BULK ANALYSIS OF MALICIOUS PDF DOCUMENTS

by

Shauna M. Policicchio

B.S., Saint Vincent College, 2013

Submitted to the Graduate Faculty of
the School of Information Science in partial fulfillment
of the requirements for the degree of

Master of Sciences

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

SCHOOL OF INFORMATION SCIENCE

This thesis was presented

by

Shauna M. Policicchio

It was defended on

March 11, 2015

and approved by

Prashant Krishnamurthy, PhD, Associate Professor

Jonathan Spring, CERT Coordination Center

Leigh Metcalf, PhD, CERT Coordination Center

Balaji Palanisamy, PhD, Assistant Professor

Konstantinos Pelechrinis, PhD, Assistant Professor

Thesis Advisors: Prashant Krishnamurthy, PhD, Associate Professor,

Jonathan Spring, CERT Coordination Center

BULK ANALYSIS OF MALICIOUS PDF DOCUMENTS

Shauna M. Policicchio, M.S.

University of Pittsburgh, 2015

From 2007 onward, the PDF document has proven to be a successful vector for malware infections, making up 80% of all exploits found by Cisco ScanSafe in 2009 [1]. Creating new PDF documents is very easy and the volume of PDF documents identified as malicious has grown beyond the capabilities of security researchers to analyze by hand. The solution proposed by this thesis is to automatically extract features from the PDF documents to group and classify them, so that similar malware may be identified without manual analysis, thus reducing the workload of the malware analyst. These features may also be studied to identify trends within the PDF documents, such as similar exploits or obfuscation techniques. Our results show that the object graph structure of the PDF document is an effective way to create an initial grouping of malicious PDF documents.

Finding similarities in PDF documents reveals further information about a data set. In our first case study, we examine the entire data set to identify large groups of similar PDF documents and make conjectures about their origins. In our second case study, we use a PDF document of known origin to find similar PDF documents within a data set. Through the two case studies, we were able to identify 50.3% of our data set with very little manual analysis of the malicious PDF documents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

There are so many people I would like to thank. A big thank you to Jonathan Spring for his countless reviews and edits of my paper. Thank you to Dr. Prashant Krishnamurthy for his guidance and support. Thank you to Dr. Leigh Metcalf and Ed Stoner for allowing me to work with them and for their help with my project. Thank you to Michael Appel for help with the project and help coding the feature extraction tool. Thank you to my fellow SFS students for all of your help and support. Thank you to my parents, family, and friends for your support in everything I do. A huge thank you to my husband for all of your support throughout this entire endeavor. Finally, thank you to God for making all of this possible. This work was funded in part by NSF-DGE Award #1027167.

Copyright 2014 Carnegie Mellon University
This material is based upon work funded and supported by Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Department of Homeland Security or the United States Department of Defense.
References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.
NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGI-

NEERING INSTITUTE MATERIAL IS FURNISHED ON AN AS-IS BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution.

Carnegie Mellon®, CERT®, and CERT Coordination Center® are registered marks of Carnegie Mellon University.

DM-0001941

1.0 INTRODUCTION

End-user technology has spread beyond Desktop PCs to a multitude of devices such as tablets, smart phones, and other smart appliances. Data creation and sharing is at an all-time high, so document formats must be portable across various environments [2]. The Portable Document Format (PDF) offers a standard format to produce and read documents across platforms. PDF documents can be created and read across operating systems, mobile devices, tablets, and also by printers and copiers. The portability of the PDF has led to its widespread adoption and use, making it a commonly-used file format today.

The Adobe Portable Document Format was created by John Warnock in 1993 [3] and was named an open standard for electronic documents by the International Organization for Standardization (ISO) in 2008 [4]. In 2010 roughly 90% of computers had a version of Adobe Acrobat or Adobe Reader installed [5]. The flexibility of PDF documents led to their success: they not only allow for text boxes and character encodings, but also embedded JavaScript and ActionScript (Adobe Flash), dynamic forms, action triggers, and live data retrieval (via network URL) [6]. However, the variety of content allowable in PDF documents provides adversaries with many vectors of attack, causing the PDF to become one of the most popular avenues for delivering malicious content.

In 2001, the first malicious PDF, Peachy, was discovered [7]. Peachy was a simple attack; it affected Adobe Acrobat users and mailed itself out to everyone in a user's Microsoft Outlook contact list. Common Vulnerabilities and Exposures (CVE) is a scheme used by Mitre to uniquely name publicly known software flaws [8]. The first CVE targeting Adobe software was published in 2006, marking malicious PDF documents as a real threat [9]. From there, the number of PDF attacks skyrocketed, making up 80% of all exploits in 2009 [1]. In 2011, Adobe released Adobe Reader X, which comes with Protected Mode, a sandbox that isolates Adobe Reader from the rest of the operating system [10]. Since then, several exploits that bypass the Protected Mode have been seen both in academia and in the wild, but lack popularity.

Targeted and untargeted social engineering attacks using PDF documents have proven to be successful for malware infections [11]. Clever guises such as billing invoices and scanned documents sent from the copier fool the unwitting user into opening the PDF and infecting his or her system [12]. These techniques are used in mass-mailed PDF documents: malicious PDF documents sent to a multitude of users with the intent of achieving as many successful infections as possible. Mass-mailed PDF documents also tend to entice users by claiming to contain information relating to current events [11]. Targeted PDF documents are sent to fewer users, typically with specific victims in mind. With a little research, adversaries make these PDF documents more appealing to the targeted users by using personal information to appear legitimate.

Drive-by-downloads occur when a user unknowingly visits an infected web site [11]. Malicious PDF documents are a popular vector in drive-by-download attacks. The PDF document opens in the background and infects a user without his or her knowledge. Malicious PDF documents hosted on a web page are generally smaller than emailed PDF documents, and do not contain any content other than the exploit and payload, while spammed malicious PDF documents sometimes contain legitimate-looking text and images along with the exploit [11].

The flexibility of PDF documents causes problems in the detection and analysis of malicious PDF documents. A change in one small part of the PDF may defeat document-signature-based detection but still deliver the same malicious payload. Manual analysis of a single malicious document is a slow process that requires both static and dynamic analysis by a malware analyst. Since malicious PDF documents are easy to change, there are collections of hundreds of thousands of different PDF documents waiting to be manually analyzed by security researchers, an impossible feat. Automated analysis alleviates pressure on human analysts. *Therefore, this thesis proposes a set of features to be used in automated analysis of a large collection of malicious PDF documents.* The results of this analysis lead us to question the possibility of efficient automated detection of malicious PDF documents or other malicious documents, and if a stricter document structure might make automated detection easier.

This thesis builds on previous work by focusing on a novel area: malicious PDF document classification and grouping for analysis of a large data set of malicious documents. Due to the ubiquitous nature of PDF documents, creating new PDF documents is very easy and the volume of PDF documents identified as malicious has grown beyond the capabilities of security researchers to analyze by hand. The solution proposed by this thesis is to automatically extract features from the PDF documents to group and classify them, so that similar malware may be identified without manual analysis, thus reducing the workload of the malware analyst. These features may also be studied to identify trends within the PDF documents, such as similar exploits or obfuscation techniques.

The following chapter will cover the necessary background information; Section 2.1 briefly describes the Portable Document format and Section 2.2 reviews related work regarding malicious PDF document analysis. Current tools for PDF analysis and creation are listed in Section 2.3. Chapter 3 details our experiments and results; more specifically, Section 3.1 describes the methods and techniques used on the data followed by the proposed feature set in 3.2.1, justification for the feature set in 3.2.2, and methods for automated feature extraction in 3.2.3. The results are listed in Section 3.3 and discussed in Section 3.4. Chapter 4 describes use cases in which artifacts of specific exploit kits are found in the data set. The first case study identifies the Blackhole Exploit Kit in the data set in Section 4.1, and the second case study in Section 4.2 identifies PDF documents created by the Phoenix Exploit Kit using known samples for reference. Finally, Chapter 5 provides a summary and describes potential future work.

2.0 BACKGROUND AND RELATED WORK

This chapter will provide a brief overview of the Portable Document Format and will discuss some related work in regards to malicious document detection and analysis. Documents created using the Portable Document Format follow a restrictive standard. Despite this fact, Adobe Acrobat and Adobe Reader allow for loose interpretation of the standard and will attempt to open severely malformed files. This loose interpretation provides adversaries with more options in the creation of malicious PDF documents since they are not restricted to the exact PDF standard. The Portable Document Format is not the only format targeted by malicious documents; Microsoft Office products are also vulnerable to malicious document creation. As a result, much research has been done in the area of malicious document detection and analysis. Section 2.1 will give an overview of the Portable Document Format, Section 2.2 will survey related work in malicious document analysis, and Section 2.3 will introduce some tools for PDF document analysis.

2.1 PORTABLE DOCUMENT FORMAT

The Portable Document Format follows the ISO 32000-1:2008 standard [4]. However, the Adobe Reader allows for loose interpretation of the standard, often opening, and attempting to fix, severely malformed files. The variety of features accepted within a PDF document and the loosely interpreted standard allow malware developers multiple vectors for infection with a single PDF document. Some of the vectors, such as Adobe Flash, are similar to other web exploits, while others, such as JavaScript, have intricacies unique to PDF documents. For example, Adobe provides its own API for JavaScript that allows the author to access and change different parts of the document using JavaScript. Understanding the PDF document structure is important for the analysis of malicious PDF documents.

A PDF document is composed of an object tree with a header and trailer, and a catalog dictionary at the root [13]. The header consists of `%PDF-` followed by the version number of the PDF specification that the PDF document follows. Figure 1 shows the PDF trailer, a dictionary that allows the program to quickly find specific objects within the PDF document by listing the object and its byte offset. It also contains the offset of the cross-reference table. The `/Size` tag specifies the number of entries in the cross-reference table, and `startxref` lists the byte offset of the cross-reference table. `/Root` contains an indirect reference to the catalog dictionary, which in this example is located in object 1. The last line of the PDF document should be the end of file marker `%%EOF`.

```
trailer
      <<  /Size 5
          /Root 1 0 R
      >>
startxref
38152
%%EOF
```

Figure 1: An example of the trailer of a file.

Figure 2 demonstrates the catalog dictionary of a PDF document. The `/Root` entry of the file trailer contains the catalog dictionary, which refers to other objects within the file that define attributes of the document, such as contents, outline, et cetera.

The `/Pages` and `/Outlines` entries in the catalog dictionary contain references to other objects in the document. An object may reference the values of any other object in the document, creating a complex hierarchy of dependencies.

An indirect object is an object that has an ID number so that other objects within the document can reference it. The cross-reference table is used for random access of indirect objects within the PDF document. The table begins with the `xref` keyword, followed by cross-reference subsections containing the byte offsets of indirect objects within the PDF

```
1 0 obj
     <<  /Type /Catalog
         /Pages 2 0 R
         /PageMode /UseOutlines
         /Outlines 3 0 R
     >>
endobj
```

Figure 2: An example catalog dictionary for a PDF document.

document. Each subsection starts with the first object in the subsection followed by the number of objects in the subsection. Each entry within the subsection begins with a ten digit byte offset, followed by a five digit generation number, and finally terminated by an `n` or `f` designating each object as in-use or free, respectively. An object that is in-use has content for the PDF viewer to render; a free object is used to add more content if necessary. The cross-reference table shown in Figure 3 contains three indirect objects numbered 0-2. The first entry is for object 0, which is free. Object 1 begins at offset 21,345 and is in-use. Object 2 has been reused, and has a generation number of 1.

```
xref
0 3
0000000000 65535 f
0000021345 00000 n
0000027891 00001 n
```

Figure 3: An example xref table.

Objects. The PDF standard supports several types of objects: dictionary objects, array objects, Boolean objects, strings, numbers, names, null objects, and streams.

Dictionary Object A dictionary object is composed of a set of keys and values.

Array Object An array object is a collection of objects.

Boolean Object A Boolean object represents a **true** or **false** value.

String Object A string object is a series of zero or more bytes. It can be a literal string enclosed by parentheses (), or a hexadecimal string enclosed by angle brackets <>. A string may be plain text, or encoded in hexadecimal or octal.

Number Object A number object can be an integer or real number.

Name Object A name object is a unique identifier used within the document.

