Empir Software Eng
https://doi.org/10.1007/s10664-018-9600-2

On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files

Tomasz Kuchta[1] · Thibaud Lutellier[2] · Edmund Wong[2] · Lin Tan[2] · Cristian Cadar[1]

Abstract Electronic documents are widely used to store and share information such as bank statements, contracts, articles, maps and tax information. Many different applications exist for displaying a given electronic document, and users rightfully assume that documents will be rendered similarly independently of the application used. However, this is not always the case, and these inconsistencies, regardless of their causes—bugs in the application or the file itself—can become critical sources of miscommunication. In this paper, we present a study on the correctness of PDF documents and readers. We start by manually investigating a large number of real-world PDF documents to understand the frequency and characteristics of cross-reader inconsistencies, and find that such inconsistencies are common—13.5% PDF files are inconsistently rendered by at least one popular reader. We then propose an approach to detect and localize the source of such inconsistencies automatically. We evaluate our automatic approach on a large corpus of over 230 K documents using 11 popular readers and

Communicated by: Paolo Tonella

Tomasz Kuchta and Thibaud Lutellier contributed equally to this paper.

✉ Cristian Cadar
c.cadar@imperial.ac.uk

Tomasz Kuchta
t.kuchta@imperial.ac.uk

Thibaud Lutellier
tlutelli@uwaterloo.ca

Edmund Wong
e32wong@uwaterloo.ca

Lin Tan
lintan@uwaterloo.ca

[1] Imperial College London, London, UK

[2] University of Waterloo, Waterloo, ON, Canada

Published online: 09 March 2018

our experiments have detected 30 unique bugs in these readers and files. We also reported 33 bugs, some of which have already been confirmed or fixed by developers.

Keywords Cross-software inconsistencies · Document correctness · Image comparison · Error-message clustering

1 Introduction

Many different applications exist for displaying a given type of electronic document, and inconsistencies between these applications can be critical sources of miscommunication. For example, there are many Portable Document Format (PDF) readers (such as Acrobat Reader, Evince, and Firefox), image file readers (such as ACDSee, Eye of GNOME, and Geeqie), and word document readers (such as Microsoft Word, Abiword, and Libreoffice). Electronic documents are increasingly displacing paper documents for delivering important information including medical advice, bills, maps, and tax information. To avoid miscommunication, it is crucial to display an electronic file consistently across different file readers.

The PDF format was created to alleviate the portability problems of electronic files. Unfortunately, there are still many inconsistencies among PDF file readers. For example, the Chrome and Mozilla support forums contain hundreds of complaints from users about PDF files being displayed differently across readers. These issues include drug information sent to doctors that cannot be properly displayed or opened (Google Chrome Help Forum 2015), customers unable to read their online bills (Mozilla Support Forum 2013), and web designers worrying that customers cannot correctly display the PDF files on their websites (Chromium Bug Tracker 2016). There are two main causes of inconsistencies between PDF readers.[1]

(1) **Bugs in readers:** PDF readers, such as `Acrobat Reader`, `Evince`, and `Chromium`, contain bugs. Figure 1 presents such a bug in `Firefox`'s embedded PDF reader. The image on the left shows the rendering of a PDF by `Chromium`, while the image on the right shows the same document rendered by `Firefox`. `Firefox` fails to display the map properly and fills large areas with black colour. `Firefox` developers confirmed our bug report (Bugzilla@Mozilla 2016).

(2) **Bugs in files:** PDF files contain bugs, causing readers to display them inconsistently, or even fail to load them. For example, if a special font is not embedded in a PDF file, some readers fail to display the contents of this file on a computer that does not contain this font. If one can automatically detect those inconsistencies, the creators or owners of PDF files could modify them to ensure the files they share will be displayed as intended by all users.

There is a blurry line between bugs in readers and bugs in files, as some PDF readers are more tolerant to errors than others. In addition, we learned that in many cases, developers are willing to provide workaround patches to the readers to tolerate bugs in malformed

[1] In this paper, we use the term *PDF reader*, or just *reader*, to refer to any application that takes a PDF file as input, such as PDF viewers, editors and processing utilities.

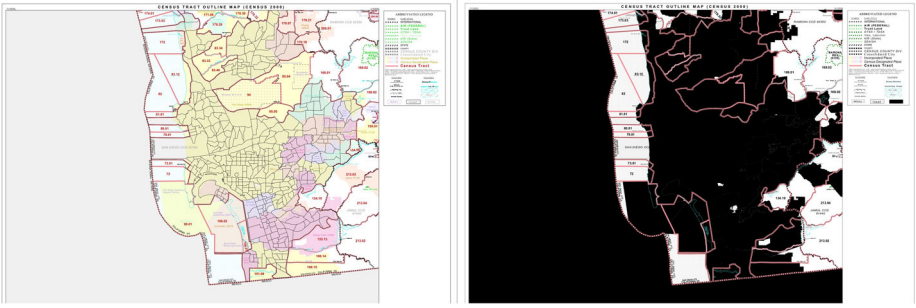


Fig. 1 An example of a bug in `Firefox`. `Chromium` rendering on the left, `Firefox` rendering on the right

files. For example, we reported a damaged file to the `Poppler` rendering library. In the comments posted in the bug tracker (Bugzilla 2016) the developers confirmed that the file was broken, but since key parts (e.g., the object catalogue) were undamaged, they decided that `Poppler` should be able to render this file and proposed a fix.

Regardless of whether the bugs are in readers or files, they cause content to be displayed inconsistently, affecting the correct communication of information. As a result, it is important to design techniques to automatically detect *cross-reader inconsistencies*, which could reveal such critical bugs affecting electronic documents and can also lead to better document recovery techniques, recently tackled by the research community (Demsky and Rinard 2005; Long et al. 2012; Kuchta et al. 2014).

In this paper, we conduct a large-scale empirical study to understand the severity of cross-reader inconsistencies, and propose new techniques to automatically detect them. Specifically, we study the inconsistent display of popular PDF readers on more than 230,000 real-world PDF files previously mined from US government websites (Garfinkel et al. 2009). Since those files are offered by the government, it is essential that they are opened and rendered consistently by a wide range of PDF readers.

By manually inspecting how different readers behave on a small fraction of these PDF files, we identify several challenges of automatically detecting bugs in readers and files. First, a pixel-by-pixel comparison of the rendered images is too strict, leading to many spurious alarms, e.g., due to tiny differences related to how certain characters are displayed. Second, capturing and comparing rendered images accurately is expensive and impractical for such a large data set, even on a cluster of machines. Third, many inconsistencies are caused by the same underlying error in a reader or PDF file, resulting in many redundant inconsistencies to be detected.

To address these challenges, we have devised a multi-stage approach for automatically detecting crashes and inconsistencies. First, we load every document in a portfolio of readers, and (1) report any crashes, and (2) record all warnings and errors logged by the readers. Second, for each reader, we cluster documents based on the warnings and errors issued in the first phase. Third, we select a representative from each cluster, load it in several readers, and then capture and compare the rendered images across readers. Then, we use a form of delta debugging (Zeller and Hildebrandt 2002) to precisely localize the source of inconsistency and generate a reduced PDF file that only displays a small number of inconsistent objects. Finally, we analyze any detected inconsistencies and report them to developers.

We apply our approach to the 230 K files in the Govdocs1 database, which automatically detected 30 unique bugs in popular PDF readers such as the ones embedded by Chromium and Firefox. We also reported 33 bugs (including 11 manually detected on Windows readers), some of which had already been confirmed or fixed by developers. These bugs affect the correctness of the displayed PDF file, causing e.g., readers to crash or PDF elements to be skipped.

In summary, we make the following contributions:

- We conduct the first (to the best of our knowledge) study of cross-reader inconsistencies by manually examining and categorising a random sample of 2,313 PDF files from the Govdocs1 database. We found that cross-reader inconsistencies are common—314 out of 2,313 (13.5%) files are displayed inconsistently—the displayed image is different in at least one reader. Common symptoms include missing images and font inconsistencies.
- We design a technique to automatically detect crashes and inconsistencies across PDF readers based on error filtering, clustering, and image processing techniques.
- We apply our techniques to the 230 K files in the Govdocs1 database to find cross-reader inconsistencies, which detected 30 unique bugs in popular PDF readers.
- We develop a technique to precisely locate document elements causing an inconsistency. This can help developers identify potential bugs in document readers faster.
- We assemble a database of documents that expose errors in these readers, annotated with the types of errors exposed, error messages triggered and a link to the discussion of each bug report. This database could help researchers and practitioners address other software reliability challenges including testing other readers, and diagnosing and fixing bugs in readers and PDF files.

Our database is made available at http://srg.doc.ic.ac.uk/projects/pdf-errors.

2 Technique

Our technique takes a set of PDF documents and a portfolio of PDF readers as input, cross-checks whether these readers open the same PDF document consistently, and outputs a list of documents that crash readers or display differently in the readers. Display inconsistencies in turn indicate either bugs in the readers or in the PDF document. For each document that is detected inconsistent, we use delta debugging to produce a reduced PDF file that only displays inconsistent objects. Our technique also outputs clusters of PDF documents, each of which contains PDF documents that are likely to expose the same bug.

Figure 2 presents a high-level overview of our approach, which consists of three main phases: the filtering phase (Section 2.1), the inconsistency detection phase (Section 2.2) and the inconsistency localization phase (Section 2.3). In the filtering phase, PDF documents are opened in a portfolio of PDF readers. If a reader crashes on a PDF file, our technique reports a crash bug. Otherwise, it collects error messages and warnings emitted by the readers to standard error, and then clusters the documents that produce similar error messages. From each cluster, our approach randomly selects a candidate and proceeds to the second phase.

In the inconsistencies detection phase, our technique opens the candidates in a set of PDF readers and captures screenshots. It then cleans the screenshots by removing some captured elements of the graphical user interface (e.g., different background colour) and applies an image similarity technique on the cleaned screenshots to detect display inconsistencies. A

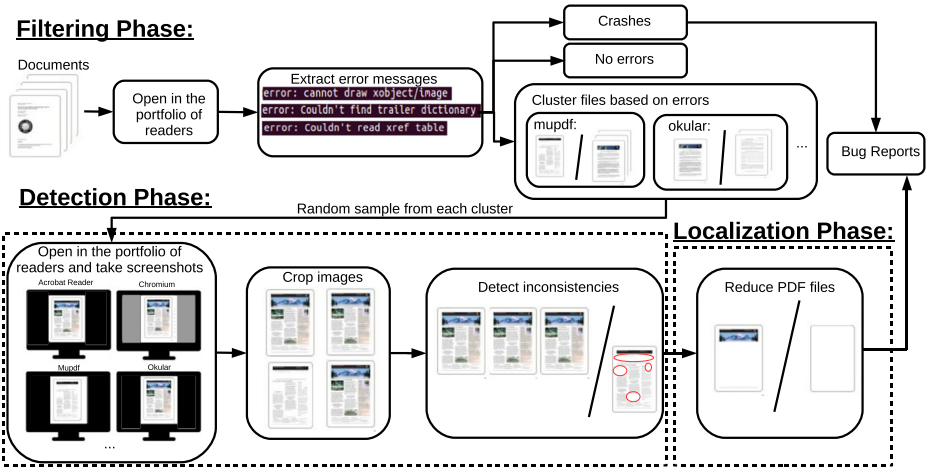


Fig. 2 Overview of our approach, which consists of three main phases: the filtering phase (Section 2.1), the inconsistency detection phase (Section 2.2), and the inconsistency localization phase (Section 2.3)

display inconsistency between two readers happens when parts of the same document are shown differently in the two readers.

