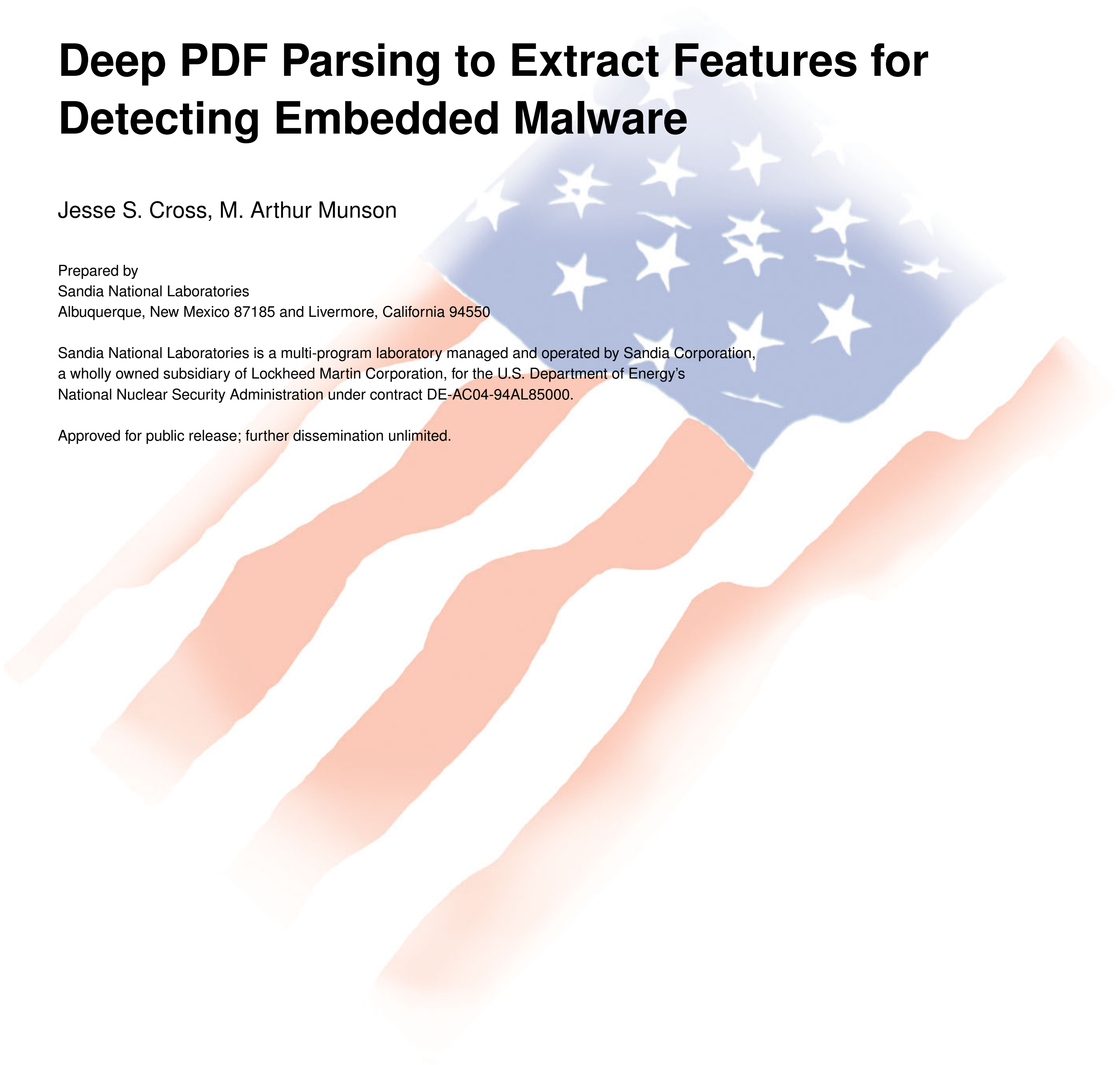
SANDIA REPORT

SAND2011-7982
Unlimited Release
Printed September 2011

Deep PDF Parsing to Extract Features for Detecting Embedded Malware

Jesse S. Cross, M. Arthur Munson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550



