NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SOURCE FINGERPRINTING IN ADOBE PDF FILES

by

John P. Donaldson

December 2013

Thesis Advisor: Chris S. Eagle
Second Reader: George W. Dinolt

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2013	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE SOURCE FINGERPRINTING IN ADOBE PDF FILES	**5. FUNDING NUMBERS**
6. AUTHOR(S) John P. Donaldson	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Science Foundation Grant No. DUE- 0912048 National Science Foundation Grant No. DUE- 1241432	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Adobe Portable Document Format (PDF) documents are increasingly used as a vector for targeted attacks. Although there exist a number of tools and methodologies for performing content-level analysis to identify unwanted or malicious behavior or characteristics in these documents, these forms of analysis are hampered by increasingly complex obfuscation techniques and usually require execution of potentially malicious code. This thesis proposes a static analysis method that uses structural elements of PDF documents to identify the tools used to generate them. This method may be used to attribute malicious PDFs to particular toolkits.

14. SUBJECT TERMS Static analysis, Adobe, Portable Document Format, PDF, structural analysis, n-gram analysis, document authorship	**15. NUMBER OF PAGES** 59
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SOURCE FINGERPRINTING IN ADOBE PDF FILES

John P. Donaldson
Civilian, Department of the Navy
B.S., Northwest Nazarene University, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
December 2013**

Author: John P. Donaldson

Approved by: Chris S. Eagle
Thesis Advisor

George W. Dinolt
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Adobe Portable Document Format (PDF) documents are increasingly used as a vector for targeted attacks. Although there exist a number of tools and methodologies for performing content-level analysis to identify unwanted or malicious behavior or characteristics in these documents, these forms of analysis are hampered by increasingly complex obfuscation techniques and usually require execution of potentially malicious code. This thesis proposes a static analysis method that uses structural elements of PDF documents to identify the tools used to generate them. This method may be used to attribute malicious PDFs to particular toolkits.

THIS PAGE INTENTIONALLY LEFT BLANK

DISCLAIMER

Partial support for this work was provided by the National Science Foundation’s CyberCorps®: Scholarship for Service (SFS) program under Award No. DUE- 0912048 and DUE-1241432. 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 the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
DOCX	Office Open XML Document format
DVI	Device Independent File Format
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
PDF	Portable Document Format
PDF/A	PDF for Archive
PDF/E	PDF for Engineering
PDF/X	PDF for Exchange
SCAP	Source Code Author Profiles
TIFF	Tagged Image File Format
XRef	Cross-reference table

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. INTRODUCTION

The Adobe Portable Document Format is a widely-used format for online document interchange. Roughly 23.4 % of the 986,278 documents in the NPS Govdocs1 corpus consists of PDF documents [1]. A wide variety of tools can, either as part of their core functionality or as an add-on, read or write PDF documents. It is the PDF format's very ubiquity that has made it a tempting vehicle for targeting vulnerable applications.

In 2012, Symantec reported that 11% of all targeted attacks involved exploits against PDF-rendering applications, a figure that does not include non-targeted, opportunistic attacks [2]. Figures from F-Secure for 2008–2010 indicate that anywhere from 28–61% of targeted attacks involved PDF exploits, with a steadily increasing trend [3], [4]. In the last quarter of 2012 alone, Microsoft reported more than a 100% increase in the number of detected attacks that used vulnerabilities in Adobe Reader and Adobe Acrobat to execute malicious Javascript payloads [5]. The mechanisms for the most common of these attacks are detailed in Chapter II, Section A.

In order to better profile the authors behind these attacks, there is interest in identifying the tools that they are using to create their malicious PDF documents. If an attacker uses an uncommon or custom tool, and that tool's signature can be identified in a document, regardless of the particular content or possible exploitation mechanisms and payloads, it may be possible to more easily identify potentially-malicious documents or attribute documents to a common source. Analysis of in-band, PDF-specific metadata, such as the reported creating program, can be insightful, but metadata can also be deleted or modified with various tools [6].

This thesis will attempt to identify a method for identifying the toolkits used by content authors to generate PDF documents. This will include benign documents. It will not attempt to identify signatures of particular exploits or malicious content, and will, to the greatest extent possible, avoid using signatures that are based on user-provided content or easily-modified data, such as document-provided metadata. Additionally, any selected methods should be fast enough to support real-time analysis.

B. ORGANIZATION OF THESIS

This thesis is organized into five chapters. The first chapter introduces the topic of PDF document forensics. The second explains background information useful to understand the structure of PDF documents, as well as related prior work. The third chapter describes the experimental methodology for identifying and classifying these documents. The fourth discusses results from the identification and classification process and gives a conclusion for the thesis. Finally, the fifth chapter addresses possible directions for future development.

II. BACKGROUND

A. PORTABLE DOCUMENT FORMAT

The Adobe PDF standard was first released in 1993, coinciding with the release of Adobe Acrobat 1.0 [7], [8]. Each release of the standard has been backwards compatible with prior releases, allowing a PDF 1.5-compliant reader to open a PDF 1.5-, 1.4-, 1.3, 1.2, 1.1, or 1.0-compliant file, and so on. Releases have added new supported features, such as encryption, embedded images, forms, or digital signatures. In 2008, the eighth release of the PDF standard, PDF 1.7, was codified as the ISO standard ISO 32000–1 [9, 10]. There are further specialized extensions to the standard, such as PDF/A [11], PDF/E, and PDF/X, which are intended for archival, engineering, and graphics interchange purposes, respectively. Adobe has continued to add features to the PDF format outside of the ISO standard, in the form of supplements [12].

A canonical PDF documents is composed of four main sections, a header, the body, a cross-reference table, and a trailer (see Figure 1) [13], [14].

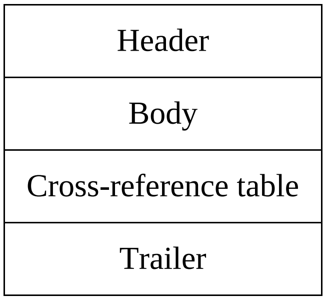
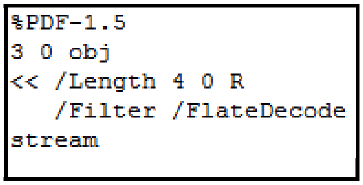


Figure 1. The four major sections of a PDF document

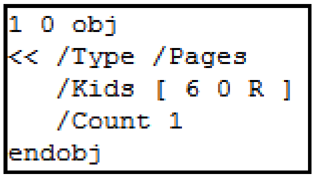
The header contains the version number of the PDF standard that the document uses, in the format “%PDF-<version_number>,” and is followed by a newline character (see Figure 2).



```
%PDF-1.5
3 0 obj
<< /Length 4 0 R
   /Filter /FlateDecode
stream
```

Figure 2. An ASCII representation of the first 80 bytes of a sample PDF document, including the document header and the beginning of an indirect object and stream.

