          if 'Encrypt' in i:
80                                  features['Encrypt'] += 1
81                          if 'ObjStm' in i:
82                                  features['ObjStm'] += 1
83                          if 'JS' in i:
84                                  features['JS'] += 1
85                          if 'JavaScript' in i:
86                                  features['JavaScript'] += 1
87                          if 'AA' in i:
88                                  features['AA'] += 1
89                          if 'OpenAction' in i:
90                                  features['OpenAction'] += 1
91                          if 'AcroForm' in i:
92                                  features['AcroForm'] += 1
```

```
93                          if 'JBIG2Decode' in i:
94                                  features['JBIG2Decode'] += 1
95                          if 'RichMedia' in i:
96                                  features['RichMedia'] += 1
97                          if 'Launch' in i:
98                                  features['Launch'] += 1
99                          if 'AsciiHexDecode' in i:
100                                 features['AsciiHexDecode'] += 1
101                         if 'Colors > 2^24' in i:
102                                 features['Colors'] += 1
103
104         return features
105
106
107 def writeVector(features,filename,label): #Write feature vectors to
        file
108
109         #SparseData format: [id,]label fid1:fval1 fid2:fval2
110         vector = str(filename)+","+label
111
112         for key,value in features.iteritems():
113                 vector += " "+key+":"+str(value)
114
115         vector += "\n"
116
117         outfile = open("dataset_"+label,"a")
118         outfile.write(vector)
119
120
121 if __name__ == '__main__':
122         #Get time for usage calulations
123         start_time = time.time()
124
125         #Get directory and label from commandline
126         path = sys.argv[1]
127         label = sys.argv[2]
128
129         #Get files in directory and open
130         files = os.listdir(path)
131
132         for f in files:
133                 node = open(path+f,'r')
134                 data = node.read()
135                 node.close()
136
137                 #Parse PDF
138                 print "Parsing "+ str(f)
139                 mypdf = pdf.pdf(data,file)
140
```

```
141                 #Extract features
142                 print "Extracting features..."
143                 features = getFeatures(data,mypdf)
144 
145                 """
146                 #Uncomment for printing to stdout
147                 for k,v in features.iteritems():
148                         print k,v
149                 """
150 
151                 #Dumping javascript if available
152                 if features['JS'] != 0 or features['JavaScript'] !=
                        0:
153 
154                         decoded,decoded_headers = mypdf.getJavaScript
                            ()
155 
156                         if len(decoded) > 0:
157                                 try:
158                                         js_out = open('./java/'+label
                                            +'/'+f+'.js','w')
159                                         head_out = open('./java/'+
                                            label+'/'+f+'.headers','w'
                                            )
160                                 except IOError:
161                                         os.makedirs('./java/'+label)
162                                         js_out = open('./java/'+label
                                            +'/'+f+'.js','w')
163                                         head_out = open('./java/'+
                                            label+'/'+f+'.headers','w'
                                            )
164 
165                                 if js_out:
166                                         js_out.write(decoded)
167                                         js_out.close()
168                                 if head_out:
169                                         head_out.write(
                                            decoded_headers)
170                                         head_out.close()
171 
172                         else:
173                             print 'JavaScript seems to be present,
                                but no script dumped'
174 
175                 #Writing SpareDataset vector
176                 print "Writing to vectorfile..."
177                 writeVector(features,f,label)
178 
179         #Calculate used time for performance measures
```

```
180         end_time = time.time()
181         used_time = end_time - start_time
182         print "Work done in " + str(used_time) + " seconds."
```

Listing A.4: ngramextractor.py

```
from __future__ import division
from pyngram import calc_ngram
from PyML import *
import sys
import os
from sqlite3 import *
import StringIO
from time import time

outfile = open("dataset.sparse", 'a')
label = sys.argv[2]

#Set n
n_value=5









#Define method to write output to file:

def write_output(result):
    print "Writing vectors to outfile..."
    outfile.write(label+' ')
    vector_string = ''
    try:
        for i in result:
                key = str(i[0])
                value = str(i[1])
                vector_string = key+':'+value+' '
                outfile.write(vector_string)
    except NameError:
        print "Name error in write_output()"
    if vector_string:
        outfile.write('\n')




#Define method to write output to file if alternate method is used:
def alt_write_output(curs):
    print "Writing vectors to outfile..."
    outfile.write(label+' ')
    vector_string = ''
    #try:
    curs.execute('select * from ngrams order by value desc')
    for row in curs:
```