Sandia National Laboratories

SAND2011-7982
Unlimited Release
Printed September 2011

Deep PDF Parsing to Extract Features for Detecting Embedded Malware

Jesse S. Cross
Emerson Electric Co. Hall
Missouri University of Science and Technology
301 W. 16th St., Rolla, MO 65409-0040
jsctx5@mst.edu

M. Arthur Munson
Sandia National Laboratories
P.O. Box 969, MS 9159
Livermore, CA 94551
mamunso@sandia.gov

Abstract

The number of PDF files with embedded malicious code has risen significantly in the past few years. This is due to the portability of the file format, the ways Adobe Reader recovers from corrupt PDF files, the addition of many multimedia and scripting extensions to the file format, and many format properties the malware author may use to disguise the presence of malware.

Current research focuses on executable, MS Office, and HTML formats. In this paper, several features and properties of PDF Files are identified. Features are extracted using an instrumented open source PDF viewer. The feature descriptions of benign and malicious PDFs can be used to construct a machine learning model for detecting possible malware in future PDF files.

Acknowledgment

The authors thank Ken Chiang and Michael Karres for providing the PDF samples used in this study, and Christopher Nebergall for helpful discussions. This work was funded by the Laboratory Directed Research & Development (LDRD) program at Sandia National Laboratories, a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Contents

Appendix

Figures

Tables

1 Introduction

The first version of the Portable Document Format (PDF) was released in 1993. During the 1990s, PDF was seen as a secure file format that was recommended over other formats such as MS Word due to vulnerabilities in macros. In 2001, the first PDF virus, known as Peachy, was released [6]. Peachy was only activated when opened by Adobe Acrobat Professional and not by Adobe Reader. This virus set the stage for many other malicious exploits for the PDF. Since 2001, Adobe has extended the format with additional compression and encryption algorithms, scripting support, and multimedia support (e.g., embedded Flash). All of these additions can be used to embed malware in a PDF file or hide the presence of embedded malware.

The detection rate of PDF malware by current antivirus software is very low. A PDF file is easy to edit and manipulate because it is a text format, providing a low barrier to malware authors. Analyzing PDF files for malware is nonetheless difficult because of a) the complexity of the formatting language, b) the parsing idiosyncrasies in Adobe Reader, and c) undocumented correction techniques employed in Adobe Reader. In May 2011, Esparza demonstrated that PDF malware could be hidden from 42 of 43 antivirus packages by combining multiple obfuscation techniques [4]. One reason current antivirus software fails is the ease of varying byte sequences in PDF malware, thereby rendering conventional signature-based virus detection useless. The compression and encryption functions produce sequences of bytes that are each functions of multiple input bytes. As a result, padding the malware payload with some whitespace before compression / encryption can change many of the bytes in the final payload.

In this study we analyzed a corpus of 2591 benign and 87 malicious PDF files. While this corpus is admittedly small, it allowed us to test a system for collecting indicators of embedded PDF malware. We will call these indicators **features** throughout the rest of this report. The features are extracted using an instrumented PDF viewer, and are the inputs to a prediction model that scores the likelihood of a PDF file containing malware. The prediction model is constructed from a sample of labeled data by a machine learning algorithm (specifically, decision tree ensemble learning). Preliminary experiments show that the model is able to detect half of the PDF malware in the corpus with zero false alarms. We conclude the report with suggestions for extending this work to detect a greater variety of PDF malware.

2 PDF File Format

The PDF is published as an ISO Standard and Adobe updates its own extensions to the format [5]. A PDF file is composed of a header, a series of objects, a cross reference (xref) table, a trailer section, and an end of file (EOF) marker. PDF files can be incrementally updated. Such incremental updates append more objects, cross reference tables, trailers, and EOFs after the first EOF.

The PDF header is of the form `%PDF-1.X` where X is between 1 and 7, inclusive. The objects are of the form:

```
num gen obj ... endobj
```

where `num` and `gen` are the number and generation of the object, respectively. The `obj` and `endobj` tags define the object start and end boundaries.