The body of the file contains a sequence of objects that define the content and appearance of the document. There are two types of objects: direct objects and indirect objects. Direct objects are unlabeled primitives that cannot be referred to by other objects, and are similar to literal values present in other programming languages. Indirect objects are labeled in such a way that other objects can refer to them. Indirect objects appear in the file preceded by an object number, a generation number, and the keyword “obj,” and are terminated with the keyword “endobj” (see Figure 3). The object number and generation number are used to uniquely identify objects. Generation numbers identify the version of an object, and initially start at 0, growing as large as 65,535 as an object is updated or reused. This functionality is intended to allow a program to deallocate an object’s memory space, and then place new content in its place, to avoid having to rewrite an entire file for small updates.



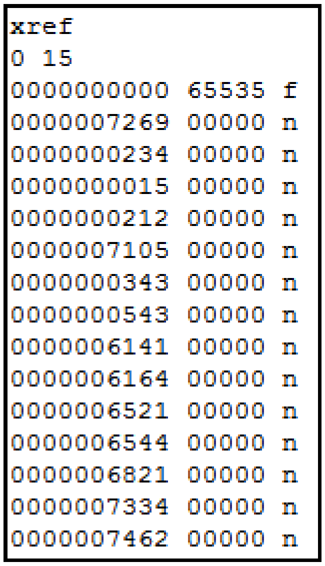
```
1 0 obj
<< /Type /Pages
   /Kids [ 6 0 R ]
   /Count 1
endobj
```

Figure 3. A sample PDF indirect object, object 1, generation 0, of the type “Pages,” which includes a field, “Kids,” with an indirect reference to generation 0 of object 6, and a field, “Count,” equal to 1.

Indirect objects may contain embedded direct objects, such as strings, streams of arbitrary binary data, and numeric values, or references to other indirect objects. These objects may contain dictionaries of fields that describe aspects of the object, such as its type, hierarchical relationships to other objects, font definitions, or displayed contents. A record of recognized object types is maintained by Adobe, and new types must be registered through them.

Data stored in streams may be encoded or encrypted through the use of filters. Filters support a variety of transformations, such as mapping binary data to a hexadecimal representation, or a number of compression schemes. Additionally, content may be encoded using any of several encryption algorithms, such as RC4, DES, 3DES, or AES. The format also supports PKI mechanisms to protect or authenticate content. These filter mechanisms only apply to complex content and not to primitive numbers, Boolean values, or entire objects, except for objects that appear in object streams. Object streams are a type of stream, introduced in PDF 1.5, which allows for objects to be grouped together, with the goal of allowing more efficient compression of large numbers of objects.

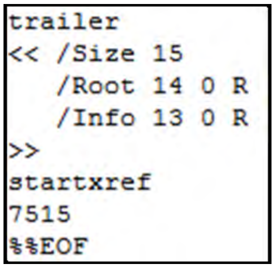
The cross-reference table allows PDF parsers to quickly locate indirect objects within a document. It begins with the keyword “xref” and can contain multiple subsections, each beginning with a line containing the number of the first object and the number of objects in the subsection. Each following line, one per object in the subsection, includes a 10-digit byte offset into the file, a five-digit generation number, and either the character ‘f’ or ‘n’, indicating whether the object defined on that line is either free or in use, respectively. The space occupied by free objects may be reused when new content is added to a document, instead of appending the new object to the end of the document. The first object in the cross-reference table is always free and has the largest possible generation number, 65,535 (see Figure 4). Further cross-reference tables can be defined as indirect objects, with the type “XRef.”



```
xref
0 15
0000000000 65535 f
0000007269 00000 n
0000000234 00000 n
0000000015 00000 n
0000000212 00000 n
0000007105 00000 n
0000000343 00000 n
0000000543 00000 n
0000006141 00000 n
0000006164 00000 n
0000006521 00000 n
0000006544 00000 n
0000006821 00000 n
0000007334 00000 n
0000007462 00000 n
```

Figure 4. A sample PDF cross-reference table, with one subsection and 15 items

The final section of a PDF document is the trailer. This section begins with the keyword “trailer,” immediately followed by a dictionary containing information about how to start constructing the document. Required keys include the size of the cross-reference table and the root of the document’s catalog. Following the “startxref” keyword is the byte offset into the file of the cross-reference table. The final line contains the keyword “%%EOF” (see Figure 5).



```
trailer
<< /Size 15
   /Root 14 0 R
   /Info 13 0 R
>>
startxref
7515
%%EOF
```

Figure 5. A sample PDF trailer section

It is possible for a document to contain multiple trailers and cross-reference tables. In this case, the trailer dictionary will contain a key, “Prev,” which points to the offset of the previous cross-reference tables. This usually happens when an application makes incremental updates to a document, allowing it to add material without altering earlier content, by chaining together cross-reference and trailer sections.

The PDF specification does not specify the order in which indirect objects should appear in a document, physically or logically, with the exception of Linearized” PDFs. These are documents whose internal representation has been ordered in such a way that a reader may begin to render the document before it has been fully parsed or loaded, typically by including objects with content that appears earlier in the document closer to the beginning of the file. Additionally, these documents contain “hint tables,” which provide a reader application with information about where objects exist within the file, before reading a cross-reference table, allowing the reader to locate objects that may have already been loaded or request out-of-order portions of the document from the backing storage device. The only ordering requirement for Linearized PDFs is that an object containing a linearization parameter dictionary must appear in the first 1024 bytes of the document. This dictionary object contains basic information about the document, such as the location of the hint tables and number of pages, and serves as an indicator to reader applications that they are opening a Linearized PDF without having to process more than the first 1024 bytes.

The elements of PDF documents described in this section are taken from the most current PDF specification. However, many readers, including Adobe’s products, will accept documents as valid that do not conform to the specification. For example, some readers will accept documents which entirely omit the cross-reference section in lieu of cross-reference objects, or which contain no trailer section, both of which are described as elements of a canonical PDF document. The PDF specification indicates that the keyword “%%EOF” must be the final element on a valid document, but, in an appendix to the PDF 1.7 specification, Adobe indicates that their products will validate a document if the keyword appears anywhere in the final 1024 bytes. As a result of these acknowledged inaccuracies in the specification’s implementations, the validity of a PDF document is perhaps not best described in terms of how well it adheres to the specification, but in whether or not Adobe Reader will open it.

Logically, the root of a PDF document is the document catalog. The document catalog is an indirect object containing rendering information such as dictionaries of pages, supported extensions, forms, layout information, metadata, tables of contents, and actions that are performed when a document is opened. The extensions dictionary is the vehicle by which non-ISO updates are supported to add new features.

Most attacks involving a maliciously-crafted PDF document leverage flaws in a reader application's implementation of specific content handlers. For example, an attacker may embed a JPEG or TIFF image that exploits a vulnerability in the handler for those image types. Since PDF 1.3, Javascript may be embedded in PDF documents, and many reader applications, including Adobe's Acrobat line of products, include an embedded Javascript engine [13], [15]. The inclusion of Javascript not only provides an additional attack surface for adversaries, but also a rich API for interacting with the system, beyond what is normally possible for a PDF document. A document may specify actions that are to take place automatically when it is opened, or upon certain user actions, such as clicking a button or entering text into a form. These actions may include the execution of Javascript or system commands. Some applications notify users of these actions, or require confirmation, but this is often poorly implemented, and it may be possible to use social engineering to convince users to permit these behaviors [16].