Null Object The null object is unequal to all other objects. There is only one, and using it as the value of a dictionary entry is the same as excluding that entry from the dictionary.

Stream Object A stream object is a sequence of bytes of unlimited length. Images and page descriptions are represented as streams since they are generally large. Streams may be encoded using one of the supported filters as specified in the PDF standard.

Objects are numbered with an object ID, but may appear in any order in the PDF document regardless of ID number. The list of IDs need not index numbers sequentially; `65`, `23`, and `115` would be a perfectly acceptable list of IDs.

Forms. Static and interactive forms use collections of fields to collect information from the user. A PDF document may contain any number of fields appearing on any number of pages. Fields may contain default values or have values supplied by the user. These values may be updated or validated by JavaScript within the PDF document. There are many types of fields, including text boxes, radio buttons, check boxes, and combo boxes. Once a user has filled in a form, he or she may select the submit-form action, which will transmit the data to a URL. In this instance, the PDF document acts as a web client with limited functionality, increasing its potential impact once exploited.

JavaScript. JavaScript code may be used to dynamically update the contents of a PDF document, usually in regards to form processing. Adobe includes its own ECMA-compliant JavaScript engine in Adobe Acrobat and Adobe Reader to interpret JavaScript used within PDF documents. However, there are several differences between the Adobe JavaScript engine and standard JavaScript engines. Primarily, Adobe includes objects and methods for accessing and changing data within the PDF document that do not exist outside the context of the PDF reader.

ActionScript. PDF allows the use of embedded multimedia objects, such as Flash. A stream object can contain a compiled Flash file that the PDF reader will execute when the PDF document is opened. This provides another vector for attacks, as Adobe Flash is known to have several vulnerabilities (for example, [14, 15, 16]).

Embedded Files. Other files such as spreadsheets, executables, and other PDF documents may be embedded within a PDF document. In older versions of Adobe Reader and Acrobat, embedded files such as executables could be executed upon opening the document. Later versions of Adobe Reader and Acrobat issue a warning that requires user response if certain file types, such as `.exe`, are set execute on document launch. Some malicious PDF documents use social engineering techniques to convince the user to accept the warning and execute the malicious binary. In his blog, Didier Stevens has demonstrated how a user may be tricked into executing an embedded executable in a PDF document [17].

2.2 RELATED WORK

This section will discuss existing work on malicious document detection and open-source tools for analyzing PDF documents. Much of the existing work has focused on identification of malicious versus benign documents, rather than grouping and classification of similar malicious documents. A recent survey on malicious PDF documents [18] discusses many of the studies detailed in this section.

Current defensive practices for malware include intrusion detection and prevention. Intrusion detection detects the malware after the machine has been infected. Intrusion prevention identifies the malware before it infects the system, and attempts to prevent the malware from executing [19]. Anti-virus software is a kind of host-based intrusion detection and prevention system. The most difficult part of these practices is determining malicious files from benign files. Malicious documents often use the same formats as benign files and attempt to mimic benign behavior to avoid detection while also executing a malicious payload.

Much of the recent work on malicious documents has attempted to define methods for automatic identification of malicious documents for use in intrusion detection and antivirus software. There have been several attempts to identify the best method for quickly identifying malicious documents. In one study on malicious documents, Li et. al. found that in practice adversaries often insert malicious code into existing benign documents, and conclude that a hybrid static and dynamic analysis is the best solution for malicious documents [20]. Other studies have used various machine learning techniques to classify benign and malicious PDF documents, with some success [21, 22, 23, 24, 25, 26]. Several web-based tools, such as Wepawet [27] and VirusTotal [28], allow a user to upload a malicious document for the tool to classify it as malicious or benign. Wepawet uses varied approaches to provide an automated in-depth analysis of the file. VirusTotal leverages popular anti-virus software to create a score of maliciousness for a document. However, the newer the techniques used in the fil,e the less likely anti-virus software, and thus the composite VirusTotal, are to label it as malicious.

Kittilsen focused on detecting malicious PDF documents from a network component [29]. He leveraged a variety of tools to identify PDF documents in the network stream and put them on a hard drive for offline analysis. Once offline, a script extracts 18 string features and runs them through a Support Vector Machine (SVM) classifier. A SVM is a kernel method classifier that has high accuracy and is able to work with sets of high dimensionality. When run on a data set of both benign and malicious files, the true positive rate for this technique was 99.50%. Borg built on the work of Kittilsen and attempted real-time analysis of PDF documents on the network, but concluded that it was not possible using Kittilsen's techniques due to several faults, including difficulties in finding the end of the PDF document in a network stream [30].

Several studies on malicious PDF document identification focus on the JavaScript within a PDF document as a means of identification. Based on a sample set of 977,615 malicious PDF documents and 1,333,420 benign PDF documents, Vatamanu, Gavriluţ, and Benchea found that 93% of the malicious PDF documents contained JavaScript while only 5% of the benign documents had JavaScript [22]. Using this insight, they calculate the fingerprint hashes of extracted JavaScript to successfully cluster the PDF documents.

Tzermias et. al. created a tool called MDScan, to be used alongside anti-virus software for malicious document detection [26]. MDScan parses the PDF for JavaScript and runs the embedded code on a JavaScript interpreter (Mozilla SpiderMonkey [31]) that has been modified to include some of the more common Adobe JavaScript API calls. If at any point shellcode is found in the interpreter, the document is labeled as malicious. MDScan successfully detected 89% of a malicious dataset of 197 files. Limitations of the tool include parsing errors, an incomplete Adobe API in the emulator, and limited exploit checking.

One of the biggest hurdles in feature selection for PDF classification is the automated de-obfuscation of the JavaScript contained within a PDF. The Adobe Reader JavaScript engine is based off of Mozilla's SpiderMonkey JavaScript engine [31], but has been modified to contain an API unique to Adobe products. As a result, malicious code writers use native API functions to obscure the exploit JavaScript code within the PDF, serving as a sort of sandbox-detection since the malicious document will not deliver the payload without the native Adobe functions. However, Lu et. al. were able to de-obfuscate and detect 98% of malicious PDF documents in a small sample set (207 malicious documents) with no false positives using their tool Malicious PDF Scanner (MPScan) [32]. MPScan hooks the Adobe Reader native JavaScript engine to extract the JavaScript source code and opcode from a PDF. They have greater success in de-obfuscating the JavaScript using the native Adobe JavaScript API. MPScan also includes shellcode and heap spray detection based on the length and entropy of strings found within the JavaScript. A heap spray occurs by storing a JavaScript string with thousands of copies of shellcode, or malicious instructions, to fill the heap and achieve execution of the shellcode [10]. If MPScan detects shellcode or heap spray in the document, it is labeled as malicious.

Šrndić and Laskov [33] took a different route in malicious PDF document detection and discovered that benign PDF files tend to be much more complex structurally than malicious PDF documents. While malicious PDF documents typically only contain the exploit code, benign PDF documents are composed of a complex hierarchy of objects and object references. Through machine learning, they were able to identify specific features, such as number of pages in the document, that distinguish benign from malicious PDF documents. Their tool was able to successfully identify 99% of malicious PDF documents in a 130,000 file mixed sample set. However, Maiorca et. al. show that this detection method is vulnerable to active attacks: a malicious PDF writer can easily mimic the features of a benign PDF document or embed a malicious PDF document inside of a benign one [34]. The embedded PDF document will still execute the exploit but the benign PDF document will pass the static detection.

Donaldson analyzed the PDF document structure to identify benign PDF documents created by similar software [35]. He used ordered lists of object types as structural signatures of benign PDF documents. In this technique, the document signature is dependent on the type and order of the objects in the PDF documents. He parsed the document for each object and object type, and listed the object types in the order they appear in the document. The object type list is the document signature. He used regular expressions and N-gram analysis on each document signature to identify documents from popular PDF creation tools, such as Adobe Acrobat and Microsoft Office, with a high level of success.

Many of the papers detailed in this section list tools used for malicious document analysis. The following section details open-source tools in PDF analysis that are not related to academic publications.

2.3 ANALYZING, CREATING, AND PARSING PDF DOCUMENTS

Several tools have been created for PDF document creation, parsing, and analysis. The following will highlight a few such tools, and will discuss their inefficiencies with regards to analyzing malicious PDF documents.

pdfid. Created by Didier Stevens, *pdfid* scans a PDF document for the presence of suspicious keywords and produces a report that can be used to triage a PDF document and determine if it needs further analysis [36]. A simple parser, *pdfid* performs a string search on the PDF document for tags whose presence could be suspicious, such as tags indicating JavaScript or Flash. The parser includes mitigation for specific obfuscation techniques in the tag names, such as HTML encoding or extra whitespace [37].

pdf-parser. A simple tool for PDF document analysis, *pdf-parser* parses PDF documents without rendering them and includes several options for analyzing PDF documents [36]. This tool will extract objects and streams from the PDF document, although not all PDF filters are currently supported. The parser allows limited string search, but overall the tool is not robust enough for in-depth malicious PDF document analysis.

PDFMiner. *PDFMiner* is a library for parsing PDF documents [38]. It comes with two tools: pdf2txt for text extraction and dumppdf to parse the PDF document into xml. *PDFMiner* has support for more filters than *pdf-parser*, but it is still lacking some newer ones, such as DCTDecode. Also, *PDFMiner* cannot successfully parse malformed PDF documents, which is very common for malicious PDF documents.

peepdf. Malware researcher Jose Miguel Esparza created *peepdf*, a malicious PDF document analysis tool that allows for the various character encodings supported by PDF documents, as well as various filters [39, 40]. Equipped with varied functionality, *peepdf* will identify potential suspicious objects within the PDF document, attempt to decrypt encrypted data, print the object tree of the PDF document, and allow creation and editing of PDF documents. It also attempts to extract and dynamically analyze any JavaScript within the PDF document. If the JavaScript successfully de-obfuscates, it performs emulation analysis on any bytecode extracted. Initially, *peepdf* used a modified version of SpiderMonkey [31] but later versions use pyv8, a Python wrapper for Google's v8 JavaScript engine [41, 42]. In both instances the JavaScript `eval` function was changed to print the code argument rather than evaluating it, but neither engine was modified to include objects or methods from the Adobe JavaScript API, leading to a low success rate in de-obfuscation.

While this thesis will not address machine learning techniques used with PDF documents or network-based detection, we use other points from previous work. Vatamanu, Gavriluţ, and Benchea discovered the high prevalence of JavaScript in malicious PDF documents, so JavaScript is an important part of feature analysis [22]. Automated de-obfuscation of malicious JavaScript is important to finding common exploits within the PDF documents. However, the emulated JavaScript engine technique used by Tzermias et. al. in MDScan appears to be more feasible than reverse engineering Adobe Reader like in Lu et. al. [26, 32]. Finally, the work of Šrndić and Laskov, and Donaldson, on malicious PDF structure will play an important role in the feature set described in this thesis [33, 35]. The following chapter describes the feature set and the methods we used to analyze our data set of malicious PDF documents.

3.0 MALICIOUS PDF DOCUMENT ANALYSIS

For this thesis we have identified features of malicious PDF documents for use in automated extraction and analysis of a large collection of malicious PDF documents. This section will discuss the methods used in our experiment, along with detailing our feature set. We present the results in Section 3.3 and discuss implications in Section 3.4.

3.1 METHOD

The data set used for this thesis includes 518,509 PDF documents collected from various public and private sources before and during the second quarter of the 2013 calendar year. These documents are all believed to be malicious, although some benign files that cause strange errors may remain in the data set. This thesis does not include a control to ensure that the PDF documents are malicious; it classifies families of PDF malware rather than malicious versus benign PDF documents. Using the techniques described in Section 3.2.3, the scripts and tools successfully extracted features from 517,682 of the PDF documents in the data set. The following section describes the tools and techniques used to analyze the data.

We create initial bins of the documents using the object graphs, link graphs using a graph similarity algorithm, and then use other features, such as JavaScript, for further document analysis. The object graphs of the PDF documents provide an initial step for binning them into similar groups. Comparing MD5 hashes of the object graph edge list finds PDF documents with exactly-matching object graphs. Hashing the edge lists may appear too strict a matching criteria due to the sensitivity to small changes, yet the success rate demonstrates this method is justified.