We apply the inconsistency localization phase to each document that is displayed inconsistently. In particular, we apply delta debugging (Zeller and Hildebrandt 2002) to reduce the number of visible objects in the file. At the end, we obtain a reduced PDF file that only renders a small number of inconsistent objects (usually one visible object). This allows developers to know which type of objects in the PDF file is causing the issue.

We employ this three-phase approach to address the challenges described in the introduction. One challenge is that comparing rendered images is expensive, and unscalable for a large data set of 230 K PDF files. For example, it takes 40 seconds to render and capture a page with two readers and 5 seconds to compare them. Thus, the total amount of time for rendering, capturing and comparing just the first pages of the 230 K documents for only two readers is approximately 125 days. To address this issue, the filtering phase employs a lightweight approach to quickly identify PDF files that are more likely to be malformed or trigger bugs in the readers. It also aims to group PDF files that cause the same bug in a reader to address the challenge of different documents exposing the same underlying bug.

Another challenge is that a pixel-by-pixel comparison between the rendered images is too strict, reporting differences that are often unnoticeable or unimportant to human eyes. Thus, in the second phase, we leverage an advanced image similarity algorithm (Sampat et al. 2009) that is based on the human visual system to accurately detect bugs while tolerating unimportant differences. We also report how other image similarity algorithms compare on this task.

2.1 Phase 1: document filtering

The first phase of the process serves two purposes: (1) selecting the documents that are likely to be malformed or trigger presentation problems in the readers, (2) grouping the documents that exhibit similar erroneous behaviour.

Specifically, this phase opens each document in every PDF reader considered and captures the error messages printed on the terminal. Intuitively, these error messages are observable signs of potential bugs with either the reader or the document. These observable signs are features that can help us group similar bugs, as it is reasonable to assume that similar error messages are likely to indicate similar root bugs. Our results from Section 5 show that our filtering approach selects files that find relatively more unique bugs than in a random sample.

We analyzed the standard error messages produced by the readers (across our data set and other studied PDF documents) and prepared a set of 501 regular expressions corresponding to the observed classes of error messages. An example of a regular expression is `(Syntax Warning|Error): Could not parse ligature component ``(char|old|chart)'' of ``(char_[0-9] +|.+_old|bar_chart)'' in parseCharName`. For each document that produces at least one error message, we try to match each message against our set of regular expressions. The matching is performed separately for each document and PDF reader. We ignore a message if it does not match any of the regular expressions. Many times the same error message is emitted by different documents. We leverage this fact to cluster documents based on the similarity of these emitted error messages.

We create feature vectors with each element of the vector corresponding to a single class of error messages (as encoded by the regular expressions). The elements in the vector have a value of 1 if the corresponding error was matched and 0 otherwise. These feature vectors are then used by a clustering algorithm to group files which trigger similar error messages.

We use the popular *K-means clustering* algorithm (MacQueen 1965; MacQueen et al. 1967). K-means iteratively groups a population of data points around K centroids representing the centres of the clusters. We use the *K-means++* algorithm (Arthur and Vassilvitskii 2007) to choose the initial locations of the centroids and the sum of squares (*inertia*) as a distance metric. We configure the algorithm to cluster ten times and return the best assignment.

Establishing the right value of K—the number of clusters—is a known challenge in K-means clustering. Having too few clusters results in data points with smaller degree of similarity being grouped together, while having too many clusters poses a risk of putting similar data points in separate clusters. To tackle this problem, we use a known selection algorithm for K: the *Silhouette* (Rousseeuw 1987) method. Silhouette is a coefficient describing how well each of the data points in the set fits in its cluster. It is calculated by measuring how far away a data point is from its cluster neighbours and also from the other clusters. The values of the coefficient fall into the range of $[-1, 1]$, where -1 means a bad clustering (elements not in the right clusters), 0 means overlapping clusters and 1 means a good clustering. To select the right K, we start with value 2, perform the clustering and calculate the Silhouette coefficient. We then increase K by 1 and repeat the process. We stop when the Silhouette coefficient is at least 0.9 and we use Euclidean distance as a metric.

For each reader, we generate a different set of clusters of PDF files. The rationale is that a given file may expose different errors in different readers. For example, for a file F: reader R1 may generate no error messages, while reader R2 generates one error message, and reader R3 generates a different error message. This likely indicates that F exposes no bug in reader R1, but two different bugs in R2 and R3. By clustering PDF files for each reader separately, our approach can capture such differences.

2.2 Phase 2: inconsistency detection

Error messages from PDF readers alone are not enough to detect inconsistencies accurately. For example, two readers may emit identical error messages for one file, but this file may still be displayed differently by the two readers because they have different recovery strategies to address the error. Therefore, a graphical detection technique, our inconsistency detection phase, is necessary to detect inconsistent images.

This phase performs three steps: (1) capturing screenshots of the PDF files rendered by different PDF readers, (2) cropping the screenshots to remove the background and GUI elements and (3) running an image similarity algorithm to detect inconsistent displays of the same file with different readers.

Due to the size of our data set, we only consider the first page of each document when detecting inconsistencies. We explain the rationale behind that choice in more detail in Section 4.1.

Capturing rendered images A straightforward approach to capturing the images rendered by PDF viewers consists of running PDF readers in full-screen mode, displaying the result on a real monitor, and taking a screenshot. However, this simple technique has several flaws. First, the screenshot quality depends on the monitor's display size. Second, taking screenshots on a real monitor is slow, and does not scale to a large set of PDF files.

To improve the scalability of the screenshot capture process and the quality of the images, we use `Xvfb` (2010), a tool that renders graphics in a buffer instead of a real monitor. Using `Xvfb`, we can set up a high screen resolution and ensure that the captured screenshots are high-quality images. In addition, because the PDF files are rendered in memory, it significantly increases the speed and scalability of the screen capture.

Cropping screenshots As we display PDF files in full-screen mode, most of the GUI control elements (e.g., for opening a file) are hidden. However, the screenshots still contain elements that differ across PDF readers that are not related to document inconsistencies. For example, if the screen resolution is larger than the rendered page, the PDF reader will display a uniform background colour that can differ across PDF readers. Therefore, we use PDFBox to extract the page size, and crop the screenshots to keep only the part of the image containing the actual document.

Detecting display inconsistencies Once we obtain cropped screenshots for each document and PDF reader pair, we apply image processing techniques to detect display inconsistencies.

We use a state-of-the-art algorithm for image similarity detection, called *Complex Wavelet Structural Similarity Index (CW-SSIM)* (Sampat et al. 2009). This algorithm aims to detect elements of an image that appear similar to the human eye by focusing on the structure of the objects in the image. This is important because external sources can add noise (e.g., interpolation or anti-aliasing of the software and libraries used for capturing and cropping screenshots.) The CW-SSIM index is robust to such nonstructural transformations.

The goal of this similarity metric is to decompose images into families of wavelets, (i.e., visual channels) that are similar to the decomposition done by human eyes to recognize patterns (Solomon et al. 1994). Then the CW-SSIM index between two images is measured by comparing discrete values of the wavelets obtained for the two images.

The CW-SSIM index ranges from 0 to 1. A value close to 1 indicates that the two compared images are similar, hence no inconsistencies are detected. A low CW-SSIM value indicates visual inconsistencies. For n readers, we select a base reader (`Acrobat Reader`), and compare all remaining $n-1$ readers with this base reader to obtain $n-1$ CW-SSIM indexes per file. If one of these indexes is below a certain threshold, the file is considered *inconsistent*, i.e., the file is displayed differently in at least one PDF reader. If all the CW-SSIM indexes are above the threshold, the file is considered *consistent* across all the tested readers. We empirically chose a threshold of 0.88 by running a preliminary study on a small set of PDF files.

We also experimented with seven other similarity algorithms and justify our choice of CW-SSIM in Section 5.1.

2.3 Phase 3: inconsistency localization

Once inconsistent files have been detected, we attempt to automatically find which elements of the PDF file are causing the inconsistency. To locate the inconsistent elements, we leverage a debugging technique called delta debugging (Zeller and Hildebrandt 2002). The delta debugging algorithm we used is described in Algorithm 1. T is the similarity threshold and sim the similarity algorithm we used to compare screenshots (i.e., `CW-SSIM`).

Algorithm 1 Delta Debugging Algorithm

Input: Inconsistent File F with objects $\{o_1, ..., o_n\}$
Output: List of objects included in the minimal PDF file
Initialization: count=0
DD(F,$\{o_1, ..., o_n\}$):
1: **if** ($n = 1$ **or** count $= 10$) **then**
2: **return** $\{o_1, ..., o_n\}$
3: **end if**
4: F_1=F$\{o_1, ..., o_{n/2}\}$
5: F_2=F$\{o_{n/2+1}, ..., o_n\}$
6: **if** (sim(F_1) $< T$ **and** sim(F_2) $\geq T$) **then**
7: **DD**(F_1,$\{o_1, ..., o_{n/2}\}$)
8: count $+= 1$
9: **else if** (sim(F_2) $< T$ **and** sim(F_1) $\geq T$) **then**
10: **DD**(F_2,$\{o_{n/2+1}, ..., o_n\}$)
11: count $+= 1$
12: **else if** (sim(F_1) $< T$ **and** sim(F_2) $< T$) **then**
13: **random**(**DD**(F_1,$\{o_1, ..., o_{n/2}\}$),**DD**(F_2,$\{o_{n/2+1}, ..., o_n\}$))
14: count $+= 1$
15: **else**
16: **return** $\{o_1, ..., o_n\}$
17: **end if**

The main idea is to iteratively remove visible objects of an inconsistent PDF file to generate a minimum valid PDF file only containing a small number of inconsistent objects. At each stage, we generate two PDF files, one containing half of the visible objects, and the other containing the other half. Then, we run the inconsistency detection phase on each PDF file and keep the one that still reveals the inconsistency. If both new files reveal the inconsistency, we randomly select one. This process is then repeated until one of the following

three conditions is reached: (1) we obtain a PDF file containing only one inconsistent object, (2) the new PDF file is no longer inconsistent, (3) we reach the tenth iteration. Condition (2) can be reached due to false negatives in the image similarity algorithm and cases where there is no single object responsible (e.g., performance bugs). Condition (3) is to ensure that our algorithm remains scalable for PDF files containing tens of thousands of visible objects and only happened for five files.

We implement the inconsistency localization algorithm using the Apache PDFBox library, a Java library that can be used to manipulate or generate PDF files.

3 Experimental setup

In this section we present the experimental setup used in our study. We describe our data set in Section 3.1, our portfolio of readers in Section 3.2 and our infrastructure in Section 3.3.

3.1 Data set

`Govdocs1` (Garfinkel et al. 2009) is a data set of about one million documents crawled from US government web pages (the `*.gov` domain). The set contains documents and files of various formats, including PDF documents.

