Reversible Data Hiding in PDF Document Exploiting Prefix Zeros in Glyph Coordinates

Neelesh Nursiah, KokSheik Wong
School of Information Technology
Monash University Malaysia, Malaysia.
{nnur15@student., wong.koksheik@}monash.edu

Minoru Kuribayashi
Graduate School of Natural Science and Technology
Okayama University, Okayama, Japan
kminoru@okayama-u.ac.jp

***Abstract*—In the contemporary world of information technology, PDF (Portable Document Format) has become the de facto document standard which allows users to exchange and view electronic documents across various platforms. PDF is the most widely exchanged document format since Internet gained popularity. Although PDF has a good authenticity system by making use of digital signatures, the file format is still susceptible to copyright infringement as there are many libraries available on the Internet to bypass the digital signature of a PDF. Therefore, claiming ownership for PDF has become a paramount issue that needs to be addressed. This paper proposes the idea of hiding data in the glyph positioning coordinate value. To suppress bit stream size increment, the reverse zero-run length coding technique is adopted. Experiments are conducted to verify the basic performance of the proposed data hiding method. In the best case scenario, 0.62 bits of data can be embedded into each Byte of the PDF file. The injected leading zeros can be removed to restore the original PDF file.**

***Index Terms*—PDF, copyright infringement, glyph positioning coordinate value, data hiding, reverse zero-run length**

I. Introduction

Due to massive growth of Internet, Portable Document Format, commonly referred to as PDF [1], has become the staple for worldwide electronic document exchange. This electronic file format is preferred over other formats because it has many fitting functionalities, hence allowing better page navigation, interactive viewing, and so on. It also provides propitious security to maintain the authenticity, accessibility, confidentiality and integrity of a document. Many organizations are realizing the importance of this file format and are gradually replacing paper-based information circulation by PDF.

Data hiding is a technique which injects payload (data to be hidden) into a container such as video, audio, PDF and so on. Among numerous methods in data hiding, the mostly investigated ones are steganography and watermarking. The former is derived from two Greek words, i.e., steganos which means concealed and graphos which means writing. The concealment of the secret communication by means of container is the main purpose of the steganography, while the watermark is used to protect the container by inserting information for the purpose of copyright protection and fingerprinting. In steganography, only the sender and the intended recipient are aware of the fact that data has been concealed into the container [2].

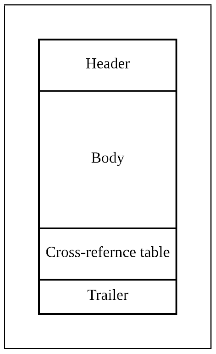
This work was partly supported by E-Science grant (01-02-10-SF0327) by MOSTI.

It is commonly recognized that the trade-off among payload, transparency, and robustness should be considered for a good data hiding method. In addition, the secrecy of the existence of the hidden message should be managed against malicious party. In case of PDF, the expansion of the file size should be suppressed so as not to be noticed. In conventional works [3]–[8], the spaces between words and characters are slightly modified to insert message. Bitar et al. [6] used the STDM (spread transform dither modution) [9] to maintain the secrecy of message, although the payload is small. Kuribashi et al. [7], [8] utilized an ordinary DM (dither modulation) [9] combined with a random permutation prior to frequency transformation to enhance the secrecy. It can control the transparency and robustness by flexibly changing the payload. However, even if the distortions caused by the modification is imperceptible, transparency is sacrificed in these methods.

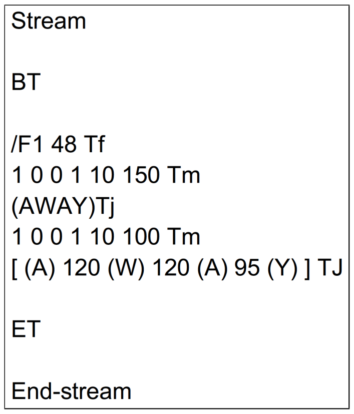
In this paper, we propose a novel technique to hide data into PDF file without causing any visual distortion. The data is hidden by manipulating the glyph positioning coordinate values. To suppress bit stream size increment, the reverse zero-run length coding technique [10] is adopted. Compared to other state of the art data hiding techniques based on space length manipulation, the proposed method causes zero distortion to the visual appearance while the conventional methods leads to some changes in spacing. Thanks to the functionalities in PDF format, the permissions to be viewing/editing can be managed by giving a password. Hence, it enables us to give a privilege to a user who has the password can receive the hidden message, and other users may not notice the existence of the message. In this case, a stagananalyzer must check the irregularity of PDF file without the password. As the visual appearance is not changed, he/she must analyze the document of the file. Due to the permission setting of the file, such an analysis is difficult. As the experimental result shows that the expansion of the file size is small, the hidden message can be kept secret in the proposed method.