```
49              key = row[0]
50              value = row[1]
51              vector_string = key+':'+str(value)+' '
52              outfile.write(vector_string)
53      #except NameError:
54      #   print "Name error in alt_write_output()"
55      if vector_string:
56              outfile.write('\n')
57
58  def alt_method(content):
59      print "MemoryError processing file. Using alternative alternative
            method."
60      content = StringIO.StringIO(content)
61      #file = open(path+f,'rb')
62      #Open sqlite3 database for alternative method for big files
63      os.remove('ngram.db')
64      conn = connect('ngram.db')
65      curs = conn.cursor()
66      curs.execute('''create table ngrams (ngram text primary key,value
            integer)''')
67      ngram_dict = {}
68
69      lines = content.readlines()
70      no_lines = str(len(lines))
71      line = 0
72
73      for l in lines: #content.readlines(): #Read lines one at a time
            and pass to functions
74          line += 1
75
76          print "Processing line number: " +str(line)+ " of " +no_lines
77          hex_content = l.encode("hex")
78          result_part = calc_ngram(hex_content, n_value*2)
79
80          for i in result_part:
81                  key = i[0]
82                  value = i[1]
83
84
85                  try:
86                      if key in ngram_dict:    #Put results in a
                            dictionary for summing frequencies
87                          ngram_dict[key] = ngram_dict[key] + value
88                      else:
89                          ngram_dict[key] = value
90
91                  except MemoryError:              #When dictionary is
                        full, dump to sqlite3 database
92                      print 'Dumping to DB'
```

```
93                      for k,v in ngram_dict.iteritems():
94                          try:
95                              curs.execute('INSERT␣INTO␣ngrams␣VALUES␣
                                    (?,?)',(k,v))
96
97                          except IntegrityError:
98                              curs.execute('UPDATE␣ngrams␣SET␣value␣=␣
                                    value+?␣where␣ngram=?',(v,k))
99
100
101                     try:    #Count in the key,value pair that caused
                            the memory error
102                         curs.execute('INSERT␣INTO␣ngrams␣VALUES␣(?,?)
                                ',(key,value))
103
104                     except IntegrityError:
105                         curs.execute('UPDATE␣ngrams␣SET␣value␣=␣value
                                +?␣where␣ngram=?',(value,key))
106
107                     conn.commit()
108                     print 'Dump␣done'
109                     ngram_dict.clear() #Clear the dumped data from
                            dictionary
110
111     if ngram_dict:
112         for k,v in ngram_dict.iteritems():          #Dump last (non
                full) dictionary to database
113             try:
114                 curs.execute('INSERT␣INTO␣ngrams␣VALUES␣(?,?)',(k,v))
115
116             except IntegrityError:
117                 curs.execute('UPDATE␣ngrams␣SET␣value␣=␣value+?␣where
                        ␣ngram=?',(v,k))
118             conn.commit()
119         ngram_dict.clear()
120
121
122     #file.close()
123     content.close()
124     del content
125     del ngram_dict
126     alt_write_output(curs)
127     conn.close()
128
129
130 #MAIN:
131
132
133 starttime = time()
```

```
134 #Get directory from commandline
135 path = sys.argv[1]
136
137 #Get files in directory
138 files = os.listdir(path)
139 no_files = len(files)
140 counter = 0
141
142 print "Read␣"+ str(no_files) + "␣files␣to␣label␣as␣" + label
143
144 for f in files:
145
146     #Open a file and read into string
147     file = open(path+f,'rb')
148     in_file = file.read()
149     print "\nProcessing␣file:␣"+file.name
150     #print 'Reading '+path+f
151     file.close()
152
153     #Convert to hex and calculate n-gram frequencies:
154
155     try:
156
157         hex_content = in_file.encode("hex")
158         result = calc_ngram(hex_content, n_value*2)# method expects
                inputstring as 1st arg, size of n-gram as 2nd arg
159         write_output(result)
160         del result
161
162     except MemoryError: #if file is too large to process, process one
            line at a time and use Sqlite3 database
163         alt_method(in_file)
164
165
166     #Track progress
167     counter += 1
168     progress = int((counter/no_files)*100)
169     sys.stdout.write("\r" + str(progress) + "%␣done")
170     sys.stdout.flush()
171     del in_file
172
173 outfile.close()
174 endtime = time()
175 print "\nDone.␣Processed␣"+ str(no_files) + "␣in␣" + str(endtime-
    starttime) + "␣seconds."
```

Listing A.5: pdfidsvmgrid.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from PyML import *
import sys, traceback

#Get command line arguments:
infile = sys.argv[1]
outfile = sys.argv[2]

#Import experiment data:
data = SparseDataSet(infile)

#Do a grid search for opitimal values of C and gamma of the Gaussian
    kernel:
param = modelSelection.ParamGrid(svm.SVM(ker.Gaussian()), 'C', [0.1,
    1, 10, 100, 1000],'kernel.gamma', [0.01, 0.1, 1, 10])
model = modelSelection.ModelSelector(param)

#Train the SVM using the optimal values:
print 'Training SVM using ' + infile
svm = model.train(data)

#Save the SVM to file for later use:
print "Saving trained SVM to" + outfile
model.save(outfile)

#Perform cross-validation over data to get performance statistics
print "Performing ten-fold cross validation"
result = model.cv(data,10)


#Save several useful statistics to file:
result_string = '\n\n\nSensitivity: '+ str(result.getSensitivity()) +
     '\nSuccessrate: '+str(result.getSuccessRate()) + '\nBalanced
    successrate: '+ str(result.balancedSuccessRate) +'\nAuROC: '+str(
    result.getROC())
print result_string

try:
    print str(result.getConfusionMatrix())
except:
    pass

try:
    result.plotROC('roc.png')
except:
    print "#"*40
    print "No ROC"
```

```
44      print "#"*40
45      pass
46
47
48
49  f = open(outfile+"_stats", 'w')
50  f.write("\nSVM␣params:␣\n␣"+ str(svm))
51  f.write(result_string)
52
53  try:
54      f.write(str(result.getConfusionMatrix()))
55  except:
56      pass
57
58  f.write('\n\nDecision␣Function:␣'+ str(result.getDecisionFunction()))
59
60
61  f.write("\n\n\n####INFO:####\n")
62  f.write(str(result.getInfo()))
63  f.write("\n\n\n####LOG:####\n")
64  f.write(str(result.getLog()))
65  f.close()
66
67  f = open(outfile+"_labels", 'w')
68  given = result.getGivenLabels()
69  predicted = result.getPredictedLabels()
70  cntr = 0
71  label_list = []
72  label2_list = []
73  for i in given:
74      temp_list = [str(i), str(predicted[cntr])]
75      label_list.append(temp_list)
76      cntr += 1
77  for item in label_list:
78    f.write("%s\n" % item)
79
80  f.close()
```