After an initial binning by MD5 hash, graph similarity algorithms provide another method of graph comparison. Graph comparison using MD5 hashes of the edge lists is sensitive to slight changes in the graph structure, such as changing an object number. In the case where an object number is changed, the graph structure is the same, but the hash would be different. A graph similarity tool compares the graphs and finds documents in the data set based on graph structure rather than object number in the PDF document. Graph similarity calculations use in-house software to compute scores for the object graphs.

The similarity algorithm computes the degree sequences of two graphs and calculates Pearson's coefficient to determine the similarity in sequences. The degree sequence of a graph is the ordered sequence of vertex degrees of the graph [43]. If the degree sequences of two graphs are different lengths, the smaller graph is zero-padded to match the larger graph. Pearson's coefficient provides a similarity score between 0 and 1, with scores close to 1 indicating high similarity. This number represents the linear relationship between two variables, in this case the degree sequences of the graphs [44]. A Pearson score is more meaningful than other similarity scores because it is bounded on both sides by 0 and 1, so each score can be computed as a percentage of similarity. However, a Pearson's coefficient may result in a similarity score of 1 for two graphs that are not identical. If the degree sequences of both graphs are not identical, but change at the same rate, the algorithm will result in a correlation of 1. Yet these graphs are still highly similar due to the correlation of the degrees of their vertices, so we include them in our results. Comparing all of the graphs in the data set results in the identification of similar or identical graphs that do not use the same object numbers and reduces any limitations caused by the naïve string hashing used to create the graph hashes, but is computationally more expensive.

There are many JavaScript samples extracted from the data set, both obfuscated and de-obfuscated, that are not exactly the same but share code. Many PDF documents use similar obfuscation techniques for different exploits, and de-obfuscated JavaScript can share the same exploit code but differ by a few bytes in the shellcode payload, usually the URL to which it calls out. Exact string matching will not identify these similarities, but other methods can. Sdhash [45] is a tool that calculates a fuzzy hash of a binary and creates a similarity score for two files by comparing them based on common binary strings. It will create a score from 0-100, where 0 are two completely dissimilar, possibly random, files. Two identical files would produce a score of 100, but a score of 100 does not mean that the files are identical, just that they are strongly similar. Scores below 21 are said to have no correlation, so this analysis chooses a threshold of 25 to record scores at all.

To compare a set of files using sdhash, first an sdhash is computed for each file by calculating entropy scores to find the statistically improbable features of the file and creating a fingerprint based on the features. Then the tool compares two sdhashes to compute the similarity score for two files. These scores can identify files that are highly similar.

There are several drawbacks to the sdhash tool. Foremost, while sdhash works well for larger file sizes, it requires a minimum of 512 bytes to compute its digest; any input smaller than 512 bytes is ignored. Also, common JavaScript obfuscation techniques, such as long strings of randomized characters in comments and randomized variable names, may defeat the similarity score calculation. Still, sdhash triages malicious binaries and has well-known failure conditions. Therefore, we used sdhash to compare the JavaScript in the data set.

The final set of features we use to cluster malicious PDF documents are related to URLs. Exploit kits that use PDF documents to gain access to a victim's computer download further malware from the exploit kit server. Many exploit kits use URL parameters to record the successful exploit that was executed to reach the server. The domain, file name, and URL parameters of each of these URLs often follow a pattern unique to an exploit kit. Experts have identified URL patterns for some of the more popular exploit kits; an analyst can compare these patterns to classify URLs from a set of malicious PDF documents.

We relied on exploratory analysis to identify the features in our feature set. The analysis revealed interesting qualities about sets of documents in the data set. Next we discuss one of these qualitities, in which the obfuscated JavaScript is broken and does not successfully de-obfuscate.

Preliminary analysis showed that there are 61,990 total PDF documents that match JavaScript to the character, but none of them successfully de-obfuscate using the methods discussed in this section. There appears to be an error in the obfuscated JavaScript that does not allow it to de-obfuscate successfully. Manual analysis found the error and also discovered that there are two layers of obfuscated JavaScript that must run successfully before a third layer of JavaScript that contains the exploit is revealed.

The first layer of obfuscated JavaScript in these samples uses the `getAnnots()` function from the Adobe JavaScript API to collect data from two Annotation objects located within the PDF document. It splits the data from one of the Annotation objects on the '-' character, creating an array of hexadecimal digits that it converts into a string of JavaScript using the `String.fromCharCode()` method. However, the data from the Annotation object is not delimited by '-'. Instead, the delimiter varies from several combinations of digits, such as 'z', 'xyz', or 'mz'. We write a script to modify this code and to retrieve the de-obfuscated JavaScript. The exact delimiter for the data in each PDF would be difficult to detect automatically, but we mitigate the error in the obfuscated JavaScript by changing the split statement in the code to delimit by non-hexadecimal digits instead of '-'.

When the corrected code runs, a new layer of obfuscated JavaScript is revealed. This layer decrypts an encrypted string and evaluates the result. The code contains a check for sandboxes: an if statement checks for the `app` object and then retrieves the data from the subject of the other Annotation object. The JavaScript code then decrypts the retrieved data, revealing the final layer of JavaScript.

Finally, the third layer contains the exploit code. Due to the volume of incorrect PDF documents, it is possible that these PDF documents were generated by an exploit kit generator that contained an error in code obfuscation that was not detected until the exploit kit was used in production. But we do not confirm this conjecture in this thesis.

We use these mitigation techniques in conjunction with a modified JavaScript engine to de-obfuscate PDF documents that have the incorrect obfuscated JavaScript.

3.2 FEATURES

Through manual analysis of PDF documents, we identify several features to analyze a large set of PDF documents. Table 1 lists the features, which include: a graph of the PDF document object structure, a cryptographic hash of the graph, obfuscated and de-obfuscated

JavaScript, any shellcode found within the JavaScript, URLs, image data, and Adobe Flash. By identifying trends among malicious PDF documents, similar exploits can be identified without manual analysis of each document. The following chapter will identify and explain the features chosen for our sample set, justify the choices, and provide recommendations for automated extraction of features.

Table 1: The feature set used in analysis.

Feature	**Definition**
Object Graph	A graph of the object structure of the document.
Graph Hash	A MD5 hash of the object graph.
JavaScript (obfuscated)	The first layer of JavaScript found directly in the document.
JavaScript (de-obfuscated)	The second layer of JavaScript revealed after the obfuscated JavaScript is run.
Shellcode	A payload encoded in hexadecimal or Unicode found in the JavaScript and image exploits.
URLs	Found in the shellcode. Have subfeatures such as domains, IP addresses, and URI parameters.
Images	Image exploits found within the document.
Flash	Flash exploits found within the document.

3.2.1 Feature Set

The feature set is composed of parts of PDF documents used to uniquely identify and analyze a PDF document. Table 1 lists the feature set, which includes the object structure of the PDF document, JavaScript, shellcode, URLs, images, and Flash. This section discusses each of these features in greater detail.

In their PDF classifier, Šrndić and Laskov discovered the importance of PDF structure in classification of malicious PDF documents [33]. One feature they use is a graphical representation of the object structure of the PDF document. The native object hierarchy in a PDF document is a tree structure, composed of a root with each PDF object a child of the root. Due to indirect object references, edges form between the object nodes, creating a cyclic graph. Figure 4 demonstrates how we stored the graph as a list of edges. In fact, computing a cryptographic hash of the edge list for each PDF document can create a fast initial grouping of the PDF documents with identical object graphs.

```
0 1
0 2
0 3
1 3
```

Figure 4: A PDF graph consisting of three objects, with an indirect object reference from object 1 to object 3.

JavaScript is the most commonly used vector for PDF exploits, thus the JavaScript code itself is composed of several important features. Normally, a malicious PDF has one or more layers of obfuscation which must be removed to reveal the exploit code and payload. In order to avoid automated detection, the initial JavaScript in the PDF document is obfuscated to hide the code that is actually executed. When the JavaScript is executed, it creates the JavaScript exploit that will be run and executes it, usually by invoking the native `eval` function. However, businesses use obfuscation to hide proprietary code so obfuscation within a PDF does not always imply maliciousness [46]. Malware authors use multiple JavaScript obfuscation techniques in concert, which makes automated malicious code analysis a formidable task, since it hides the true nature of the malicious code.

We now survey the obfuscation techniques observed in our data set. Some obfuscation techniques used by malicious JavaScript code embedded in PDF documents are used by all malicious code. For example, a naïve but successful evasion of static and signature-based detection is to rename all variable and function names to nonsensical values that provide no information as to the purpose of the code. A malware author can also avoid signature-based detection by changing one character of each variable or function to create a slightly different file that executes exactly the same. When the adversary changes variable names, it mostly hinders manual static analysis, although simple anti-virus software may also be avoided this way. This technique only changes local variable names and does not change the names of native functions that anti-virus software could identify, such as `eval` and `unescape` because these functions need to be called using the name recognized by the JavaScript engine. Both of these functions are common in malicious or obfuscated code observed in our data set: `eval` evaluates an expression or several statements of JavaScript given as a string argument, and `unescape` is used to decode a string that contains hexadecimal escape sequences (such as in URI encoding). Occasionally, multiple exploits will use the same obfuscated variable name to mean the same thing. For example, malware authors may use the anagram “yarsp” to refer a variable dealing with the “spray” of a heap spray. Arbitrary comments throughout the code can help defeat string-based signatures and searching, as well as hinder manual static analysis without changing the functionality of the code. Often HTML encoding [47] or octal encoding [48] is used to obfuscate the JavaScript code within the PDF document. The PDF reader will decode the HTML, but anti-virus software or other JavaScript engines may not be set up to automatically decode the HTML. String operations such as concatenation, replacement, substring, splitting, etc., can be used to hide key terms (such as `eval` and `unescape`) within the code. For example, malicious code attempting to obfuscate its use of the `eval` function may look like the code in Figure 5. The string “&#97;” is HTML encoding for the character ‘a’.

```
var l = "l";
var e = "e@v".replace("@", "") +"&#97;" + l;
e(payload);
```

Figure 5: Malicious JavaScript obfuscates use of the `eval` function.

Frequently, the code to be executed is created character-by-character by accessing indices of an arbitrary string in the obfuscated code. There are various ways to arrange the indices to be accessed: they can be hard-coded, located in another array, or placed in a string that is then split to create an array of indices. These techniques hide the true purpose of the code from both anti-virus software and malware analysts.

The majority of the methods discussed thus far involve creating a string of code that is sent to the `eval` function. Thus, a great way to retrieve de-obfuscated code is to override the native `eval` function to print the code instead of evaluating it. In response to this analysis technique, malware developers have begun to use `eval` on code that is necessary to de-obfuscate the code that is executed. In this scenario, the initial `eval` call must execute correctly in order to create the de-obfuscated exploit string to be evaluated.

Malicious obfuscated JavaScript is not only used to avoid static and signature-based detection, but also dynamic detection: by calling on functions and objects native to the Adobe JavaScript engine, the obfuscated JavaScript can defeat analysis by JavaScript tools that do not use the Adobe engine. For example, malicious code developers can hide values or code in other objects within the PDF document and access them using the Adobe API. If the JavaScript is run in a non-Adobe environment, it will fail, preventing analysis. Values and code can also be hidden in the `rawValue` attribute of PDF fields used in forms, and can be retrieved using JavaScript. For example, parts of the code could be hidden in an Annotation object within the PDF document and could be retrieved using the `getAnnots()` function [49]. Also, in the Adobe JavaScript environment, if a function is called on `null` or `undefined`, it returns the native `Doc` object, which contains the `eval` function. Adversaries use this idiosyncrasy to avoid JavaScript sandboxes and obfuscate the call to `eval`. In the code in Figure 6, the function `String.prototype.slice()` is called on an empty string, so `r` becomes the `Doc` object, from which `eval` can be called. In a non-Adobe JavaScript engine, this code will fail.

The obfuscation techniques used in a PDF document generated by an exploit kit could hint at which version of the exploit kit created the PDF document, since obfuscation techniques used by exploit kits are updated more frequently than the exploit code or the PDF template.

The de-obfuscated JavaScript will contain the code that exploits the PDF reader in an

```
q = "slice";
b = "ghbf";
z = b[q];
r = z();
e = r["e" + "val"];
```

Figure 6: `String.protoype.slice()` is called on an undefined object, which becomes the `Doc` object. `eval()` can be called from the `Doc` object.

attempt to gain arbitrary code execution. Arbitrary code execution is generally accomplished through the use of a heap spray or through the exploitation of a vulnerability in the Adobe JavaScript engine.