II. Preliminaries

The structure of a PDF container is as shown in Fig. 1. It consists of 4 parts namely, Header, Body, Cross-reference table and Trailer. Specifically, the header contains information about the file so that a PDF parser/reader can detect whether the file is a PDF or not, and it also indicate the version of the



(a) Layout of PDF File



```
Stream

BT

/F1 48 Tf
1 0 0 1 10 150 Tm
(AWAY)Tj
1 0 0 1 10 100 Tm
[ (A) 120 (W) 120 (A) 95 (Y) ] TJ

ET

End-stream
```

(b) Syntax of PDF stream

Fig. 1. Layout of PDF file and syntax of PDF stream

PDF. The body consists of the actual document itself, which is made up of nine types of objects. These objects are the basic building blocks of the PDF, including null, Boolean, integer, real, name, string, array, dictionary and stream. Each object has either a direct- or indirect reference marker and a generation number (i.e., binary offset). On the other hand, the cross-reference table allows quick reference to every indirect object that the PDF processor requires at any particular time. Finally, the trailer is a dictionary with key-value pairs that provides information (e.g., size and root) that is needed in order to parse / process the PDF file.

In the body of the PDF, the syntax for rendering text, images and figures is enclosed in the stream object, i.e., between *stream* and *end-stream*. Within this stream object, we are interested in the TJ operator enclosed between *BT* and *ET*, which stands for *Begin Text* and *End Text*, respectively. The TJ operator takes an array as an input, where the array consist of one or more strings with numbers. The number is referred to as the *glyph positioning value* expressed in thousandth of a unit. The number appears between strings, for the purpose of adjusting the position of the characters / words. An example of the TJ operator and its input array is

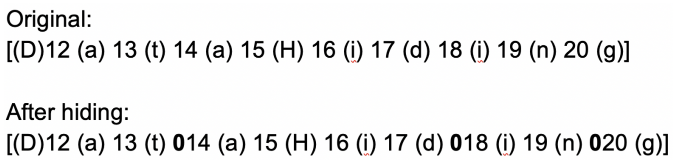
$$[\ (A)\ 120\ (W)\ 120\ (A)\ 95\ (Y)\ ]\ \text{TJ} \tag{1}$$

which is shown in Fig. 1(b).

The operator and input array are utilized to display the word AWAY, and the glyph positioning values adjust the placement of alphabets in a more appealing manner.

III. Proposed Data Hiding Method

In PDF, a number (viz., float or integer) can have a leading zero, which does not change the value of that number. For instance the number 12.3 can be written as 012.3. The look-and-feel of the document rendered from a modified PDF file appears to be exactly the same as that of its original unprocessed counterpart. This has been verified by visual



Original:
[(D)12 (a) 13 (t) 14 (a) 15 (H) 16 (i) 17 (d) 18 (i) 19 (n) 20 (g)]

After hiding:
[(D)12 (a) 13 (t) **0**14 (a) 15 (H) 16 (i) 17 (d) **0**18 (i) 19 (n) **0**20 (g)]

Fig. 2. Changes observed in an input array due to data hiding.

inspection[1]. By exploiting the redundant (leading) zero, data can be inserted. Specifically, the presence of a leading zero in a glyph positioning value will signify the bit ‘1’ (e.g., 012.3). On the other hand, the absence of leading zero signifies the bit ‘0’ (e.g., 12.3), where the value remains unchanged. The following steps are performed to hide data into a PDF:

1) Let μ be the data (represented in binary format) to be hidden.
2) A PDF file is decompressed using the Qpdf library [] since the stream objects are usually packed in some compressed form.
3) Locations of the TJ operator are identified.
4) For each identified TJ operator, its input array is parsed to manipulate the glyph positioning coordinate values. Here, a leading zero is injected if ‘1’ is to be embedded. Otherwise, the original representation remains unchanged to signify ‘0’.
5) Step 4 is repeated until all data segments μ_i to be embedded are processed, or when all TJ operators or glyph positioning values are exhausted.
6) A new PDF file is constructed by replicating the unmodified parts of the PDF bit stream, which includes basically all elements, except the potentially modified glyph position values along with the TJ operators.
7) The PDF file is compressed by using the Qpdf library.
8) The PDF file is then repaired using the PDFtk tool [11].