Listing A.6: pdfidsvmnorm.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from PyML import *
import sys, traceback

#Get command line arguments:
infile = sys.argv[1]
outfile = sys.argv[2]

#Import experiment data:
data = SparseDataSet(infile)

#Specify type of kernel, optimal values from previous experiments,
    and type of normalization
data.attachKernel('gaussian', C=100, gamma=0.1, normalization = '
    cosine')

#Create the SVM:
s = SVM()

#Perform cross-validation over data to get performance statistics
print "Performing ten-fold cross validation"
result = s.cv(data,10)




#Save several useful statistics to file:

f = open(outfile+"_stats", 'w')
f.write("\nSVM params: \n "+ str(svm))
f.write('\n\n\nSensitivity: '+ str(result.getSensitivity()) + '\
    nSuccessrate: '+str(result.getSuccessRate())+'\nConfusionmatrix\n'
    )
f.write('\n\nDecision Function: '+ str(result.getDecisionFunction))
try:
    f.write(str(result.getConfusionMatrix()))
except:
    pass

f.write("\n\n\n####INFO:####\n")
f.write(str(result.getInfo()))
f.write("\n\n\n####LOG:####\n")
f.write(str(result.getLog()))
f.close()

f = open(outfile+"_labels", 'w')
given = result.getGivenLabels()
predicted = result.getPredictedLabels()
```

```
45 cntr = 0
46 label_list = []
47 label2_list = []
48 for i in given:
49     temp_list = [str(i), str(predicted[cntr])]
50     label_list.append(temp_list)
51     cntr += 1
52 for item in label_list:
53   f.write("%s\n" % item)
54
55 f.close()
```

Listing A.7: pdfidsvm.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from PyML import *
import sys, traceback

#Get command line arguments:
infile = sys.argv[1]
outfile = sys.argv[2]

#Import experiment data:
data = SparseDataSet(infile)

#Attach optimal kernel from earlier experiments:
data.attachKernel('gaussian', gamma = 0.1, C = 100)

#Create the SVM:
s = SVM()

#Perform cross-validation over data to get performance statistics
print "Performing ten-fold cross validation"
result = s.cv(data,10)


#Save several useful statistics to file:
result_string = '\n\n\nSensitivity: '+ str(result.getSensitivity()) +
    '\nSuccessrate: '+str(result.getSuccessRate()) + '\nBalanced
   successrate: '+ str(result.balancedSuccessRate) +'\nAuROC: '+str(
   result.getROC())
print result_string

try:
    print str(result.getConfusionMatrix())
except:
    pass

try:
    result.plotROC('roc.png')
except:
    print "#"*40
    print "No ROC"
    print "#"*40
    pass



f = open(outfile+"_stats", 'w')
f.write("\nSVM params: \n "+ str(svm))
f.write(result_string)
```

```
46
47 try:
48     f.write(str(result.getConfusionMatrix()))
49 except:
50     pass
51
52 f.write('\n\nDecision␣Function:␣'+ str(result.getDecisionFunction()))
53
54
55 f.write("\n\n\n####INFO:####\n")
56 f.write(str(result.getInfo()))
57 f.write("\n\n\n####LOG:####\n")
58 f.write(str(result.getLog()))
59 f.close()
60
61 f = open(outfile+"_labels", 'w')
62 given = result.getGivenLabels()
63 predicted = result.getPredictedLabels()
64 cntr = 0
65 label_list = []
66 label2_list = []
67 for i in given:
68     temp_list = [str(i), str(predicted[cntr])]
69     label_list.append(temp_list)
70     cntr += 1
71 for item in label_list:
72   f.write("%s\n" % item)
73
74 f.close()
```