Table 1. XRef Table Example

```
xref
 0      5
 0    65535   f
 10     0     n
 20     0     n
200     0     n
600     0     n
```

The xref table form is shown in Table 1. The first line contains the identifier label, `xref`. The second line shows that there are five objects in the PDF file and the object 0 is the first object. The remaining lines are of the form:

file offset in bytes, revision number, free/in use marker.

The trailer object contains information about how to parse the xref table. It is immediately followed by the `startxref` tag which specifies the byte offset of the xref table in the PDF file.

3 Antivirus Evasion

The PDF has many features conducive to malware such as morphologic manipulation, ease of file cloaking, encryption, compression, and attempted forward compatibility. **Morphologic manipulation** is evident in that every character in tags used in the PDF specification may be represented by their hex or octal equivalent. For example, `/OpenAction` is equivalent to `/Open#41ction`. Obfuscation like this can thwart a simple text search for tags of interest (e.g., `/JavaScript`).

File cloaking is possible because Adobe PDF Reader will accept a PDF that has up to 1 kB of garbage before the PDF header, even though this contradicts the ISO standard. The beginning of the cloaked file looks like another file type (e.g., an image file), but if the file is opened by Adobe Reader the PDF contents are displayed (ignoring the non-PDF preamble). A cloaked malware PDF file may avoid being scanned for viruses. This presents many problems.

- First, PDFs are capable of accessing external data, including external PDFs. An apparently benign PDF can open a cloaked PDF file (downloaded as a non-PDF file) and execute its exploit.
- Second, an attacker can use social engineering to convince an authority to digitally sign a PDF file cloaked as a harmless picture (e.g., a fluffy kitten). Once the picture is signed, the attacker can change the file extension to `.pdf`, creating a digitally signed PDF malware.

The PDF offers a variety of compression and encryption algorithms. Only streams (arrays of data) are compressed or encrypted; however, it is possible to embed an entire malicious PDF file into a stream. Both compression and especially encryption are able to change many of the resultant bits with a simple change like adding whitespace. The ease of these changes makes it extremely difficult to generate useful antivirus signatures. Documents encrypted with the null password will open transparently to the user in Adobe Reader.

Finally, forward compatibility attempts in the format simplify hiding malware in PDF documents. The PDF specification mandates that current PDF readers should ignore any unrecognized data. This is supposed to make a document generated with a later version of PDF partially viewable using an older PDF reader. An unintended side effect is the ability to hide parts of malicious code between objects. Because the between-object data is unexpected, Adobe Reader blithely ignores the code snippets. Current examples of malware are capable of calling the added malicious code.

4 Malicious Features

The features collected from the PDF describe its look and content, and are listed in Table 2. The presence of any of these features will not conclusively identify a PDF file as malware. Instead, the hope is that a machine learning algorithm can identify statistical feature patterns that, as a group, are reliable indicators of whether a PDF file should be categorized as benign or malicious.

Table 2: Features Collected

FEATURE	TYPE	DESCRIPTION
xref_length	Integer	Size of the xref table.
xref	Boolean	Does the xref table exist? Absence indicates poorly formed file.
startxref	Boolean	Is there a startxref? Absence indicates poorly formed file.
trailer	Boolean	Does the trailer section exist? Absence indicates poorly formed file.
repaired	Boolean	Was the PDF repaired in order to correctly parse?
num_pages	Integer	Number of pages. In our corpus, malicious PDF files tend to have lower page counts.
OpenAction	Integer	Does the PDF perform an action upon opening? If so, how many tags were used?
AA	Integer	Does the PDF respond to user actions?
JavaScript	Integer	How many blocks of JavaScript are in the PDF?
JavaScript_in_OpenAction	Boolean	Does JavaScript run when the PDF is opened?
JavaScript_in_AA	Boolean	Does JavaScript run in response to a user action?
JS	Integer	Number of single line JavaScript statements in PDF.
JS_in_OpenAction	Boolean	Does JavaScript run when the PDF is opened?
JS_in_AA	Boolean	Does JavaScript run in response to a user action?
JBIG2Decode	Integer	How many times is the JBIG2Decode filter used? (This filter was used in a recent zero day attack on PDF Reader.)
Launch	Integer	In how many places does the PDF attempt to launch a program / script?
RichMedia	Integer	How many blocks of Flash content are embedded?
XML	Integer	How many times is XML used?
EmbeddedFile	Integer	How many files are embedded in the PDF?
XObject	Integer	Number of XObjects in the PDF.
ObjStm	Integer	Number of object streams in the PDF. Object streams may be used to hide tags associated with malicious PDF files.
Names	Integer	How many catalog names listings are there?