For instance to hide 001000101, the input array for a TJ operator will undergo some changes as illustrated in Fig. 2, where a leading zero is injected at the 3rd, 7th, and 9th numbers. On the other hand, for data extraction, the following steps are carried out using an empty queue Q:

1) The PDF is decompressed using the Qpdf library.
2) The locations of the TJ operators are located.
3) For each TJ operator, its input array is parsed to locate the glyph position coordinate values. For each located glyph position coordinate value, if a leading zero is encountered, ‘1’ is added to the queue Q. Otherwise, ‘0’ is added to Q.
4) Step 3 is repeated until all glyph position coordinate values are considered.
5) The queue Q then becomes the extracted data μ'.

It is noteworthy that the zeros added during the data hiding phase can be removed to reconstruct the original PDF file, hence achieving reversibility.

[1]The original and processed PDFs are rendered and exported as images. The pixel-to-pixel difference between two images is considered and found to be zero

IV. Suppression of file size increment

Recall that a leading (redundant) zero is introduced to a glyph positioning coordinate value when '1' is to be inserted. Since the data to be hidden is usually random in nature (e.g., encrypted), to hide the data μ, a zero is injected 50% of the time. This may leads to significant file size increment, which is undesirable.

To suppress file size increment, reverse zero-run length (RZL) encoding technique is deployed [10]. Specifically, the data μ is pre-processed so that the resulting data ν is hidden instead of the original μ. First, μ divided into segments of k-bit for $k > 1$, and each segment is processed one at a time. Next, the segment μ^i is converted into the decimal value (denoted by μ^i_{10}) and ν_i is the concatenation of μ^i_{10} number of zeros, followed by unity. For example, $\mu_i = 110$ implies that $\nu_i = 0000001$ (i.e., 6 zeros followed by '1') and $\mu_i = 000$ implies that $\nu_i = 1$ (i.e., no zeros followed by '1') for $k = 3$. The process is repeated until all segments μ_i is processed. Finally, ν is defined as

$$\nu := [\nu_1, \nu_2, \cdots]. \tag{2}$$

The same process detailed in Section III is invoked to embed ν, instead of μ. Similarly, at the receiver's end, ν is first obtained then it is parsed to reconstruct μ. Specifically, each segment of ν_i is delimited by the value '1'. For example, the array

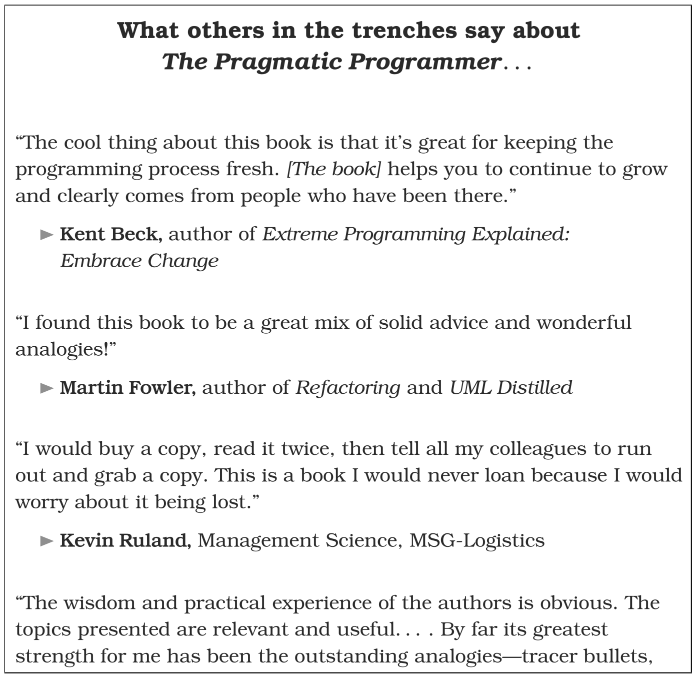
$$\nu = 00001\ 01\ 001\ 1 \tag{3}$$

implies that there are 4 segments (since there are 4 occurrences of '1'). The number of zeros before a '1' is then computed and converted to its binary-equivalent value, with extra zeros padded on the left to make up the length of k bits. For the example given above, 4 data segments are retrieved, namely, $\nu_1 = 00001, \nu_2 = 01, \nu_3 = 001$ and $\nu_4 = 1$, which translate to $\mu_1 = 110$, $\mu_2 = 001$, $\mu_3 = 010$, and $\mu_4 = 000$, respectively. Note that the value $k = 3$ in the illustrated example needs to be communicated to the receiver to decode the data.