Listing A.8: flow2pdf.py

```
#!/usr/bin/python
import os, sys, time, subprocess

if __name__ == '__main__':
        #Get time for usage calulations
        start_time = time.time()

        #Get directory and label from commandline
        alert_file = sys.argv[1]
        root_path = '/home/master/master/SVM/expREAL/'
        log_path = root_path+'pdf/'
        tmp_path = root_path+'tmp/'
        pcap_path = root_path+'pcaps/'
        pdf_path = root_path+'pdfs/'
        flow_path = root_path+'flow/'
        new_flow = root_path+'new_flow/'
        #Convert all to pcap
        files = os.listdir(log_path)
        for f in files:
                if 'snort' in f:
                        subprocess.check_call(['cp',log_path+f,
                            tmp_path])
        files = os.listdir(tmp_path)
        for f in files:
                pcap_stime = time.time()
                subprocess.check_call(["u2boat",tmp_path+f,pcap_path+
                    f+".pcap"])
                #print "/var/log/snort/u2boat",pcap_path+f,pcap_path+
                    f+".pcap"
                pcap_etime = time.time()
                print "PCAP conversion per file" + str(pcap_etime-
                    pcap_stime)

        #Parse alert file
        f0 = open(alert_file,'rb')
        alerts = f0.readlines()
        f0.close()

        time_pattern = '%Y/%m/%d-%H:%M:%S'

        os.chdir(flow_path)

        for line in alerts:
                parts =  line.split()
                ttime = "2011/"+ str(parts[0].partition('.')[0])
                epoch = int(time.mktime(time.strptime(ttime,
                    time_pattern)))
                src = parts[9]
```

```
44                  dst = parts[11]
45                  src_addr = src.partition(':')[0]
46                  src_port = src.partition(':')[2]
47                  print epoch,src,dst,src_addr,src_port
48
49
50                  #Get flow
51                  files = os.listdir(pcap_path)
52                  for f in files:
53                          flow_stime = time.time()
54                          subprocess.check_call(["tcpflow","-r",
                                pcap_path+f,"src port "+src_port+" and
                                host "+src_addr])
55                          flow_etime = time.time()
56                          print "Per file pcap to flow" + str(
                                flow_etime-flow_stime)
57
58          #Get files in directory and open
59          files = os.listdir(flow_path)
60
61          for f in files:
62                  header_stime = time.time()
63                  node = open(flow_path+f,'rb')
64                  data = node.read()
65                  node.close()
66
67                  new_data = data.partition('%PDF-')
68                  new_data = new_data[1]+new_data[2]
69                  #new_data = new_data.partition('%%EOF')
70
71                  outfile = open(pdf_path+f+".pdf","w")
72                  outfile.write(new_data)
73                  outfile.close()
74                  header_etime = time.time()
75                  print "Per file ehader removal" + str(header_etime -
                        header_stime)
76
77          #Calculate used time
78          end_time = time.time()
79          used_time = end_time - start_time
80          print "Work done in " + str(used_time) + " seconds."
```

Listing A.9: classifynew.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from PyML import *
from PyML.classifiers.svm import loadSVM
import sys, os
import pdfextract



#Get command line arguments:
svm_file = sys.argv[1]
dataset = sys.argv[2]
path = sys.argv[3]

#Load trained SVM:
data = SparseDataSet(dataset)
data.attachKernel('gaussian', gamma = 0.1, C = 100) #For testing

SVM = loadSVM(svm_file,data)

#SVM = SVM()  #For testing
#SVM.train(data, saveSpace = False) # For testing
#SVM.save(svm_file)

#Extract features from infiles:

files = os.listdir(path)

for f in files:
    node = open(path+f,'r')
    data = node.read()
    node.close()


    features = pdfextract.external_call(data,f)

    vector = ''
    for key,value in features.iteritems():
                    vector += key+":"+str(value)+" "


    temp_file = open("tmp","w")
    temp_file.write(vector)
    temp_file.close()
    test_data = SparseDataSet("tmp")

    result = SVM.test(test_data)
    print f
```

```
49      print vector
50      print result.getPredictedLabels()
```

Listing A.10: parsemsse.py

```
#!/usr/bin/python
import re, os, sys, time


if __name__ == '__main__':

    in_file = sys.argv[1] #Pass input file from command line

    f1 = open(in_file,'rb')

    exploit_count = {}
    counted_files = []
    match = "DETECTION_ADD"


    print "Reading and parsing log..."
    #loglines = get_exploits(log)

    for l in f1: #read each line in log-file
        if match in l: #If DETECTION_ADD lines found
            params = l.split(' ') #Split the line
            file_name = params[5].split('\\')[5].rstrip('\n') #Get
                the file name
            if file_name in counted_files:
                print "%s already counted!" %file_name
            else:
                counted_files.append(file_name)
                try:
                    exploit_count[params[2]] += 1 #Count exploit
                        frequency
                except KeyError:
                    exploit_count[params[2]] = 1 #Add new exploits

        else:
            pass #If no match, do nothing

    f1.close()
    print "Writing to file..."

    f2 = open('test_out','a') #Write exploit names and count to file
    for i,j in exploit_count.iteritems():
        f2.write(i)
        f2.write(' %d' %j)
        f2.write('\n')

    f2.close()
    print "Done."
```

B Threats Detected by MSSE

As of 15th of November 2011.