For this study, we extracted all the files with *.pdf extension from the data set. The total number of extracted files is 231,232. We found that amongst the extracted files there are 10 files which have the suffix `.pdf` but are not PDF files; that is 0.004% of all the files. Furthermore, we have found that there are 1,309 duplicates within the extracted files; that accounts for 0.57% of all the files. Neither the non-PDF files, nor the duplicates could have negatively influenced the results as their percentage compared to the set size is negligible.

According to the metadata information attached to the `Govdocs1` corpus, there are another 3,688 files categorized as PDF. However, since these files do not have *.pdf extension we do not consider them in our study.

In further sections of the paper, when we refer to *data set* we mean the PDF files that we use, rather than the whole `Govdocs1` set, which also includes other types of files (unless we explicitly state otherwise).

The data set is appropriate for our experiments for the following reasons:

- **Widely-used publicly-available data set.** The `Govdocs1` data set has been widely used by the digital forensics community (Grajeda et al. 2017) and the original paper (Garfinkel et al. 2009) discussing in detail the need for a curated corpus like `Govdocs1` has over 200 citations up to date. Also, the data set is freely available, which makes results accessible and reproducible.
- **File Content Diversity.** While it is true that `Govdocs1` files were collected from a common source (governmental web pages), the searches were performed using words randomly chosen from a dictionary, random numbers and a combination of the two. In total 25,330 unique search terms were used while crawling for the files in our PDF set. We believe that such a high number of the search terms used contributes to the data set diversity. We encountered a broad range of types of PDF documents in the set, such as forms, scanned paper documents, scientific articles, and maps.
- **File Size Diversity.** The files in the set are also diverse in terms of file size and the number of pages, as presented in Table 1.

Table 1 Basic statistics, extracted using `pdfinfo`, for the `GovDoc1` data set; `pdfinfo` failed for 15 documents

	Min	Max	Median	Average	Total
File size	931 B	68.8 MB	155.1 KB	579.9 KB	127.8 GB
Pages	1	3,200	10	27	6,443,316

– **Creating Software Diversity.** We inspected the files for optional metadata fields, *Creator* and *Producer*.[2] The *Creator* field is meant to store the name of the original software that created the document for those documents which were initially created in a different format and then converted to PDF. The *Producer* field is meant to store the name of the software that converted the documents to the PDF format. While these fields are optional, 90% of the files in our data set contain both of them.

  Using the Creator and Producer fields, we extracted the information on the operating system (OS) used to create a file. As expected, the majority of the files have been created on Windows (68%). 14% of the files were created on Unix-based systems (mostly Mac and Solaris). We could not extract OS information for the remaining files.

  We also looked at the software used to create the files. There are ∼18,000 unique creators and ∼1,400 unique producers in our PDF set. These numbers are high because they consider different versions of a product as different software. In order to perform further analysis, we grouped different versions of similar software.

  We found 54 applications that were used in the creation of at least 100 documents (either as a producer or a creator). *Adobe Acrobat Distiller* is the most popular software used by far (34% of the files were created by Distiller). This is not surprising as Distiller is the most popular way to create PDF files on Windows. The second most common tool to create PDF files in this data set is *PScript* (12%). *PScript* is a document converter used by many different programs (e.g., *Microsoft Office*, *Ghostscript*). *Acrobat PDFMaker* and the MacOS printer driver *PDFWriter* are also quite common (respectively 8% and 7%). Other commonly used software includes Microsoft Office suite (*Word*, *PowerPoint*, *Excel*), Adobe suite (*Capture*, *Photoshop*, *InDesign*, etc.), LaTeX converters, printer drivers (Hewlett-Packard, Canon, Lexmark, etc.), browsers (*Internet Explorer* and *Mozilla*), commercial document creator tools (*QuarkXPress*, *Corel WordPerfect*, *Amyuni*), open source document converters (*PrimoPDF*, *Ghostscript*, *OpenOffice*, *iText*), and Apple software (*Quartz*, *Keynote*).

  We also found a small number of files (673) for which the Creator or Producer fields have been changed to "U.S. Gov. Printing Office" or "Government Accountability Office." It is possible that these files have been generated using software specific to the US government and therefore are not representative of PDF documents which can be found elsewhere. However, they only represent 0.1% of all the Creator and Producer fields.

  Overall, our analysis of Creator and Producer metadata demonstrates the diversity of the tools used to generate the PDF files, despite the fact that all of the files have been crawled from a single source of US government domains.

– **PDF Standard Revision and Creation Date Diversity.** Table 2 presents a detailed breakdown of the number of files in our data set for each of the PDF standard version. There are representatives for every PDF standard revisions from 1.0 up to 1.7. The files

[2] http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf, pg. 844

Table 2 Numbers of documents in our data set for various versions of the PDF standard

	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
Files #	954	6,665	41,082	52,804	87,008	22,123	19,304	1,276
Percentage	0.4%	2.9%	17.8%	22.8%	37.6%	9.6%	8.3%	0.6%

creation date ranges from early nineties to 2009.[3] Since version 1.7, the PDF format has not changed much, with extensions regarding forms in 2008 and 2009, security in 2011 (change in the password checking algorithm), and a new format 2.0 in July 2017. Since we do not deal with password-protected files, we believe that the 2011 update has little impact on our study. While the files created with the 2017 standard could create different types of inconsistencies, this update is too recent and possibly fewer documents in this format have been created so far. The diversity of PDF versions and document creation times is a desirable feature of the data set for our study.

– **Real-world documents.** The documents from the `Govdocs1` corpus are real-world documents that have not been crafted, fuzzed or cherry-picked to highlight incorrect behaviors or security issues of a specific PDF reader. This is important because in our study we want to find out about inconsistencies in "common" PDF files, in order to better understand how often the problem occurs in the real-world scenarios.
– **Privacy and contents issues.** Creating a new documents corpus, e.g., by crawling the web, is challenging from the legal and privacy point of view. Collected data might contain personal information that should not be redistributed, copyrighted material or illegal content, e.g., pornography. `Govdocs1` corpus was designed in a way that it can be used freely without violating the law.

3.2 Portfolio of readers

Many PDF readers are available for different operating systems. For this study, we focus on readers running on the Linux operating system, and evaluate 11 popular PDF readers. Three of the readers are command line utilities, but as mentioned in the introduction we use the term *reader* to denote any application that processes PDF files.

In our portfolio we have three command line tools: `pdfinfo`, `pdftk` and `ps2pdf`, six popular PDF readers: `Acrobat Reader`,[4] `Xpdf`, `Evince`, `Okular`, `qpdfview`, `MuPDF`, and two web browsers: `Firefox` and `Chromium`.

`Evince`, `Okular`, `qpdfview` and `pdfinfo` use the same PDF library, `Poppler`. However, the codebases are quite different and these readers behave differently. A good example is one of the bugs we reported[5] in which an inconsistency is visible for `Evince` but not for `Okular`. The PDF backend is only a part of the final result, as every PDF reader also needs the rendering component that shows results on the screen. Because of that, we believe it makes sense to test readers based on the same backend, even though some of the bugs are common for all of them.

[3] We filtered out all dates reported prior to the introduction of PDF 1.0 in 1992 and after the `Govdocs1` paper publication year, i.e., 2009.

[4] `Acrobat Reader` is not supported on Linux. We used the last available version from 2013, which is newer than files in the data set.

[5] https://bugs.freedesktop.org/show_bug.cgi?id=97485

Table 3 presents the readers that we use in each phase. In the filtering phase we do not consider the output from `Firefox`, `Chromium` and `Acrobat Reader`, because we found that these readers do not produce any useful PDF parsing messages on the terminal (of these three, only `Firefox` emitted parsing-related messages, but only for five of all the tested documents). In the inconsistency detection and localization phases we do not use `pdfinfo`, `pdftk` and `ps2pdf`, as they are command-line tools and do not display visual representation of a PDF document.

3.3 Infrastructure

Phase 1: Filtering We load all documents by using readers from our portfolio. To speed up this time-consuming process, we leverage a cloud infrastructure. The infrastructure is a set of virtual machines running in an `Apache CloudStack`-based cloud. We used `Vagrant` with `VirtualBox` to create the machines. Each virtual machine is configured to have 16 CPUs running at 1GHz, with 16GB of RAM and a 80GB hard drive. We allocated around 30 such virtual machines in our cloud. Inside each machine there are 14 `LXC` containers, each of them representing a separate execution environment. Both the VMs and the containers are running `Ubuntu 14.04`. Each container is constrained to use only one core, leaving 2 free cores per each VM for the host OS. The containers are configured to run the graphical interface `LXDE` in headless mode using `Xdummy` or `Xvfb`.

We split the document set into a number of partitions. Each container is assigned one partition at a time. In our experiments we used a centralized storage mounted by all the VMs in the cloud via the `NFS` protocol.

We control the containers using `Ansible`, which is a cloud automation tool. Each container is loaded with an `Ansible` playbook (a sequence of commands) which performs the necessary setup and invokes a `Bash` script which executes the experiment.

To cluster error messages, we implemented a `Python` v3 script which extracts feature vectors from the captured standard error messages. The script makes use of the `scikit-learn` package (Scikit learn 2017) for K-means clustering and the calculation of Silhouette coefficient.

We use `xdotool` to automatically perform the necessary graphical user interface interactions such as mouse clicks or getting the name of the window.

Table 3 Portfolio of PDF readers used in each phase

Reader	Version	Phase 1	Phases 2	Phase3
`pdfinfo`	`0.24.5`	✓		
`pdftk`	`2.01`	✓		
`ps2pdf`	`9.1`	✓		
`Xpdf`	`3.03-16`	✓	✓	✓
`Evince`	`3.10.3`	✓	✓	✓
`Okular`	`0.19.3`	✓	✓	✓
`qpdfview`	`0.4.7`	✓	✓	✓
`MuPDF`	`1.3-2`	✓	✓	✓
`Firefox`	`44.0.2`		✓	
`Chromium`	`48`		✓	✓
`Acrobat Reader`	`9.5.5`		✓	✓

Phase 2 and 3: Inconsistency Detection and Localization These two phases were performed on a different machine, equipped with an Intel i5-2400 3.10GHz CPU and 6GB of RAM, running `Ubuntu 14.04`. Due to technical problems, the screenshots for `Firefox` were captured in the cloud VMs. For related technical reasons, the `Firefox` screenshots were not used in inconsistency localization.[6] We use the `Xvfb` virtual screen buffer to open the files in high resolution.

To compare screenshots, we use `MATLAB` version `R2015` and two libraries: `Steerable Pyramids` (Matlabpyrtools 2016) and `CW-SSIM` (Complex-wavelet structural similarity index (cw-ssim) 2013).

Performance It takes around a week to load all the documents in the cloud. Once standard errors are captured, it takes several hours to cluster the files. Comparing two images with CW-SSIM in our setting takes 5s; we need seven comparisons (`Acrobat Reader` vs the rest) per file, which requires 35–40s per file. Inconsistency localization takes roughly 8-11 minutes per file. We believe these numbers are reasonable, give the data set size and that we operate on relatively large images (3000×3000 pixels before cropping).