An open source PDF reader, MuPDF 0.8.165, was re-instrumented to extract the features in Table 2. This tool exceeds the parsing capabilities of the current publicly available PDF analysis tools. Some malware uses many layers of filters and a very liberal interpretation of the PDF standard to confuse current analysis tools. As a full PDF parser capable of rendering PDF documents, MuPDF is capable of cutting through many layers of obfuscation. In contrast, current publicly available PDF analysis tools are script-based (e.g., Python or Ruby) and do not implement the full range of PDF compression methods, encryption methods, document repair, etc. that is necessary to see past today's advanced malware obfuscation. Because it is written in C, MuPDF also parses PDF files much faster than the script analysis tools—despite parsing files more thoroughly. The instrumented MuPDF is currently capable of analyzing about 300 files per minute on average. Appendix A describes the organization of the MuPDF source code and the modifications made to instrument the reader.

5 Preliminary Machine Learning Results

To check the feasibility of detecting PDF malware with these features, we trained a machine learning model from half the data and tested the model on the other half of the data. We used bagging [1] to construct an ensemble of Hellinger distance decision trees [2]; this learning algorithm is robust to class imbalances in the training data [2]. The model takes as input the feature description of a PDF file (produced by MuPDF) and outputs the estimated probability that the file is malware.

We used 5x2 cross-validation [3] for the experiment. This design repeats two-fold cross-validation five times. In each of the five runs, the data are randomly divided into two equal sized sets, aka two data folds, that we denote as A and B. A model is trained using fold A and tested on fold B, and then a second model is trained on fold B and tested on fold A. This ensures that every data point is used for testing — either for the first model or the second model. The process is repeated five times with different random data splits.

Because only 87 of the 2677 examples are malware, an accuracy of 96.8% is trivially obtained by always predicting non-malware. Instead, we measure the model's performance on the test data using precision and recall. Precision is the percentage of predicted malware cases that are actually malware. Recall is the percentage of actual malware cases detected as malware by the model. Formally,

$$\text{precision} = \frac{\text{\# correct malware detections}}{\text{\# malware predictions}} \qquad \text{recall} = \frac{\text{\# correct malware detections}}{\text{\# malware}}$$

Note that perfect recall can be obtained by predicting that all files are malware, and in general higher precision can be obtained by only predicting a file is malware for obvious cases. Further, we can trade off precision for recall, and vice versa, by adjusting the threshold for declaring a case malware. For example, instead of declaring a file malware if the predicted probability exceeds 0.5 (the default threshold), we might choose a threshold of 0.75 to reduce the number of false positives. Ultimately, a range of precision-recall performance tradeoffs can be achieved with a model by using different thresholds.

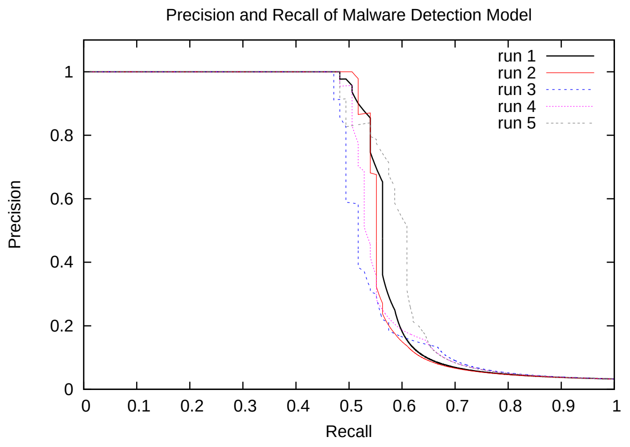


Figure 1. Roughly half of PDF malware in corpus can be detected with zero false positives. More malware can be detected (higher recall) using a different threshold (see text), but false positives increase rapidly (lower precision).