```
Exploit:Win32/Pdfjsc.CX 45
Exploit:Win32/Pdfjsc.CZ 1
Exploit:Win32/Pdfjsc.CS 30
Exploit:Win32/Pdfjsc.CR 1600
Exploit:Win32/Pdfjsc.CU 1
Exploit:Win32/Pdfjsc.CH 18
Exploit:Win32/Pdfjsc.CK 2
Exploit:Win32/Pdfjsc.CJ 2
Exploit:Win32/Pdfjsc.CM 15
Exploit:Win32/Pdfjsc.CL 1
Exploit:Win32/Pdfjsc.CO 8
Exploit:Win32/Pdfjsc.CA 4
Exploit:Win32/Pdfjsc.CC 4
Exploit:Win32/Pdfjsc.CG 65
Exploit:JS/Mult.BW 4
Exploit:JS/Mult.BQ 2
Exploit:JS/Mult.BX 2
Exploit:JS/Mult.BB 3
Exploit:Win32/Pdfjsc.UR 2
Exploit:Win32/Pdfjsc.UQ 2
Exploit:Win32/Pdfjsc.UP 2
Exploit:Win32/Pdfjsc.UW 8
Exploit:Win32/Pdfjsc.UV 2
Exploit:Win32/Pdfjsc.UU 2
Exploit:Win32/Pdfjsc.UT 2
Exploit:Win32/Pdfjsc.UY 2
Exploit:Win32/Pdfjsc.UX 2
Exploit:JS/Pdfcmi.K 6
Exploit:Win32/Pdfjsc.UG 2
Exploit:Win32/Pdfjsc.UF 2
Exploit:Win32/Pdfjsc.UE 8
Exploit:Win32/Pdfjsc.UD 2
Exploit:Win32/Pdfjsc.UK 2
Exploit:Win32/Pdfjsc.UJ 2
Exploit:Win32/Pdfjsc.UI 2
```

```
Exploit:Win32/Pdfjsc.UH 2
Exploit:Win32/Pdfjsc.UO 2
Exploit:Win32/Pdfjsc.UN 2
Exploit:Win32/Pdfjsc.UM 4
Exploit:Win32/Pdfjsc.UL 2
Exploit:Win32/Pdfjsc.JO 4
Exploit:Win32/Pdfjsc.JI 16
Exploit:Win32/Pdfjsc.JX 20
Trojan:JS/Proxas.A 2
VirTool:Win32/VBInject.gen!DG 4
Exploit:Win32/Pdfjsc.JP 506
Exploit:Win32/Pdfjsc.JQ 10
Exploit:JS/Heapspray 5
Exploit:Win32/Pdfjsc.GO 312
Exploit:Win32/Pdfjsc.GI 6
Exploit:Win32/Pdfjsc.GJ 24
Exploit:Win32/Pdfjsc.GG 2
Exploit:Win32/Pdfjsc.GF 106
Exploit:Win32/Pdfjsc.GX 4
Exploit:Win32/Pdfjsc.GZ 10
Exploit:Win32/Pdfjsc.GT 124
Exploit:Win32/Pdfjsc.GV 13
Exploit:Win32/Pdfjsc.GQ 11
Exploit:Win32/Pdfjsc.GP 8
Exploit:Win32/Pdfjsc.GR 168
Exploit:Win32/Pdfjsc.IC 2
Exploit:Win32/Pdfjsc.IK 254
Exploit:Win32/Pdfjsc.PE!gen 872
Exploit:Win32/Pdfjsc.PM 64
Exploit:Win32/Pdfjsc.PN 18
Exploit:Win32/Pdfjsc.PO 2
Exploit:Win32/Pdfjsc.PH 120
Exploit:Win32/Pdfjsc.PI 10
Exploit:Win32/Pdfjsc.PK 2
Exploit:Win32/Pdfjsc.PG 2
Exploit:Win32/CVE-2010-2883 4
Exploit:Win32/Pdfjsc.PX 6
Exploit:Win32/Pdfjsc.PY 8
Exploit:Win32/Pdfjsc.PZ 2
Exploit:Win32/Pdfjsc.PT 4
Exploit:Win32/Pdfjsc.PU 2
Exploit:Win32/Pdfjsc.PW 2
```