The de-obfuscated JavaScript contains shellcode to be executed once the exploit has run. The shellcode can tell a malware analyst what the adversary was trying to achieve with the malicious PDF document. Usually, a malicious PDF document is used to download further malicious code, so the shellcode will contain an URL from which to download the code. We collect both the shellcode and the URLs for further analysis.

The most common PDF exploit that does not need to involve JavaScript is CVE-2010-0188, the TIFF file buffer overflow [50]. The exploit includes overflowing a TIFF image to achieve arbitrary code execution. Like the JavaScript, the shellcode payload and any URLs it contains can be extracted from the image stream in the document.

Adobe Flash within a PDF document is vulnerable to malicious exploits as well, although Flash exploits are not as common as the JavaScript exploits in PDF documents in our data set. Adobe Flash is written in ActionScript and then compiled for execution before being embedded into a PDF document. The compiled Flash can be extracted from the PDF document and the ActionScript, or a dump of the bytecode, can be retrieved. Sometimes the ActionScript will also contain a shellcode payload that can be extracted and analyzed separately.

3.2.2 Feature Justification

This section will provide justification for each of the features discussed in the previous section. The object graph and graph hash were discovered through exploratory analysis, while other features, such as JavaScript and Flash, are justified because they have been observed to deliver PDF exploits.

During exploratory analysis, many of the PDF documents had exactly the same PDF graph, including identical object numbers. Deeper analysis of two PDF documents with identical graphs showed that they both contained JavaScript within the same object of the PDF. The rote JavaScript was very different, but the de-obfuscated JavaScript was nearly identical. It only differed a few bytes in the bytecode payload which was a different URL to access exploits. Since initial manual comparison of the PDF graphs led to the identification of similar PDF documents, we included the feature in automated analysis.

Exploit kits cause the similarity in PDF structures with identical payload, because exploit kits mass-generate PDF documents as an initial attack vector. These PDF documents either contain the malicious code themselves or download it from a server once the exploit has successfully run. The exploit-kit generators use a PDF template and then fill in the exploit code and payload accordingly, leading to thousands of PDF documents with the same object structure and same origin.

Through manual exploratory analysis, we confirmed these similarities through different PDF documents created by the same exploit kit. Table 2 displays results derived from samples of malicious PDF documents identified by Contagio as the Phoenix Exploit Kit 2.0 [51] and Blackhole Exploit Kit 2.0 [52], comparing the object hierarchies of the known PDF documents. Both documents from the Blackhole Exploit Kit share the same graph hash. Closer inspection of the PDF documents reveals that they share similar obfuscated and de-obfuscated JavaScript as well. Likewise, six out of seven of the Phoenix Exploit Kit PDF documents share the same graph hash. These six PDF documents all contain JavaScript exploits that have identical obfuscated JavaScript and similar de-obfuscated JavaScript, although they use different exploits. The seventh Phoenix Exploit Kit PDF document uses an image exploit that does not require JavaScript, so it does not require the obfuscation techniques used within the other PDF documents. This difference explains why its graph hash differs from the other Phoenix documents.

Table 2: PDF documents from known exploit kits and their tree hashes.

PDF	**Graph MD5**
phoenix_2.0_newplayer.pdf	06055c4d82813cce7aaad42d283b3181
phoenix_2.0_allv7.pdf	06055c4d82813cce7aaad42d283b3181
phoenix_2.0_printf.pdf	06055c4d82813cce7aaad42d283b3181
phoenix_2.0_geticon.pdf	06055c4d82813cce7aaad42d283b3181
phoenix_2.0_all.pdf	06055c4d82813cce7aaad42d283b3181
phoenix_2.0_collab.pdf	06055c4d82813cce7aaad42d283b3181
blackhole_2.0_1.pdf	8def19035c1e5652c02a8199786280ed
blackhole_2.0_2.pdf	8def19035c1e5652c02a8199786280ed
phoenix_2.0_libtiff.pdf	fe4102d6db98dd3e13e261ff55212e35

Since Adobe employs its own JavaScript engine with new vulnerabilities, many malicious PDF documents target the Adobe JavaScript engine. Vatamanu, Gavriluţ, and Benchea found that 93% of malicious PDF documents contain JavaScript [22]. The JavaScript found within malicious PDF documents is usually obfuscated, to hide the exploit that is being used. The obfuscation techniques used in the JavaScript could provide details about the origins of the PDF documents. Obfuscation techniques may be specific to a particular exploit kit or version of an exploit kit. Due to time constraints, we were unable to pursue this thought further.

Likewise, the de-obfuscated JavaScript provides details about a PDF document. Some exploit kits use the same JavaScript for each PDF document, but obfuscate the JavaScript in each PDF document to make the documents appear different. In this scenario, the de-obfuscated JavaScript will be the same for each PDF document created by the exploit kit, and provides a way to identify PDF documents created by the exploit kit.

The JavaScript also contains the initial malicious, post-exploit payload in the form of shellcode, usually encoded in hexadecimal or Unicode. This can provide information about the end goal of the malicious PDF document. For most exploit kits, the goal of a PDF document is as the initial infection vector which will subsequently download further malware. The shellcode will contain the URL from which to download further malware. The domains used in the URLs or the URI parameters may be specific to an exploit kit, and can be used to identify malicious campaigns.

While less popular than JavaScript exploits, images and Adobe Flash are also vectors of compromise for PDF documents. Their exploits, shellcode, and URLs tell similar stories about their sources.

3.2.3 Feature Extraction

In order to begin analysis of the features detailed in Section 3.2.1, the malware analyst must extract the features from the PDF documents. Due to the large size of malicious PDF document collections, an analyst requires automatic extraction of the features. This section details some feature extraction methods for malicious PDF documents. Feature extraction begins with parsing the PDF document, a process for which several open-source tools are readily available. Modifications to other open-source tools that provide stubs for some Adobe API calls, combined with a JavaScript engine, automatically de-obfuscates JavaScript for a large number of malicious PDF documents. Finally, Flash files have a specific header that makes them easily identifiable for removal. We created a tool that we will refer to as Dugmare which combines the open source tools and custom scripts described in this section to extract the features from PDF documents.

There are several open-source tools for parsing a PDF document. As mentioned in Section 2.3, *PDFMiner* [38] comes packaged with dumppdf.py, a tool that parses a PDF document and outputs it as xml. Many programming languages already have libraries that will parse xml, such as lxml in Python [53], making the xml a more feasible option for parsing. As previously mentioned, the *PDFMiner* tool fails on malformed PDF documents;

a PDF document that is missing the end of file tag `%%EOF` will not be successfully parsed. We modified *PDFMiner* to ignore the errors caused by malformed PDF documents so that they can be parsed. From the xml created by *PDFMiner*, Dugmare parses the object structure of the PDF document and constructs the edge list of the graph of the PDF document. Dugmare compares the PDF graphs by calculating an MD5 hash on the edge list. During parsing, Dugmare also searches the PDF document for JavaScript, images, or Flash content.

In its interactive console, *peepdf* has a `js_code` command that will extract any JavaScript from a specified object in a PDF document [39, 40]. *Peepdf* tokenizes the contents of the objects and produces a weighted score as to whether it is JavaScript or not. If the weighted score is above a threshold, *peepdf* extracts the JavaScript. We reproduced this code and adjust the weights to increase or decrease sensitivity for malicious JavaScript extraction. This thesis required the data to contain at least five distinct strings and have at least 15 tokens in the set {`'var '`, `';'`, `')'`, `'('`, `'function '`, `'='`, `'{'`, `'}'`, `'if '`, `'else'`, `'return'`, `'while'`, `'for '`, `','`, `'eval'`, `'unescape'`, `'.replace'`}. The only difference between these values and those used in *peepdf* are the addition of `'unescape'` and `'.replace'` to the token set.

After extraction, the analyst must de-obfuscate the JavaScript to determine what it does, including the exploit being used. Manual de-obfuscation methods involve finding the string that is sent to the `eval` function and printing it rather than letting it evaluate. For PDF documents written for Adobe products, the JavaScript engine must emulate Adobe API functions and values. The analyst must manually find any missing values and add them to the code and make accommodations for API functions or remove their calls from the code. Automation of this analysis, including iteratively changing the JavaScript code to get the correct output, is much trickier.

Malicious PDF document tool *peepdf* contains a `js_analyse` module that attempts complete de-obfuscation of JavaScript that is extracted from the PDF document [39]. The tool uses the pyv8 wrapper for the v8 JavaScript engine [41] which tracks the values of variables and local functions in the *context* of the current execution. The `eval` function is modified within the context to store any values rather than evaluating them. However *peepdf* does not implement Adobe API functions. Therefore *peepdf* has a low success rate for obfuscation techniques that use the API.

The de-obfuscation attempts by *peepdf* provide a good start for creating a more successful PDF JavaScript de-obfuscation tool. Many initial errors during the *peepdf* de-obfuscation on the data set are caused by HTML/xml tags within the JavaScript code. Before the code is executed, Dugmare uses regular expressions to comment out any lines beginning with <. After that, the majority of the errors were caused by calls to native Adobe objects (such as `app` and `event`), which do not exist in the v8 engine. The native Adobe objects are very large and their scope is dependent on the context from which they are called within the PDF document. Due to time constraints, the tool includes emulations for only a few of the functions and some partial objects. The JavaScript for Acrobat API Reference contains more details about native PDF objects, such as their attributes and operation [54].

To add objects from the Adobe API to the pyv8 context, Dugmare uses *PDFMiner* to transform the PDF document into parseable xml and extracts common PDF object values such as subject, author, creator, etc. to add to the JavaScript context. Attributes of the `app`, `this`, and `info` objects are easily found within a PDF document. The lxml module in Python [53] can search the xml string created by *PDFMiner* for the attribute name and retrieve its value in the document. In some cases, the value references another object. The lxml module searches the xml for the referenced object, and retrieves the data of that object.

Obfuscated PDF JavaScript calls several Adobe API functions to prevent the JavaScript from being run in a non-Adobe environment. Dugmare adds stubs or modified versions of these functions to the v8 context to allow code execution. Dugmare only includes functions called by the obfuscated JavaScript; it should not include stubs for other native functions exploited by the de-obfuscated JavaScript to prevent the possibility of an infection from the malicious JavaScript.

Dugmare sets two of the functions used by obfuscated JavaScript, the `app.eval()` and `this.eval()` functions, to the `eval()` function within the context of the current execution. To ensure that it is being run in an Adobe environment, some malicious JavaScript checks if Adobe API functions are defined within the JavaScript engine. The JavaScript converts the function to a string and checks each of the first three characters to make sure they match ‘f’, ‘u’, ‘n’, the beginning of ‘function’. This check ensures that the function exists.

To overcome the check, the tool initializes functions that are checked this way but difficult to emulate, such as `app.getString()` and `app.newDoc()`, as empty functions within the JavaScript context.

Malicious JavaScript sometimes retrieves values stored in Annotation objects within a PDF document. Annotation objects are the "sticky notes" used to add comments to a file. Two functions are necessary to retrieve the objects: `app.doc.syncAnnotScan()` scans the PDF for Annotation objects and builds an array of them, and `app.doc.getAnnots()` retrieves all Annotations within the array that meet a specified criteria. Dugmare collects Annotation objects while the PDF document is parsed for API object attributes, so the `app.doc.syncAnnotScan()` function is added as an empty function. Dugmare emulates the second function, `app.doc.getAnnots()`, to retrieve the Annotations from within the context.

The Adobe JavaScript engine has several inconsistencies with other JavaScript engines, however Dugmare mitigates the inconsistencies by catching exceptions and changing the pyv8 context to increase successful de-obfuscations. For example, occasionally PDF JavaScript uses the dollar sign character, `$` without being initialized. In this scenario, `$ = this`, but this association is not present in the v8 engine. If a ReferenceError is thrown by pyv8 and the undefined variable is `$`, Dugmare sets `$` equal to `this` within the v8 context.

As a result of the variability of PDF document content, sometimes the JavaScript code contains multiline values within HTML/xml tags that a simple regular expression will not extract. If a ReferenceError or a SyntaxError is thrown and it is not due to a different inconsistency, the error message will contain the offending line number and the Dugmare will comment out the line.