Figure 1 shows the precision-recall tradeoff curves for the malware detection model for five different runs. In all five runs, slightly less than half the malware can be detected (recall < 0.5) with perfect precision. Recall can be increased at the expense of adding false alarms, but the model's precision drops precipitously. The exact location of the steep drop varies by run.

6 Discussion

Many features are present in both malicious and benign PDF files. The results from the previous section show that a machine learning algorithm can correlate multiple features to categorize previously unseen PDF files. There is much room for improvement in these preliminary results. Future research should examine the errors of the current model to determine why half of the PDF malware is incorrectly categorized. Two potential solutions are training a model from a larger corpus, and defining new features to characterize the differences between benign PDFs and the non-detected malware.

In our manual analysis of the corpus, the features JavaScript_in_OpenAction and JS_in_OpenAction stood out as a clear indicator of malicious intent. None of the benign PDF files executed JavaScript upon opening, while many of the malicious PDFs did exactly this. External research confirms that it is common for malicious PDF files to execute JavaScript upon opening. Surprisingly, the machine learning models did not heavily use these features. The models did, however, use the JavaScript features to help categorize files as benign or malicious. Further investigation is needed to a) understand why the models did not take greater advantage of JavaScript_in_OpenAction and JS_in_OpenAction, and b) assess the broader reliability of these features in larger data sets.

The current data set contains additional patterns that could be exploited to detect more PDF malware. It is evident that some malicious files were created with an exploit kit. In these files, the file structure and objects were nearly identical except for the JavaScript contained in one object. It should be possible to look for signatures to identify the output of exploit kits, in order to hamper the malicious writers ability to quickly churn out PDF files with identical formats but different behavior.

Finally, many additional features could be collected by the MuPDF parser to detect potential malware. Known exploit mechanisms that could be easily instrumented include:

- The `Colors` tag is followed by a value greater than 2^{24}.
- A `Fonts` tag specifier string is too long.
- The range specifier for a TIFF image is incorrect.

If any of the above features exist, the PDF file should be considered malicious or corrupted to the point where it should not be rendered. While these examples have been patched by Adobe, malware authors continue to use old exploits since many computers still run unpatched versions of Adobe Reader.

7 Conclusion

Reliable detection of features in a PDF requires complete parsing. Current PDF analysis tools are not successful when multiple obfuscation techniques are combined to hide malicious intent. The reinstrumented MuPDF is capable of the level of parsing required to defeat PDF language based obfuscation. In our corpus, the execution of JavaScript upon file opening is a clear indicator of potential PDF malware. The complete lack of JavaScript execution upon opening in the benign corpus suggests that this feature is indicative of malware with a very low (or zero in the case of the current data set) false positive rate. The other features do not, individually, separate benign from malicious PDFs, but our preliminary results suggest that machine learning algorithms can synthesize these noisy features to identify PDF malware.

References

[1] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[2] D. A. Cieslak, T. R. Hoens, N. V. Chawla, and W. P. Kegelmeyer. Hellinger distance decision trees are robust and skew-insensitive. *Data Mining and Knowledge Discovery*, pages 1–23, 2011.

[3] T. G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998.

[4] J. M. Esparza. Obfuscation and (non-)detection of malicious PDF files. Presented at CARO 2011 Workshop in Prague, Czech Republic., May 2011. Slides downloaded from `http://securityblog.s21sec.com/2011/05/obfuscation-and-non-detection-of.html`.

[5] PDF specifications. `http://www.adobe.com/devnet/pdf/pdf_reference.html`. Accessed 2011/8/8.

[6] VBS/PeachyPDF@MM - malware, Aug. 2001. [online] `http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=99179`.

A MuPDF Modifications

This appendix describes the high level organization of the MuPDF source code and the changes made to instrument the PDF reader for feature extraction.

A.1 MuPDF Directory and File Structure

MuPDF may be compiled for X11, win32, and Android. The modifications do not remove this ability; however, no testing outside of Ubuntu has been conducted. Project files for Debian, win32, and Android, exist in directories of the same name.