```
Exploit:Win32/Pdfjsc.PP 2
Exploit:Win32/Pdfjsc.PQ 490
Exploit:Win32/Pdfjsc.PR 8
Exploit:Win32/Pdfjsc.PS 28
Exploit:Win32/Pdfjsc.PL 2
Exploit:Win32/Senglot.D 4
Exploit:Win32/Pdfjsc.N 1
Exploit:Win32/Pdfjsc.L 2
Exploit:Win32/Pdfjsc.J 375
Exploit:Win32/Pdfjsc.H 46
Exploit:Win32/Pdfjsc.G 368
Exploit:Win32/Pdfjsc.D 1
Exploit:Win32/Pdfjsc.C 3
Exploit:Win32/ShellCode.E 1
Exploit:Win32/ShellCode.A 64
Exploit:Win32/Pdfjsc.X 2
Exploit:Win32/Pdfjsc.NO 6
Exploit:Win32/Pdfjsc.Q 93
Exploit:JS/Mult.AB 1
Exploit:JS/Mult.AG 20
Exploit:JS/Pdfjsc.R 48
Exploit:JS/Pdfjsc.I 6
Exploit:JS/Pdfjsc.D 4
Exploit:JS/Pdfjsc.E 96
Exploit:Win32/Pidief.B 2
TrojanDownloader:JS/Qakbot.B 2
Exploit:Win32/Pidief.D 15
TrojanDownloader:JS/Qakbot.D 12
TrojanDownloader:JS/Qakbot.G 12
Exploit:Win32/Senglot.U 2
TrojanDownloader:JS/Qakbot.H 2
Exploit:Win32/Pidief.J 54
Exploit:Win32/Pidief.K 14
Exploit:Win32/Pidief.M 4
Exploit:Win32/Pidief.O 662
Exploit:Win32/Pidief.P 98
Exploit:Win32/Pdfjsc.MZ 134
Exploit:Win32/Pidief.R 1
Exploit:Win32/Pidief.S 1
Exploit:Win32/Pidief.T 2
Exploit:Win32/Pidief.U 1
Exploit:Win32/Pidief.V 2
```

```
Exploit:Win32/Pidief.W 8
Exploit:Win32/Pidief.X 3
Exploit:Win32/Pidief.Y 1
Trojan:SWF/Sprayload.A 1
Exploit:SWF/CVE-2010-1297.A 2
Exploit:SWF/CVE-2010-1297.F 5
Exploit:SWF/CVE-2010-1297.E 2
Exploit:Win32/Pifde 18
Exploit:Win32/Pidief.gen!A 2
Exploit:Win32/Pidief.gen!C 128
Exploit:Win32/Pidief.gen!B 4
Exploit:Win32/Pdfjsc.BG 1
Exploit:Win32/Pdfjsc.BE 1
Exploit:Win32/Pdfjsc.BN 1
Exploit:Win32/Pdfjsc.BO 2
Exploit:Win32/Pdfjsc.BL 1
Exploit:Win32/Pdfjsc.BK 28
Exploit:Win32/Pdfjsc.BH 90
Exploit:Win32/Pdfjsc.BI 112
Exploit:Win32/Pdfjsc.BU 18
Exploit:Win32/Pdfjsc.gen!B 82
Exploit:Win32/Pdfjsc.DG 99
Exploit:Win32/Pdfjsc.gen!C 22
Exploit:Win32/Pdfjsc.TP 18
Exploit:Win32/Pdfjsc.TU 2
Exploit:JS/Pdfjsc.BA 1
Exploit:Win32/Pdfjsc.TA 2
Exploit:Win32/Pdfjsc.TB 10
Exploit:Win32/Pdfjsc.TC 34
Exploit:Win32/Pdfjsc.TD 2
Exploit:Win32/Pdfjsc.TE 2
Exploit:Win32/Pdfjsc.TF 2
Exploit:Win32/Pdfjsc.TG 2
Exploit:Win32/Pdfjsc.TI 2
Exploit:Win32/Pdfjsc.TJ 2
Exploit:Win32/Pdfjsc.TK 20
Exploit:Win32/Pdfjsc.TL 18
Exploit:Win32/Pdfjsc.TM 12
Exploit:Win32/Pdfjsc.TN 16
Exploit:Win32/Pdfjsc.TO 4
Exploit:Win32/CVE-2010-2883.A 82
Exploit:Win32/Pdfjsc.AY 2
```

```
Exploit:Win32/Pdfjsc.AV 20
Exploit:Win32/Pdfjsc.AS 22
Exploit:Win32/Pdfjsc.AJ 1
Exploit:Win32/Pdfjsc.AD 4
Exploit:Win32/Pidief.AR 1
Exploit:Win32/Pidief.AP 2
Exploit:Win32/Pidief.AU 58
Exploit:Win32/Pidief.AZ 45
Exploit:Win32/Pidief.AX 1
Exploit:JS/Mult.DC 2
Exploit:Win32/Pidief.AF 1
Exploit:Win32/Pidief.AG 1
Exploit:Win32/Pidief.AD 1
Exploit:Win32/Pidief.AE 1
Exploit:Win32/Pidief.AJ 2
Exploit:Win32/Pidief.AK 1
Exploit:Win32/Pidief.AH 1
Exploit:Win32/Pidief.AN 2
Exploit:Win32/Pidief.AL 1
Exploit:Win32/Pidief.AM 1
Exploit:Win32/Pdfjsc.FI 3
Exploit:Win32/Pdfjsc.FA 2
Exploit:Win32/Pdfjsc.FF 1
Exploit:Win32/Pdfjsc.FG 106
Exploit:Win32/Pdfjsc.FD 2
Exploit:Win32/Pdfjsc.FE 2
Exploit:Win32/Pdfjsc.FS 100
Exploit:Win32/Pdfjsc.FQ 1
Exploit:Win32/Pdfjsc.FW 4
Exploit:Win32/Pdfjsc.FU 16
Exploit:Win32/RdrJmp.A 1
Exploit:Win32/Pdfjsc.KE 1
Exploit:Win32/Pdfjsc 7
TrojanDropper:Win32/Pidrop.A 44
Exploit:Win32/Pdfheap.A 3
Exploit:Win32/Pdfjsc.HA 1
Exploit:Win32/Pdfjsc.HT 1
Exploit:Win32/Pdfjsc.HW 1
Exploit:Win32/Pdfjsc.HQ 4
Exploit:Win32/CVE-2010-0188 5
Exploit:Win32/Pdfjsc.HX 11
Exploit:Win32/Pdfjsc.SI 2
```

