PDF Mirage: Content Masking Attack Against Information-Based Online Services

Ian Markwood[*†], Dakun Shen[*†], Yao Liu[†], and Zhuo Lu[†]

[†]University of South Florida, Tampa, FL, U.S.A
[*]Co-First Authors

Abstract

We present a new class of content masking attacks against the Adobe PDF standard, causing documents to appear to humans dissimilar to the underlying content extracted by information-based services. We show three attack variants with notable impact on real-world systems. Our first attack allows academic paper writers and reviewers to collude via subverting the automatic reviewer assignment systems in current use by academic conferences including INFOCOM, which we reproduced. Our second attack renders ineffective plagiarism detection software, particularly Turnitin, targeting specific small plagiarism similarity scores to appear natural and evade detection. In our final attack, we place masked content into the indexes for Bing, Yahoo!, and DuckDuckGo which renders as information entirely different from the keywords used to locate it, enabling spam, profane, or possibly illegal content to go unnoticed by these search engines but still returned in unrelated search results. Lastly, as these systems eschew optical character recognition (OCR) for its overhead, we offer a comprehensive and lightweight alternative mitigation method.

1 Introduction

Designed as a solution for displaying formatted information consistently on computers with myriad hardware and software configurations, Adobe's Portable Document Format (PDF) has become the standard for electronic documents. Academic and collegiate papers, business write-ups and fact sheets, advertisements for print, and anything else meant to be viewed as a final product make use of the PDF standard. Indeed, there is an element of constancy implied in the creation of a PDF document. End users cannot easily change the text of a PDF document, so most come to expect a degree of integrity present in all PDF documents encountered.

Attacks are studied and corresponding defenses developed dealing with arbitrary code execution through some allowances made by Adobe to execute JavaScript within the rendering process of a PDF file [1] [2] or from other rendering vulnerabilities [3] [4]. These typically allow data exfiltration, botnet creation, or other objectives unrelated to the PDF file itself aside from using it as a delivery mechanism [5] [6] [7] [8]. We present a class of attacks against the *content* integrity of PDF documents themselves, and following this, describe and test a comprehensive defense method against these attacks. Without changing the appearance of a PDF, we are able to alter how several information-based services see it, with the following implications:

1. We demonstrate how academic paper writers can collude with multiple conference reviewers, by altering a paper invisibly to humans, to be assigned to those reviewers by automatic reviewer assignment systems, such as that used by the IEEE International Conference on Computer Communications (INFOCOM) [9] that openly publishes its automated algorithm. We simulate this reviewer assignment system using 100 sample academic papers and a corpus of 2094 papers from 114 reviewers of a past security conference, finding that we can cause any of said sample papers to match with any reviewer.

2. We show how an unethical student can invisibly alter a document to avoid plagiarism detection, namely the dominant market share Turnitin [10], and generalize methods to target specific small plagiarism similarity scores to simulate the few false positives such systems typically detect. We illustrate this attack by inducing plagiarism scores, as measured by Turnitin, from 0-100% in 10 academic papers without changing their appearance.

3. Lastly, we show real-world examples of making leading search engines display arbitrary (potentially spam, offensive material, etc.) results for innocuous keywords. We have successfully caused Bing, Yahoo!, and DuckDuckGo to index five documents under keywords not displayed in those documents.

These systems have in common the need to scrape

PDFs for their content for further processing or searching within. Online conference paper or other document repositories and companies that index the Internet require text from PDFs so they may be located via search. Natural language processing tools scrape PDFs to discover the topics within, and this information is used in several large conferences to assign unpublished work to conference reviewers as well as in document repositories to categorize large volumes of works without manual effort. Finally, plagiarism checkers require text from new articles for comparison against currently published work to detect impermissible similarity.

Scraping of PDF documents can be done in an automated setting by text extraction tools such as the PDFMiner package [11]. However, fonts of any name may be embedded in the PDF document, and these tools cannot check the fonts' authenticity. A font is actually akin to an encoding mechanism, which maps keys pressed on a keyboard to glyphs representing those keys. Without some way to check the validity of fonts in a PDF, which glyphs a font maps keys to is arbitrary. Moreover, humans reading a PDF read the rendered version of what a tool such as PDFMiner reads, meaning that machines and humans are on opposite ends of this encoding mechanism and may be caused to read different information.

Consequently, the various PDF document scraping environments may be misused through the remapping of keys to arbitrary rendered glyphs. Using one or more custom fonts, an attacker may cause a word to be rendered as another word by switching the glyph mapping within the font file, or rather change the underlying text while keeping a constant rendered output. That is to say, in a document containing the word "kind" an attacker may force that to be rendered as "mean" with a custom font mapping *k* to *m*, *i* to *e*, *n* to *a*, and *d* to *n*, so the human now sees "mean" while the machine still sees "kind"; or to avoid human detection an attacker can change the underlying text to "mean" and use a font with the reverse mapping to render it as "kind" for the human to see. The latter tactic subverts aforementioned end applications, while still rendering PDFs in all appearances normal to humans. We refer to this as a *content masking* attack, as humans are caused to view a masked version of the content these computer systems read.

To assign papers to reviewers for a conference, several large conferences employ automated systems to compare the subject paper with a corpus of papers written by each reviewer to find the best match. This matching is executed upon the most important topics, or keywords, found in the paper via natural language processing methods. If an author replaces the keywords of a paper with those of a reviewer's paper, a high match is guaranteed, and the two may thereby collude. By creating custom glyph mappings for characters, the masked paper can make perfect sense to the human eye, while the underlying text read by the machine has many substituted words which would not make sense to a human reader. This exploit has the technical challenges of replacing words of differing lengths (larger and smaller replacements require different methods) and also constructing multiple fonts required for different mappings of the same letter (for example, to map the word "green" to "brown" requires two different font mappings for *e*). A naive defense could check the number of fonts embedded, so in Section 4 we design algorithms to minimize the number of auxiliary fonts used, in order to avoid detection. To evaluate, we construct our own automatic reviewer assignment system reproducing the current INFOCOM system [9], and show that for 100 test papers, targeting a specific reviewer is possible by masking 4-9 unique words in most papers and no more than 12 for all tested.

This content masking attack also undermines plagiarism detection. In this case, we need only switch out isolated characters to change plagiarized text to text never written before, while again masking these changes as the original text to the human reader. In fact, as most papers have a small (false positive) percentage of similarity present due to common phrases within the English language, this method simulates that by varying the number of characters changed, to simulate the usual small but nonzero plagiarism percentage. Only one font is required to make this mapping, as the resultant text does not need to make sense to the plagiarism detector. Thus, say, all rendered *e*'s may be represented by some other letter in a font that maps that key to the glyph *e*, and other letters may be changed similarly, building a one-to-one mapping covering at most all letters. The challenge is to target a small plagiarism percentage, but accomplishing that as we do in Section 5, a single embedded font bearing the name of a popular font will cause no suspicion.

Finally, search engines and document repositories may be subverted to display unexpected content also. Here, we may replace the entire text of a PDF without changing the rendered view, with a variety of implications. One may hide advertisements in academic papers or business fact sheets, for example, to spam users searching for information. In this exploit, the attacker should replace an entire document with the fewest number of fonts necessary, to avoid seeming particularly unusual. This must be done in a different way than for the topic matching exploit, due to changing the entire document rather than a few words, so we outline another method in Section 6. We then test it on popular search engines, finding that Yahoo!, Bing, and DuckDuckGo are susceptible.