B. RELATED WORK

There is little work dedicated to non-content-based analysis of PDF documents, but there are a number of tools and methodologies intended to perform content-level analysis. Smutz and Stavrou developed a means to identify malicious content in PDF documents through structural analysis [17]. Additionally, they were able to further classify attacks involving these documents as either broadly-distributed opportunistic attacks, or targeted attacks. However, their approach is intended to identify malicious behaviors and signatures, and not to identify or classify toolkits used to create the documents.

There is a large body of work on authorship identification in documents, but most of the work focuses on analyzing human languages in plain-text documents, which is not particularly well-suited to the binary content in PDF documents. Frantzeskou et al. developed an approach, called SCAP, which uses binary n-gram analysis to identify human authors of compiled programs written in a variety of programming languages [18]. However, this approach is highly dependent on the programming language being profiled, and it is unknown how effective it may be against the PDF specification language, or how it may interact with embedded content, such as images, which may appear in a PDF document. Additionally, we were not certain if these approaches could be desensitized enough to provide coarse enough results, which would be resistant to changes caused by human-provided input and content.

More closely related is work by Rosenblum et al., related to analyzing compiled programs to identify their compilers [19]. Their approach identifies idiomatic instruction sequences particular to each compiler and uses the sequences to fingerprint the compiler. However, their methodology depends on disassembling binaries from arbitrary byte-offsets, and it is unclear if it could be adapted to work with similar analysis of PDF files.

Most PDF tools are intended to process or manipulate the content of a document, and only a few tools exist that are intended to inspect the structure of PDF documents. Pyew is a Python-based tool for statically analyzing a variety of file formats, including PDFs [20]. Although Pyew does offer some support for structural analysis, it is best suited for extracting and analyzing content from PDF objects. Jose Esparza’s Peepdf tool also offers similar capabilities for PDF documents, and can generate a representation of the logical structure of a document [21]. Oragami-pdf offers similar features, in addition to content modification capabilities [22]. However, both Peepdf and Oragami-pdf are limited to inspecting the logical structure of a document, and not the actual ordering of structural elements as they appear in the raw file Didier Stevens has produced a number of tools intended for analyzing PDF documents, as well as documentation on analysis and obfuscation techniques, largely with a focus on identifying potentially malicious documents [23], [24], [25].

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN AND METHODOLOGY

A. ENVIRONMENT

Our experiments were conducted in a VMWare virtual machine running 32-bit Fedora 16 with one GB of RAM, running on a host with four GB of RAM and a 1.66Ghz Intel Atom processor.

B. METHODOLOGY

1. Document Generation

We generated 57 documents using nine tools (see Table 1) with a variety of settings and inputs to serve as a training set, against which to build and test signatures.

Authoring Tool	Operating System
AbiWord 2.8.6 [26]	Fedora 16
Adobe Acrobat XI Professional [27]	Windows 8
FreePDFConvert.com [28]	Web-based
LibreOffice 3.4 340m1(Build:602) [29]	Windows 8
Microsoft Office 2007 SP3 [30]	Windows 8
Neevia [31]	Web-based
Nitro Pro 8 [32]	Windows 8
PDFOnline [33]	Web-based
WordPerfect X6 [34]	Windows 8

Table 1. Software used to generate sample PDF documents

The documents were generated with one page of text, two pages of text, or embedded images. We created this content natively with some tools, but other tools converted a pre-existing document into a new PDF document (see Table 2). In the case of native functionality, we typed content or inserted images directly into the tool, then saved the document as a PDF file. When using a tool to convert a DOCX [35] file into a PDF, we created an intermediate DOCX file using Microsoft Office 2007. When a tool supported importing a PDF document, the PDF generated by Microsoft Office was used as input. One of the evaluated tools, Nitro Pro 8, supported plaintext ASCII files as input.

Authoring Tool	**Content Generation Methods**
AbiWord	Native (text only)
Adobe Acrobat	Native, import from PDF
FreePDFConvert.com	Import from DOCX
LibreOffice	Native
Microsoft Office 2007	Native
Neevia	Import from DOCX
Nitro Pro 8	Native, import from DOCX, import from ASCII plaintext
PDFOnline	Import from DOCX
WordPerfect X6	Native, import from PDF

Table 2. Software content generation options used

Several tools included additional output options, beyond their default settings. We tested several of these output options (see Table 3). The most common among these include “optimized” or “minimum” formats, referred to in the Adobe PDF Reference as Linearized PDF. The next common output option was to create PDF/A-compatible documents. PDF/A documents use a subset of the features in normal PDF documents, and must embed all resources, such as fonts, into the document, that might normally be expected to be provided by the host system. Finally, Adobe PDFMaker [36] is a generic printer driver plugin that allows software without native PDF-generation capabilities to produce PDF documents through their print functionality. It is installed as a part of the Adobe Acrobat XI Professional suite.

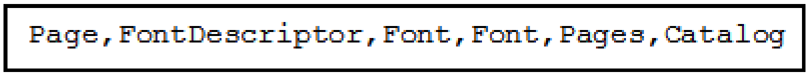
Authoring Tool	**Output Formats**
AbiWord	Default
Adobe Acrobat	Default, Optimized, PDF/A
FreePDFConvert.com	Default
LibreOffice	Default, PDF/A
Microsoft Office 2007	Default, Minimum, PDFMaker
Neevia	Default
Nitro Pro 8	Default
PDFOnline	Default
WordPerfect X6	Default

Table 3. Software output methods and options used

Additionally, we downloaded 5267 PDF documents from the Govdocs1 corpus [37] to serve as real-world examples. These documents were used to test structural signature generation performance on a larger and more complex set of documents than the ones that we generated for training purposes.

2. Structural Signature Generation

We developed two methods for generating structural signatures for PDF documents. Both methods generate a series of tokens, representing an ordering of the types of all of the indirect objects within a document. For convenience, we represent this series as a comma-delimited list (see Figure 6). The tools for both methods produce a list, in this format, as output, which is later used as input for our classification tools. For development purposes, these structural signatures are created once and saved separately from the original document, so that, while testing classification tools and methodologies, the signature did not need to be regenerated each time.



```
Page,FontDescriptor,Font,Font,Pages,Catalog
```

Figure 6. A sample PDF structural signature

The first method leverages Didier Stevens pdf-parser.py tool [23] to parse a PDF document and identify indirect objects. Pdf-parser.py accepts a PDF document as input and produces a listing of object information or document statistics, or extracts the contents of objects. We process the object listing of a document to generate an ordered set of object types, as they appear in the file. We refer to this method later in this document as Method #1. Code for this method is located in Section A of the Appendix.

The second method scans the raw content of a PDF document with a regular expression to identify instances of the keyword "Type," and extracts the object type following the keyword, using a Python program. We refer to this method later in this document as Method #2. Code for this method is located in Section B of the Appendix.

Timing information for both methods was obtained by using the total execution time from the Linux "time" command while generating signatures for our simple generated files, the entirety of our sample from the NPS Govdocs1 corpus [37], and a smaller subset of 969 documents from the NPS corpus. This subset was chosen by grouping documents from the larger sample by their metadata-indicated creator, and selecting the largest group.