```
Exploit:Win32/Pdfjsc.SH 2
Exploit:Win32/Pdfjsc.SK 2
Exploit:Win32/Pdfjsc.SJ 2
Exploit:Win32/Pdfjsc.SM 2
Exploit:Win32/Pdfjsc.SL 2
Exploit:Win32/Pdfjsc.SO 2
Exploit:Win32/Pdfjsc.SN 2
Exploit:Win32/Pdfjsc.SA 6
Exploit:Win32/Pdfjsc.SC 2
Exploit:Win32/Pdfjsc.SB 10
Exploit:Win32/Pdfjsc.SE 2
Exploit:Win32/Pdfjsc.SD 2
Exploit:Win32/Pdfjsc.SG 2
Exploit:Win32/Pdfjsc.SF 2
Exploit:Win32/Pdfjsc.SY 2
Exploit:Win32/Pdfjsc.SX 2
Exploit:Win32/Pdfjsc.SZ 2
Exploit:Win32/Pdfjsc.SQ 2
Exploit:Win32/Pdfjsc.SP 2
Exploit:Win32/Pdfjsc.SS 2
Exploit:Win32/Pdfjsc.SR 2
Exploit:Win32/Pdfjsc.SU 2
Exploit:Win32/Pdfjsc.ST 2
Exploit:Win32/Pdfjsc.SW 2
Exploit:Win32/Pdfjsc.SV 2
Exploit:Win32/Pdfjsc.EB 1158
Exploit:Win32/Pdfjsc.EK 5
Exploit:Win32/Pdfjsc.EJ 340
Exploit:Win32/Pdfjsc.EH 1490
Exploit:JS/Pdfupf.A 8
Exploit:Win32/Pdfjsc.EM 5142
Exploit:Win32/Pdfjsc.EL 6
Exploit:Win32/Pdfjsc.ES 204
Exploit:Win32/Pdfjsc.ER 162
Exploit:Win32/Pdfjsc.EP 210
Exploit:Win32/Pdfjsc.EW 1
Exploit:Win32/Pdfjsc.EV 3
Exploit:Win32/Pdfjsc.EU 128
Exploit:Win32/Pdfjsc.ET 1
Exploit:Win32/Pdfjsc.EZ 58
Exploit:Win32/Pdfjsc.EY 14
Exploit:Win32/Pdfjsc.EX 212
```

```
Exploit:Win32/Pidief.BK 1
Exploit:Win32/Pidief.BH 1
Exploit:Win32/Pidief.BO 1
Exploit:Win32/Pidief.BM 407
Exploit:Win32/Pidief.BA 295
Exploit:Win32/Pidief.BE 1
Exploit:Win32/Pidief.BD 442
Exploit:Win32/Pidief.BX 1
Exploit:Win32/Pidief.BT 1
Exploit:iPhoneOS/Pidief.A 1
Exploit:JS/Pdfjsc.AZ 2
Exploit:Win32/Pdfjsc.JV 314
Exploit:JS/Pdfjsc.AX 2
Exploit:JS/Pdfjsc.AY 4
Exploit:Win32/Pdfjsc.JW 104
Exploit:Win32/Pdfjsc.WG 4
Trojan:Win32/Swrort.A 46
Exploit:Win32/Pdfjsc.LH 12
Exploit:Win32/Pdfjsc.LL 2
Exploit:Win32/Pdfjsc.LM 18
Exploit:Win32/Pdfjsc.LA 2
Exploit:Win32/Pdfjsc.LC 1
Exploit:Win32/Pdfjsc.LY 1
Exploit:Win32/Pdfjsc.LU 17
Exploit:Win32/Pdfjsc.LV 52
Exploit:JS/Mult.CF 1
Exploit:JS/Mult.CD 3
Exploit:JS/Mult.CA 2
VirTool:JS/Obfuscator.AG 24
VirTool:JS/Obfuscator.AD 2
Exploit:JS/Mult.CM 4
Exploit:JS/Mult.CJ 6
VirTool:JS/Obfuscator.T 2
Exploit:Win32/Pdfabdic 12
Exploit:Win32/ShellCode.gen!C 14
VirTool:JS/Obfuscator.K 4
Exploit:Win32/Pdffir.A 65
Exploit:Win32/Pdfjsc.KA 2
Exploit:Win32/Pdfjsc.KB 6
Exploit:JS/ShellCode.gen 52
Exploit:Win32/Pdfjsc.KG 2
Exploit:Win32/Pdfjsc.KI 16
```