As stated in the prior section on obfuscation techniques, if a native Adobe function is called on null or undefined, it returns the native `doc` object. This idiosyncrasy is often used to call the `eval()` function, as seen in our data set. The undefined function call is not usually straightforward to find programmatically, but when a "TypeError: function called on null or undefined" is thrown, Dugmare uses regular expressions to replace the undefined function with the `doc` or `app` object within the code.

Finally, occasionally a "TypeError: undefined is not a function" error is thrown. In this scenario, Dugmare uses regular expressions to substitute `eval()` for the undefined function in the code.

These mitigation techniques address some of the obfuscation techniques found within the data set. These techniques are not all-encompassing, but greatly increase the effectiveness of our PDF JavaScript de-obfuscation tool. We tested these techniques on a non-random sample of 500 documents from our data set to determine their effectiveness. When run the sample, *peepdf* achieved an 8% de-obfuscation rate. After our improvements to the JavaScript engine, we obtained a de-obfuscation rate of over 50% on the same set of 500 documents.

Once Dugmare extracts the de-obfuscated JavaScript, it retrieves the payload from the JavaScript. The payload is typically a hexadecimal or Unicode string. Dugmare uses regular expressions bounded by single quotation marks, double quotation marks, or the character sequence '\&' to extract these strings for further analysis. Once extracted, Dugmare converts the Unicode strings to hexadecimal to maintain a standard encoding for storage. A regular expression finds URLs in a shellcode by searching for the hexadecimal encoding of "http://" and pulling out the following characters. These URLs are for further analysis of the PDF documents.

PDF images deliver an exploit by filling the data section with specially crafted values that overflow the image boundaries and allow for arbitrary code execution. Data streams contain PDF images, designated by an image tag with the image data as the value. Dugmare searches the PDF document for the image tags and retrieves the values in between them to extract image data for further analysis. The image tags analyzed for this work were TiffImage, Image, and xapGimg. Dugmare decodes the image data from base64 to ascii or Unicode and then searches the result for a URL.

Our initial attempts at automated image extraction were not highly successful. Further inspection found that many malicious PDF documents obfuscate the image tags by encoding part of the tag in HTML. For example, a `<TiffImage>` tag might instead read `&#60;Ti&#102;fImag&#101;&#62;`. The JSAnalysis module of *peepdf* contains a function called `unescapeHTMLentities` which checks for HTML entities within a string and converts them to plain text or Unicode [40]. When Dugmare uses `unescapeHTMLentities` on the stream data, the number of PDF documents with extracted image data more than doubles from 1,431 extracted images to 4,725 images.

Automated identification and extraction of Adobe Flash is easier than JavaScript extraction. Compiled Adobe Flash contains one of two prefixes to the compiled code. Compiled Flash begins with three characters "FWS" followed by bytes of compiled code. If the Flash is compressed, the compressed code begins with "CWS." The tool checks the first three characters of decoded streams for the either Flash prefix to identify Adobe Flash within a PDF document, then extracts the entire stream. The Python tool *swf_mastah* uses the parsing power of *peepdf* to search for the Flash headers and extract the `swf` file [55]. From there *swftools* [56] creates a dump of the Flash bytecode, or *furnace-avm2* decompiles the `swf` to ActionScript 3 [57]. The dump or decompilation results provide the Flash file in human-readable form that facilitates analysis.

The features discussed in this chapter are used to group malicious PDF documents without manual analysis of each document. However the features are not readily extractable. Our tool achieves automated extraction of features through the use of open source tools and simple scripts that parse the malicious PDF documents and extract the features. The following sections discuss analysis of these features to group malicious PDF documents and the results of analysis on the data set.

3.3 RESULTS

This section will present the results of automatically extracting the feature set and using the features to group a data set of 518,509 malicious PDF documents. Our tool called Dugmare, composed of a series of Python scripts and modified open source tools, parsed the PDF documents and extracted the graph, JavaScript, ActionScript, and other features. From there, we used other scripts and open-source tools on the extracted data, for example to compute graph similarities or decompile the ActionScript. The results of each feature are listed below.

Our tool successfully produced object graphs for 505,854 of the PDF documents, or

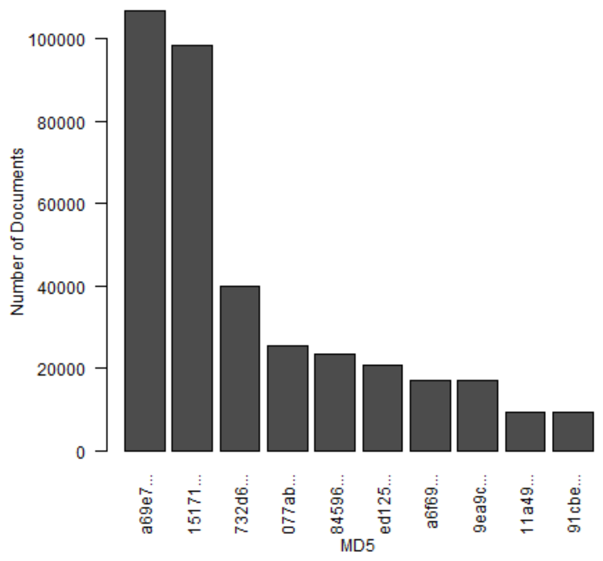
97.6% of the data set. Parsing errors in the modified *PDFMiner* affected the production of xml for some of the PDF documents. Without the xml, the tool cannot parse for the object structure of the PDF document. The object graphs produced 10,141 unique graph hashes. Figure 7 displays the distribution of PDF documents for the top 10 graph hashes.

Initial analysis found 488,354 distinct JavaScript scripts, both obfuscated and automatically de-obfuscated using the methods described in Section 3.2.3. The tool extracted obfuscated JavaScript from 470,910 malicious PDFs in the sample set, which is 90.8% of the entire sample set. This number describes the PDF documents that contain at least one layer of JavaScript, and includes PDF documents that contain identical JavaScript. Cursory glances at the documents from which obfuscated JavaScript was not extracted suggest that not all of the JavaScript in the data set was successfully extracted, possibly due to obfuscation or incorrect sensitivity in the identifier used. Several PDF documents that did not have JavaScript extracted did contain JavaScript, but the JavaScript was short and did not include enough tokens to alert the JavaScript identifier in the tool. Of the JavaScript that was extracted, there were 342,070 unique samples from 470,910 PDF documents, with 139,391 of the PDF documents containing JavaScript that matched at least one other PDF document.

Dugmare initially de-obfuscated the extracted obfuscated JavaScript from 270,141 of the PDF documents with extracted JavaScript, which is an initial 57.37% success rate. There are only 146,282 unique de-obfuscated scripts, and 123,859 malicious PDF documents share de-obfuscated JavaScript as well. At the end or our analysis, we were able to improve the de-obfuscation success rate to 62.9%, or 296,425 PDF documents and 178,385 unique scripts. Overall, there were 145,333 PDF documents (49%) that shared de-obfuscated JavaScript with at least one other PDF document.

The top two graph hashes from Table 7 had high success rates for de-obfuscation, 91.6% and 72.2% respectively, while the third graph hash had no initial successful de-obfuscations. Closer inspection of the 40,120 PDF documents matching the third graph hash found that all of them have the exact same obfuscated JavaScript, but none de-obfuscated successfully. These PDF documents contain the 'broken' obfuscated JavaScript described in Section 3.1.

Figure 7: The top 10 graph MD5s based on population within the data set.



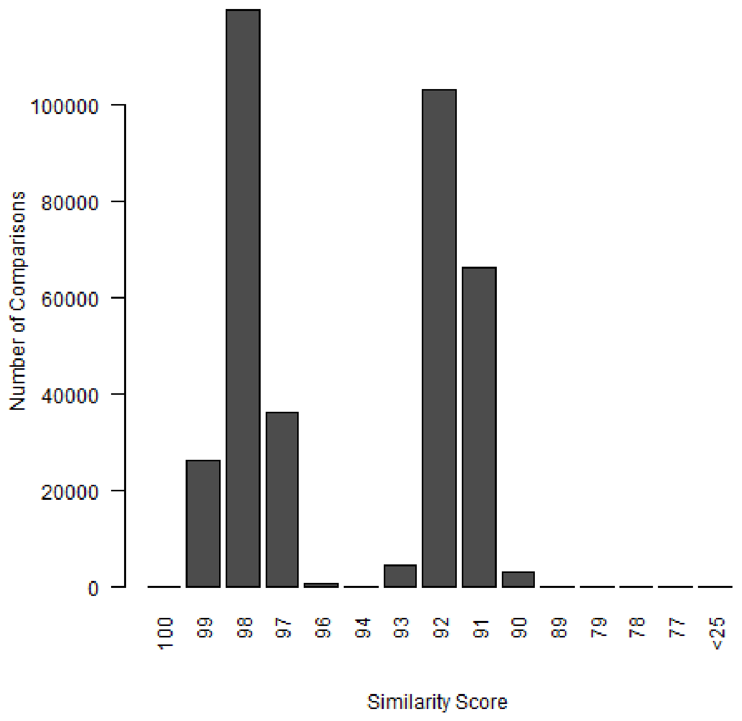
The techniques described in Section 3.1 were able to automatically successfully de-obfuscate 31,386 out of 40,120 documents that did not de-obfuscate initially.

There are large sets of documents in the data set that share exactly the same JavaScript. The group of PDF documents with 'broken' JavaScript is the largest group of PDF documents with exactly the same obfuscated JavaScript at 61,990 PDF documents. The largest group of identical de-obfuscated JavaScript is much smaller, only 3,339 PDF documents, which are unrelated to the set of documents with the 'broken' JavaScript. Of this set with identical de-obfuscated JavaScript, 3,024 documents have a matching graph hash.

We were able to identify a large set of highly similar graphs using the graph similarity algorithm and used sdhash to compare the JavaScript in the documents with these graphs to confirm that they are similar. From the graph comparisons, there is a set of 1,150 graphs belonging to 10,693 PDF documents that all have a similarity score of 1. All but 4 of the PDF documents have successfully-extracted JavaScript. Of these 10,689, 69 JavaScript samples successfully de-obfuscated. Since a large portion of the documents have obfuscated

JavaScript but not many have de-obfuscated JavaScript, we used sdhash to compare the obfuscated JavaScript to determine if the documents are similar. Figure 8 shows the sdhash results from 39 files compared against the rest of the set, for a total of 359,502 comparisons with 359,463 scores above the sdhash threshold of 25. Each comparison computes an sdhash score for two JavaScript samples. Figure 8 groups the comparisons by sdhash score, showing that 99% of the JavaScript samples have a sdhash similarity score of over 90. The complete set comparisons did not finish in time for this paper, but these results suggest the JavaScript in the PDF documents with matching graph hashes is highly similar.

Figure 8: Obfuscated JavaScript sdhash similarity scores for PDF documents belonging to a set of 1,150 graphs with similarity score of 1.



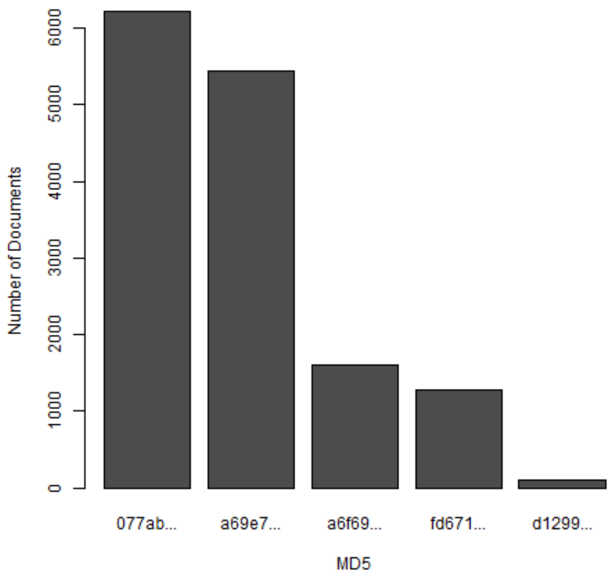
We modified *swf_mastah* [55] to accept a byte string of input rather than using *peepdf* to parse the PDF document itself. This change allows *swf_mastah* to check all string and stream data parsed by the tool without using *peepdf*. Dugmare successfully found Adobe Flash for 2,063 PDF documents. Of these, Dugmare extracted JavaScript for 1,978 PDF documents, suggesting that JavaScript and Flash exploit use is correlated. From the Flash files, furnace-avm2 successfully decompiled 23 distinct ActionScript samples from 749 PDF documents. 673 PDF documents match in decompiled ActionScript, and 670 of them match in graph hash, "d41d8cd98f00b240e9800998ecf8427e".The sdhash comparison revealed that some of the other ActionScript samples are very similar - differing by a few bytes in a payload or in some errors raised by furnace-avm2.