3. Developing Structural Signature Classifiers

a. Regular Expressions

By comparing the structural signatures of our training documents, grouped by the tools that were used to create them, we visually identified unique patterns or positions of elements common only to that tool. These patterns were represented as regular expressions. As additional classifiers were built, it became necessary to refine some regular expressions in order to prevent false positives. Some tools required multiple regular expressions in order to capture the signatures of their output.

b. N-gram Analysis

Word n-grams of various sizes were extracted from the structural signatures, and used to build a simple profile for each of the authoring tools. N-grams are sequences of tokens, in this case, of structural elements, of arbitrary length, n. To build a profile, we calculated the probability of a given n-gram appearing in all of the n-grams associated with one tool (see Table 4). This was repeated for all n-gram lengths less than or equal to the length of the longest signature in our dataset.

N-gram	Score
StructTreeRoot-Pages	0.076923
StructElem-StructElem	0.153846
Pages-Catalog	0.115385
Page-Page	0.076923
FontDescriptor-Font	0.115385
Metadata-StructElem	0.076923
Page-OutputIntent	0.076923
Page-Pages	0.038462
OutputIntent-Metadata	0.076923
StructElem-StructTreeRoot	0.076923
Font-Page	0.115385

Table 4. Sample n-gram scores for documents generated with LibreOffice, with n-grams of size two

4. Classifying Document Structures

a. Regular Expressions

After creating regular expressions for each tool, we could test their ability to accurately classify documents that were created by the tools. Our script, classifyStructure.py (see Section C of the Appendix), accepted a comma-delimited list of structural elements as input, as generated by our signature-generation tools, then tested the input against each regular expression. If a regular expression matched the input, we output the respective matching program as a possible match. For the best possible accuracy, we allowed the possibility of multiple matching programs.

b. N-gram Analysis

Given an unknown sample, we extracted n-grams of the same size as those used to generate the profiles. For each extracted n-gram that matched an n-gram in our profiles, we added the score for each associated tool to a running total for each tool. After examining all n-grams in the sample, we returned the highest-scoring program as the most likely tool to have created the unknown sample (see Tables 5 and 6. Five-fold cross-validation was performed to estimate the accuracy of the classifier for several sizes of n-grams.

adobe	0.105882
nitro	0.071429
wordperfect	0.142857
pdfonline	0.238095

Table 5. Sample programs scores for the n-gram “Catalog,Page”

Program	**N-gram**					**Total Score**
	Page FontDescriptor	FontDescriptor Font	Font Font	Font Pages	Pages Catalog	
Abiword	0.200	0.200	0.200	0.200	0.200	1.000
PDFOnline		0.095				0.095
LibreOffice		0.115			0.115	0.231
Nitro			0.095			0.095
Neevia					0.152	0.152
FreePDFConvert					0.029	0.029

Table 6. Sample classification scores for the sample document in Figure 6. using n-grams of size two, indicating that the tool most likely used to create the document was Abiword.

IV. RESULTS AND CONCLUSIONS

A. RESULTS

1. Signature Generation

Both signature generation methods had advantages and disadvantages. The first method took more time to generate a signature, up to tens of seconds, on average (see Tables 7 and 8), but generated more reliable output that is more closely indicative of what a PDF reader program will actually recognize as indirect Type tokens. The second method was significantly faster, but may generate incorrect signatures if the keyword “Type” appears in a context where it does not indicate an indirect object’s type, or if it exists in an invalid object that is not referenced by a cross-reference table. Additionally, some embedded direct objects contained type definitions, but these objects were not extracted by the first method. In our training set, these variations appeared regular and predictable, so we were able to make small changes to our regular expression-based identifying fingerprints to account for the differences with no loss in accuracy.

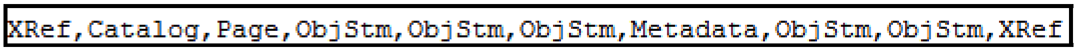
Number of files	Generation Method	Total Time	Average time (per file)
58			
	Parsing (method #1)	1m 14.877s	1.291s
	Raw (method #2)	0m 0.411s	0.007s

Table 7. Time to extract structural signatures for training documents

Number of files	Generation Method	Total Time	Average time (per file)
5267			
	Parsing (method #1)	Approx. 3 days	
	Raw (method #2)	3m 21.547s	0.038s
969			
	Parsing (method #1)	301m25.740s	18.664s
	Raw (method #2)	0m59.363s	0.061

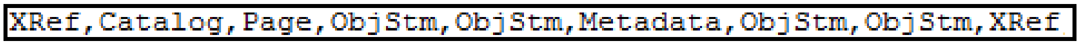
Table 8. Time to extract structural signatures for selected NPS Govdocs1 documents

Documents that were created with identical input to different tools produced different structural signatures. For example, the documents whose structure is represented in Figures 7, 8, 9, and 10. are visually similar when opened with a PDF reader. The structures represented in Figure 7 and Figure 10 were generated from the same document, but with method #1 and method #2, respectively.



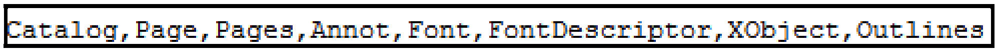
```
XRef,Catalog,Page,ObjStm,ObjStm,ObjStm,Metadata,ObjStm,ObjStm,XRef
```

Figure 7. A structural signature, generated with method #1, for a sample document created with Adobe Acrobat



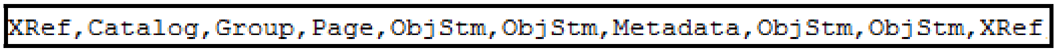
```
XRef,Catalog,Page,ObjStm,ObjStm,Metadata,ObjStm,ObjStm,XRef
```

Figure 8. A structural signature, generated with method #1, for a sample document created with Microsoft Word and saved using Adobe PDFMaker



```
Catalog,Page,Pages,Annot,Font,FontDescriptor,XObject,Outlines
```

Figure 9. A structural signature, generated with method #1, for a sample document created with Nitro8



```
XRef,Catalog,Group,Page,ObjStm,ObjStm,Metadata,ObjStm,ObjStm,XRef
```

Figure 10. A structural signature, generated with method #2, for a sample document created with Adobe Acrobat

2. Document Classification

a. Regular Expressions

We were able to create regular expressions to classify each of our training documents by the tool that created them. One to three object labels was sufficient to identify each of our training files with no false positives or negatives. In most cases, a single regular expression was sufficient to identify all documents associated with a particular tool. Documents created by Nitro Pro 8 carried one of two possible identifying signatures, depending on whether they were created natively within the program, or imported from another file (see Table 9).