Having enumerated these vulnerabilities, as these systems eschew optical character recognition (OCR) for its overhead, we offer a comprehensive and lightweight alternative mitigation method in Section 7. While a naive

method would perform OCR over the full document, we instead render the unique characters used within the document and perform OCR on these. This font verification method has several technical challenges in its implementation, due to the number and variety of glyphs within font files, and all these issues are overcome in the algorithm we provide. We find it performs at a roughly constant speed regardless of document length (a tenth of that for full document OCR at 10 pages), with glyph distinction accuracy just under 100%, and with 100% content masking attack detection rate.

2 Background Information

PDF Text Extraction: The Adobe PDF standard contains eight basic types of objects, including strings. Strings house the text in a document, including plain text, octal or hexadecimal representations of plain text, or text with some type of encoding [12]. PDF rendering software treats each string as a series of character identifiers (CIDs), each mapping to its corresponding glyph within the font associated with that string via the Character Map (CMap) [13]. A series of glyphs is thus displayed.

Text information extracted from PDF files by using tools like the Python package PDFMiner. These tools extract text by copying the plaintext from all string objects in a PDF file. Though these tools can extract the font name for each string as well, a whitelist will not defend against this attack, as fonts may be given any name.

Topic Matching: The exponential growth of human knowledge/record keeping and the ease of its access demands an efficient means of providing context-relevant search results, stemming the research field of natural language processing. This field extracts the specific subject of a document without the need for human classification. The ultimate goal of useful search results prompts the companion research field of matching keywords to topics which has been tackled by the leading search engines.

Latent Semantic Indexing (LSI) is a popular natural language processing algorithm for extracting topics from documents. The LSI approach infers synonymous words/phrases to be those with similar surrounding contexts, rather than constructing a thesaurus. These detected patterns can allow singular value decomposition to reduce the number of important words in a document such that it may be represented by a small subset. This small subset, of cardinality k, then contains frequency data for each element, such that the document may be represented by a dot in k-space. Similarity between documents is easily calculated via their Euclidean distances apart in this geometric representation [14].

Latent Dirichlet Allocation is a newer popular topic extraction algorithm, which is generally speaking a probabilistic extension of LSI [9]. Topics are generated as collections of related words, using supervised learning. The probability of a document corresponding to each of the predefined topics is calculated based on how well the words within the document correspond to the words within each topic [15, 16].

Topic matching is used within the automation of the review assignment process for several large conferences, such as the ACM Conference on Computer and Communications Security (CCS) or the IEEE International Conference on Computer Communications (INFOCOM). These conferences receive many submissions and have many reviewers, and the manual task of finding the most suitable reviewers for each paper is onerous, so they automate by comparing topics extracted from subject papers and papers published by reviewers. The authors of [9] execute a performance comparison between LSI and LDA for use in the present (as of 2016) INFOCOM reviewer assignment system, which uses PDFMiner for text extraction, finding LSI to work well with the academic papers submitted to that conference. We accordingly perform our experiments using LSI to determine the important keywords of each paper, and note that the attack functions equivalently using LDA.

Plagiarism Detection: Turnitin, LLC has the dominant market share for plagiarism detection software. Its software is proprietary, but current documentation states "Turnitin will not accept PDF image files, forms, or portfolios, files that do not contain highlightable text..." [10], indicating that PDFMiner or some similar internally developed tool is used to scrape the text from PDF documents. We may assume from the lack of support for image files that optical character recognition (OCR) is not used, meaning that our proposed attack should succeed, which is proved in Section 5.2.

Additionally, the Turnitin documentation states that "All document data must be encoded using UTF-8 character set" [17]. As mentioned in Section 2, text may have custom encodings, but here we find they are not permitted by Turnitin. This disallows any attack where text, gibberish in appearance, is translated via decoding into legible text. However, no restriction on fonts is in place, due to the necessary ability for Turnitin's client institutions to specify their own format requirements.

Document Indexing: Extracting topics from a document is somewhat of a subproblem to the larger issue of document indexing. As information highly relevant to a search may appear in a small portion of a document, simply relying on the overall topic of every document to infer relevancy to a search may miss some useful results. A search engine should do more than simply topic modeling to show results for a query. In fact, Google uses more than 200 metrics to determine search relevancy [18], including its famous PageRank system of inferring quality of a site based on the number of sites linking to it [19].

Though documentation is sparse on other search engines such as Bing or Yahoo, Google does host some discussion of its treatment of PDF files. It states that they can index "textual content ... from PDF files that use various kinds of character encodings" [20] but that aren't encrypted. "If the text is embedded as images, we may process the images with OCR algorithms to extract the text" [20], but for our content masking attack, text is not embedded as images, so logically the system would not perform OCR. Our experiment finds out for sure for Google, Bing, Yahoo, and DuckDuckGo in Section 6.2.

3 Masking Font Creation

The content masking attack is facilitated by the ability to embed custom fonts within PDF documents. In fact, having all fonts embedded is a formatting requirement for the submission of academic papers to conferences. However, no integrity check is performed on those fonts as to the proper correlation between text strings within the PDF file and the respective glyphs rendered in the PDF viewer. An attacker may map characters to arbitrary glyphs and alter the text extracted from a PDF document while it appears unchanged to humans in a PDF viewer. This requires two steps, firstly to create the requisite font files and secondly to encode the text via these font files.

The first step may employ one of the multiple open source multi-platform font editing tools such as FontForge [21]. With this tool, one can open a font and directly edit the character glyphs with the typical vector graphics toolbox, or copy the glyph for a character and paste it into the entry for another character. One can then edit the PDF file directly with open source tools such as QPDF [22], or in the case of manipulating academic papers, quicken the process by adding custom fonts in LaTeX, and aliasing each to a simple command [23]. We employ the latter method for its greater ease. It employs the program ttf2tfm, included with LaTeX, to convert TrueType fonts to "TeX font metric" fonts which are usable by LaTeX. Two LaTeXcode files are supplied by [23]: T1-WGL4.enc for encoding, and t1custom.fd for easy importing of the font into a LaTeXdocument.

The second step of choosing how to mask this content and what in a document to encode with custom fonts depends on the system targeted, and the technique and evaluation for each of the three scenarios introduced in Section 1 appears in the following three sections.

4 Content Masking Attack Against Conference Reviewer Assignment Systems

As learned in Section 2, topic matching works from groups of words constituting the main topic of the document. Assignment of conference paper submissions to reviewers is accomplished by finding the highest similarity between detected topics within submissions and those within a corpus of reviewers' papers. Meanwhile, a lazy paper writer may wish to collude with specific reviewers, know of some more generous to papers, or just think reviewers may be less critical of papers not within their specializations. This lazy writer needs to change the paper topic to target a specific reviewer, replacing words corresponding to the topic of the paper with words comprising the topic of a paper from the reviewer's corpus, while being masked as the original words to still make visual sense. We now discuss the challenges for this attack and methods to target one or more reviewers, and subsequently evaluate the attack efficacy.