Dugmare parsed images and JavaScript for URLs, and extracted 147,420 unique URLs from 259,705 PDF documents. Within these URLs, there were 10,458 unique domains. We used Python socket.gethostbyname function to resolve domains to IP addresses. We ran each of the URLs through the function, but only 39,564 non-unique domains resolved to 1,473 IP addresses, possibly due to the short lifespan of malicious servers. The URLs and IP addresses that we found further group the PDF documents. For example, 14,677 PDF documents contain 744 unique URLs with domains that resolve to the IP address 78.111.51.123, an IP address associated with the Blackhole Exploit Kit [58]. These PDF documents are composed of five graph hashes, shown in Figure 9.

The pattern 'http://[domain]/download_file.php?e=[exploit string]' matches 411 unique URLs of the data set contained in 2,169 unique PDF documents. There were two exploit strings used to finish the URL parameter: 'Adobe-80-2010-0188' and 'Adobe-90-2010-0188', with 201 URLs matching the first exploit string and 210 matching the second. The PDF documents containing URLs matching this pattern have just two graph hashes. There are 43 PDF documents with a graph hash of "419152986fa55d0e18f9369a091ff773", and all 43 of them contain image exploits that match this URL pattern. The other graph hash, "3f092f00e5bba6a1e0e0d1ba70a7bedf", has 3,568 PDF documents, of which images were extracted from 3,558 files. However, only 2,126 of these documents match this URL pattern. Table 3 lists the other URL patterns for PDF documents with this graph hash. Many of these URLs are similar to the original pattern.

The extracted features were beneficial in grouping the malicious PDF documents of the data set. The graph hashes provided a useful initial grouping, and the graph similarity scores

Figure 9: Graph hashes of documents that contain some URL that resolves to the IP address 78.111.51.123.



added to these groupings. A large portion of the PDF documents contained JavaScript exploits, and a high percentage of the JavaScript samples were similar or identical. The ActionScript was less prevalent as a feature. The URLs extracted from the JavaScript and image exploits matched domains and URL patterns of known exploit kits. Grouping by these features demonstrates successful tactical assistance to our analysis of a large number of malicious PDF documents.

Table 3: Other URL patterns found.

Other URL Patterns
/showthread.php?t=[6-digit number]
/dl.php?5
/payload.php?e=Adobe-80-2010-0188
/payload.php?e=Adobe-90-2010-0188
/load.php?spl=Adobe-80-2010-0188
/load.php?spl=Adobe-libtiff
/download.php?e=Adobe-PDF-8
/download.php?e=Adobe-PDF-9
/file.php?e=Adobe-80-2010-0188
/file.php?e=Adobe-90-2010-0188
/drop.php?e=Adobe-80-2010-0188
/drop.php?e=Adobe-90-2010-0188
/exe.php?x=tiff

3.4 DISCUSSION

The above analysis of PDF documents based on their structure, JavaScript, Flash, images, and URLs allowed us to group the PDF documents based on the feature set. This section will look further into some of the most prevalent results from the graph hashes, JavaScript, graph similarity, and URLs. Using the graph hashes, we grouped over 40% of the PDF documents into two groups. In fact, 98% of the PDF documents share a graph hash with at least one other document. Running a graph similarity algorithm on the graphs identified more documents with a similar graph structure. Finally, we identified a group of PDF documents created by the Bleeding Life exploit kit using URL patterns.

As Table 7 demonstrates, the top two graph hashes account for a large portion of the malicious PDF document population. The first, “a69e72586baa6b6cfb69d1106f014469”, accounts for 107,046 PDF documents or 20.7% of the parsed PDF documents. The second graph hash, “15171cb907dfdd161c6125ff35dea40f”, is close behind, with 98,426 or 19.0% of the PDF documents. Initial analysis of PDF documents suggests that despite different obfuscated JavaScript samples, the de-obfuscated JavaScript for PDF documents within these two hashes, are nearly matching. For case studies of the PDFs represented by these two hashes, see Section 4.1.

Of the 10,141 graph hashes, 8,906 of them (87.8%) are unique to a single PDF document; 1,236 graph hashes belong to more than one PDF document. 496,948 PDF documents (98% of those successfully hashed) share a graph MD5 with at least one other PDF document. Therefore, the graph hash calculations resulted in an initial grouping of 98% of the PDF documents with at least one other PDF document. The tool used the similarity algorithm to compare all of the 10,141 distinct graphs from the data set; 3,314 graphs have a similarity score of 1 with at least one other graph, grouping 33% of the 2% that was ungrouped.

The fact that 26.9% of all of the PDF documents share obfuscated JavaScript with at least one other PDF document and 23.9% share de-obfuscated JavaScript suggest that many of the JavaScript exploits and PDF documents share common sources. Two explanations for this phenomenon are exploit kits that mass produce malicious PDF documents or leaked JavaScript exploits from previous malicious PDF documents. This chapter and the following two chapters will list further evidence of exploit kits within the data set.

The graph similarity algorithm found 1,150 identical graphs that did not have identical graph hashes. The 10,693 PDF documents that have these graphs share similar properties, such as obfuscated JavaScript that failed to de-obfuscate. The sdhash scores suggest a high similarity among the obfuscated JavaScript samples. These results show that using graph similarities along with the graph hashing can provide further identification of similar PDF documents. The case studies in Sections 4.1 and 4.2 will demonstrate the use of graph similarity in greater detail.

The URL pattern ‘http://[domain]/download_file.php?e=[exploit string]’ is similar to URL patterns used by exploit kits. It is plausible that the exploit strings ‘Adobe-80-2010-0188’ and ‘Adobe-90-2010-0188’ refer to an Adobe vulnerability and the version of Adobe that it attacks: 80 for version 8.0, 90 for version 9.0. Finally, it lists the CVE number used in the exploit, CVE-2010-0188, which is the Adobe TIFF File vulnerability [50]. In fact, all PDF documents that match this URL pattern contain an image exploit, which is generally used to exploit CVE-2010-0188. Further research suggests this URL string pattern is used by the Bleeding Life exploit kit to record the exploit used so that the authors can calculate accurate success rates for each exploit [59].

These results show that the feature set is an effective way to group malicious PDF documents for classification and analysis. This work has further applications in identifying malicious PDF documents from benign PDF documents. While we did not test the graph hashing on benign PDF documents, the graph hash could triage the documents for quick identification of malicious PDF documents from exploit kits. However, this technique would be highly susceptible to slight changes in PDF structure. The case studies in Chapter 4 will demonstrate how to identify a similar group of malicious PDF documents from the data set and how to find similar PDF documents given a set of known malicious PDF documents.

4.0 CASE STUDIES

Simply sorting the malicious PDF documents into like groups is beneficial because it reduces the work of malware analysts, however, the ability to associate these groupings with known malware families creates an even greater benefit. In this section, we will explore two case studies that show different strategies for associating malicious PDF documents with known malware families, using the feature set from this thesis. In the first case study, we look at the features of the largest malicious PDF document groupings and link them to the Blackhole Exploit Kit. In the second case study, we analyze known samples from the Phoenix Exploit Kit and use our feature set to identify other possible Phoenix samples within the data set.

4.1 CASE STUDY I: ALL ROADS LEAD TO BLACKHOLE

In the first case study, we will discuss the high prevalence of the Blackhole Exploit Kit within the data set, and how we came to this conclusion using the features extracted from the data set. This is an important use of the feature set; there may be other scenarios where an analyst is given a large collection of malicious documents with no knowledge of their origin. In such a scenario, it is useful to be able to link the largest groupings of documents to specific families of malware. We were able to achieve this with our data set, using PDF documents as an example.

The first version of the Blackhole Exploit Kit was released in 2010 and quickly became one of the most popular exploit kits ever created [60]. The kit originates from Russia and uses a rental model that allows customers to rent time on a Blackhole server or a license option that allows customers to host their own Blackhole servers. The kit targets vulnerabilities in

Java, Adobe Flash, Adobe Acrobat, Internet Explorer, and Windows, and contains a variety of exploits for each. The Blackhole Exploit Kit is used to deliver other malware to a host, such as the Zeus banking trojan or Cridex malware. Blackhole Exploit Kit remained one of the most prominent sources of malware until the arrest of its author in October 2013 [61]. Analysis of the data set suggests that 258,250 of the PDF documents were created by the Blackhole Exploit Kit generator. The exploit kit created the most common malware seen in the wild at the time of data collection.

The top two graph hashes from the data set, "a69e72586baa6b6cfb69d1106f014469" and "15171cb907dfdd161c6125ff35dea40f" both appear to be from the Blackhole Exploit Kit. Both sets of PDF documents contain varying JavaScript obfuscation techniques but PDF documents within each set match nearly exactly in de-obfuscated JavaScript. The de-obfuscated JavaScript from "a69e72586baa6b6cfb69d1106f014469" matches Blackhole PDF exploit Type 1 and "15171cb907dfdd161c6125ff35dea40f" matches Blackhole PDF exploit Type 2 as described by Howard [60] and Desai and Haq [62].

The de-obfuscated JavaScript of the PDF documents matching the Blackhole graph hashes contains shellcode in Unicode or hexadecimal that includes a URL. Early versions of Blackhole contain URLs that link to a PHP file on varying domains, and include two arguments: `f` and `e`. It is believed that `f` refers to the customer who created the PDF (so that the correct payload is downloaded) while the value of `e` is used to track the successful exploit that was used [60]. The Blackhole exploit kit control panel lists the exploits offered by the kit and real-time statistics about their effectiveness. A later document notes the exploit values in the URL string may change with the version of the exploit kit used [63]. This thesis found 101,392 URLs from 210,205 PDF documents that contain a parameter `e`. Table 4 lists the values of `e`, their meaning in Blackhole, and the number of URLs found containing the parameter.

The PDF Type 1 and PDF Type 2 were to be expected, since PDF documents containing the Type 1 and Type 2 JavaScript have been identified in the data set. However, the IE MDAC exploit, an older exploit that targets the Microsoft Data Access Component (MDAC) in Internet Explorer, was an unexpected result since the data set is composed of PDF documents. This could be due to changes in the values of `e` for different versions of

Table 4: The prevalence of the `e` URL parameter within the data set for PDF documents with URLs containing `f` and `e` parameters.

e	Exploit	Number of PDF documents
0	Java	0
1	Flash	0
2	MDAC	0
3	PDF Type 1	91,647
4	PDF Type 2	38,613
5	IE MDAC	29,723
6	Unknown	50,222
7	IE MSXML	0

the exploit kit.

Table 5 breaks down each value of `e` by graph hash within the data set. Particular exploit parameters seem to be associated with each PDF exploit type. For example, graph hash "a69e72586baa6b6cfb69d1106f014469", which uses PDF exploit Type 1, makes up a large percentage of URLs with `e` parameter values of 3 or 5, while graph hash "15171cb907dfdd161c6125ff35dea40f" makes up a large percentage of URLs with `e` parameter values of 4 or 6. In this way the analyst can identify other possible PDF documents created by Blackhole by looking for PDF documents that match the URL pattern.

The comparison of the two main Blackhole graph hashes did not return any identical graphs for graph hash "a69e72586baa6b6cfb69d1106f014469", but Table 6 shows five graph hashes that have a similarity score of 1 with "15171cb907dfdd161c6125ff35dea40f". The first two hashes can be found in Table 5, "a6f69b666889e4bc4c705014c8ca7a9b" and "fec1abe91f68c37a0a400b74f1ae1ea4", with URLs containing e values of 4 and 6 respectively, both of which the "15171cb907dfdd161c6125ff35dea40f" set also has. Furthermore, "a6f69b666889e4bc4c705014c8ca7a9b" contains URLs that resolve the same domain used by other Blackhole graph hashes [58] (see Table 9).