Program Name	Regular Expression
AbiWord 2.8.6	`^Page,.*Catalog$`
Adobe Acrobat XI Professional	`^XRef`
FreePDFConvert.com	`^Catalog,Pages,.+Metadata$`
LibreOffice 3.4 340m1(Build:602)	`^FontDescriptor,.*Catalog$`
Microsoft Office 2007 SP3	`^Catalog,.*XRef$`
Neevia	`^Page,.*Catalog,Metadata$`
Nitro Pro 8 (native)	`^Catalog,.*XObject,.*(?:Outlines|XObject)$`
Nitro Pro 8 (converter)	`^Metadata,.*Pages$`
PDFOnline	`Page,Pages(?:$|,FontDescriptor)`
WordPerfect X6	`^Catalog,.+Pages,Metadata$`

Table 9. Regular expressions used for classifying fingerprints generated with method #1

Some identifying fingerprints were easy to develop, but others required refactoring, as new samples were added, to avoid false classifications. For example, the first indirect object in documents generated by Adobe tools, including both Adobe Acrobat and PDFMaker, is always a cross-reference table object, or XRef. A simple regular expression, identifying a single XRef object at the beginning of a document's structure, was sufficient to identify all Adobe-generated documents, and never falsely classified a document that was created by another tool.

b. N-gram Analysis

Our n-gram classification process yielded encouraging results against our training data for a variety of n-gram sizes. Profiles generated with signatures created with method #2 resulted in slightly higher classification accuracy than method #1 when trained and exercised against our entire training set (see Tables 10 and 11, and Figure 11). The accuracy of both methods peaked at around 92% for n-gram sizes of between two and four elements. Even with an n-gram size of one, accuracy was better than 50%, although this may have been an artificiality introduced by our small documents and selection size . The reduction of accuracy at higher n-gram sizes was likely due to the limited size of our sample documents. For example, no 12-grams exist in a document with only 11 structural elements, so we would be unable to train against or classify documents with 11 or fewer structural elements.

n	Number of correctly-classified samples	Classification Accuracy
1	29	0.518
2	51	0.911
3	52	0.928
4	53	0.946
5	50	0.892
6	48	0.857
7	42	0.750
8	34	0.607
9	22	0.393
10	12	0.214
11	7	0.125
12	6	0.107
13	5	0.089
14	3	0.054
15	0	0

Table 10. N-gram classification results against 56 training documents with n-grams of size n and signature generation method #1

n	Number of correctly-classified samples	Classification Accuracy
1	34	0.607
2	52	0.928
3	52	0.928
4	52	0.928
5	48	0.857
6	48	0.857
7	46	0.821
8	39	0.696
9	32	0.571
10	23	0.411
11	13	0.232
12	8	0.143
13	7	0.125
14	5	0.089
15	2	0.036
16	0	0

Table 11. N-gram classification results against 56 training documents with n-grams of size n and signature generation method #2

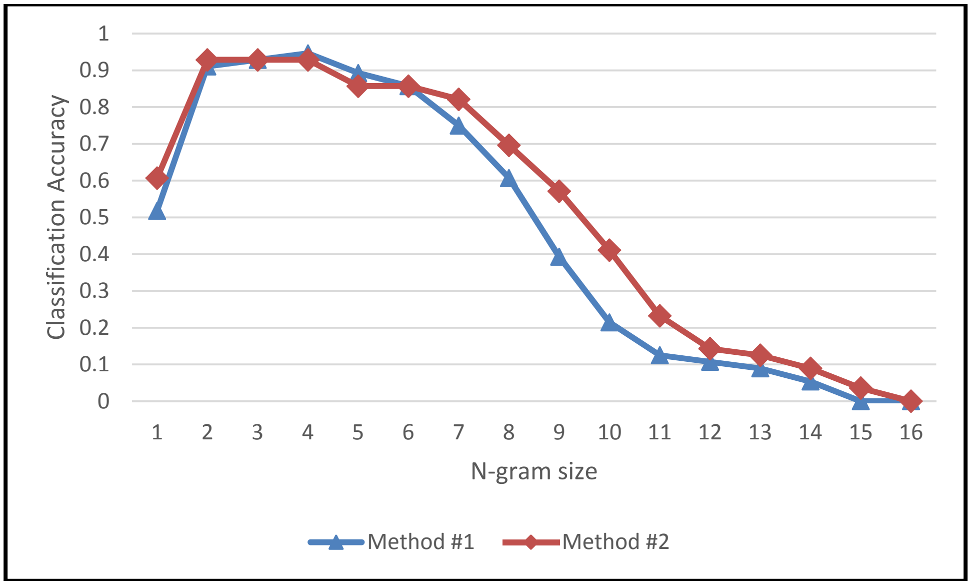


Figure 11. Graph representing n-gram classification accuracy against 56 training documents for n-gram sizes one through 16 and signature generation methods #1 and #2

Five-fold cross-validation yielded similar results (see Figure 12). The long tail is likely due to three documents, which each contained the same 14 indirect object sequences. Because of the limited number of training documents that were created with some tools, some training sets did not contain documents created by all possible tools, resulting in lower accuracy.

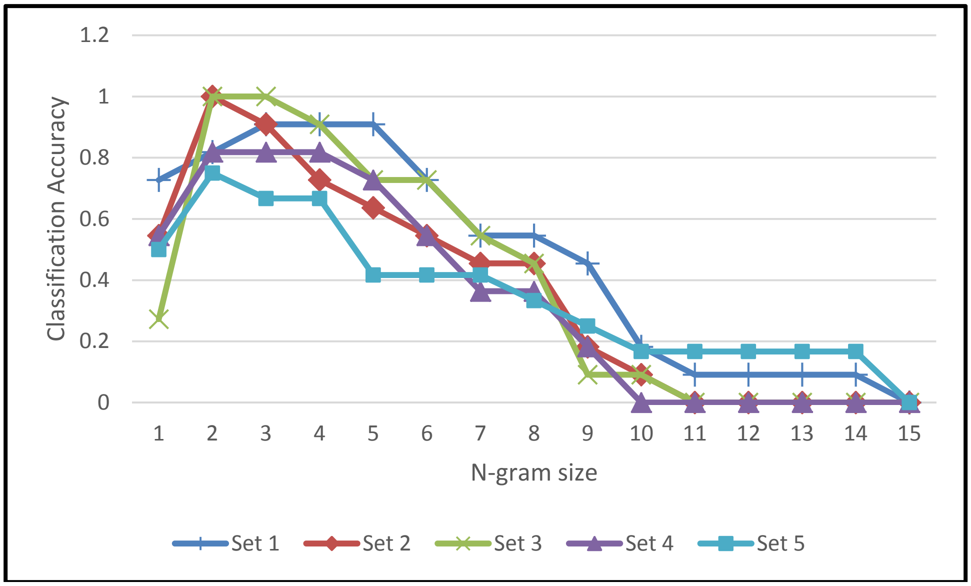


Figure 12. Graph representing n-gram classification accuracy in five-fold cross-validation, per-set, for n-gram sizes one through 16 and signature generation method #1

Using n-grams of size four, further analysis was conducted using five-fold cross-validation to obtain more specific accuracy information for each program (see Tables 12 and 13). The only program that seemed to be classified differently as a result of using different signature generation methods was LibreOffice. It is also interesting to note that the classification accuracy did not seem to be directly correlated with the number of training samples. For example, both AbiWord and Adobe had similar classification accuracies, even though our training corpus only had two samples from AbiWord, and 14 from Adobe tools. In cross-validation with training partitions that only contained one AbiWord-generated document, we were still able to identify another AbiWord-generated document.