4.1 Construct Word and Character Maps

We primarily require a list of original words within the subject document to change, and a list of words from the target document to which to change these original words. The new words will then be masked to display as the original words using the masking fonts described in Section 3. First, any stopwords within the document should be eliminated from consideration. These are common words within the paper's language, such as "the," "of," "her," or "from." Stopwords may be removed by using existing tools like the Natural Language Toolkit (NLTK) Python package [24]. From here an attacker can replace the most frequently used words in the subject paper with the most frequently used words in the target reviewers paper. This will result in the most frequently used words in the target paper also appearing in the subject paper, for a high similarity score as measured by the LSI method within the automatic reviewer assignment system.

Consider word lists A and B having constituent words $\{a_1, a_2, ..., a_n\}$ and $\{b_1, b_2, ..., b_n\}$ which are in descending order of appearance within the subject and target papers, respectively. An attacker wishes to replace words A with topic B and must therefore replace each word a_i within the text of the subject paper with a word b_i, encoded using some font(s) to render b_i the same graphically as a_i (a *word mapping*). No other words should/need be changed. Consequently, the objective is to construct a mapping between the letters of each b_i and a_i (a *character mapping*). If a_i and b_i are character arrays $\{a_i[1], a_i[2], ..., a_i[p_i]\}$ and $\{b_i[1], b_i[2], ..., b_i[q_i]\}$, then the attacker should construct a *masking font* such that the character $b_i[1]$ maps to the glyph $a_i[1]$, $b_i[2]$ to $a_i[2]$, etc. We may consider this analogous to a map data structure, where $b_i[1]$ is a key and $a_i[1]$ its value, and so on. Two challenges naturally arise in constructing the required character mappings:

One-to-Many Character Mapping: From the brief

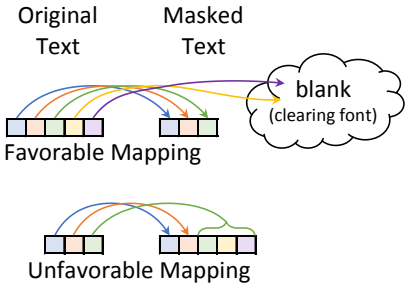

Figure 1: Handling the word length disparity challenge. Each box represents a character.

Algorithm 1 Build Character Map

Input: subject paper s, target paper t
Output: character mapping $C : B \to A$, encoding fonts $F = \{f_1, f_2, ..., f_x\}$

1: $A \leftarrow$ top k topic words of LSI(s)
2: $B \leftarrow$ top k topic words of LSI(t)
3: $C \leftarrow$ empty character map
4: **for** $i \leftarrow 1$ to k **do**
5: $\quad p_i \leftarrow length(a_i)$
6: $\quad q_i \leftarrow length(b_i)$
7: $\quad$ **if** $p_i < q_i$ **then** ▷ favorable mapping
8: $\quad\quad$ **for** $j \leftarrow 1$ to p_i **do**
9: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], a_i[j])\}$
10: $\quad\quad$ **for** $j \leftarrow p_i + 1$ to q_i **do**
11: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], \emptyset)\}$
12: $\quad$ **else if** $p_i > q_i$ **then** ▷ unfavorable mapping
13: $\quad\quad$ **for** $j \leftarrow 1$ to $q_i - 1$ **do**
14: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], a_i[j])\}$
15: $\quad\quad rest \leftarrow$ combine $\{a_i[q_i], ..., a_i[p_i]\}$
16: $\quad\quad C \leftarrow C \cup \{(b_i[q_i], rest)\}$
17: $\quad$ **else** ▷ equal word length
18: $\quad\quad$ **for** $j \leftarrow 1$ to q_i **do**
19: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], a_i[j])\}$
20: $x \leftarrow$ largest number of key collisions in C
21: $temp \leftarrow C$
22: **for** $i \leftarrow 1$ to x **do** ▷ build fonts
23: $\quad f_i \leftarrow$ empty font
24: $\quad$ **for each** $c \in C$ **do**
25: $\quad\quad$ **if** value in c is $\emptyset$ **then**
26: $\quad\quad\quad C \leftarrow C \setminus \{c\}$
27: $\quad\quad\quad$ use clearing font for key in c
28: $\quad\quad$ **else if** no key collision between c, f_i **then**
29: $\quad\quad\quad C \leftarrow C \setminus \{c\}$
30: $\quad\quad\quad f_i \leftarrow f_i \cup \{c\}$
31: $\quad F \leftarrow F \cup f_i$
32: $C \leftarrow temp$
33: **return** C, F

example in Section 1 of changing the word *green* to *brown*, we know that in terms of a map data structure there is a collision for the key e and the values o and w, such that an attacker will require two masking font "maps" to render *green* as *brown*. The first challenge is to minimize the number of fonts required in the document, so as to avoid suspicion, while fully switching topic A for B. This problem is not delimited by word: some character mappings may be reused in the same or other words, and many may not. Additionally, changing all of the words in A to those in B may be unnecessary, which also impacts the number of one-to-many mappings and resultant number of required font files. If fewer words must be changed while ensuring the required similarity between papers, fewer fonts may be required, and a naive font count threshold defense will be less effective.

Word Length Disparity: Further, the lengths p_i and q_i of words a_i and b_i may differ, causing a_i to be longer than b_i or vice versa. If $p_i > q_i$, to render b_i as a_i, a font file entry is necessary for the letter $b_i[q_i]$ mapping to the last $p_i - q_i + 1$ letters of a_i. Several additional fonts may be necessary if some $b_i \in B$ have the same last character. Thus, we define a *favorable keyword mapping* as a word mapping $b_i \to a_i$ such that $p_i < q_i$. In this case, only a single *clearing font* is needed, wherein all characters map to a blank glyph of no width. Figure 1 illustrates handling favorable and unfavorable mappings. In practice, a blank glyph of no width is in fact a single dot, of width (and height) equal to the smallest unit of measure within a font drawing program. In contrast, an *i* is 569 units wide (and a *w* is 1500 units wide), so this dot will not be rendered at all. And because this clearing font has all letters map to no-width blanks, it will be the only additional font required if $\forall i, p_i < q_i$, hence its favorability.

4.2 Matching One or More Papers to One Reviewer

Mapping of words from B to A is by their original descending order of frequency within the target and subject papers, respectively. Algorithm 1 shows the overall encoding process and begins by running the LSI model on the subject and target papers, then constructing a map between characters in k of the topic words returned. Then, the mapping is added to C for each character, for each word of B, to the corresponding character(s) in the corresponding word of A. Here, comments (Lines 7, 12, 17) indicate the steps taken for favorable and unfavorable mappings and the case when both words are of the same length. Finally at Line 22, the mappings in C are broken up into collections to be made into custom masking fonts, with the exception of those characters from favorable mappings which map to null, for which the previously introduced single clearing font is used. Resulting

from this algorithm are fonts to be used for each character of the words in B to mask them as the words in A. If the attacker has multiple papers under submission, this process may be repeated independently for each paper.

4.3 Matching One Paper to Multiple Reviewers

For a better chance at cheating the peer review process and to collude with multiple reviewers, the content masking attack can be adapted to split up the masked words among two (or more) different lists of frequently used words. Instead of mapping between word lists A and B, the attacker will map between A and B and A and C, such that a_1 will be replaced with b_1 part of the time and c_1 the rest of the time, and so on. The method is otherwise the same as shown in Algorithm 1, but has its own challenge.