Table 5: The URL parameter `e` for Blackhole broken down by graph MD5 within the data set.

e	Graph Hash	Number of PDF documents
3	a69e72586baa6b6cfb69d1106f014469	68,238
3	077abc1b91cde57ef299bf7787ebbb80	23,378
3	e01eb3fd21c121ef472ae7f0e130f052	30
3	2f1af874839fd14ce0b4d14523b8bacc	1
4	15171cb907dfdd161c6125ff35dea40f	18,879
4	a6f69b666889e4bc4c705014c8ca7a9b	15,110
4	fd6716d9ecf7f6a832f87d239b5df24a	2,272
4	d1299c27d2008c465c23c6361ae5f703	1,466
4	1b5d20ad33f549c05a52e9c7d00ded2a	867
4	a63ec13bb8319147fd94a78a660f5e89	19
5	a69e72586baa6b6cfb69d1106f014469	29,180
5	cef4f4da91ab7ea1c0d43b6225b88367	147
5	8684f0fd4be6e61bd6c3816372a5ecf6	131
5	ec2ab4c835e8d24e12b00af406c7f2a6	77
5	b960da88098d49cd3a6af257858b4824	59
5	61672da6468343c6a961524da7475878	35
5	2c9d12a0534c433ac0ad9cf48a8c4158	28
5	ee4b338103288e74f10af7dd1ad1281d	27
5	c4420f43f812c520c8a43b6461efbdcf	6
5	c046f60078f67c879e8c218449bc8a12	1
6	15171cb907dfdd161c6125ff35dea40f	50,219
6	fec1abe91f68c37a0a400b74f1ae1ea4	2
6	85f5df8cced26b3bb30e3e8d25fb8bc8	1

The rest of the graph hashes require manual analysis to determine if they are Blackhole

PDF documents. There is a parsing error in the stream extraction for object 8 of PDF documents with graph hash “ba8a7101b03c6449a2075aac7b674588” that prevents all of the stream from being extracted. Manual analysis of the JavaScript exploit within this stream confirms that it is PDF exploit Type 2 from the Blackhole Exploit Kit. Likewise, a parsing error for PDF documents with graph hash “4a9fe1141a1eefb4493c2637318edbfb” prevents the object 8 data from being added to the xml for the PDF documents. Since the JavaScript exploit is in object 8 for “15171cb907dfdd161c6125ff35dea40f” PDF documents, this parsing error could explain why no JavaScript was extracted from these files. In fact, manual analysis of object 8 of a few of the “4a9fe1141a1eefb4493c2637318edbfb” documents reveals that they do in fact contain the PDF Type 2 exploit for the Blackhole Exploit Kit. Both of these graph hashes display the power of the graph hash as a grouping mechanism; we were able to identify errors that can be commonly solved for an entire group of PDF documents. Finally, manual analysis of “f1bffd255b04365a91bb4c44ca34e43e” files found that their obfuscation techniques and de-obfuscated JavaScript do not match the other Blackhole samples, so they most likely are not from the Blackhole Exploit Kit.

Table 6: Prevalence within the data set of graphs with a similarity score of 1 with the Blackhole graph hash “15171cb907dfdd161c6125ff35dea40f”.

Graph Hash	**PDFs**	**Obfuscated JS**	**De-Obfuscated JS**
a6f69b666889e4bc4c705014c8ca7a9b	17,247	16,541	15,061
fec1abe91f68c37a0a400b74f1ae1ea4	5	5	2
ba8a7101b03c6449a2075aac7b674588	2,235	1,251	0
f1bffd255b04365a91bb4c44ca34e43e	21	21	0
4a9fe1141a1eefb4493c2637318edbfb	95	0	0

The study of various features of the top two graph hashes in the data set determined they were created by the Blackhole Exploit Kit. Finding other PDF documents in the data set with similar features led to the identification of 23 related graph hashes. The identified graphs belong to a total of 258,250 PDF documents. This feature analysis led us to conclude that an estimated 49.8% of the data set was created by or is related to the Blackhole Exploit Kit. This high percentage is not unexpected due to the high popularity of the exploit kit at the time that the data set was collected. This grouping assists a malware analyst in classifying a large portion of the data set without manual analysis of every document.

4.2 CASE STUDY II: KNOWN TO UNKNOWNS

This case study discusses finding PDF documents within the data set similar to a known sample set. The process of using known malicious indicators, such as features of malicious PDF documents, to discover other related indicators is called indicator expansion [64]. While we do not use this approach exactly, the idea is similar since we use our features to expand a set of similar PDF documents. Furthering the indicator set for a particular malware using indicator expansion will increase the ability of security specialists to identify the malware on their systems and in large collections such as the one used in this thesis. The sample set used for this case study is a set of seven PDF documents created by the Phoenix Exploit Kit version 2.0 distributed for research use by Contagio [51]. The Phoenix Exploit Kit was a browser exploit kit written in PHP that provided exploits for 16 different vulnerabilities, including several Adobe vulnerabilities. It was originally detected in 2007 but did not gain prominence among malware seen in the wild until 2009 [65].

As demonstrated in Table 2, six of the seven Phoenix 2.0 samples contain JavaScript and the same graph MD5: "06055c4d82813cce7aaad42d283b3181". This graph hash matches 1,908 PDF documents within the data set. Of those, 1,170 have JavaScript that successfully de-obfuscated and 1,093 of the JavaScript samples contain 'fix_it', a function name common to each of the de-obfuscated JavaScript samples from the sample set. The graph hash has a similarity score of 1 with three other graphs in the data set. PDF documents with these graph hashes and number of successful JavaScript extractions are listed in Table 7. Unfortunately, obfuscated and de-obfuscated JavaScript was only found for one of the graph hashes, and the code does not have similarities with the code found in the Phoenix samples. While these PDF documents could be from a different version of the Phoenix Exploit Kit, we cannot make this assumption without other samples with which to compare. Thus in this scenario, graph similarity scores failed to find more similar documents.

Table 7: Prevalence within the data set of graph hashes matching the Phoenix samples containing JavaScript. Further analysis cannot connect the JavaScript exploits to the Phoenix samples.

Graph Hash	PDFs	Obfuscated JS	De-Obfuscated JS
7b75b83afcab213e75d928d317ea35fa	74	74	52
716d44efa6284713045421c4353594ee	6	0	0
f80b93a28e986339d8128e006438fc7b	1	0	0

The obfuscated JavaScript for each Phoenix 2.0 sample is identical and matches 306 PDF documents within our data set. All 306 of those PDF documents match the graph hash "06055c4d82813cce7aaad42d283b3181" of the JavaScript exploit PDF documents from the Phoenix 2.0 sample set. These exact matches occur despite the fact that there are comments of random alphanumeric characters scattered throughout the obfuscated JavaScript. The fact that these comments match for so many samples suggests that the comments are not changed for each new PDF document created by the exploit kit. Despite the identical obfuscated JavaScript, the de-obfuscated JavaScript for the Phoenix 2.0 samples does not match any de-obfuscated JavaScript within the data set. The sdhash comparisons found another JavaScript sample that had a similarity score of 90 with the Phoenix 2.0 obfuscated JavaScript. This JavaScript belongs to a graph hash matching four PDF documents, two of which match the Phoenix 2.0 JavaScript and two which do not. Without further information about the Phoenix Exploit Kit, this thesis cannot conclude that the two that do not match are associated with the Phoenix Exploit Kit. By broadening the analysis to include all files in the dataset matching the Phoenix 2.0 JavaScript graph hash, we can identify another graph hash that has over 300 documents that exactly match obfuscated JavaScript for some of the files with the Phoenix hash. These files are possibly related to Phoenix 2.0, but more evidence is necessary.

The graph hash for the Phoenix 2.0 image exploit sample differs from that of the Phoenix 2.0 JavaScript samples; it is “fe4102d6db98dd3e13e261ff55212e35”. There are 32 PDF documents within the data set that match this graph hash, and each one has an image exploit with an URL similar to the one in the Phoenix 2.0 sample. The graph hash has a similarity score of 1 with seven other graph hashes. Table 8 lists PDF documents with these graph hashes and successful image and JavaScript extractions. The first two entries in the table, “2b673e1764a93aa421cf875c567c697f” and “5e4515b91fbdcc56c0fb44bc7da553d4” contain image exploits with URLs matching the Phoenix 2.0 URL pattern, so they appear to be other Phoenix 2.0 documents. However, the last five all contain JavaScript exploits with similar obfuscation techniques so while they are possibly related to each other, they are probably not related to the Phoenix 2.0 image exploit sample.

Table 8: Prevalence within the data set of graph hashes matching the Phoenix sample containing an image exploit.

Graph Hash	PDF Documents	Images	Obfuscated JS
2b673e1764a93aa421cf875c567c697f	68	68	0
5e4515b91fbdcc56c0fb44bc7da553d4	98	98	0
6b6d59f74f14a9da9ebf2ce4f224dd22	1	0	1
775ec88ba274c93b709b1ab30b03dd04	21	0	21
e677fc7c7472e0ccc3bcbba3be25cae0	1	0	1
f6c4b0f4a15aaa2c816485f652160c27	3	0	2
fb6397534acf9b778c6df5a8359afb06	1	0	1

Each PDF document from the Phoenix 2.0 samples with a JavaScript exploit contains the

URL ‘http://111.gosdfsdjas.com/l.php?i=6’. One PDF document in the set of unknown malicious PDF documents matches this URL exactly. There are 1,779 PDF documents with 1,751 unique URLs that match the URL pattern of ‘/l.php?i=’, with different values for the `i` parameter. Table 9 displays these graph hashes, values, and number of PDF documents. Each of these graph hashes has already been identified in this analysis as a possible Phoenix 2.0 document. The first four are the graph hash of the Phoenix 2.0 samples with JavaScript. The next graph hash, “56cd016287cc4476810e21d6fce37fa1”, has a sdhash similarity score of 100 with other PDF documents with the Phoenix 2.0 JavaScript exploits graph hash. The final three include the image exploit graph hash and two of the graph hashes that have a similarity score of 1 with the image exploit graph hash. It is interesting to note that all of the image exploits have the same value for the parameter `i`: ‘8’. If the `i` parameter represents the exploit being used, a value of ‘8’ could represent an image exploit.

Thus, by extracting features from known Phoenix Exploit Kit 2.0 samples this thesis was able to identify PDF documents within the data set that were similar to or created by the exploit kit. This thesis identified five different graph hashes that match the samples in some way, and two other files that also match, for a total of 2,454 documents or .5% of the data set.

Table 9: The values for parameter `i` for suspected Phoenix Exploit Kit documents in the data set.

Graph Hash	i	Number of PDF documents
06055c4d82813cce7aaad42d283b3181	4	393
06055c4d82813cce7aaad42d283b3181	5	287
06055c4d82813cce7aaad42d283b3181	6	294
06055c4d82813cce7aaad42d283b3181	16	298
56cd016287cc4476810e21d6fce37fa1	13	311
5e4515b91fbdcc56c0fb44bc7da553d4	8	98
2b673e1764a93aa421cf875c567c697f	8	68
fe4102d6db98dd3e13e261ff55212e35	8	30

5.0 CONCLUSION

As with all malware, there exist far too many samples of malicious PDF documents for security researchers to analyze manually. This thesis proposes several features of malicious PDF documents and techniques to automatically extract them from the PDF documents. We use these features to observe trends and to classify the malicious PDF documents. Section 3.2.1 identifies and describes the features used in the feature set, and Section 3.2.2 provides justification for the features. Section 3.2.3 explains how the features can be extracted automatically using open source tools and scripts. Automatic feature extraction allows analysis of a large collection of PDF documents. From there, we complete further analysis using the techniques described in Section 3.1. Section 3.3 reports the most prominent results found from the feature analysis. The results showed that the object structure of the document is an effective way to create an initial grouping of malicious PDF documents, which is confirmed through the use of other features such as JavaScript. The URLs can also identify similar PDF documents and exploit parameters used by the PDF documents. The analysis identified PDF documents with URL patterns possibly matching the Bleeding Life Exploit Kit. Sections 3.4 provides a discussion of the results. Chapter 4 shows the results of two case studies which found PDF documents that were possibly created by known exploit kits. The two case studies identified 260,704 possible PDF documents, or 50.3% of the data set.