Software Name	False-positive	False-negative	True-positive	True-negative
AbiWord	0.000	0.000	1.000	1.000
Adobe	0.000	0.000	1.000	1.000
FreePDFConvert	0.000	0.600	0.400	1.000
LibreOffice	0.000	0.300	0.700	1.000
Microsoft Office 2007	0.062	0.000	1.000	0.938
Neevia	0.020	0.000	1.000	0.980
Nitro	0.000	0.100	0.900	0.800
PDFOnline	0.000	0.300	0.700	1.000
WordPerfect	0.000	0.000	1.000	1.000

Table 12. Per-program n-gram classification accuracy for n-grams of size four and signature generation method #1

Software Name	False-positive	False-negative	True-positive	True-negative
AbiWord	0.000	0.000	1.000	1.000
Adobe	0.000	0.000	1.000	1.000
FreePDFConvert	0.000	0.600	0.400	1.000
LibreOffice	0.000	0.000	1.000	1.000
Microsoft Office 2007	0.062	0.000	1.000	0.938
Neevia	0.020	0.000	1.000	0.980
Nitro	0.000	0.100	0.900	0.800
PDFOnline	0.000	0.300	0.700	1.000
WordPerfect	0.000	0.000	1.000	1.000

Table 13. Per-program n-gram classification accuracy for n-grams of size four and signature generation method #2

B. CONCLUSIONS

Our results indicate that PDF document structure may be a reliable means for identifying the tool that was used to create a document. Structural fingerprints are harder to forge than metadata or content-specific signatures, and would require reordering a document’s objects to obfuscate or obscure. Most of the tools that we used to generate our training corpus create unique fingerprints that are identifiable in documents containing diverse content types. The signature-generation process can be easily scalable to support near-real-time analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

V. FUTURE WORK

We were able to build reliable identifying fingerprints for a small number of tools and content types using documents that we generated (see Table 1). However, given the limited sample size and the simplicity of the generated documents, this is not especially surprising or useful. It remains unclear if these approaches can scale to the larger sample sizes and more complicated documents in such a way that is suitable for real-time classification. When we tested our fingerprints against samples from the NPS Govdocs1 corpus [37], many of our classification results did not match what document metadata indicated. It is possible that these documents had incorrect or misleading metadata, or that our fingerprints could not be generalized to this collection, but, without information about the tools that actually created the documents, we cannot make claims about these hypotheses. Additionally, our ability to make strong claims about the effectiveness of our methods was limited by small sample sizes for each tool. We leave it to future work to build similar fingerprints for other tools and more complex documents.

Of particular interest are documents that are generated by tools that may provide more precise control to users over how a document's structure is constructed. We did not test documents that were generated by tools such as LaTeX or compiled from DVI or PostScript. It may be possible for an adversary to evade or forge structural signatures, such as the ones explored in this thesis, if the compiler does not leave a signature that is distinguishable from user-provided content.

All of the regular expression-based identifying fingerprints were built by hand after visually inspecting the structural signature of each sample documents. This approach would likely not scale well in real-world use, as it requires human interaction to analyze the structures of each of the training documents in order to identify unique patterns and is difficult to convert into a reliably repeatable process..

Our n-gram-based approach yielded encouraging results against our training data, but it is unsure how well the approach scales to larger and more complicated documents. The training documents were simple, with no more than 14 indirect objects each.

Additionally, we did not take into account the positions of each n-gram within the greater structure, which may serve as a more reliable indicator in future work.

It would also be of interest to examine how resistant both the regular-expression and n-gram analysis classification methods are to malicious or targeted subversion of an analyzed document’s structure. If a human is able to craft content in a tool which results in a radically different signature than other documents which were created by the same tool, it may be challenging to accurately classify a document.

There is room for future work in examining the structure of incrementally-modified PDF documents. When we used tools which were able to open and modify existing PDF documents, the new documents bore the signature of the most recent tool that was used to modify the file. However, it is possible for an incrementally-modified PDF document to be modified and contain signatures for multiple tools. When examining the Govdocs1 corpus, we found very few examples of documents that appeared to have been modified in this way, with chained trailers and cross-reference tables, but were unable to determine which tools were used to create them.

We briefly investigated the idea of combining metadata analysis and idiomatic artifact identification with our other approaches to identify suspicious or potentially-subverted documents. If a document’s metadata indicates that it was created with one tool, but the structural signature indicates another, it is possible that an author has modified their document beyond what the original tool generated.

During the process of this research, we discovered a lack of large collections of PDF documents with known provenance. Although there are resources, such as the NPS Govdocs1 corpus, for large numbers of documents found in the wild, we were unable to leverage these documents for reliable training purposes because we were unable to verify which tools created them without blindly trusting the documents’ metadata. There are, occasionally, small collections of documents, associated with a single tool, which are intended to demonstrate that tool’s capabilities. It is essential to future work that large bodies of PDF documents, from a wide variety of tools, with a variety of content types, and of known provenance are available. This corpus should include documents which utilize the full range of capabilities of the PDF specification, such as encryption, forms, and scripted actions, as well as readily comparable documents which were created from identical inputs and have similar appearance. Additionally, tools with different modes of operation, such as ones which allow native content creation, as well as conversion between formats, should be fully explored.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX

A. SCRIPT USED TO GENERATE METHOD #1 SIGNATURES (GETSTRUCTURE.SH)

```
#!/bin/bash
# File: getStructure.sh
# Structural signature generation method #1
# Uses Didier Stevens’ pdf-parser to extract object names
./bin/pdf-parser.py “$1” | grep -a -A 1 “obj” | sed -rn
“s/\s*Type:\s*\/(\S+)/\1/p”
```

B. SCRIPT USED TO GENERATE METHOD #2 SIGNATURES (GETSTRUCTURE.PY)

```
#!/usr/bin/python

# File: getStructure.py
# Structural signature generation method #2
# Extracts “Type” values from the input file(s)
# Written by John Donaldson

import re
import mmap
import sys

typere = re.compile(“Type\s*\/(\w+)”)

def getStructure(filename):
    structure = []
    with open(filename, “rb”) as fileHandle:
        data = mmap.mmap(fileHandle.fileno(),0,prot=mmap.PROT_READ)
        for result in typere.finditer(data):
            structure.append(result.group(1))

    return ,.”“join(structure)

if __name__ == “__main__”:
    for filename in sys.argv[1:]:
        print(getStructure(filename))
```

C. SCRIPT USED TO CLASSIFY STRUCTURAL SIGNATURES WITH REGULAR EXPRESSIONS (CLASSIFYSTRUCTURE.PY)

```
#!/usr/bin/python

# File: classifyStructure.py
# Classifies a PDF document by its structure, using provided fingerprints

import sys
from StructureFingerprints import fingerprints
import getStructure

def classify(structure):
    structure = ,.”“join(structure)
    structure = structure.strip()
    guesses = []
    for fingerprint, name in fingerprints.iteritems():
        if fingerprint.search(structure) != None:
            if name not in guesses:
                guesses.append(name)

    if len(guesses) == 0:
        print(“Couldn’t classify”)
    elif len(guesses) == 1:
        print(guesses[0])
    else:
        print(“-------------Multiple possibilities-----------------”)
        for guess in guesses:
            print(guess)

def classifyFile(filename):
    return classify(getStructure.getStructure(filename))


if __name__ == “__main__”:
    if (len(sys.argv) > 1):
        for filename in sys.argv[1:]:
            classify(map(lambda x: x.strip().lower(), open(sys.argv[1]).readlines()))
    else:
        classify(map(lambda x: x.strip().lower(), sys.stdin.readlines()))
```