Intuition would suggest replacing a_1 half of the time with b_1 and half of the time with c_1. However, the requirement for the attacker's paper to be the most similar of a large number of papers to a reviewer's paper and also the most similar of all others to another reviewer's paper is quite stringent. The intuitive method fails as the similarity score for one target reviewer will be high enough but the other too low. So we use an iterative refinement method which tunes the replacement percentages according to the calculated similarity scores until they are both the highest among their peers. This is generalizable to more than two reviewers, by refining the percentages proportionally according to the successive differences in similarity scores between the subject paper and each of the target papers. We match one paper to three reviewers in Section 4.4, the typical number of reviewers to which papers are assigned (barring contention in reviews, which would not happen during collusion).

4.4 Experiment

We have built a conference simulation system reproducing the INFOCOM automatic assignment process described in [9]. We imported into this system 114 TPC members from a well-known recent security conference as reviewers, and downloaded a collection of each of these reviewers' papers published in recent years. In total, this comprised 2094 papers used as training data for the automatic reviewer assignment system. For testing data, we also downloaded 100 papers published in the greater Computer Science field. Our experiment, then, is to test the topic matching of the test papers with the training papers, via our content masking attack. Following are evaluations of the content masking attack matching one paper to one reviewer, multiple papers to one reviewer, and one paper to multiple reviewers.

Matching one paper to one reviewer: The automatic reviewer assignment process compares a subject paper with every paper from the collection of reviewers' papers to gather a list of similarity scores. The reviewer with the highest similarity score is assigned the paper to judge (if available). We therefore aim to change a testing paper topic to a training paper topic, and to examine how well this works with all papers. For each such pair of papers, then, we replace the frequently appearing words A in the testing paper with those frequently appearing words B in the training paper via Algorithm 1. We test the topic matching of each of the 100 testing papers against our training data to see what is required to induce a match.

For each pair of training and testing papers, we replace important words in the testing paper one by one, to see how many replacements are needed to make that pair the most similar. Figure 2 illustrates this iterative process for one example training/testing paper pair, showing resultant similarity scores. The box plots show where the greatest concentration of the 2094 similarity scores dwell, while red pluses show outliers. The blue stars which emerge to the top correspond to the similarity scores between the testing paper and the target training paper. Figure 2 shows a clear separation of that similarity score from the rest after replacing 9 words, meaning that for this pair, content masking all appearances of those 9 unique words in the testing paper will result in its assignment to the reviewer who wrote that training paper.

Performing this process for all 100 testing papers, we compile the results into Figure 3, which displays the cumulative distribution function (CDF) for the number of words requiring replacement. Evidently, all 100 papers may be matched with the target with 12 words or fewer masked. The sharp jump appearing from 4-9 words indicates that most papers can be successfully targeted to a specific reviewer masking between 4 and 9 words. The font requirements for replacing these words is then represented in Figure 4. A majority of papers require 3 or fewer masking fonts, while almost all of them need only as many as 5. This is a comparatively small number and should go unnoticed among the collection of fonts natural to academic papers. For example, this paper has some 19 embedded fonts, between bold/italic variants, fonts used in figures, and one picture font used in Table 1.

Matching multiple papers to a single reviewer: Should an author wish to have multiple submitted papers all assigned to a target reviewer, the author may simply repeat the content masking process on each paper. While in the previous case we find that an average of 3 or 4 fonts is necessary to make a single test paper sufficiently similar to the target training paper, that needs not directly translate to 3 or 4 fonts per paper with multiple papers. Some fonts may be reused among papers, resulting in

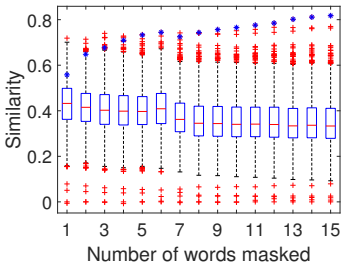

Figure 2: Similarity scores relative to amount of words masked. Blue stars show the desired matching.

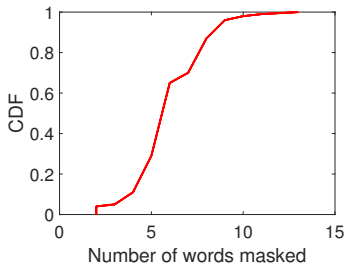

Figure 3: Word masking requirements for all 100 testing papers.

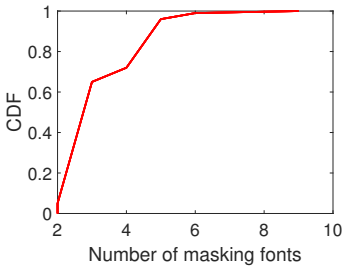

Figure 4: Masking font requirements for all 100 testing papers.

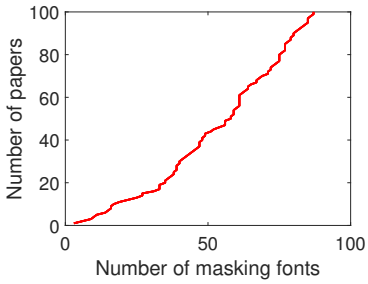

Figure 5: Masking font requirements for matching from 1 to all 100 testing papers to a single reviewer.

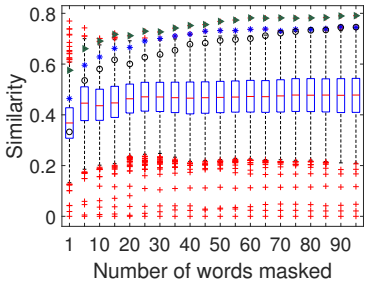

Figure 6: Similarity scores relative to amount of words masked, between a paper and three reviewers. Blue stars, black circles, and green triangles show the desired matchings.

fewer overall fonts used. Figure 5 confirms this, showing a trend more logarithmic than linear.

Matching a paper to multiple reviewers: Finally, we evaluate the iterative refinement method to split masked words among three reviewers' papers as discussed in Section 4.3. Figure 6 shows that the similarity scores for the three target reviewers (blue star, black circle, and green triangle) consistently increase; after some 70 words masked, the subject paper is more similar to the three target papers than any others.

5 Content Masking Attack Against Plagiarism Detection

While a method similar to the topic matching subversion technique just outlined may be used to hide plagiarism, fewer requirements constrain the plagiarist than the lazy author targeting a specific reviewer in a conference. Specifically, an attacker needs only make the underlying text *different* than the rendered, plagiarized text. The underlying text does not need to be actual words, and so only one font is needed, ensuring the naive defense of limiting fonts is defeated. This *scrambling font* is just a random scrambling of the characters. Each original letter is replaced with the letter which displays as the original. Resulting is a human-legible PDF document which appears as gibberish to Turnitin and necessarily has a similarity score of 0%. Details and options for this method are below, followed by an evaluation of each option.

5.1 Targeting a Specific Plagiarism Score

Because Turnitin is a similarity checker, not a plagiarism detector, it relies on the human factor to actually detect plagiarism. Turnitin informs the individual with grading duties of any pieces of similar prose, which naturally arise due to the plethora of written work in existence and the human tendency toward common patterns and figures of speech. It is unlikely then, and would stand out to the grader, that a submission would have 0% similarity with anything ever written. We offer and evaluate two methods an attacker can use to target a specific (low but non-zero) similarity score and more likely go unnoticed.