3.4 Research questions

In Sections 4, 5 and 6 we present a series of experiments. The experiments were conducted to answer the following research questions:

1. **Section 4**: How frequent are cross-reader inconsistencies and what are the main types of inconsistencies?
2. **Section 5**: How well does our automated technique perform in terms of finding cross-reader inconsistencies?
3. **Section 6**: How well does our inconsistency localization based on delta debugging perform?

4 A study of cross-reader inconsistencies

We first manually study a reasonably large random sample of 2,313 PDF files from the `Govdocs1` database. This empirical study has two main goals. The first is to understand how severe cross-reader inconsistencies are, i.e., how often PDF files are displayed inconsistently and what the common symptoms are.

The second goal is to produce a random set of PDF files with ground-truth knowledge about cross-reader inconsistencies. Such information allows us to measure the standard metrics of precision and recall of the automated detection techniques: *precision* is the fraction of reported inconsistencies that are true inconsistencies, and *recall* or *true positive rate* is the fraction of all true inconsistencies that are detected. F1 is the harmonic mean of precision and recall.

[6] A bug in `Xvfb` prevented us to display the PDF files with Firefox in Fullscreen mode, making the screenshots incorrect.

4.1 Methods

It is impractical to manually examine the 230K files of the `Govdocs1` database. Thus, we study a one percent random sample of 2,313 documents. We open each document with the eight graphical PDF readers (described in Section 3.2) and identify any rendering differences. To reduce manual effort, we first automatically took screenshots of the displayed images in each reader. Then one of the authors manually compared the screenshots to determine if two screenshots generated from the same PDF file reveal any inconsistencies. If they do, we then manually open the file in the two readers and compare the displayed PDF images to confirm the inconsistency; it is to ensure that bugs in the screen capture tool do not affect our results.

On average, a PDF file in our sample has 28 pages, and more than 64,000 pages for all 2,313 files. For eight readers, there would be more than 500,000 images to check—prohibitively expensive for a manual analysis. Therefore, we focus on inconsistencies found on the first page of each document only.

Because we are not the authors of the PDF files, we do not know what is the correct rendering for each file. Therefore, we use the "majority" rule to classify inconsistent displays. If five readers or more have identical renderings and the others display the page differently, we consider the five readers to be correct and the others incorrect.

When a reader cannot render a non-embedded font, it will replace it with a default font. Most readers use different default fonts, and because there is no standardized behaviour on the correct rendering of a missing font, we cannot determine which reader behaves correctly. As a result, we only report the total number of such files.

One may consider using `Acrobat Reader` as the ground-truth, as it is developed by the same company that created the PDF format. However, `Acrobat Reader` also has bugs, and furthermore the Linux version is no longer supported, with several known bugs present (Archlinux 2015).

4.2 Inconsistencies are common

Table 4 shows that cross-reader inconsistencies are quite common—out of the 2,313 pages manually verified, 13.5% are rendered differently by at least one reader. Since these PDF files originate from US government websites, they are meant to be correctly read by users and were not generated to be displayed inconsistently in different readers on purpose. Yet, a significant portion is not. The bottom part of Table 4 shows the agreement between readers, e.g., in 41% of the cases 7 out of the 8 readers behave similarly.

To understand if our random sample size is big enough, we calculate the margin of error for this sample, which is 1% with 95% confidence. This means that with 95%

Table 4 Number of consistent and inconsistent files

	# of files	%
Consistent with all readers	1999	86.5%
Inconsistent	314	13.5%
7/8 readers agree	128	41%
6/8 readers agree	92	29%
5/8 readers agree	38	12%
≤4/8 readers agree	56	18%

For the latter, we report how many readers behave similarly ("agree")

confidence, the percentage of inconsistent files in the entire Govdocs1 database would be 13.5%±1%.

When projected on the entire 230K database, this percentage represents 28K to 33K inconsistent files, which can cause many of the miscommunication issues described in the introduction. Since only the first pages of those documents have been studied, and an inconsistency can happen anywhere in the document, the number of inconsistencies would likely be bigger than 13.5% when all pages of the documents are considered.

4.3 Types of inconsistencies

Table 5 shows the different types of inconsistencies that we encountered. The numbers from this table do not map directly to the number of files triggering inconsistencies presented in Table 4, because one file can expose several issues in the same reader (e.g., missing an image and displaying an incorrect colour) or the same issue in multiple readers (considered as multiple bugs since they are present in multiple readers), and multiple files may expose the same issue in one reader (e.g., the 26 missing-image inconsistencies of MuPDF shown in Table 5 are all caused by a single MuPDF bug).

We describe each type of inconsistency below:

Performance issues The first type of inconsistencies one notices when opening a PDF file concerns performance. This is not a graphical bug by itself, but our tool was able to detect it nonetheless. While we can ensure that the reader is fully loaded before taking the screenshot, it is not trivial to ensure the PDF page is fully rendered before taking a screenshot. If the reader takes too long to render the file, the screen capture might occur before the file is fully displayed. We decided to allow a reasonable time of 30s for the reader to display the file. If after this time the file is still not rendered correctly, we report a performance bug. Our results indicate that MuPDF, Acrobat Reader, Evince and Xpdf always render the PDF files within 30s, while the other PDF readers experience performance problems for some files.

Missing images We observed several cases of missing images during our study. While in principle those issues could also be created by bugs in the PDF files (e.g., incorrectly compressed images), the 28 issues we encountered were caused by bugs in Chromium, Firefox and MuPDF. The bug in MuPDF was already known, while the other bugs were new and confirmed by the developers. Figure 3 shows the missing image bug in MuPDF. This bug happened because of specific image encodings that were not well supported by the readers. This issue is critical when it occurs in scanned documents for which each page consists in one large image. In such cases, the buggy reader only displays blank pages.

Map bugs Maps are challenging to render because they are generally made of several superposed layers of graphic vectors and often contain advanced features such as transparent objects. In the data set, we found several maps that were incorrectly rendered, similar to the example in Fig. 1. These are bugs in the PDF readers.

Colour inconsistencies can either be bugs in the reader or in the PDF file. Figure 4 shows a colour bug in the Ubuntu version of Acrobat Reader 9.5.5. In this example, the background colour rendered by Acrobat Reader is different from the background colour rendered by other readers. Colour bugs can also be caused by buggy files, if the colour space of the file is incorrectly encoded. While no information is usually lost, this significantly affects the design of the document and can be a major issue for graphic designers.

Table 5 Issues detected in the random sample, sorted by type and reader

Issue type	Acrobat Reader	Chromium	Evince	Firefox	MuPDF	Okular	qpdfview	Xpdf	Total
Performance issues	0	1	0	1	0	15	8	0	25
Missing images	0	1	0	1	26	0	0	0	28
Map bugs	0	0	2	4	0	0	0	0	6
Colour inconsistencies	44	19	5	28	7	3	0	0	106
Gradient inconsistencies	0	0	5	6	3	1	0	0	15
Form inconsistencies	12	12	0	0	0	0	0	0	24
Others	2	4	4	16	25	6	0	0	57
Font issues	178								
Total	58	37	16	56	61	25	8	0	261
Total Unique Bugs	3	5	4	9	8	2	0	0	31

Acrobat denotes Acrobat Reader

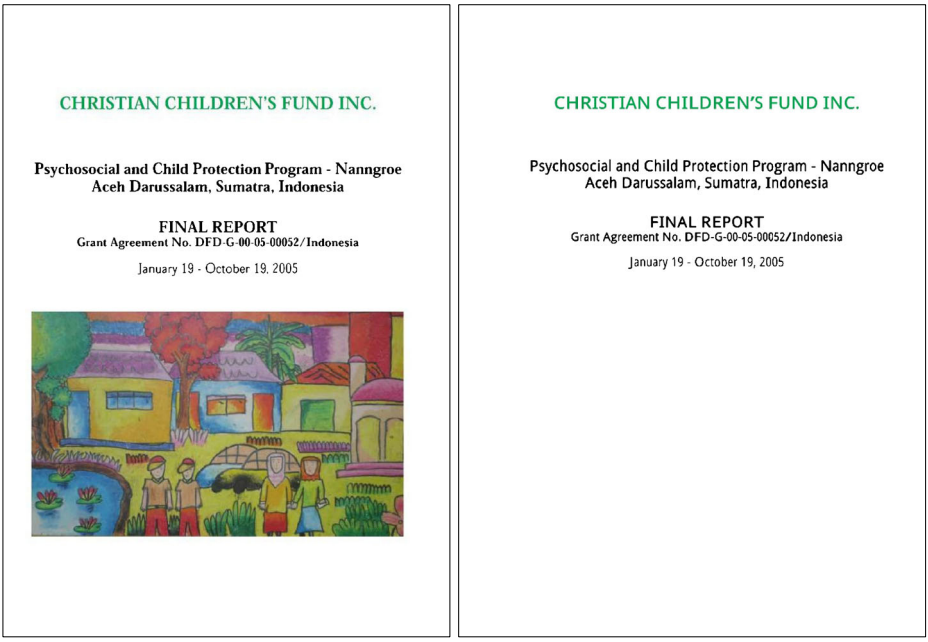


Fig. 3 An example of a missing image bug in `MuPDF`. `MuPDF` on the right, other readers on the left

Form inconsistencies These inconsistencies occur when forms are included in the PDF file. `Chromium` and `Acrobat Reader` highlight editable fields while other readers do not. We believe such discrepancies are neither bugs in the PDF files, nor in the readers.

Others This category contains a wide range of inconsistencies that appear rarely and are generally caused by bugs in a specific reader or rendering library. For example, we found a bug in `Evince` that occurs only in rare cases when a PDF file contains images that have a

Fig. 4 Example of colour discrepancy between `Acrobat Reader` (left) and other readers (right)

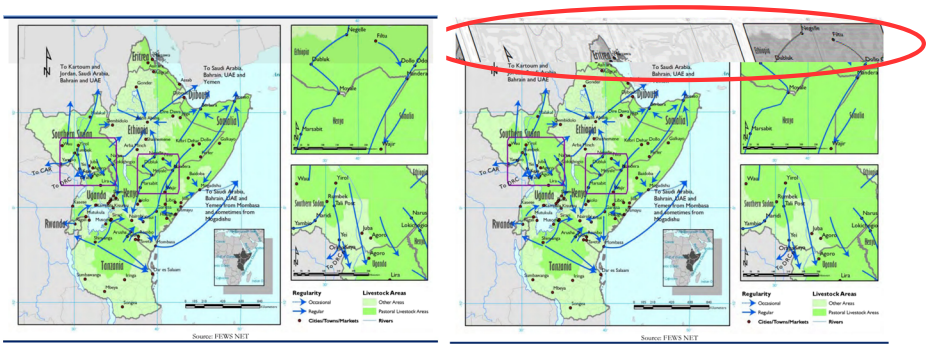


Fig. 5 An example of "other" types of inconsistencies. `Chromium` rendering on the left, distorted `Evince` rendering on the right

colour depth inferior to 8 bits. Figure 5 displays an example of this bug. We filed this bug against the `Poppler` rendering library and the developers fixed it. [7]

Font inconsistencies As we cannot know which readers display the correct font, we only report the total number of files that had at least one font inconsistency (178 in Table 5). Indeed, 178 out of 314 files that reveal inconsistencies contain a font inconsistency. This inconsistency occurs when an uncommon font is not embedded in the PDF file. In this case, the reader will either use a default font to replace the non-embedded font or simply not display the text. Figure 6 shows and example of this issue.