V. Experiments

The proposed data hiding method is proposed using Python (version 3.5.2). Qpdf [11] (version 8.0.2) and pdfTK toolkit [12] (version 2.02) are deployed. For 10 random PDFs are considered to verify the basic performance of the proposed data hiding method. It is verified that the original and processed PDF appear to be exactly the same. As a representative example, Fig. 3 shows the original and processed PDF files. In addition, the hidden data can be extracted, and the original PDF can be restored from the processed PDF by removing the injected zeros.

Table I records the file size of each PDF before and after hiding data using different value of k for RZL. Here, the compressed file size is reported, where the results for the processed PDF (i.e., with data hidden) is obtained by decompressing the original PDF, inserting data into the decompressed PDF, repairing the PDF with hidden data, and then re-compressing the resulting PDF. Note that when $k = 1$, the procedure detailed in III is invoked to hide data, i.e., not using RZL. In general, the file size increases when data is hidden into the PDF file, irregardless of the value of k. This is an expected outcome because zeros are injected into the PDF file to encode additional data. In terms of percentage of variation, the range is $[-18\%, 18\%]$. When k increases, the increase in file size also decreases accordingly. Nonetheless, there are some interesting cases where the PDF containing additional data


What others in the trenches say about
***The Pragmatic Programmer*...**

"The cool thing about this book is that it's great for keeping the programming process fresh. *[The book]* helps you to continue to grow and clearly comes from people who have been there."

► **Kent Beck,** author of *Extreme Programming Explained: Embrace Change*

"I found this book to be a great mix of solid advice and wonderful analogies!"

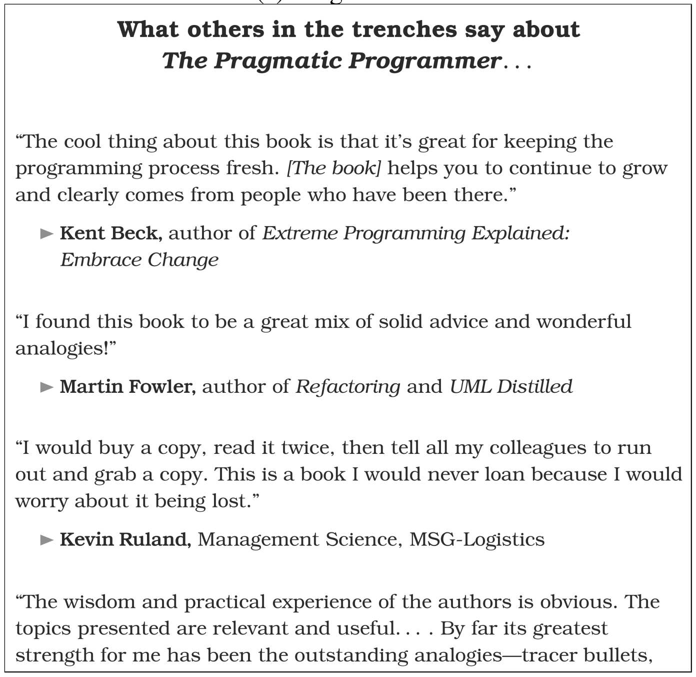
► **Martin Fowler,** author of *Refactoring* and *UML Distilled*

"I would buy a copy, read it twice, then tell all my colleagues to run out and grab a copy. This is a book I would never loan because I would worry about it being lost."

► **Kevin Ruland,** Management Science, MSG-Logistics

"The wisdom and practical experience of the authors is obvious. The topics presented are relevant and useful. . . . By far its greatest strength for me has been the outstanding analogies—tracer bullets,

(a) Original PDF


What others in the trenches say about
***The Pragmatic Programmer*...**

"The cool thing about this book is that it's great for keeping the programming process fresh. *[The book]* helps you to continue to grow and clearly comes from people who have been there."

► **Kent Beck,** author of *Extreme Programming Explained: Embrace Change*

"I found this book to be a great mix of solid advice and wonderful analogies!"

► **Martin Fowler,** author of *Refactoring* and *UML Distilled*

"I would buy a copy, read it twice, then tell all my colleagues to run out and grab a copy. This is a book I would never loan because I would worry about it being lost."