By Letter: Here, the attacker begins with a scrambling font and removes characters from being scrambled successively until a target percentage of the text is not being replaced. Intuitively, this small target percentage would then appear plagiarized, yielding a credible similarity score. This may be done in a calculated fashion using the known frequency of usage of letters in the English (or other) language. The letters may be listed by their

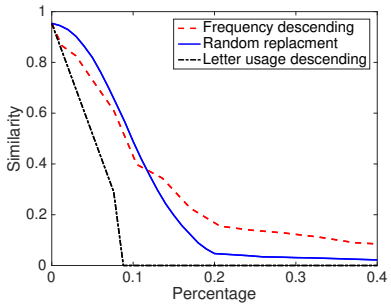

Figure 7: Effects of the percentage of text changed upon plagiarism similarity scores for 10 sample documents.

frequency in ascending or descending order (we evaluate both) and then excluded from scrambling in that order until the target percentage of unaltered text is reached.

By Word: This method is similar to the previous, but instead of leaving some characters unscrambled in the custom font, the attacker leaves some words unaltered by not applying the custom scrambling font to them. Here, words within the document may be listed in frequency of appearance, ascending or descending, and excluded from the scrambling font in that order (we again evaluate both). We also consider changing words at random with a probability targeting some similarity score. This method may be more effective for an attacker in the long run, if Turnitin implements a requirement that some percentage of words be found in a dictionary, English or otherwise. In that case, this attack may be augmented by the previously described method of replacing real words for other real words rendered as the originals.

5.2 Experiment

We use 10 already published papers retrieved from the Internet and mask the content in varying degrees to see the effects on Turnitin's returned similarity scores. We vary the amount a scrambling font is applied to the text according to the previously described methods and upload the resultant papers to Turnitin. Again, we target a specific range of similarity scores, between 5% and 15%, such that a human grader is unlikely to suspect foul play.

Figure 7 plots the three methods. "Frequency descending" refers to the method of masking words in the order of their frequency of appearance in the document, while "Letter usage descending" refers to masking letters by their frequency of usage. Ascending order proved unwieldy in both cases and not worth displaying. Finally, "Random replacement" refers to the method of iterating over all words and masking them with a probability of 1-100% in increments of 1%. These are all plotted in terms of the percentage of text changed. Masking letters by their frequency of usage results in a similarity curve that is too steep to be manageable for selecting a small range of similarity scores. In contrast, the other two methods are very suitable for comfortably picking a specific range. Any probability between 17% and 20% will net a similarity score in our desired 5-15% range in the case of randomly chosen masking. When words are replaced in order of their frequency of appearance, the 5-15% range may be achieved by replacing anywhere between 20 and 40% of the words, offering a very wide range of safety for the plagiarist.

6 Document Indexing Subversion

The final direction of this attack is against search engines, whether for the entire web or for small document repositories or websites. Websites can implement a simple search returning pages housing the query text, or they can use custom search engines offered by Google [25] or Yahoo! [26]. Microsoft Bing also offers its API [27]. As small sites are unlikely to have a more sophisticated search mechanism than the leading search engines, we target and demonstrate our attack against these.

6.1 Method

We here consider modifying the entire content of a PDF to render as something else. Both the underlying text extracted by PDFMiner (or otherwise) and the rendered text should make sense in this case, so that an individual searching for certain terms will be caused to find a PDF holding those words but displaying something entirely different. This results in a more extreme version of the one-to-many character mapping challenge from the attack against topic matching. Instead of masking a small finite number of words, we now examine masking the entire content. However, this is facilitated by the realization that these masks are not necessarily delineated by spaces as before; the attacker can treat the entire document as a single word to be masked. It consequently encounters the word length disparity challenge, to treat the variation in length between real and rendered text, but only once.

Nevertheless, the strategy of adding new fonts, ad hoc, to cover each new mapping quickly balloons out of control, in terms of the attacker needing to keep track of what mappings appear in what font. The number of fonts will increase with the number of characters to be masked, to an upper limit of every character needing a map to every other. Considering (for English) upper and lower case letters, numbers, and common punctuation (22 symbols, dependent upon count), all $26+26+10+22=84$ characters must each map to the other 83 different characters, as well as themselves for those cases which a character and its mask are the same. This requires 84 fonts and represents $84^2 = 7056$ mappings. Code can certainly be

Search Engine	Indexed Papers	Attack Successful	Evades Spam Detection	Not Later Removed
Google	✓	✗	✗	✗
Bing	✓	✓	✓	✓
Yahoo!	✓	✓	Flagged / Cleared	✓
DuckDuckGo	✓	✓	✓	✓

Table 1: Results of content masking attack on search engines.

written to automatically construct all these mappings, but to make this more efficient, we offer an alternative - 84 fonts, in each of which all characters map to one masking character. For example, in font "MaskAsA" character *a* maps to *a*, *b* to *a*, *4* to *a*, *!* to *a*, etc. To mask a document as another, the attacker may simply apply fonts, character by character, that correspond to the desired mask. At the end of the documents, the three end behavior options presented as part of Algorithm 1 and illustrated in Figure 1 function here as well, to handle the length variation.

6.2 Experiment

To demonstrate the efficacy of this attack, we obtained a handful of well-known academic papers, masked their content, and then placed them on one author's university website to be indexed by several leading search engines. For this simple proof of attack, we only used one masking font which scrambled the letters for rendering. The resulting papers have legible text that renders to gibberish, meaning that if they can be located by searching for that legible text, the search engine is fooled.

We submitted the site housing these papers to Google, Bing, and Yahoo! and searched for them some days later. Search engine DuckDuckGo does not accept website submissions but we searched there as well. Table 1 lists the results of our content masking attack on these search engines. "Indexed Papers" indicates the search engine listed the papers in its index. "Attack Successful" means they are indexed using the underlying text, not the rendered gibberish. After a successful attack, the papers may later be put behind a spam warning or removed from the index, as shown in the last two columns. We found similar results for each of the 5 papers tested: that Bing, Yahoo!, and DuckDuckGo all indexed the papers according to the masked legible text, and none removed them later (at time of writing). Yahoo! did mark them as spam after two days but confusingly some days after that removed the spam warning.

Figure 8 illustrates this for one of tested paper. The masked paper is shown in Figure 8a and contains no rendered English words beyond what is shown. Figures 8b, 8c, and 8d show the search results for the legible underlying text, and Figure 8e shows the spam warning appearing days later but later disappearing. Each query was

(a) Gibberish paper

(b) Bing result for the gibberish paper

(c) DuckDuckGo result for the gibberish paper

(d) Yahoo! result for the gibberish paper

(e) Temporary Yahoo! spam warning

Figure 8: Results of the content masking attack against popular search engines. The attack was not successful against Google.

appended with "site:XXX.edu" to isolate the university website where they are hosted for this proof of concept.

Interestingly, Google indexed the papers, but according to the rendered gibberish, not the underlying text. This indicates, of these four engines, only it performs OCR on PDF files it indexes rather than extracting the text through PDFMiner or the like. After two days, the papers were removed from Google's index, before the authors obtained screenshots. We conclude that Google has a robust defense against the content masking attack, while the other three engines remain susceptible.