```
Exploit:Win32/Pdfjsc.KH 1
Exploit:Win32/Pdfjsc.KJ 2
Exploit:Win32/Pdfjsc.KL 2
Exploit:Win32/Pdfjsc.KO 2
Exploit:Win32/Pdfjsc.KQ 1
Exploit:Win32/Pdfjsc.KP 2
Exploit:Win32/Pdfjsc.KS 2
Exploit:Win32/Pdfjsc.KT 16
Exploit:Win32/Pdfjsc.KY 2
Exploit:Win32/Pdfjsc.KZ 1
TrojanDownloader:Win32/Small.gen!C 40
Exploit:Win32/Pdfjsc.MB 9
Exploit:JS/ShellCode.F 2
Exploit:JS/ShellCode.M 8
Exploit:Win32/Pdfjsc.RV 2
Exploit:Win32/Pdfjsc.RW 2
Exploit:Win32/Pdfjsc.RT 2
Exploit:Win32/Pdfjsc.RU 2
Exploit:Win32/Pdfjsc.RR 2
Exploit:Win32/Pdfjsc.RS 2
Exploit:Win32/Pdfjsc.RP 2
Exploit:Win32/Pdfjsc.RQ 2
Exploit:Win32/Pdfjsc.RZ 2
Exploit:Win32/Pdfjsc.RX 2
Exploit:Win32/Pdfjsc.RY 2
Exploit:Win32/Pdfjsc.RD 2
Exploit:Win32/Pdfjsc.RB 2
Exploit:Win32/Pdfjsc.RC 2
Exploit:Win32/Pdfjsc.RA 2
Exploit:Win32/Pdfjsc.RN 2
Exploit:Win32/Pdfjsc.RO 2
Exploit:Win32/Pdfjsc.RH 22
Exploit:Win32/CVE-2009-1862.A 4
Exploit:JS/Mult.DD 5
Exploit:Win32/Pdfjsc.MI 2
Exploit:Win32/Pdfjsc.OP 16
Exploit:Win32/Pidief.I 1
Exploit:Win32/Pdfjsc.MA 154
Exploit:Win32/Pdfjsc.OK 3
Exploit:Win32/Pidief 2
Exploit:Win32/Pdfjsc.DA 5
Exploit:Win32/Pdfjsc.DB 2
```

```
Exploit:Win32/Pdfjsc.DC 1
Exploit:Win32/Pidief.Q 1
Exploit:Win32/Pdfjsc.DI 6
Exploit:Win32/Pdfjsc.DJ 3
Exploit:Win32/Pdfjsc.DK 2
Exploit:Win32/Pdfjsc.DM 3
Exploit:Win32/Pdfjsc.DN 20
Exploit:Win32/Pdfjsc.DR 8
Exploit:Win32/Pdfjsc.DZ 1
Exploit:Win32/Pidief.F 1
Exploit:Win32/Pdfjsc.VL 4
Exploit:Win32/Pdfjsc.QO 2
Exploit:Win32/Pdfjsc.QN 2
Exploit:Win32/Pdfjsc.QM 2
Exploit:Win32/Pdfjsc.QL 2
Exploit:Win32/Pdfjsc.QK 2
Exploit:Win32/Pdfjsc.QJ 2
Exploit:Win32/Pdfjsc.QI 2
Exploit:Win32/Pdfjsc.QH 2
Exploit:Win32/Pdfjsc.QG 2
Exploit:Win32/Pdfjsc.QF 10
Exploit:Win32/Pdfjsc.QE 2
Exploit:Win32/Pdfjsc.QD 4
Exploit:Win32/Pdfjsc.QC 2
Exploit:Win32/Pdfjsc.QA 10
Exploit:Win32/Pdfjsc.QZ 2
Exploit:Win32/Pdfjsc.QY 2
Exploit:Win32/Pdfjsc.QX 6
Exploit:Win32/Pdfjsc.QW 2
Exploit:Win32/Pdfjsc.QV 2
Exploit:Win32/Pdfjsc.QU 16
Exploit:Win32/Pdfjsc.QT 2
Exploit:Win32/Pdfjsc.QS 8
Exploit:Win32/Pdfjsc.QR 2
Exploit:Win32/Pdfjsc.QQ 2
Exploit:Win32/Pdfjsc.QP 2
Exploit:Win32/Pdfjsc.CV 3178
Exploit:Win32/Pidief.BZ 1386
Exploit:Win32/Pdfjsc.EQ 55
Exploit:Win32/Pdfjsc.gen!A 1399
Exploit:Win32/Pdfjsc.FR 1
Exploit:JS/Pdfjsc.A 190
```

```
Exploit:JS/Pdfjsc.B 720
TrojanDownloader:JS/SetSlice 28
```