Reader reliability Our experiments enable us to assess the relative reliability of the eight readers. The row *Total* of Table 5 shows the total number of bugs exposed in each reader, while the row *Total Unique Bugs* presents the number of unique bugs. We assessed uniqueness manually based on the type of inconsistency, potential identical warnings and error messages, the similarity of the documents, and whether the exact same set of readers behave similarly. For example, if two PDF files have the same inconsistency (e.g., a jpeg image is not displayed) with the same readers, then we consider that these two files reveal only one unique bug. On the other hand, if the image in the first file is incorrectly displayed with one specific reader (e.g., Evince) and the image in the second file is incorrectly displayed with a different reader (e.g., Chromium), then these two PDF files reveal two unique bugs. Visual inconsistencies are often subjective, so best effort judgement seems to be a reasonable approach.

We can use both measures as an approximation of reader's reliability. With all other factors being equal, the higher the number of unique bugs, the lower the quality of a reader's codebase. On the other hand, under the assumption that the `Govdocs1` data set is representative of real-world documents (which we discuss in Section 3.1), the total number of bugs in each reader correlates with its frequency of failure in the field. Looking at these two metrics, `MuPDF` is the least reliable reader with 8 unique bugs and 61 bugs, while the most reliable reader is `Xpdf` with no bugs.

Summary The experiments presented in this section try to answer the research question *How frequent are cross-reader inconsistencies and what are the main types of inconsistencies?*

[7] https://bugs.freedesktop.org/show_bug.cgi?id=94371

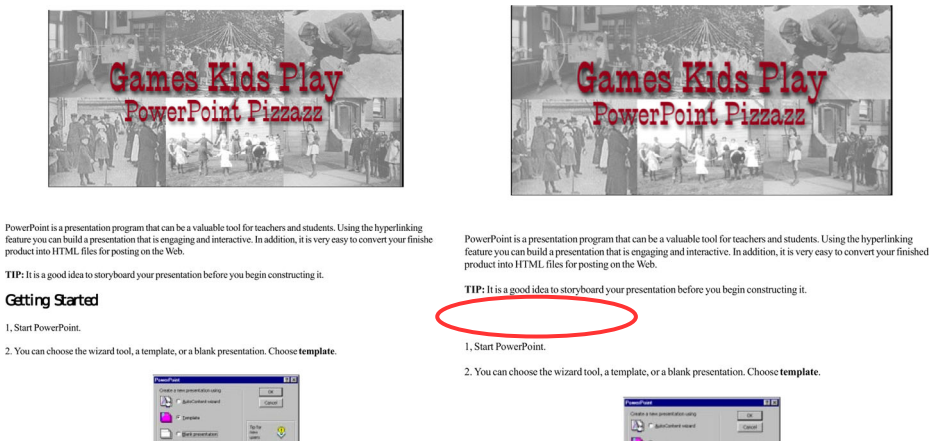


Fig. 6 `Chromium` (left) can render the incorrectly embedded characters, while other readers (right) cannot

As we found out, cross-reader PDF inconsistencies are surprisingly common and we can categorize the inconsistencies into several common types. Finally, some PDF readers exhibit more visual inconsistencies than others.

5 Automatic results

In this section, we evaluate our automatic inconsistency detection technique described in Section 2. We devised two experiments, one using the random sample of documents that we manually inspected in Section 4, and the other using the entire 230 K database.

5.1 Results for the random sample

We perform the first experiment on the sample of documents studied in Section 4[8]. This experiment evaluates the inconsistency detection phase of our approach: we perform image comparison on all files in the evaluation set without filtering to potentially identify all inconsistent PDF files. The aim of this experiment is to use the manually-determined ground truth to establish the precision and recall of our automated image comparison technique.

Our automatic tool detected 189 true inconsistent files, which revealed 21 unique bugs in the readers and 124 incorrect files. We consider a file as *incorrect* if it does not follow the PDF format specification or does not embed or subset non-standard fonts. While font embedding is not included in the PDF specifications, it is included in many publisher standards. For example, the minimum requirements for PDF published on IEEE Xplore platform include “embed or subset all fonts” (ENGINEERING 2008).

The detection precision, recall and F1 for our approach are 33%, 60%, and 43% respectively. While 43% is the highest F1 value, we can tune the threshold of our technique to

[8]For technical reasons we needed to exclude one file from the sample, because PDFBox was unable to parse it.

obtain different precision and recall values. In other words, because our inconsistency detection tool is based on a similarity score, we can modify the threshold to either reduce the number of false positives or false negatives. Figure 7 shows the ROC curve associated with the CW-SSIM similarity algorithm we used. This curve shows that we can get a reasonably good true positive rate (60%) while keeping a low false positive rate (20%). Increasing the true positive rate to 80% can be done if we accept to increase the false positive rate to 50%.

We also experimented with other image similarity algorithms: Absolute Error (AE), Mean Absolute Error (MAE), Mean Squared Error (MSE), square Root Mean Squared Error (RMSE), Normalized Cross Correlation (NCC), Peak Signal to Noise Ratio (PSNR) (Huynh-Thu and Ghanbari 2008) and Perceptual Hash (PHASH) (Zauner 2010). Figure 7 shows the ROC curves for each of them. CW-SSIM and PHASH are the best performing image comparison metrics. We select CW-SSIM as it was our first choice; it is also a bit faster (we did not perform a thorough time measurement but we believe that one second is a good estimate).

False positives 73% of the false positive results are due to limitations of CW-SSIM. Some documents contain few or no structured elements (e.g., almost blank pages), making it difficult for CW-SSIM to identify structural similarities. These are edge cases where simpler techniques such as a histogram comparison might provide more accurate results. Future work could be done to use different image comparison algorithms depending on the visual features of the PDF file being evaluated. 11% of the false positives are due to GUI problems that are not correctly removed from the screenshots. In 10% of the cases, the files were correctly detected as inconsistent by our algorithm, but the manually generated ground truth was incorrect. The remaining false positive results were mostly due to spurious graphical bugs. We do not consider these spurious graphical bugs as true positives because their root cause is external to the PDF reader (e.g., bugs in the Unity graphical shell).

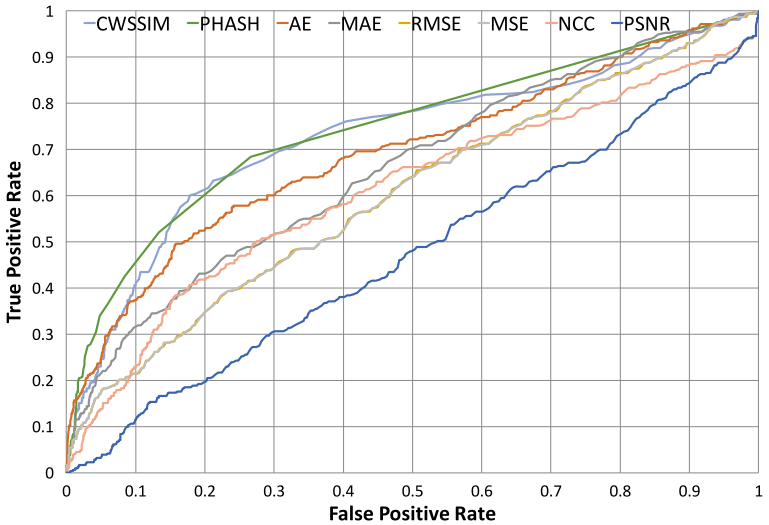


Fig. 7 ROC curves for different image similarity algorithms

False negatives Local inconsistencies such as font inconsistencies can be challenging to detect. For example, consider a document in which only a few words are rendered with an incorrect font; automatically detecting those few incorrectly rendered words is hard. This concerns the vast majority of the inconsistent files classified as correct by our tool. Those "local" discrepancies are very hard to spot when looking at the entire pages because most of the page is actually rendered correctly. Possible improvements could be done by extracting the locations of the different text boxes from the PDF's metadata and running the image comparison on specific parts of the screenshot.

5.2 Results for the entire set

In the second experiment we evaluate our end-to-end approach, as described in Section 2. We analyze the whole data set of over 230,000 files: in the first stage, we load each document in every PDF reader used for this phase, detect crashes and capture emitted error messages. We then create clusters of documents for each reader based on the emitted error messages, randomly select one document from each cluster and run our inconsistency detection tool on the first page of the document. The reason for random selection of a single document from each cluster is the assumption that clustering should group documents with similar issues together. This allows to minimize the number of tested files to just one file per cluster; in our experiments, this reduces the test set size from 230K to 103. Although it is a heuristic, our experiments show that it yields good results.

Error messages The number of documents in the `Govdocs1` data set that cause the readers to emit an error message of interest varies between 3,771 and 28,438, with an average of 15,293 and a median of 14,085 across the PDF readers. In total 65,406 files generated warnings or error messages in at least one of the readers, which accounts for 28% of documents in the set.

Crashes In the first phase of our approach, we detect PDF reader crashes, such as segmentation faults and aborts. Table 6 presents the number of non-spurious crashes observed in `ps2pdf`, `pdftk`, `Evince` and `qpdfview` (the other readers had no crashes or only spurious ones).

Inconsistency bugs Table 7 shows the number of clusters created for each PDF reader after running the filtering phase. This varies between only 2 clusters for `pdftk` and `Okular` and 41 for `Evince`, for a total of 103 clusters.

The column *Clusters for* indicates the PDF reader that generated the error messages that were used for clustering. The column *Number of clusters* presents the number of cluster obtained.

We randomly sampled one file (or candidate) from each cluster and compared the rendered images both manually and using our automated technique. We assume that the file selected from the cluster is representative of the other files in the cluster. In the table we

Table 6 Number of crashes detected

Reader	ps2pdf	pdftk	Evince	qpdfview	Others	Total
Crashes	2 (2)	5,281 (3)	112 (2)	8 (2)	0	5,403 (9)

The number of unique errors is shown in parentheses

Table 7 Inconsistencies for cluster candidates

Clusters for	Number of clusters	Inconsistent candidates	Bugs triggered by candidates
`pdfinfo`	5	2	2
`pdftk`	2	1	1
`ps2pdf`	19	7	7
`Xpdf`	9	2	2
`Evince`	41	16	23
`Okular`	2	2	3
`qpdfview`	13	4	5
`MuPDF`	12	6	6
Total	103	40	49 (10)

The third column shows the number of inconsistent cluster candidates. The number of unique bugs is shown in parentheses

only display the results of manual inspection in order to illustrate the usefulness of clustering independently of the accuracy of image comparison, but we also present the results of our automatic tool below.

The column *Inconsistent candidates* reports the number of candidates that are inconsistent across the readers and the column *Bugs triggered by candidates* displays the number of bugs triggered by the inconsistent candidates. An inconsistent file can trigger different bugs in different readers, so this number might be higher than the number of inconsistent candidates.