7 Defense Against Content Masking

As intoned through this paper, Optical Character Recognition (OCR) is able to move the text extraction process from targeting the underlying text to the rendered version, preventing this masking attack. OCR is required for print documents scanned to PDF, but for documents with rendered text, system designers have been loath to use OCR in lieu of PDFMiner or its ilk. OCR is far more complex and requires more processing time than simply running the PDF file through a lightweight parser to collect its strings. We propose here a lightweight font verification method that enables the use of OCR in a highly efficient way to prevent the content masking attack. The intuition is simple; we render each character in the fonts embedded in the subject PDF file and then perform OCR on those characters rather than the rendered PDF file itself. Where an academic paper may be some 50,-75,000 characters, the fonts embedded therein usually contain at most just a couple hundred characters.

Challenges and Technical Details: While the intuition is simple, some challenges arise in its realization. First, while most PDF generation tools will embed only those letters used in the document, it is possible through Adobe InDesign, as one example, to embed the whole font. Some fonts accommodate many characters used in many other languages, and the upper limit on font character capacity is $2^{16} = 65,536$ because characters have a two-byte index. Clearly, performing OCR on a font of that size will be equivalent to performing OCR on an academic paper in terms of computational overhead. Consequently, we scan the document to extract the characters used, and only render those characters (in their respective fonts) for OCR verification. This requires iterating over the entire document, but the overhead introduced here is much less than with full-document OCR, as the process just builds a list from the series of character codes rather than executing image processing techniques on all character glyphs. OCR is then performed on the series of character codes used in each font only.

Second, the existence of many special characters within a font prompts the question of what characters OCR can distinguish and how to handle those it can't. Theoretically, OCR may mature to the point where it can distinguish any sort of accent mark over normal letters, any characters used in languages other than English, and any additional special characters used in typeset mathematics, etc., and some OCR software may be currently in development working on a subset of these problems. However, we aim to provide a defense method readily integrable into current systems. Additionally, such an advanced software will likely incur overhead beyond that of a current OCR package to achieve the requisite precision, where our solution must be sufficiently lightweight to fit within systems where full-document OCR has not been applied due to computational complexity. We define a *normal set* of character codes as those representing upper and lowercase English letters, numbers, and common punctuation, which English OCR packages target, and then we check if the extracted character codes appear in this normal set or not. A letter in the normal set appearing as something other than itself is evidence of the content masking attack, as is a letter outside the normal set having the glyph of one inside. OCR is performed on all used characters in the font, as previously mentioned, and those within the normal set are required to have the correct respective glyph, while those outside the normal set are constrained not to have a distinguishable glyph (i.e. one appearing in the normal set).

The third issue arises with the fact that many special characters have high similarity with normal characters, especially for those fonts in common use which have many thousands of available characters. If one such special character is used legitimately in the text, the scheme just described will flag it as a content masking attack due to its similar appearance with a normal set character. Worse, common OCR tools available presently will conflate characters which humans can easily tell apart but for which the software is not precise enough to do so. For example, it is easy to tell visually that π and n are different characters, but not by common OCR tools.

Font Training Step: We therefore introduce a training step, wherein OCR is performed on the font and lists of intersections compiled. When we perform OCR on each represented character and the detected glyph for a special character but appears like a normal letter, we check the list of characters similar to that normal letter. If the special character appears on that list, we recognize that it may be valid and that we cannot know if it is being used legitimately or as part of a content masking attack. As the purpose of the content masking attack is to disguise the visually rendered text as some other text for the computer to see, we simply replace the extracted character code for this letter as the normal letter it looks like, and pass this on to the end application. If content masking is occurring, the rendered text is sent to the plagiarism

detector, reviewer assignment system, etc., thwarting the attack. Otherwise, the string in which this special character appears is with high probability not an English word and would not be useful to the end application anyway. A reviewer assignment system or plagiarism detector will not make use of mathematical equations when assigning reviewers, as these are not discernible words, so if πr^2 is extracted as nr^z, no loss of function is suffered.

This training solution prompts one further issue, which is that different fonts will need to be trained independently as their nuances cause different sets of characters to appear similar. For the reviewer assignment and plagiarism detection problems, we know a limited number of fonts should be used, due to academic formatting requirements favoring a small set of fonts. Nevertheless, for other applications, such as search indexing, the only limit on the number of fonts that can be trained is that those fonts must be legible enough for an OCR tool to parse. These lists do not occupy too much space; for example our lists for Times New Roman and Arial fonts are 29.4KB and 36.2KB, respectively. This database compiled, the OCR tool will be used to discern the real name of each font used in the document, to counteract the problem mentioned early in this paper, that an attacker may name a font anything desired. Open source OCR tools such as Tesseract OCR [28] provide this functionality.

Font Verification Overview: The training process begins by gathering a collection of fonts and training the system on each. For each character in a font's normal set, all special characters are tested for OCR similarity, and any identified as similar are added to the list for that normal character. Testing a new PDF file is outlined in Algorithm 2, wherein the list of characters and their fonts is reduced to unique combinations of those attributes, and each then tested with OCR. Content masking attacks are detected in lines 12 and 17 when the underlying character index is a normal character other than the OCR-extracted character or when the underlying character index is a special character that does not appear in the similarity list for the OCR-extracted character. In these cases, this pseudocode exits to notify of the attack, though other behavior could be inserted here. This protects all end applications, except in the attack against plagiarism detection in which the attacker replaces normal characters with special characters similar in appearance. That specific attack is identified as possible at line 15, in the case that the underlying character is a special character which does appear in the similarity list for the OCR-extracted character; in this case all instances of this character in the text extracted from this file are replaced with the OCR-extracted character for use in the end application.

Algorithm 2 Extract Rendered Text

Input: font list $F = \{f_1, f_2, ..., f_p\}$, normal character index set $N = \{n_1, n_2, ..., n_q\}$, special character index set $S = \{s_1, s_2, ..., s_r\}$, document character list $D = \{d_1, d_2, ..., d_s\}$
Output: extracted text $T = \{t_1, t_2, ..., t_s\}$

1: Unique character index/font map list $U = \emptyset$
2: **for** $i \leftarrow 1$ to s **do**
3: **if** $d_i \notin U$ **then**
4: $U \leftarrow U \cup (d_i, \text{FONT}(d_i))$
5: $m \leftarrow |U|$
6: OCR-extracted character index set $O = \{o_1, o_2, ..., o_m\}$
7: **for** $i \leftarrow 1$ to m **do**
8: $o_i \leftarrow \text{OCR}(u_i)$
9: $f \leftarrow u_i.font$
10: $L \leftarrow$ list of similar character lists $\{l_1, l_2, ..., l_v\}$ for f
11: **if** $u_i.index \in N$ **then**
12: **if** $o_i \neq u_i.index$ **then** ▷ Attack Detected
13: break
14: **else if** $u_i.index \in S$ **then**
15: **if** $u_i.index \in l_{o_i}$ **then** ▷ Attack Possible
16: $u_i \leftarrow o_i$
17: **else** ▷ Attack Detected
18: break
19: $T \leftarrow$ Apply modified U to D
20: **return** T

Font Verification Performance: The implementation for this defense method is written in Python and employs PDF-Extract [29] to extract font files from PDFs, textract [30] to extract the text strings, and pytesseract [31], a Python wrapper for Tesseract OCR [28]. The alternative to our font verification method is to perform OCR on the entire document, so we use Tesseract OCR for this purpose also for a fair comparison. This comparison will illustrate not only that our method detects/mitigates the content masking attack as well as the naive full document OCR method, but that it performs far better in several scenarios common to PDFs both in and out of the presence of our content masking attack.