D. STRUCTUREFINGERPRINTS.PY

```
import re

# File: StructureFingerprints.py
# Contains fingerprints for identifying PDF document creators
# Fingerprints are only for method #1

reFlags = re.I

fingerprintsBase = {

    “^Catalog,.*xref$”:”Microsoft Office 2007+,”
    “^FontDescriptor,.*Catalog$”:”LibreOffice,”
    “^Page,.*Catalog$”:”AbiWord,”
    “^xref”:”Adobe,”
    “^Page,.*Catalog,Metadata$”:”Neevia,”
    “^Catalog,.+Pages,Metadata$”:”WordPerfect X6,”
    “^Catalog,Pages,.+Metadata$”:”FreePDFConvert.com,”
    “^Metadata,.*Pages$”:”Nitro8-convert,”
```

```
   “^Catalog,.*XObject,.*(?:Outlines|XObject)$”:”Nitro8-native,”
   “Page,Pages(?:$|,FontDescriptor)”:”PDFOnline,”
}

fingerprints = {}

for fingerprint, name in fingerprintsBase.iteritems():
   fingerprints[re.compile(fingerprint,reFlags)] = name
```

E. N-GRAM ANALYSIS AND CLASSIFICATION PROGRAM (NGRAMCLASSIFY.PY)

```
#!/usr/bin/python

from nltk.tokenize import word_tokenize
from nltk.util import ngrams
import os
import os.path

trained = False
trainedData = {}


def getFileTokens(filename):
        tokens = []
        with open(filename,”r”) as infile:
                tokens = word_tokenize(infile.read().replace(,,”““\n”))
        return tokens

def getFileNgrams(filename, ngramSize):
        return(ngrams(getFileTokens(filename),int(ngramSize)))

# Walk through a path, extracting n-grams from files and associating
# them with the directory that contains them. E.g., the file
# ./my_docs/foo/bar.struct would be associated with the program “foo”
def loadProgramNgrams(ngramSize,basepath=.”/,”normalize=True):
        programNgrams = {}
        for root, dirs, files in os.walk(basepath):
                program = os.path.basename(root)
                if  program != ‘‘ and program not in programNgrams:
                        programNgrams[program] = {}
                        for filename in files:
                                filepath = os.path.join(root,filename)
                                currNgrams = getFileNgrams(filepath,ngramSize)
                for ngram in currNgrams:
                    if ngram in programNgrams[program]:
                        programNgrams[program][ngram] += 1
                    else:
                        programNgrams[program][ngram] = 1

        # Normalize
        if program != ‘‘ and normalize:
            totalNgrams = 0
            for ngram,count in programNgrams[program].iteritems():
                totalNgrams += count

            for ngram in programNgrams[program]:
                programNgrams[program][ngram]                    =
programNgrams[program][ngram]/float(totalNgrams)


    return programNgrams



def printNgrams(ngramSize,basepath=.”/”):
    ngramCounts = loadProgramNgrams(ngramSize,basepath)
    for program,ngrams in ngramCounts.iteritems():
        print(“{0}:.”format(program))
        for ngram,count in ngrams.iteritems():
            print(“\t{0} -- {1}.”format(ngram,count))

def getTrainingData(ngramSize,basepath=.”/”):
    global trained
    global trainedData

    # If we have already trained, just return the cached results
    if trained:
        return trainedData

    programNgrams = loadProgramNgrams(ngramSize,basepath,normalize=True)
    ngramProgramScores = {}
    for program, ngrams in programNgrams.iteritems():
        for ngram,score in ngrams.iteritems():
            if ngram not in ngramProgramScores:
                ngramProgramScores[ngram] = []

            ngramProgramScores[ngram].append((program,score))

    trainedData = ngramProgramScores
```

```
        trained = True

        return ngramProgramScores

def  printNgramProgramAssociations(ngramSize,basepath=.”/”):
        ngramProgramScores = getTrainingData(ngramSize,basepath)
        for ngram,programs in ngramProgramScores.iteritems():
                print(“{0}:.”format(ngram))
                for program,score in programs:
                        print(“\t{0} -- {1}.”format(program,score))

def getScores(ngramSize,unknownSample,basepath=.”/”):
        ngramProgramScores = getTrainingData(ngramSize,basepath)
        unknownNgrams = getFileNgrams(unknownSample,ngramSize)
        programScores = {}

        for ngram in unknownNgrams:
                if ngram in ngramProgramScores:
                        for possibleProgram,score in ngramProgramScores[ngram]:
                                if possibleProgram not in programScores:
                                        programScores[possibleProgram] = 0
                                programScores[possibleProgram] += score

        # Normalize scores back down to between 0 and 1. These aren’t percentages,
        # but give us a nicer range of numbers to work with.
        totalScore = 0.0
        for score in programScores.values():
                totalScore += score

        for program in programScores:
                programScores[program] /= totalScore

        return programScores

def classify(ngramSize,unknownSample,basepath=.”/”):
        scores = getScores(ngramSize,unknownSample,basepath)
        mostLikely = ““
        topScore = 0
        # This information is interesting if we care about how close the next-best result is
        secondScore = 0
        secondLikely = ““

        for program, score in scores.iteritems():
                if score > topScore:
                        secondScore = topScore
```

```
            secondLikely = mostLikely
            topScore = score
            mostLikely = program

    if topScore == 0:
        mostLikely = “Unknown”
        topScore = 1.0
    return mostLikely

# Check to see if a sample, whose provenance we know, is correctly
# classified. Returns True if it matches, False if we classified incorrectly
def validate(ngramSize,unknownSample,basepath):
    expected = os.path.basename(os.path.dirname(unknownSample))
    result = classify(ngramSize,unknownSample,basepath)
    print(“Expected {}, got {}.”format(expected,result))
    if expected == result:
        return True
    else:
        return False



if __name__ == “__main__”:
    import sys
    if len(sys.argv) > 2:
        ngramSize = sys.argv[1]
        trainingLocation = sys.argv[2]
        files = sys.argv[3:]
        for filename in files:
            result = classify(ngramSize,filename,trainingLocation)
            print(“{:50}: {}.”format(os.path.basename(filename),result))
    else:
        print(“Usage: {} <ngram size> <path to training data> <files to classify
...>.”format(sys.argv[0]))
        print(“\tMultiple files may be classified at once”)
        print(“\tTraining data is expected to be in the following format, under the
provided path:”)
        print(“\tProvided path \\”)
        print(“\t\tname_of_program1 \\”)
        print(“\t\t\tstruct_file_from_program1_1”)
        print(“\t\t\tstruct_file_from_program1_2”)
        print(“\t\t\t. . .”)
        print(“\t\tname_of_program2 \\”)
        print(“\t\t\tstruct_file_from_program2_1”)
        print(“\t\t\tstruct_file_from_program3_2”)
        print(“\t\t\t. . .”)
        print(“\t\t. . .”)
        print(“\n\tTraining files should contain structural elements, either line- or comma-delimited”)
```