For example, the first row shows that documents which generate error messages in `pdfinfo` were split into five clusters. From each of the clusters we randomly sample one candidate, resulting in a total of five candidates. Two out of these five candidates are inconsistent across PDF readers. Further manual examination of these two candidates indicates that they reveal two bugs.

In total, 40 out of 103 candidates have inconsistencies. Therefore 38% files selected via clustering are inconsistent compared to only 13.5% inconsistent files in our random sample discussed in Section 4. Across the inconsistent cluster candidates we found a total of 49 bugs, out of which 10 were unique. The rate of unique bugs to the number of files for cluster candidates is 10 / 103 = 9.7%, which is higher that the rate of 31 / 2,313 = 1.3% for the random sample. The higher percentages of inconsistent files and unique bugs suggest that the clustering approach is indeffective.

Results of image comparison on the candidates Finally, if we use the automatic image comparison-based inconsistency detection, we get 24 cluster representatives detected as inconsistent, which contain a total of 31 bugs, 6 unique bugs and 14 incorrect files.

Correlation with file producer With `pdfinfo`, we retrieved the authorship data of the files producing standard errors or having non-zero return status. We extracted creator, producer and author of the file to see whether there is a correlation between inconsistent files and the software that produced them. The results are non-conclusive although there are many files that share common origins: there are ∼18,000 distinct creators, ∼1,400 distinct producers and ∼48,000 distinct authors.

We further analyzed the metadata of the files in the random sample, as presented in Table 4. First, we analyzed the correlation with the version of PDF document.

Table 8 presents the percentage of files for each PDF version in our sample, as well as the percentage of inconsistent files for each version. First, we can see the distribution of

Table 8 Comparison between the distribution of files across PDF versions in the entire random sample and in the inconsistent files from the random sample

Percentage	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
Whole random sample	0.6%	3.0%	16.8%	23.2%	37.8%	9.2%	8.8%	0.7%
Inconsistent files	2.5%	3.5%	20.4%	21.3%	25.8%	12.1%	13.7%	0.6%

files across PDF versions in our random sample closely follows the distribution of our entire data set showed in Table 2.

Second, the distribution of inconsistent files across PDF versions roughly follows the distribution of the PDF versions in the sample, with a few exceptions. The largest difference occurs for PDF 1.4. Indeed, this version represents 37.8% of the random sample, but only 25.8% of the inconsistent files. Additionally, we measured the Pearson correlation between inconsistent files and PDF versions and did not find any correlation. For all PDF versions, the Pearson coefficient is between -0.1 and 0.1.

We also analyzed the correlation between creators and producers that were used to produce more than 20 documents in our random sample.

We found 12 creators used to generate more than 20 documents: InDesign, Pscript, Capture, PDFMaker, Print Server, PageMaker, POP90, Office, FrameMaker, WordPerfect, Microsoft Word, and QuarkXPress.

Only three creators have a small correlation with inconsistent PDF files. InDesign and QuarkXPress have a small positive correlation (Pearson coefficient between 0.1 and 0.3) with files rendered inconsistently. PScript presents a small negative correlation (Pearson coefficient between -0.1 and -0.3). For all the other creators, we found no correlation with incorrectly rendered documents.

There are seven producer fields used in more than 20 documents: Acrobat Distiller, Acrobat PDF Writer, Acrobat PDF Library, Ghostscript, Corel PDF Engine, Etymon, and PDFContext. None of them shows any significant correlation with inconsistently rendered PDF documents.

Summary of inconsistencies and bugs detected in both experiments Our technique automatically detected inconsistencies and bugs in both the experiments in Sections 5.1 and 5.2. In total, our approach detected 229 inconsistent files and 5,403 crashes/exceptions automatically, including 30 unique bugs in the readers and 138 incorrect files. The number of unique bugs (30) is not the sum of the number of bugs detected in each experiment, as some bugs overlap.

Bug reports One of the outcomes of the project was filing 33 bug reports of which 17 were confirmed or fixed. We filed the bugs throughout the project, with some of them being spotted during the development of the tool. We did not report all bugs, as some of them had already been reported or fixed.

The list of bug reports is presented on our project website at https://srg.doc.ic.ac.uk/projects/pdf-errors/results.html. Currently out of 33 reported bugs, 8 have already been fixed, 9 have been confirmed as true bugs, 5 have been triaged as “won’t fix”, and 11 await confirmation. Note that one of the fixed bugs (#17 on the list) was marked as a duplicate as there was a parallel report of a problem. We still count the bug towards the 33 unique bugs as that was our judgement at the time of reporting.

Summary The experiments presented in this section try to answer the research question *How well does our automated technique perform in terms of finding cross-reader inconsistencies?* We show that the technique is able to find more inconsistencies in cluster candidates compared to a random selection of documents. We also show that our algorithm of choice for visual inconsistency detection is one of the two algorithms with the best ROC curve.

6 Inconsistency localization evaluation

To evaluate the accuracy of our inconsistency localization technique, we run it on the 189 true inconsistent files detected by our approach on the random sample (see Section 5.1). The inconsistency localization algorithm was able to generate a reduced PDF file in 86% of cases. Most files (139) were reduced to only two visible objects, 13 to only one, and 21 to more than two.

We evaluated the correctness of the reduced PDF files by manually examining them to verify whether they actually reveal an inconsistency. 84% of the reduced PDF files display an element that is inconsistent. Figure 8a shows an example of inconsistent file incorrectly displayed by `Okular` and Fig. 8b shows the same issue on the reduced file obtained after

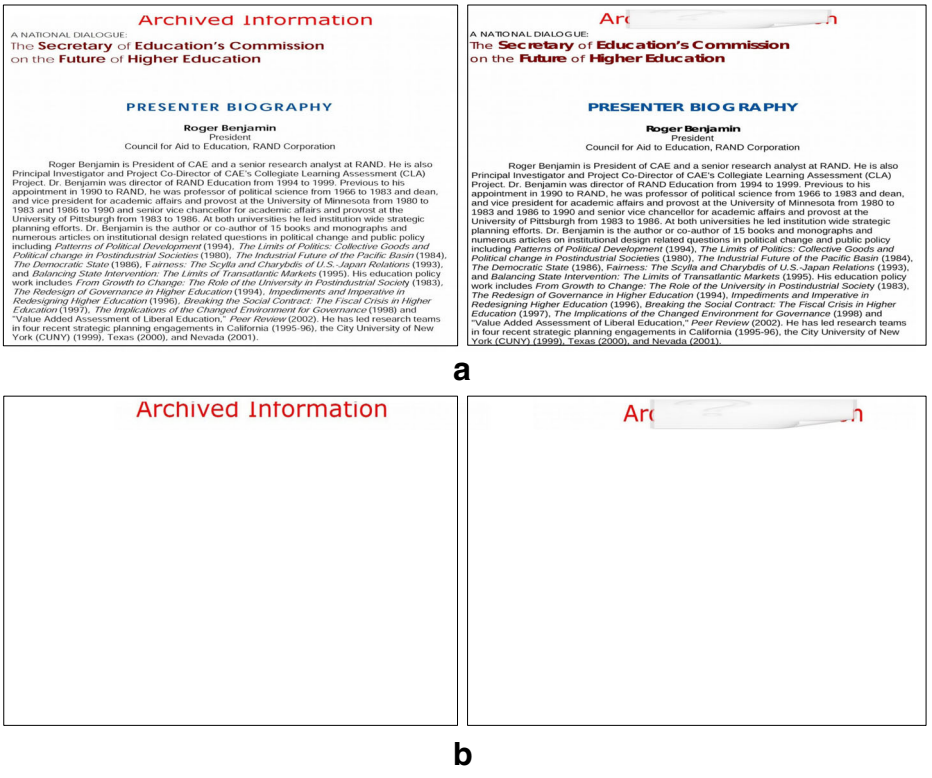


Fig. 8 Inconsistency between Adobe Reader (left) and Okular (right) before (a) and after reduction (b)

inconsistency localization. As we can see, the file has been reduced to only contain the inconsistent element.

43% of the incorrectly reduced PDF files are due to the inconsistency not being caused by a specific element. For example, `MuPDF` has a bug in its fullscreen functionality that does not work correctly when the size of the page is too large or too small compared to the screen's resolution, regardless of the elements in the file. The other 57% of the cases seem to be caused by false positives in CW-SSIM.

In number of bytes, reduced files are 31% smaller than the original file. This is a relatively small reduction compared to the one in number of displayed objects, as a PDF file must contain many base structures to remain valid. We refer to the size of these base structures as the *base file size*. Using the base file size, we can also compute the relative size reduction, i.e., $\frac{reduced\ file\ size - base\ file\ size}{original\ file\ size - base\ file\ size}$. Our approach has a substantial relative size reduction of 86%.

Summary The experiments presented in this section try to answer the research question *How well does our inconsistency localization based on delta debugging perform?* We show that in most cases the localization algorithm was able to reduce the "buggy" document to just a couple of visible objects.

7 Cross-OS inconsistencies

Since there are more Windows users than Linux users, there is a possibility that developers of PDF readers spend more time improving the reliability of their PDF readers on Windows than on Linux, and it is possible that the inconsistencies we found are only a problem on Linux.

In this section, we aim at determining whether the files we detected as inconsistent on Linux readers also reveal bugs in Windows PDF readers. More specifically, we aim at answering the following two research questions:

Question 1: Are the inconsistencies we detected OS specific?
Question 2: Can we use the files we detected as inconsistent on Linux to detect bugs in another OS and other PDF readers?

7.1 PDF readers on windows

To understand whether the inconsistencies we found are OS specific, we chose, when possible, equivalent Windows versions of the software we tested on Linux. More precisely, `Firefox`, `Chromium` and `MuPDF` have an equivalent version for Windows. For `Xpdf` we only found a more recent version (4.00.1) while for `Evince`, only an older version was found (2.32.0.145). For `Acrobat Reader`, we chose to use the most recent version on Windows (Acrobat Reader DC) as it is likely to be the most used PDF reader on this OS. `Okular` and `qpdfview` do not have a Windows version available.

We completed this set of Windows PDF readers with two popular readers that are specific to Windows. The first one is `Edge`, Microsoft default browser on Windows 10, and the second one is `Sumatra PDF`, a popular lightweight PDF reader. In total we evaluated 8 PDF readers on Windows 10.

Table 9 Comparison between bugs on Linux and Windows version of PDF readers

PDF reader	# Linux	# Windows
Firefox	7	5 (71%)
Chromium	4	3 (75%)
mupdf	2	1 (50%)
Evince	3	3 (100%)
Total	16	12 (75%)

Linux column shows the number of bugs reported on Linux. # Windows represents the number and percentage of the reported bugs that also occur in the Windows version of the reader

7.2 Experiment details

For this experiment, we focused on the files we reported during our evaluation of Linux PDF readers. We reported 19 files revealing 22 different bugs. We focused on these files because these represent unique bugs that we found.

7.3 Results

7.3.1 Are the inconsistency we detected OS specific?

To answer this research question, we focus on bugs we reported for PDF readers that are cross-OS (e.g., `Firefox`, `Chromium`). The goal is to find out whether these bugs only occur in the Linux version or in both versions of the reader. When possible, we used the same version of the reader (`Firefox`, `Chromium`, `MuPDF`). In the case of `Evince`, the Windows version is older than the Linux version.