First, we compare the performance of the two methods with differing amounts of masked content. We generate 10 PDF files with masked characters varying from 5-20% in frequency of appearance, and apply both methods to each of these file. The results are shown in Figure 9 and show a distinct benefit to our font verification method compared with the traditional full document OCR. Here, *detection rate* refers to the correct extraction of rendered text and the consequent ability to prevent the content masking attack from occurring. For full document OCR, we generate 10 PDF documents with no con-

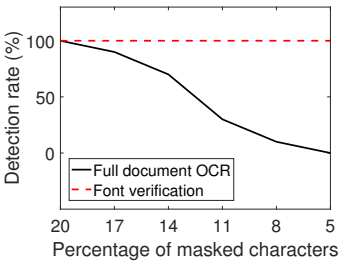

Figure 9: Attack detection under varying degrees of attack.

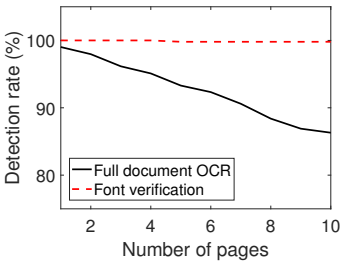

Figure 10: Attack detection on PDFs of different sizes.

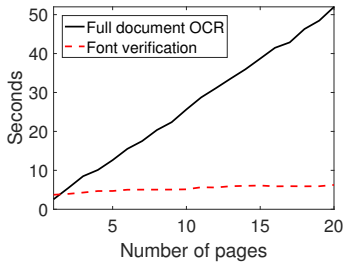

Figure 11: Attack detection time relationship with PDF size.

tent masking and measure the error in character recognition, and then we use this error as a threshold, such that the attack is flagged for one of the content masked PDF files if it is determined to have a larger difference between characters and their glyphs. That threshold was measured at 7%, and more than 20% of characters had to be masked before the full document OCR method detected the content masking attack (after this, detection was 100%). The attack is considered detected by the font verification method if Algorithm 2 flags it or the edge case approach we take of replacing special characters that look like normal letters with those normal letters will enable the end application (plagiarism/spam detector) to process the text properly and thereby flag the attack. In all cases, our algorithm detected the attack or constructed the proper English words required by the end application to detect it.

The disparity here between the methods' accuracy in the 5-20% character masking range has a few aspects involved. Fewer masked characters will appear in a sparser distribution, which make them less visible among legitimate characters. OCR is affected by the distance between characters and the resolution of the image, among other things, which we can control in the case of font verification but which are not controlled when performing OCR over an entire document. We can generate an optimal image of all relevant characters, check their validity, flag detected attacks, and in the case of special characters which appear identical to normal letters, replace them with those normal letters for proper use in the end application.

We also analyze the effects of document length on the detection rate for each method, by comparing their results on 10 PDF files ranging from 1-10 pages in length and having an even 30% distribution of masked characters. Figure 10 illustrates that while the font verification method is almost perfectly static, full document OCR gradually performs more poorly, reaching 14% misdetection by page 10. The aforementioned OCR error rate explains this problem, where while 30% masked characters is above the required 20% to guarantee detection in the previous experiment, additional pages of text steadily allow more masked text to go unnoticed. The font verification appears to be 100% throughout, but actually dips to 99.8% halfway through. Our method is not immune to the errors inherent to OCR as it also uses OCR, but its more judicious approach minimizes those errors. In this case, OCR is confusing the ';' and ':' characters; these are rare but eventual in prose.

Finally, we demonstrate the performance gain of our font verification method over the full document OCR method, on 20 PDF files ranging from 1-20 pages in length and having a 30% distribution of masked characters. In Figure 11, the full document OCR method increases linearly with pages added while the font verification method unsurprisingly remains largely static, increasing by roughly a second compared to the 45 experienced by the full document OCR method. In all, our method requires about 6 seconds to check a 20 page document, rather than 50 seconds, using one core on a laptop processor (Intel i7 at 2.7GHz). This provides far better scalability for the target systems than the alternative, and is easily applied to current systems without requiring upgrades.

8 Related Work

Most exploit research targeting the PDF standard has been in bugs surrounding various programs rendering, displaying, exporting, or otherwise handling PDF documents. The not-for-profit MITRE Corporation lists in its Common Vulnerabilities and Exposures (CVE) collection 431 entries involving the keyword "PDF" and having to do with these external programs [5]. These allow for arbitrary code execution on the host computer and all the associated security risks [6], including establishment of botnets, data exfiltration, and other high-impact security issues. They are, however, limited to basic hacking-type exploits, zero-days chased by patches, and the PDF itself is essentially a vehicle for the hack [7]. These attacks are not thematically novel, and the patches indeed follow the zero-days with reasonable speed [8].

Similarly, some exploration has been performed on the JavaScript execution ability within the PDF standard. When abused, this too allows for arbitrary code execution. Security researcher Didier Stevens offers a series of blogs discussing how to misuse this JavaScript execution, including how to encode the strings involved to create polymorphic malware resisting simple signature-based antivirus products [32]. Some research finds that writing polyglots (code valid in multiple languages) within PDFs can expose security concerns depending on what language the reader uses to interpret the code [2]. Successive updates to the PDF standard implement measures to block certain functions, such as reaching out to the Internet, placing their function behind a confirmation window for the user to view [12]. Additionally, most current antivirus products offer real-time protection using heuristics that can detect potentially malicious behaviors despite simple code obfuscation.

Some academic research regarding PDF security analyzes the JavaScript being executed to verify safety. One work analyzes a set of static features extracted from the PDF, and then instruments with context monitoring code the JavaScript within. This combination static and runtime approach is tested on a collection of 18623 PDF documents without malware and 7370 with, resulting in few false negatives and no false positives [1]. Other research targets attacks not dependent on JavaScript or other parsing vulnerabilities, including one that works to detect these attacks using machine learning on existing flagged PDF files using data extracted from the structure of the file as well as its content [3]. One may expect this strategy to suffer from the same difficulties experienced by signature-based antivirus products, namely an inability to detect malware not already discovered by researchers. Another work allows PDF documents to be opened in an emulated environment to track how they behave before doing so in the host environment [4].

Some works slightly closer to ours examine the possibility of causing PDF documents to be rendered differently on different computers, showing how to restrict the syntax of the PDF standard to prevent this from occurring [33] [34]. This attack against data consistency has some vague similarity to the concept of content masking - displaying different content for the human than the machine. However, we provide several real-world examples of how our content masking attack can subvert real systems, while the impact of the attack in this work is relatively limited to the document looking different to humans using different computers. Some works [35] [36] [37] examine poisoning search results, but this is from the perspective of presenting false data to the machine through website code or manipulations of the PageRank algorithm via botnets, an existing threat vector for which defenses have been continually adapting.