LIST OF REFERENCES

[1] "Digital Corpora >> Govdocs1—simple statistical report," 30 September 2012. [Online]. Available: http://digitalcorpora.org/corpora/files/govdocs1-simple-statistical-report

[2] Symantec Corporation, "Symantec Internet security threat report 2013: Volume 18, Appendix," April 2013. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_appendices_v18_2012_221284438.en-us.pdf

[3] F-Secure, "PDF based targeted attacks are increasing—F-Secure Weblog : News from the Lab," 9 March 2010. [Online]. Available: http://www.f-secure.com/weblog/archives/00001903.html

[4] F-Secure, "PDF most common file type in targeted attacks," 6 May 2009. [Online]. Available: http://www.f-secure.com/weblog/archives/00001676.html

[5] Microsoft Corporation, "Security intelligence report (SIR) vol. 14," 17 April 2013. [Online]. Available: http://www.microsoft.com/security/sir/default.aspx

[6] National Security Agency, "Hidden data and metadata in Adobe PDF files: publication risks and countermeasures," 27 July 2008. http://www.nsa.gov/ia/_files/app/pdf_risks.pdf

[7] Adobe Systems Incorporated, Portable document format reference manual, Addison-Wesley, 1993.

[8] L. Leurs, "The history of PDF | How the file format and Acrobat evolved," 27 January 2013. [Online]. Available: http://www.prepressure.com/pdf/basics/history

[9] Adobe Systems Incorporated, "Document management – portable document format – part 1: PDF 1.7," 1 July 2008. [Online]. Available: http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf

[10] ISO, "PDF format becomes ISO standard," 2 July 2008. [Online]. Available: http://www.iso.org/iso/home/news_index/news_archive/news.htm?refid=Ref1141

[11] *Document management – Electronic document file format for long-term preservation – Part 1: Use of PDF 1.4 (PDF/A-1,),* ISO 19005–1:2005.

[12] “Adobe supplement to the ISO 32000,” June 2008. [Online]. Available: http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/adobe_supplement_iso32000.pdf

[13] Adobe Systems Incorporated, “PDF reference, version 1.7,” November 2006. [Online]. Available: http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/pdf_reference_1–7.pdf

[14] *Document management – Portable document format – Part 1: PDF 1.7,* ISO 32000–1:2008.

[15] Adobe Systems Incorporated, “JavaScript for Acrobat | Adobe developer connection,” [Online]. Available: http://www.adobe.com/devnet/acrobat/javascript.html

[16] D. Stevens, “Escape from PDF | Didier Stevens,” 29 March 2010. [Online]. Available: http://blog.didierstevens.com/2010/03/29/escape-from-pdf/

[17] C. Smutz and A. Stavrou, “Malicious PDF detection using metadata and structural features,” in *Annual Computer Security Applications Conference,* Orlando, 2012.

[18] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, “Effective identification of source code authors using byte-level information,” in *Proceedings of the 28th International Conference on Software Engineering,* Shanghai, 2006.

[19] N. E. Rosenblum, B. P. Miller, and X. Zhu, “Extracting compiler provenance from program binaries,” in *9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering,* New York, 2010.

[20] “PDFAnalysis - pyew - Analysis of malicious PDF files - A Python tool for static malware analysis,” 7 June 2010. [Online]. Available: http://code.google.com/p/pyew/wiki/PDFAnalysis

[21] J. M. Esparza, “peepdf - PDF analysis tool | eternal-todo.com,” 16 May 2011. [Online]. Available: http://eternal-todo.com/tools/peepdf-pdf-analysis-tool

[22] G. Delugré and F. Raynal, “Origami - Sogeti ESEC lab,” Sogeti, 24 May 2011. [Online]. Available: http://esec-lab.sogeti.com/pages/Origami

[23] D. Stevens, “Quickpost: About the physical and logical structure of PDF files | Didier Stevens,” 9 April 2008. [Online]. Available: http://blog.didierstevens.com/2008/04/09/quickpost-about-the-physical-and-logical-structure-of-pdf-files/

[24] D. Stevens, “PDF, let me count the ways... | Didier Stevens,” 29 April 2008. [Online]. Available: http://blog.didierstevens.com/2008/04/29/pdf-let-me-count-the-ways/

[25] D. Stevens, “PDF tools,” November 2008. [Online]. Available: http://blog.didierstevens.com/programs/pdf-tools/

[26] “AbiWord,” AbiSource, 13 June 2010. [Online]. Available: http://www.abisource.com/

[27] “PDF converter, PDF editor, convert to PDF | Adobe Acrobat XI Pro,” Adobe Systems Incorporated, May 2013. [Online]. Available: http://www.adobe.com/products/acrobatpro.html

[28] Baltsoft Software, “PDF Converter - convert to PDF online free,” Baltsoft Software, 3013. [Online]. Available: http://www.freepdfconvert.com/

[29] “Home >> LibreOffice,” The Document Foundation, 4 April 2013. [Online]. Available: http://www.libreoffice.org/default/

[30] “Office - Office.com,” Microsoft, 25 October 2011. [Online]. Available: http://office.microsoft.com/en-us/

[31] “Free online PDF converter, batch convert doc, docx to PDF, PDF/A or image, doc converter,” Neevia Technology, 2013. [Online]. Available: http://docupub.com/pdfconvert/

[32] “Create, convert & edit PDF files | Nitro,” Nitro PDF Pty. Ltd., 21 April 2013. [Online]. Available: http://www.nitropdf.com/

[33] “Online PDF converter — create PDF & convert PDF to word —free!,” BCL Technologies, 2013. [Online]. Available: http://www.pdfonline.com/convert-pdf/

[34] “Corel WordPerfect Office – wordperfect.com,” Corel Corporation, 26 April 2013. [Online]. Available: http://www.corel.com/corel/pages/index.jsp?pgid=12100162

[35] *Information technology – Document description and processing languages – Office Open XML File Formats – Part 4: Transitional Migration Features,* ISO/IEC 29500, 2012.

[36] Adobe Systems Incorporated, “Adobe - Acrobat : for Windows : Adobe PDFMaker 1.0 for Microsoft Word 97 (Final),” Adobe Systems, 14 January 1998. [Online]. Available: http://www.adobe.com/support/downloads/detail.jsp?ftpID=483

[37] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, “Bringing science to digital forensics with standardized forensic corpora,” *Digital Investigation*, no. 6, pp. S2-S11, 2009.

[38] W.-J. Li, K. Wang, S. Stolfo, and B. Herzog, “Fileprints: identifying file types by n-gram analysis,” in *Proceedings from the Sixth Annual IEEE SMC*, West Point, 2005.

[39] D. Cao, J. Luo, M. Yin, and H. Yang, “Feature selection based file type identification algorithm,” in *Intelligent Computing and Intelligent Systems, 2010 IEEE International Conference on*, Xiamen, 2010.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California