The results displayed in Table 9 show that 75% of the bugs we reported for Linux also occur on Windows 10. The reason is that most of these bugs occur in the shared backend library that parses the PDF file and not in the OS specific source code.

7.3.2 Can we use the files we detected as inconsistent on Linux to detect bugs in another OS and other PDF readers?

To answer this question, we check how many of the inconsistent files we reported for Linux readers also reveal bugs on Windows readers. Table 10 displays the number of reported inconsistent files on Linux that also reveal bugs on Windows readers. For example, the Acrobat column indicates that 5 of the 19 inconsistent files on Linux readers reveal bugs in Acrobat Reader on Windows. We reported 11 bugs to the developers for the Windows readers (4 for `Acrobat Reader`, 3 for `Edge` and 4 for `Sumatra PDF`).

The files we reported for Linux helped us find inconsistencies in all the Windows readers tested. In particular, we found inconsistencies in Acrobat Reader DC, the most recent

Table 10 Number of reported inconsistent files on Linux that also reveal bugs in Windows readers

Acrobat	Firefox	Chromium	Edge	Sumatra	Evince	mupdf	xpdf
5/19	11/19	3/19	5/19	5/19	7/19	6/19	4/19

version of Acrobat Reader on Windows. We also found 5 inconsistencies in both Microsoft Edge and Sumatra, two PDF readers that do not have an equivalent reader working on Linux.

This study highlights that the PDF inconsistency problem is not restricted to Linux readers, and that the files which are inconsistent on Linux readers are also likely to reveal inconsistencies in Windows readers.

8 Discussion and threats to validity

8.1 Conclusion validity

Visual inconsistencies can be subjective. While most of them are clear (e.g., crashes and missing images), some are ambiguous (e.g., colour differences can be so small that some people consider them different while others see no difference). Therefore, there is a potential threat regarding the labelling of inconsistencies in our study. To mitigate this issue, the labelling was done by one of the authors, then independently verified by another author and two other students.

We empirically choose an optimal threshold and image comparison algorithm for our data set. However our study covers very diverse documents, and the optimal threshold or image comparison algorithm might be different for specific types of documents (e.g., image or text). In the future, we plan to take the content of the file into consideration for choosing the optimal threshold and image comparison algorithm to use.

8.2 Internal validity

Despite its large size of about 230K files, the PDF files used in our evaluation come from a single source of the U.S. government's websites. Thus, the results on other PDF files may be different. However, the data set should be reasonably representative for real-world PDF files. To mitigate this issue, we did an extensive study on the data set, investigated the files' metadata and found out the files were produced by a wide range of software.

We mostly focus on PDF readers available on Linux. However, we claim the PDF inconsistency issue is generalizable to other platforms. To support this claim, we did a small study on the 18 files we reported for Linux PDF readers and found that these files also reveal inconsistencies in 8 different Windows PDF readers.

8.3 Construct validity

If one PDF file exposes the same bug in all readers, and causes all readers to fail in the same way (e.g., all display an identical but wrong image), our approach would fail to detect this bug as there are no visual inconsistencies. However, in practice, we have not seen such cases during our manual examination.

8.4 External validity

Although the presented technique is tuned for PDF documents, it is not bound to the PDF format. The same methodology of capturing error messages and return codes, clustering and then comparing screenshots could be applied to other document readers, like e.g., MS Word viewers, image viewers or web browsers. The aspects that need to be adjusted for a specific application, e.g., a different document type, are the regular expressions used for

the emitted messages and the cropping functionality which removes application specific UI elements from the screenshots. The technique is applicable in a broader context because we treat programs in the portfolio in a black-box manner and capture their externally visible behaviour only.

More generally, the technique can be used to tackle the problems that arise when cross-checking a portfolio of programs on a large data set of inputs: which inputs to select (error messages), how to group similar inputs (clustering) and how to detect issues using a similarity metric (image comparison). The presented tool can be used by end-users to check that the published document is presented as expected and by developers to automatically detect bugs in a portfolio of viewers.

8.5 Practical applicability

We envision the practical applicability of this research in two directions: 1) as empirical evidence/systematic study of the PDF inconsistency problem, and 2) as a technique/methodology for cross-testing a portfolio of reader programs and for cross-verification of document correctness.

Our empirical research highlights and quantifies the, otherwise anecdotal, problem of cross-reader PDF inconsistencies. The fact that these inconsistencies appear is an interesting problem on its own due to the intended portable nature of the PDF format. The results also show the difficulty of implementing a complex document format standard and a potential for inconsistencies to appear across different implementations of the standard. We hope that our empirical study will spawn further research into document correctness and reader testing.

Our proposed inconsistency detection technique and prototype could be applied by:

- Developers to test their implementations of parsers and viewers. Our error clustering algorithm can help select interesting documents, the delta-debugging component can help to narrow down the problems and the minimized document can serve as a test case.
- Publishers/designers to make sure that the document looks consistently and as expected across multiple viewers (similarly to what is being done for web pages).
- End-users for sanity checks against bugs such as missing images or inability to load a file in certain readers.
- Other researchers who want to study similar inconsistency issues but find the data set is too large for thorough analysis.

9 Related work

The cross-reader inconsistency issue is analogous to the more studied cross-browser inconsistency (CBI) problem (Roy Choudhary 2014; Saar et al. 2014; Choudhary et al. 2010, 2012; Mesbah and Prasad 2011; Ochin 2011; Choudhary 2011; Eaton and Memon 2007). Different browsers can render the same webpage differently, and some of those inconsistencies are critical when a browser does not support a particular HTML element.

Eaton and Memon first introduced the CBI issue in Eaton and Memon (2007). They propose an approach based on an inductive model to detect bugs in web application that can lead to CBIs. This first approach to detect CBIs only focus on incorrect web applications. If a bug in a specific browser creates an inconsistency for a correct web application, then this approach would not detect it. In addition, this approach is based on HTML tags and is not transferable to the cross-reader inconsistency problem.

The CBI problem has been addressed incrementally by Choudhary et al. in several papers (Roy Choudhary et al. 2014; Choudhary et al. 2010, 2012; Choudhary 2011). First, Webdiff, described in Choudhary et al. (2010) and Choudhary (2011), combines a structural analysis of the DOM structure of a web page and a histogram comparison of screenshots of the web page opened in two different browsers to detect CBIs. CrossCheck (Choudhary et al. 2012) is built on the top of WebDiff, with the addition of a web crawler allowing to detect inconsistencies on entire web applications instead of single web pages. Finally, X-pert (Roy Choudhary et al. 2014) improves CrossCheck by only considering elements that are leaf nodes of the DOM structure. It builds a model for each browser by crawling a specific web application. The models include transitions between pages, as well as the DOM structure and a screenshot of each page. The models are then checked for equivalence using the chi-square distance between the histogram of each element of the screenshot.

Concurrently with WebDiff (Choudhary et al. 2010; Choudhary 2011), Mesbah and Prasad proposed a very similar approach to detect cross-browser inconsistencies (Mesbah and Prasad 2011). The main difference with WebDiff is that they consider the trace-level behavior of the web application.

Different causes for CBI were identified in Ochin (2011). The most common reasons for CBI are HTML tags, CSS, font rendering, DOM, scripts, add-ons and third-party entities. Out of all these reasons, only the font rendering issue is also applicable to the cross-reader inconsistency issue.

While the importance of studying cross-browser inconsistencies has been widely recognized, cross-reader inconsistencies have been under-studied. Furthermore, some of the techniques used to detect cross-browser inconsistency are not directly applicable to cross-reader inconsistencies (for example, they rely on matching the DOM, the structured representation of HTML documents, but the structure of a PDF does not change when opened by different readers). On the other hand, our approach is applicable to a wide variety of electronic documents, including HTML, because it treats the documents and readers as black boxes.

The work from the cross-browser testing area that is most closely related to ours is `Browserbite` (Saar et al. 2014). As in our approach, the technique operates in a black-box manner; it combines image processing for inconsistency detection with machine learning for improving accuracy. However, in addition to the different domain, `Browserbite` is evaluated on only 140 websites, which does not raise the same scalability challenges that we encountered for our corpus of 230 K documents, which requires solutions such as our clustering approach. We use a white-box binary search to find inconsistent locations, while `Browserbite` splits image into regions and performs a linear comparison of all regions; we also use CW-SSIM for image comparison, while `Browserbite` uses histograms.

Many techniques have been developed to detect malicious or vulnerable PDF files (Smutz and Stavrou 2012; Corona et al. 2014; Maiorca et al. 2013; Tzermias et al. 2011; Laskov and ŠRndić 2011; Laskov 2013). Indeed, PDF files can contain embedded JavaScript elements which can be potentially vulnerable or malicious. Common methods to detect such files consist of analysing the metadata and the structure of the PDF file to detect malicious JavaScript components.

MDScan (Tzermias et al. 2011) is a stand-alone tool that combines static analysis of the document and dynamic analysis to detect malicious documents. First, a static analysis of the PDF file is used to detect and extract any embedded JavaScript source code. Then this code is executed and MDScan attempts to detect malicious shellcode execution.

PJScan (Laskov and ŠRndić 2011) is the first approach using static analysis to detect vulnerable PDF files. It focuses on extracting features from Javascript code embedded in PDF

files. Smutz and Stavrou (2012) present another approach based on machine learning that aims to detect vulnerabilities. The features used to detect malicious PDF files are extracted from the metadata and the structure of the PDF (e.g., number of pages, objects, producer, javascript elements, etc.). This approach was compared to PJScan and provided much better results. An improvement of existing static detection (Laskov and ŠRndić 2011; Smutz and Stavrou 2012) was proposed in Laskov (2013). The main change was to consider features in object streams. As object streams are generally encoded, they are often used to hide malicious code. While machine learning could also be used to detect potential inconsistent elements, this approach is very different from our work. In this work, we chose a dynamic approach: both the error clustering and the screenshot comparison require executing the PDF reader.

Maiorca et al. (2013) present a new approach to generate malicious PDF files that cannot be detected by techniques based on machine learning and propose a new tool, Lux0R (Corona et al. 2014) to detect this kind of attack.

While dealing with PDF files, those studies consider a completely different problem from ours. We focus on benign documents and do not consider malware. Our technique, through detecting display and behavior inconsistencies across different readers, has the potential to identify malicious PDF targeting a specific PDF reader. However, we did not investigate this possibility and assumed that all the files mined from the US government websites are benign.

To address cross-reader inconsistencies, standards such as PDF/A and PDF/X have been proposed to ensure that PDF files will always be rendered consistently (ISO 2001, 2005). However, those specifications are highly restrictive and typically only used for archiving data. For example, when looking at the sample of PDF files we extracted from the `Govdocs1` data set, we found that none of these files conformed to the PDF/A or PDF/X standards. Another possible solution to reduce display differences is flattening the PDF file. However, flattening a PDF file may add more inconsistencies when merging the different layers of the original file. In addition, flattening may break the internal structure of the file, making some operations (e.g., selecting the text) impossible. Therefore, a new solution is necessary to help users know whether the PDF they share will be viewed correctly across the readers.