Section 2 introduces the Character Map (CMap), through which letters are mapped to entries within fonts, ultimately displaying the associated glyphs. During our literature search, we found a work [13] from a social science journal of Assessment & Evaluation in Higher Education which touches on a similar topic from a non-scientific stance. [13] discusses how the CMap can be altered to make letters map to different characters within a font. In this way, plagiarism detection can be fooled by mapping to obscure characters whose glyphs are similar in appearance to those for the typically used characters. After devising our attacks, we discovered this work also contains cursory mention of the ability to modify the glyphs within a font, but does not explore this possibility or demonstrate its practicality as we do. We evaluate new methods to target specific similarity scores such that the resultant PDF does not appear unnatural with a 0% similarity score. Further, we show how these custom fonts can be used to subvert conference reviewer-assignment systems and search indexing, developing new and distinct attack methods specific to each of these very different targets. Additionally, we provide a robust defense method, including a defense against the slightly different attack proposed in [13] involving the use of existing characters similar in appearance to normal letters.

9 Conclusion

In this paper, we have presented a new class of content masking attacks against the Adobe PDF standard. After creating algorithms for each of three content masking attack variants, we perform a comprehensive evaluation showing that each lives up to its theory and operates in present state-of-the-art systems. Our first attack allows academic paper writers and reviewers to collude via subverting the automatic reviewer assignment systems in current use by academic conferences including INFOCOM, which we simulated. This requires no visible changes to the paper being reviewed and the addition of just 3-5 custom masking fonts for almost all of the 100 papers tested, easily lost in any paper's natural fonts. We show a second attack that renders ineffective plagiarism detection software, particularly Turnitin, to the point of being able to target specific small plagiarism similarity scores to appear natural and evade detection. In our final attack, we successfully place masked content into the indexes for Bing, Yahoo!, and DuckDuckGo which renders as information entirely different from the keywords used to locate it. Lastly, we provide and test a robust font verification algorithm which is more accurate than full document OCR and requires considerably less computation power.

References

[1] D. Liu, H. Wang, and A. Stavrou, "Detecting Malicious Javascript in PDF through Document Instrumentation," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 100–111, June 2014.

[2] J. Magazinius, B. K. Rios, and A. Sabelfeld, "Polyglots: crossing origins by crossing formats," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 753–764, ACM, 2013.

[3] D. Maiorca, D. Ariu, I. Corona, and G. Giacinto, "A structural and content-based approach for a precise and robust detection of malicious PDF files," in *2015 International Conference on Information Systems Security and Privacy (ICISSP)*, pp. 27–36, Feb 2015.

[4] F. Schmitt, J. Gassen, and E. Gerhards-Padilla, "PDF Scrutinizer: Detecting JavaScript-based attacks in PDF documents," in *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pp. 104–111, July 2012.

[5] MITRE Corporation, "CVE - Common Vulnerabilities and Exposures (CVE)." `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=pdf`, 2016.

[6] K. Selvaraj and N. F. Gutierrez, *The Rise of PDF Malware*. Symantec, Recurity Response, 2010.

[7] R. Brandis and L. Steller, *Threat Modelling Adobe PDF*. DSTO Defence Science and Technology Origanisation, 2012.

[8] Adobe Security, *PDF Security Reaches New Levels with Adobe Reader XI and Adobe Acrobat XI*. Adobe, 2013.

[9] B. Li and Y. T. Hou, "The New Automated IEEE INFOCOM Review Assignment System," *IEEE Network*, vol. 30, no. 5, pp. 18–24, 2016.

[10] "Submitting a Paper." `https://guides.turnitin.com/01_Manuals_and_Guides/Student/Classic_Student_User_Guide/09_Submitting_a_Paper`, 2016.

[11] Y. Shinyama, "PDFMiner." `https://euske.github.io/pdfminer/`, 2013.

[12] Adobe, *PDF Reference*. Adobe Systems Incorporated, 2006.

[13] J. Heather, "Turnitoff: Identifying and Fixing a Hole in Current Plagiarism Detection Software," *Assessment & Evaluation in Higher Education*, vol. 35, no. 6, pp. 647–660, 2010.

[14] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, and R. Harshman, "Using Latent Semantic Analysis to Improve Access to Textual Information," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, (New York, NY, USA), pp. 281–285, ACM, 1988.

[15] M. D. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichet Allocation," *Journal of machine learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[16] L. K. Pritchard, M. Stephens, and P. Donnelly, "Inference of Population Structure Using Multilocus Genotype Data," *Genetics*, vol. 155, no. 2, pp. 945–959, 2000.

[17] "Student Paper Migrations." `https://guides.turnitin.com/01_Manuals_and_Guides/Administrator/Administrator_User_Guide/22_Student_Paper_Migrations`, 2016.

[18] "How Search Works: Algorithms." `https://www.google.com/insidesearch/howsearchworks/algorithms.html`, 2016.

[19] S. Brin and L. Page, "Reprint of: The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[20] "PDFs in Google Search Results." `https://webmasters.googleblog.com/2011/09/pdfs-in-google-search-results.html`, 2011.

[21] G. Williams, "FontForge." `https://fontforge.github.io/`, 2017.

[22] J. Berkenbilt, "QPDF." `http://qpdf.sourceforge.net/`, 2015.

[23] J. Zhao, "Custom Fonts in Latex." `http://math.stanford.edu/~jyzhao/latexfonts.php`, 2012.

[24] E. L. Bird, Steven and E. Klein, *Natural Language Processing with Python*. OReilly Media Incorporated, 2009.

[25] Google, "Custom Search Engine." `https://cse.google.com/cse/`, 2016.

[26] Yahoo!, "BOSS Hosted Search." https://boss.yahoo.com/hosted-web-search, 2016.

[27] Microsoft, "Bing Search API." https://datamarket.azure.com/dataset/5BA839F1-12CE-4CCE-BF57-A49D98D29A44, 2016.

[28] R. Smith and Z. Podobny, "Tesseract OCR." https://github.com/tesseract-ocr, 2017.

[29] K. J. Ward and V. Costan, "PDF-Extract." https://github.com/CrossRef/pdfextract, 2015.

[30] D. Malmgren, "textract." https://textract.readthedocs.io/en/stable/, 2014.

[31] S. Hoffstaetter, J. Bochi, and M. Lee, "pytesseract." https://pypi.python.org/pypi/pytesseract/0.1, 2014.

[32] D. Stevens, "PDF, Let Me Count the Ways ." https://blog.didierstevens.com/2008/04/29/pdf-let-me-count-the-ways/, 2008.

[33] G. Endignoux, O. Levillain, and J. Y. Migeon, "Caradoc: A Pragmatic Approach to PDF Parsing and Validation," in *2016 IEEE Security and Privacy Workshops (SPW)*, pp. 126–139, May 2016.

[34] J. Wolf, "Omg wtf pdf," 2010.

[35] N. Leontiadis, T. Moore, and N. Christin, "A nearly four-year longitudinal study of search-engine poisoning," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 930–941, ACM, 2014.

[36] D. Y. Wang, S. Savage, and G. M. Voelker, "Juice: A longitudinal study of an seo botnet.," in *NDSS*, 2013.

[37] K. Du, H. Yang, Z. Li, H. Duan, and K. Zhang, "The ever-changing labyrinth: A large-scale analysis of wildcard dns powered blackhat seo," in *25th USENIX Security Symposium (USENIX Security 16), USENIX Association*.