The main functions for each system exist in MuPDF_ROOT/apps/. Any modifications to the re-instrumented PDF reader should not be done inside the win_main.c and x11_main.c files in order to maintain portability. These files contain OS specific wrappers to the generic pdfapp.c file. The file, pdfapp.c, should be modified in their place. Other .c files for separate support utilities exist in this directory. Currently, they are not in use for this project.

The PDF parser source is located in MuPDF_ROOT/pdf/. The primary files of interest are pdf_parse.c, pdf_xref.c, and pdf_repair.c. These files are responsible for parsing the pdf and repairing it in the case of corruption. Modifications to these files are described below.

MuPDF makes use of the Fitz graphic library. No other applications appear to use this library. The source of this library is contained in MuPDF_ROOT/fitz/. All objects and functions imported from the library are preceded by 'fz_'. Many of the support /functions and objects such as decompression, dictionaries (hash functions), rendering, encryption, and the base object format are defined in these source files. No parts of the library need to be modified for this project.

A.2 PDF Parsing Method

The entire PDF is loaded into an array of dictionaries before the first page is shown. Each object in the PDF file is loaded into its own dictionary in the array. Dictionaries are hash functions in this context.

The file, pdf_xref.c, contains the logic for loading the PDF. It first searches the trailer section for the location of the xref table which contains the file offsets off each object in the PDF file. In the event that the xref table cannot be located; the trailer section is malformed; or the file offsets are incorrect, MuPDF will attempt to repair the PDF using the functions in pdf_repair.c. In the case of repair, the reader will search for each object, construct a new xref table, and then continue parsing normally.

Once all of the objects are located, each object will be parsed and the tags will be loaded into the dictionary as a set of keys and values. Each key-value pair has its own unique index which allows the user to loop through every pair in a dictionary. The value may be a reference to another dictionary. For example, `/Length 1234` will be loaded with `Length` as a key and `1234` as the corresponding value. When the value happens to be a reference to another dictionary, it may be read in the same manner as the parent dictionary. All objects will be read in this manner. The index in the xref table will determine the index of each object in the dictionary array. After the dictionary array is populated, the first page will be shown and the program will begin to monitor for user input.

A.3 Instrumentation Modifications

The modifications done to MuPDF include removing the display ability, scanning for features, and writing the features list out to a file. Many other files have sections that are commented out. These sections display parsing data that were used to make sense of the method MuPDF uses to parse.

In order to stop MuPDF from entering the user input and display loop, the message loop in x11_main.c was stopped by immediately setting its closing parameter to false. The same thing may be accomplished in win_main.c in order to use this tool with Windows. This has the effect of completely parsing the PDF file, writing data to disk, and immediately exiting.

The functions in pdf_xref.c associated with loading the xref table and object such as `pdf_read_xref()` have been modified to accept the features structure. This gives the functions the capability to save features associated with the xref table such as whether it was repaired or if the PDF was formatted correctly.

The remaining modifications took place in pdf_app.c. A new function, `list_dict()`, was defined. The remaining features are saved to the features structure in this function. The structure is appended to a file in pdfapp_open.

A.4 Usage Instructions

To run the instrumeted reader, run:

```
mupdf filename
```

where `filename` is a PDF to be scanned. After scanning is complete, MuPDF will append to or create a file, features.csv. This file is a comma separated value file that contains the values of the measured features. Every time MuPDF is run a new line is appended that contains the values for each file. The file, features.names, shows the names of each feature and the values allowed for that feature. A python script, loop.py, located in MuPDF_ROOT/ is provided for easy directory based analysis. The analysis directory will need to be changed in the source. Once this script is executed, MuPDF will be called once for each file in the analysis directory and append the data generated from each file to features.csv.

DISTRIBUTION:

1	MS	0813	Christopher Nebergall, 9312
1	MS	0932	Joey Ingram, 9515
1	MS	1316	Steve Plimpton, 1426
1	MS	9011	Ken Chiang, 8966
1	MS	9159	Philip Kegelmeyer, 8900
1	MS	0899	RIM — Reports Management, 9532 (electronic copy)
1	MS	0359	D. Chavez, LDRD Office, 1911

v1.36

Sandia National Laboratories