Prior work on recovering electronic documents (Kuchta et al. 2014; Demsky and Rinard 2005; Long et al. 2012) can leverage our inconsistency detection and localization techniques. Previous work in this space uses image similarity to assess the quality of repaired input (Long et al. 2012).

10 Conclusions and future work

This paper presents and quantifies the research problem of cross-reader inconsistencies, which are caused by bugs in readers and files. We conduct an empirical study on 2,313 PDF files, which shows that cross-reader inconsistencies are common. In addition, we propose techniques to detect and localize inconsistencies automatically on over 230K PDF documents. Our approach has detected 30 unique bugs on Linux. We also reported 33 bugs to developers, 17 of which have already been confirmed or fixed.

In the future, we plan to automatically fix the detected bugs in PDF files to ensure consistency across readers. The fault localisation component of our technique based on delta-debugging makes it possible to narrow down the problem to specific objects in the document. Then we could focus on removing or fixing those problematic objects. Fixing PDF documents is not a trivial task due to document format constraints such as cross-reference table that lists object offsets in the file, or object numbers which are used to cross-reference

objects within the document. Even removal of a problematic object may result in a need to update other objects referencing the removed one or object offsets in the mentioned cross-reference table.

Another potential line of work is to extend the technique to other operating systems and document types. That would involve preparing the corresponding virtual machines and tuning our screenshot capture tools. Other document/file types might also need a different similarity metric—a study of ROC curves similar to the presented one might be necessary to pick the best metric for the task.

Finally, with more engineering effort we could improve system's performance and try to analyze randomly chosen pages of the document or a whole document, rather than the first page only. We believe that the current performance is reasonable given the data set size. However, more work in this area would make it possible to further scale up the technique.

Our database of clusters of PDF files and detected bugs, which we make available at http://srg.doc.ic.ac.uk/projects/pdf-errors, could help researchers and practitioners address software reliability challenges including testing other readers, and diagnosing and fixing bugs in readers and PDF files.

Acknowledgments We would like to thank Zhou Wang for his help on image comparison, William Culhane and Lukas Rupprecht for helpful discussions, and Thomas Joseph for project infrastructure support.

This research was generously supported by EPSRC through the Early-Career Fellowship EP/L002795/1, Microsoft Research through a PhD scholarship, the Natural Sciences and Engineering Research Council of Canada, and an Ontario Graduate Scholarship award.

References

Archlinux (2015) [wontfix] wrong colours in adobe reader (acroread). https://bbs.archlinux.org/viewtopic.php?id=193918

Arthur D, Vassilvitskii S (2007) k-means++: the advantages of careful seeding. In: Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms. Society for industrial and applied mathematics, Philadelphia, pp 1027–1035

Bugzilla (2016) bug 94260 - pdf file doesn't load or is displayed inconsistently. https://bugs.freedesktop.org/show_bug.cgi?id=94260

Bugzilla@Mozilla (2016) Bug 1244729 - [PDF Viewer] Incorrect PDF display (large portions of the map appear as black). https://bugzilla.mozilla.org/show_bug.cgi?id=1244729

Choudhary SR (2011) Detecting cross-browser issues in web applications. In: 2011 33rd international conference on Software engineering (ICSE). IEEE, Piscataway, pp 1146–1148

Choudhary SR, Versee H, Orso A (2010) Webdiff: automated identification of cross-browser issues in web applications. In: 2010 IEEE international conference on Software maintenance (ICSM). IEEE, Piscataway, pp 1–10

Choudhary SR, Prasad MR, Orso A (2012) Crosscheck: combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In: 2012 IEEE fifth international conference on Software testing, verification and validation (ICST). IEEE, Piscataway, pp 171–180

Chromium Bug Tracker (2016) PDF's not displaying with Chromes PDF Distiller. https://code.google.com/p/chromium/issues/detail?id=333918

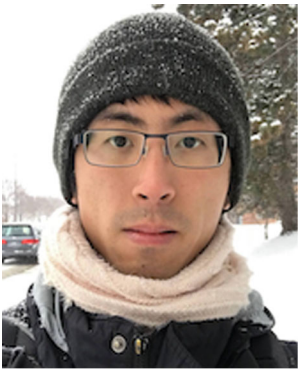
Complex-wavelet structural similarity index (cw-ssim) (2013) http://www.mathworks.com/matlabcentral/fileexchange/43017-complex-wavelet-structural-similarity-index--cw-ssim-

Corona I, Maiorca D, Ariu D, Giacinto G (2014) Lux0r: detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In: Proceedings of the 2014 workshop on artificial intelligent and security workshop. ACM, New York, pp 47–57

Demsky B, Rinard M (2005) Data structure repair using goal-directed reasoning. In: Proceedings of the 27th international conference on software engineering. ACM, New York, pp 176–185
Eaton C, Memon AM (2007) An empirical approach to evaluating web application compliance across diverse client platform configurations. Int J Web Eng Technol 3(3):227–253
ENGINEERING IC (2008) PDF Specification for IEEE xplore (Part A-core requirements). IEEE, Piscataway
Garfinkel S, Farrell P, Roussev V, Dinolt G (2009) Bringing science to digital forensics with standardized forensic corpora. Digit Investig 6:S2–S11
Google Chrome Help Forum (2015) PDF viewer bug with tcpdf. https://productforums.google.com/forum/#!msg/chrome/tVNKJhiv-XQ/tH9RZyPlJGwJ
Grajeda C, Breitinger F, Baggili I (2017) Availability of datasets for digital forensics–and what is missing. Digit Investig 22:S94–S105
Huynh-Thu Q, Ghanbari M (2008) Scope of validity of psnr in image/video quality assessment. Electron Lett 44(13):800–801
ISO (2001) Part 1, graphic technology: pre press digital data exchange. ISO, Geneva
ISO (2005) Document management: electronic document file format for long-term preservation. ISO, Geneva
Kuchta T, Cadar C, Castro M, Costa M (2014) Docovery: toward generic automatic document recovery. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering. ACM, New York, pp 563–574
Laskov P (2013) Detection of malicious pdf files based on hierarchical document structure. In: Proceedings of the network and distributed system security symposium, NDSS 2013. The internet society, Reston
Laskov P, ŠRndić N (2011) Static detection of malicious javascript-bearing pdf documents. In: Proceedings of the 27th annual computer security applications conference. ACM, New York, pp 373–382
Long F, Ganesh V, Carbin M, Sidiroglou S, Rinard M (2012) Automatic input rectification. In: 2012 34th international conference on software engineering (ICSE). IEEE, Piscataway, pp 80–90
MacQueen J (1965) On convergence of k-means and partitions with minimum average variance. In: Annals of mathematical statistics, vol 36, p 1084. INST MATHEMATICAL STATISTICS IMS BUSINESS OFFICE-SUITE 7, 3401 INVESTMENT BLVD, HAYWARD, CA 94545
MacQueen J et al (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth berkeley symposium on mathematical statistics and probability, vol. 1, Oakland, pp 281–297
Maiorca D, Corona I, Giacinto G (2013) Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security. ACM, New York, pp 119–130
Matlabpyrtools (2016) http://www.cns.nyu.edu/~lcv/software.php
Mesbah A, Prasad MR (2011) Automated cross-browser compatibility testing. In: Proceedings of the 33rd international conference on software engineering. ACM, New York, pp 561–570
Mozilla Support Forum (2013) PDF.js not being displayed correctly. https://support.mozilla.org/en-US/questions/948061
Ochin JG (2011) Cross browser incompatibility: reasons and solutions. International Journal of Software Engineering & Applications (IJSEA) 2(3):66–77
Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math 20:53–65
Roy Choudhary S, Prasad MR, Orso A (2014) X-pert: a web application testing tool for cross-browser inconsistency detection. In: Proceedings of the 2014 international symposium on software testing and analysis. ACM, New York, pp 417–420
Saar T, Dumas M, Kaljuve M, Semenenko N (2014) Cross-browser testing in browserbite. In: Web engineering. Springer, Berlin, pp 503–506
Sampat MP, Wang Z, Gupta S, Bovik AC, Markey MK (2009) Complex wavelet structural similarity: a new image similarity index. IEEE Trans Image Process 18(11):2385–2401
Scikit learn (2017) http://scikit-learn.org/stable/
Smutz C, Stavrou A (2012) Malicious pdf detection using metadata and structural features. In: Proceedings of the 28th annual computer security applications conference. ACM, New York, pp 239–248
Solomon JA, Pelli DG et al (1994) The visual filter mediating letter identification. Nature 369(6479):395–397
Tzermias Z, Sykiotakis G, Polychronakis M, Markatos EP (2011) Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the fourth european workshop on system security. ACM, New York, p 4
Xvfb (2010) x window system version 11 release 7.6. http://www.x.org/archive/X11R7.6/
Zauner C (2010) Implementation and benchmarking of perceptual image hash functions na
Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200

Tomasz Kuchta is an engineer at Qualcomm, where he works on product security. His interests span across the areas of symbolic execution, dynamic software analysis, security, systems and software reliability. Tomasz has a PhD in Computer Science from Imperial College London, a Master's degree from Cracow University of Technology, and several years of software engineering experience in the industry.

Thibaud Lutellier received the Diplôme d'Ingénieur from Télécom Saint-Etienne and the M.A.Sc. degree in computer engineering from the University of Waterloo. He is currently working toward the PhD degree at the University of Waterloo. His research interests include bug detection, program repair, and machine learning.



Edmund Wong is a PhD student at the University of Waterloo, and he had received his master's degree in 2014. His research interest includes automated documentation generation for source code comments and the application of documentation analysis to improve software reliability.

Lin Tan a Canada Research Chair, is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Waterloo. She received her PhD from the University of Illinois, Urbana-Champaign. She is an associate editor of IEEE Transactions on Software Engineering (2017-present) and an editor of the Springer Empirical Software Engineering Journal (2015-present). She was the program co-chair of MSR 2017, ICSE-NIER 2017, and ICSME-ERA 2015. Her co-authored papers have received an ACM SIGSOFT Distinguished Paper Award at FSE in 2016 and IEEE Micro's Top Picks in 2006. Dr. Tan is a recipient of an NSERC Discovery Accelerator Supplements Award, an Ontario Early Researcher Award, an Ontario Professional Engineers Award – Engineering Medal for Young Engineer, a University of Waterloo Outstanding Performance Award, two Google Faculty Research Awards, and an IBM CAS Research Project of the Year Award.

Cristian Cadar is a Reader (Associate Professor) in the Department of Computing at Imperial College London, where he leads the Software Reliability Group. His research interests involve designing practical techniques and tools for improving the reliability and security of software systems. Cristian has received several research awards, including the Jochen Liedtke Young Researcher Award, the HVC Award and the ACM CCS Test of Time Award. He was general chair for the European Conference on Computer Systems (EuroSys) 2016 and co-chair for the New Ideas Track of the International Conference on Software Engineering (ICSE NIER) 2017. He is Associate Editor for the ACM Transactions on Software Engineering and Methodology (TOSEM). Cristian has a PhD in Computer Science from Stanford University, and undergraduates and Master's degrees from the Massachusetts Institute of Technology.