► **Kevin Ruland,** Management Science, MSG-Logistics

"The wisdom and practical experience of the authors is obvious. The topics presented are relevant and useful. . . . By far its greatest strength for me has been the outstanding analogies—tracer bullets,

(b) Processed PDF

Fig. 3. Original PDF and processed PDF with $74,460$ bits embedded. PDF1 is shown here as the representative example.

TABLE I
FILE SIZE [BYTES] BEFORE AND AFTER HIDING DATA INTO 10 DIFFERENT PDF FILES IN THE COMPRESSED FORM.

k	PDF1	PDF2	PDF3	PDF4	PDF5	PDF6	PDF7	PDF8	PDF9	PDF10
Original	1,676,158	2,044,845	5,469,503	12,927,921	15,297,492	15,571,908	5,661,249	38,988,175	10,811,677	25,578,325
1	**1,695,668**	2,045,629	5,478,348	12,332,031	12,497,098	**18,404,930**	5,831,093	38,631,381	**10,837,040**	**26,324,804**
2	**1,689,598**	2,043,453	5,413,863	12,141,251	12,599,464	**18,317,220**	5,821,221	38,626,287	**10,834,802**	**26,324,591**
3	1,678,993	2,039,435	5,394,782	12,113,121	12,500,380	**18,188,208**	5,800,813	38,615,684	10,830,632	**26,324,391**
4	1,668,221	2,034,604	5,383,104	12,113,121	12,552,757	**18,185,956**	5,777,584	38,603,869	10,826,097	**26,324,035**
5	1,656,056	2,029,194	5,378,966	12,113,121	12,460,907	**18,185,956**	5,752,000	38,590,910	10,820,992	**26,323,798**

TABLE II
FILE SIZE [BYTES] BEFORE AND AFTER HIDING DATA INTO 10 DIFFERENT PDF FILES IN THE DE-COMPRESSED FORM.

k	PDF1	PDF2	PDF3	PDF4	PDF5	PDF6	PDF7	PDF8	PDF9	PDF10
Original	3,693,447	4,719,512	34,403,226	111,770,098	24,331,971	64,550,934	24,245,138	189,917,042	119,079,831	184,071,792
1	3,767,846	4,752,838	36,198,831	112,039,658	33,890,051	64,750,799	25,822,818	194,171,244	119,270,038	184,352,205
2	3,747,690	4,743,819	35,845,196	111,796,703	24,466,625	64,646,114	25,781,129	194,145,495	119,262,110	184,351,959
3	3,729,074	4,735,492	35,799,556	111,770,098	39,683,372	64,552,357	25,742,622	194,121,713	119,254,785	184,351,731
4	3,715,598	4,729,459	36,001,807	111,770,098	24,399,218	64,550,934	25,714,747	194,104,494	119,249,479	184,351,565
5	3,704,530	4,724,506	36,545,191	111,770,098	24,337,597	64,550,934	25,691,852	194,090,357	119,245,125	184,351,432

TABLE III
HIDING CAPACITY FOR THE PROPOSED METHOD WHEN USING DIFFERENT k FOR RZL (BITS).

k	PDF1	PDF2	PDF3	PDF4	PDF5	PDF6	PDF7	PDF8	PDF9	PDF10
1	74,460	33,326	1,795,605	269,560	9,558,080	199,865	1,577,680	4,254,202	190,207	280,413
2	54,304	24,307	1,441,970	26,605	134,654	95,180	1,535,991	4,228,453	182,279	280,167
3	35,688	15,980	1,396,330	0	15,351,401	1,423	1,497,484	4,204,671	174,954	279,939
4	22,212	9,947	1,598,581	0	67,247	0	1,469,609	4,187,452	169,648	279,773
5	11,144	4,994	2,141,965	0	5,626	0	1,446,714	4,173,315	165,294	279,640

actually assumes a smaller file size than that of its original counter part (e.g., see PDF1 for $k \geq 5$, or PDF5 for $k = 1$). A potential reason to this phenomenon is that the hidden data, by chance, causes the original bit stream to assume some specific bit patterns, which can be efficiently coded by using shorter codewords. This phenomenon will be further investigated as one of our future work. It is interesting to note that, for most cases, the following relation holds true:

$$FS(P) + |\mu| \geq FS(P'), \tag{4}$$

where P and P' refers to the original and processed PDFs, respectively. $FS(P)$ refers to the file size of P, and $|\mu|$ refers to the size of the payload μ in the unit of bytes. Nonetheless, there are some cases where the Eq. 4 does not hold true, and they are indicated with boldface type setting in Table I.