There are several possible limitations to our work. The data set could be skewed in some way that would make the analysis inaccurate for normal data. For example, the majority of the documents were submitted by users who believed that the documents were malicious. This form of collection can cause skew through incomplete collection. There could be sophisticated malicious PDF documents that do not cause the user to suspect they are malicious or that were downloaded without user knowledge, and thus would not be submitted to the data set. As such, our results may not accurately represent the populations of malicious PDF documents seen in the wild. Further, since this thesis was unable to successfully de-obfuscate all of the JavaScript, there could be important differences in the documents with failed de-obfuscation that were missed. Finally, there is a lack of public resources on known exploit kits to compare the features to, hindering the correct identification of documents within the data set. Incomplete collection, de-obfuscation, and resources all affect the possibility of inaccurate results for this data set.

Overall, the results from this work are promising. Previous work [33, 35] determined that the PDF document structure is important in identifying similar documents. We built on this work by grouping large collections of malicious PDF documents with identical document structures. Not only was the document structure useful in finding large groups of similar documents, but it also provided a new means of indicator expansion [64]. Given a known malicious indicator, we found documents containing the indicator and augmented the malicious indicator set by finding similar documents with different indicators using the document structure. The other features of our feature set, including URLs and JavaScript, confirmed the similarity of documents found using the document structure. This work has further implications in malicious document analysis and identification, as discussed in the next section.

5.1 FUTURE WORK

The use of document structure and other features to identify similar documents has other implications outside of analyzing a large data set. Future work could include using the structure and other features to identify malicious versus benign PDF documents, much like the work of Šrndić and Laskov [33]. The development of a malicious document detection technique using our features would require testing on benign documents to check for structural collisions or similar features to documents the malicious data set. Our work is slightly less dependent on the object structure than Šrndić and Laskov's work due to the use of other features, such as JavaScript, and the graph similarity algorithm, which will make it less susceptible to changes in document structure. However, it would still be weak to complete changes in document structure, such as evasion using an embedded document described by Maiorca et. al. [34].

Malware repositories still collect malicious PDF documents, but not at the high volume of several years ago. We can apply the techniques described in this thesis to other types of structured documents, such as Microsoft Office files. Given a large collection of structured documents, we can identify like features and structural similarities to the work in this thesis.

The study of PDF documents that did not de-obfuscate as well as PDF documents from other data sets could increase the success of JavaScript de-obfuscation. For example, several malware samples retrieved data from an xfa field but our JavaScript de-obfuscation tool did not accommodate this functionality. There are several intricacies to xfa data that increase the difficulty of automatically identifying this technique in the code and finding the data within the parsed PDF document. The incorporation of this method, for example, will increase the success rate of de-obfuscation.

BIBLIOGRAPHY

[1] D. Danchev. Report: Malicious PDF files comprised 80 percent of all exploits for 2009 | ZDNet. [Online]. Available: http://www.zdnet.com/blog/security/report-malicious-pdf-files-comprised-80-percent-of-all-exploits-for-2009/5473

[2] J. Gantz and D. Reinsel, "Big data, bigger digital shadows, and biggest growth in the far east," *DC iView: IDC Analyze the Future*, 2012.

[3] Adobe PDF 101 - quick overview of PDF file format. [Online]. Available: http://partners.adobe.com/public/developer/tips/topic_tip31.html

[4] "ISO 32000-1:2008." [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=51502

[5] A. C. Madrigal, "Flash and the PDF: Computing's last great and now endangered monopolies," *The Atlantic*, Apr. 2012. [Online]. Available: http://www.theatlantic.com/technology/archive/2012/04/flash-and-the-pdf-computings-last-great-and-now-endangered-monopolies/255403/

[6] J. Yonts, "PDF malware overview," SANS, Tech. Rep., Jul. 2010. [Online]. Available: http://www.sans.org/security-resources/malwarefaq/pdf-overview.php

[7] Not everything is peachy with PDFs - InformationWeek. [Online]. Available: http://www.informationweek.com/not-everything-is-peachy-with-pdfs/d/d-id/1011421?

[8] D. Waltermire and K. A. Scarfone, "Sp 800-51: Guide to using vulnerability naming schemes," *Recommendations of the National Institute of Standards and Technology*, 2011.

[9] Cve-2006-5857. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5857

[10] D. Stevens, "Malicious PDF documents explained," *Security and Privacy*, vol. 9, no. 1, pp. 80–82, 2011.

[11] K. Selvaraj and N. F. Gutierrez, "The rise of PDF malware," *Symantec Security Response*, 2010.

[12] Email "inovice_aug_9693495.pdf" contains malicious PDF file | mxlab - all about anti virus and anti spam. [Online]. Available: http://blog.mxlab.eu/2014/08/22/email-inovice_aug_9693495-pdf-contains-malicious-pdf-file/

[13] "Document management portable document format part 1: PDF 1.7," Adobe Systems Incorporated, Tech. Rep., 2008.

[14] Cve-2012-075. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0754

[15] Cve-2011-0611. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0611

[16] Cve-2011-0609. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0609

[17] D. Stevens. (2010, Mar.) Escape from PDF. [Online]. Available: http://blog.didierstevens.com/2010/03/29/escape-from-pdf/

[18] "Detection of malicious pdf files and directions for enhancements: A state-of-the art survey," *Computers & Security*, pp. 246–266, 2015.

[19] K. A. Scarfone and P. M. Mell, "Sp 800-94: Guide to intrusion detection and prevention systems (idps)," *Recommendations of the National Institute of Standards and Technology*, 2007.

[20] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. Keromytis, "A study of malcode-bearing documents," *Lecture Notes in Computer Science*, vol. 4579, no. DIMVA, pp. 231–250, 2007.

[21] C. Smutz and A. Stavrou, "Malicious PDF detection using metadata and structural features," *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 239–248, Dec. 2012.

[22] C. Vatamanu, D. Gavriluţ, and R. Benchea, "A practical approach on clustering malicious PDF documents," *Journal in Computer Virology*, vol. 8, no. 4, pp. 151–163, 2012.

[23] J. Cross and A. Munson, "Deep PDF parsing to extract features for detecting embedded malware," Sandia National Labs, Albuquerque, New Mexico, Unlimited Release SAND2011-7982.

[24] P. Laskov and N. Šrndić, "Static detection of malicious JavaScript-bearing PDF documents," *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[25] D. Maiorca, G. Giacinto, and I. Corona, "A pattern recognition system for malicious PDF files detection," *Machine Learning and Data Mining in Pattern Recognition*, no. 510-524, 2012.

[26] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," *Proceedings of the Fourth European Workshop on System Security*, no. 4, Apr. 2011.

[27] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," *Proceedings of the International World Wide Web Conference*, 2011.

[28] VirusTotal - free online virus, malware and URL scanner. [Online]. Available: https://www.virustotal.com/en/

[29] J. Kittilsen, "Detecting malicious pdf documents," *Gjøvik University College Institutional Archive*, 2011.

[30] K. Borg, "Real time detection and analysis of pdf-files," *Gjøvik University College Institutional Archive*, 2013.

[31] M. D. Netork. SpiderMonkey. [Online]. Available: http://www.mozilla.org/js/spidermonkey/

[32] X. Lu, J. Zhuge, R. Wang, Y. Cao, and Y. Chen, "De-obfuscation and detection of malicious PDF files with high accuracy," *46th Hawaii International Conference on System Sciences*, pp. 4890–4899, 2013.

[33] N. Šrndić and P. Laskov, "Detection of malicious pdfs files based on hierarchical document structure," *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[34] D. Maiorca, I. Corona, and G. Giacinto, "Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection," *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[35] J. P. Donaldson, "Source fingerprinting in adobe pdf files," *Naval Postgraduate School*, 2013.

[36] D. Stevens. (2008, Nov.) PDF tools | didier stevens. [Online]. Available: http://blog.didierstevens.com/programs/pdf-tools/

[37] ——. (2008, Apr.) PDF, let me count the ways | didier stevens. [Online]. Available: http://blog.didierstevens.com/2008/04/29/pdf-let-me-count-the-ways/

[38] Y. Shinyama. (2010) PDFMiner. [Online]. Available: http://www.unixuser.org/~euske/python/pdfminer/

[39] J. M. Esparza. peepdf - PDF analysis tool | eternal-todo.com. [Online]. Available: http://eternal-todo.com/tools/peepdf-pdf-analysis-tool

[40] ——. peepdf - PDF analysis and creation/modification tool - google project hosting. [Online]. Available: https://code.google.com/p/peepdf/

[41] pyv8 - python wrapper for google v8 javascript engine - google project hosting. [Online]. Available: https://code.google.com/p/pyv8/

[42] v8 - v8 JavaScript engine - google project hosting. [Online]. Available: https://code.google.com/p/v8/

[43] D. Reinhard, *Graph Theory*, 3rd ed. Berlin, New York: Springer-Verlag, 2005.

[44] R. C. Sprinthall, *Basic Statistical Analysis*, 8th ed. Allyn and Bacon, 2007.

[45] R. Vassil, "Data fingerprinting with similarity digests," *Advances in Digital Forensics VI*, pp. 207–226, 2010.

[46] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious JavaScript code: A measurement study," *7th International Conference on Malicious and Unwanted Software*, 2012.

[47] W. W. W. Consortium, "Html 4.01 specification," 1999.

[48] Microsoft Developer Network (MSDN), "Octal and hexadecimal character specifications," 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/edsza5ck.aspx

[49] J. Wolf. (2010, Jan.) Pdf obfuscation using getannots(). [Online]. Available: https://www.fireeye.com/blog/threat-research/2010/01/pdf-obfuscation-using-getannots.html

[50] CVE-2010-0188 | adobe TIFF file vulnerability | trend micro threat encyclopedia. [Online]. Available: http://about-threats.trendmicro.com/us/vulnerability/722/adobe%20tiff%20file%20vulnerability

[51] M. Parkour. (2010, May) contagio: Phoenix 2.0 exploit kit. [Online]. Available: http://contagiodump.blogspot.com/2010/05/files-from-phoenix-20-exploit-kit.html

[52] ——. (2012, Sep.) contagio: CVE-2012-4681 samples original (APT) and blackhole 2.0 (crime). [Online]. Available: http://contagiodump.blogspot.com/2012/09/cve-2012-4681-samples-original-apt-and.html

[53] S. Richter. lxml - processing XML and HTML with python. [Online]. Available: http://lxml.de/

[54] (2007, Apr.) JavaScript for acrobat API reference. [Online]. Available: http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf

[55] B. Dixon. (2011, Nov.) pdfxray_public/swf_mastah.py at master 9b/pdfxray_public github. [Online]. Available: https://github.com/9b/pdfxray_public/blob/master/builder/swf_mastah.py

[56] SWFTOOLS. [Online]. Available: http://www.swftools.org/

[57] P. Zotov. (2013, May) whitequark/furnace-avm2 GitHub. [Online]. Available: https://github.com/whitequark/furnace-avm2

[58] Malware domain list. [Online]. Available: http://www.malwaredomainlist.com/mdl.php?inactive=on&sort=Reverse&search=rat&colsearch=All&ascordesc=ASC&quantity=100&page=25

[59] G. De Maio, A. Kapravelos, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Pexy: The other side of exploit kits," *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Jul. 2014. [Online]. Available: https://www.cs.ucsb.edu/~vigna/publications/2014_DIMVA_PExy.pdf

[60] F. Howard, "Exploring the blackhole exploit kit," *Sophos Technical Paper*, 2012.

[61] "Cisco 2014 midyear security report," Cisco, Tech. Rep., 2014. [Online]. Available: https://info.sourcefire.com/2014CiscoMidyearSecurityReport.html

[62] D. Desai and T. Haq, "Blackhole exploit kit: Rise & evolution," *Malware Research Team Technical Paper*, Sep. 2012.

[63] G. Szappanos, "Inside a black hole: Part 2," *Sophos Technical Paper*, 2010.

[64] J. M. Spring, "A notation for describing the steps in indicator expansion," in *IEEE eCrime Researchers Summit*. Anti-Phishing Working Group, September 17, 2013. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=73560

[65] M. Fossi, G. Egan, E. Johnson, T. Mack, T. Adams, J. Blackbird, B. Graveland, and D. McKinney, "Symantec report on attack kits and malicious websites," *Haettu*, vol. 5, p. 2012, 2011.