For comparison purpose, the original and processed file sizes, in their decompressed form, are reported in Table II. It is noteworthy that the results show more consistent trend where the file size increases when data is hidden into a PDF file (compare 2nd row to 3rd row). In addition, file size also decreases, in general, when k increases, which is expected although some PDFs show abnormal trend (e.g., PDF3 and PDF5).

In terms of payload, each PDF can hide certain amount of data (see resulst for $k = 1$). Intuitively, PDF of larger file size (see Table I) will have higher hiding capacity, and vice versa. Nonetheless, the proposed method offers, on average, approximately 0.14 bits per Byte. The detailed results are recorded in Table III. As expected, the hiding capacity decreases as k increases, since up to 2^k positions (i.e., glyph coordinates) can potentially be utilized to hide an original k-bit segment. Although some PDFs (e.g., PDF6) do not offer any hiding capacity when $k > 5$, a smaller k parameter value can be considered. Therefore, there is a trade off between file size and hiding capacity.

The advantage of the proposed method is in terms of transparency because no visual distortion is introduced due to data hiding. If a malicious party observes the body of PDF file directly, the existence of hidden message can be easily recognized. Such a direct observation can be restricted by setting different access permission to the PDF file. As one of the practical scenario, without a password, nobody can check the document of PDF file. As the expansion of file size is kept small in the proposed method, it is difficult to guess the existence of hidden message only from a given PDF file.

VI. CONCLUSIONS

In this paper, a technique is put forward to insert data into a PDF file. Specifically, it is observed that attaching a leading zero to the glyph coordinate value will not crash the decoder and the exact same value is interpreted. A simple technique is put forward to hide data into PDF by exploiting this observation. To suppress bit stream size increment, the data is pre-processed using reverse zero run-length coding. It is verified that the original and processed PDF appear to be identical when they are rendered. It is confirmed that the original PDF file can be restored from its processed counterpart. In the best case scenario, 0.62 bits can be hidden per Byte of the PDF file.

As future work, we shall explore the utilization of leading zeros to hide more data. In addition, the possibility in combining the proposed method and conventional methods will also be investigated.

References

[1] Adobe Systems Incorporated, "Document management — portable document format — part 1: PDF 1.7," ISO 32000-1:2008, July 2008.

[2] L. Y. Por and B. Delina, "Information hiding: a new approach in text steganography," in *Proc. ACACOS'08*, 2008, pp. 689–695.

[3] S. Zhong, X. Cheng, and T. Chen, "Data hiding in a kind of PDF texts for secret communication," *Int. J. Network Security*, vol. 4, pp. 17–26, 2007.

[4] I. S. Lee and W. H. Tsai, "A new approach to covert communication via PDF files," *Signal Processing*, vol. 90, no. 2, pp. 557–565, 2010.

[5] H.-F. Lin, L.-W. Lu, C.-Y. Gun, and C.-Y. Chen, "A copyright protection scheme based on PDF," *Int. J. Innovative Computing, Information and Control*, vol. 9, no. 1, pp. 1–6, 2013.

[6] A. W. Bitar, R. Darazi, J.-F. Couchot, and R. Couturier, "Blind digital watermarking in PDF documents using spread transform dither modulation," *Multimedia Tools and Applications*, vol. 76, no. 1, pp. 143–161, 2017.

[7] M. Kuribayashi, T. Fukushima, and N. Funabiki, "Data hiding for text document in PDF file," in *Proc. IIHMSP'17*, 2017, pp. 390–398.

[8] M. Kuribayashi, T. Fukushima, and N. Funabiki, "Robust and secure data hiding for pdf text document," *IEICE Trans. Information and Systems*, vol. E102-D, no. 1, pp. 41–47, 2019.

[9] B. Chen and G. Q. Wornell, "Quantization index modulation: a class of provably good methods for digital watermarking and information embedding," *IEEE Trans. Inform. Theory*, vol. 47, no. 4, pp. 1423–1443, 2001.

[10] K. Wong, K. Tanaka, K. Takagi, and Y. Nakajima, "Complete video quality-preserving data hiding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 10, pp. 1499–1512, Oct 2009.

[11] Jay Berkenbilt, "Qpdf manual," https://www.cs.rit.edu/doc/qpdf/qpdf-manual.html, accessed July 6, 2019.

[12] Steward and Lee, "Pdftk," https://www.pdflabs.com/tools/pdftk-server, accessed July 6, 2019.