FontCode: Embedding Information in Text Documents using Glyph Perturbation

CHANG XIAO, CHENG ZHANG, and CHANGXI ZHENG, Columbia University

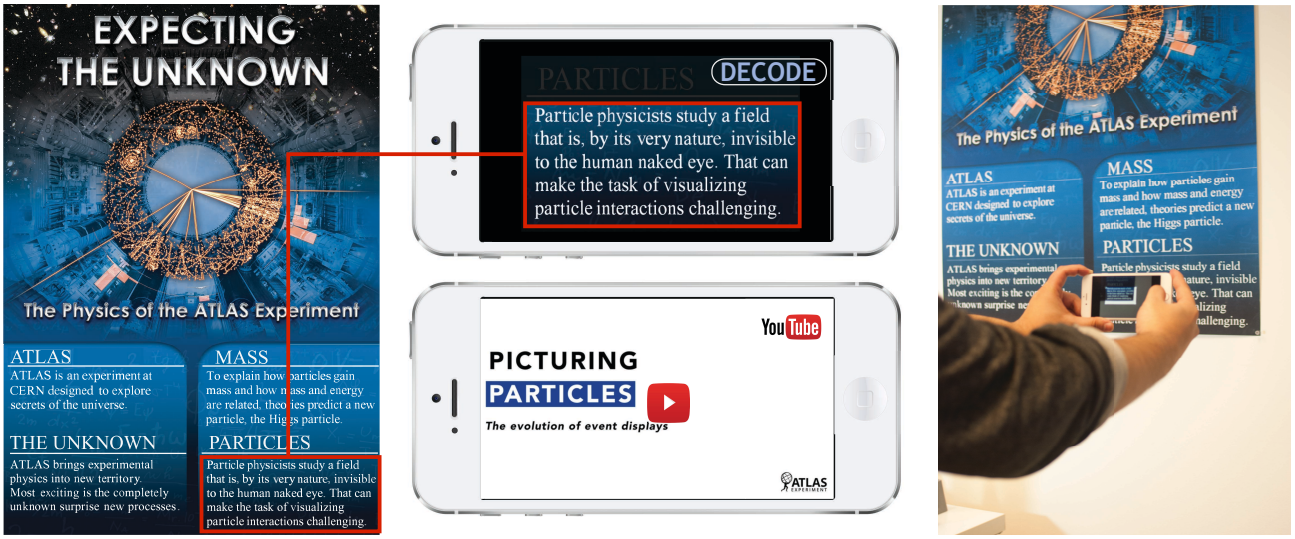


Fig. 1. **Augmented poster.** Among many applications enabled by our FontCode method, here we create a poster embedded with an unobtrusive optical barcode (left). The poster uses text fonts that look almost identical from the standard Times New Roman, and has no traditional black-and-white barcode pattern. But our smartphone application allows the user to take a snapshot (right) and decode the hidden message, in this case, a Youtube link (middle).

We introduce *FontCode*, an information embedding technique for text documents. Provided a text document with specific fonts, our method embeds user-specified information in the text by perturbing the glyphs of text characters while preserving the text content. We devise an algorithm to choose unobtrusive yet machine-recognizable glyph perturbations, leveraging a recently developed generative model that alters the glyphs of each character continuously on a font manifold. We then introduce an algorithm that embeds a user-provided message in the text document and produces an encoded document whose appearance is minimally perturbed from the original document. We also present a glyph recognition method that recovers the embedded information from an encoded document stored as a vector graphic or pixel image, or even on a printed paper. In addition, we introduce a new error-correction coding scheme that rectifies a certain number of recognition errors. Lastly, we demonstrate that our technique enables a wide array of applications, using it as a text document metadata holder, an unobtrusive optical barcode, a cryptographic message embedding scheme, and a text document signature.

CCS Concepts: • **Computing methodologies** → *Image processing*; • **Applied computing** → Text editing; Document metadata;

Additional Key Words and Phrases: Font manifold, glyph perturbation, error correction coding, text document signature

ACM Reference format:
Chang Xiao, Cheng Zhang, and Changxi Zheng. 2017. FontCode: Embedding Information in Text Documents using Glyph Perturbation. *ACM Trans. Graph.* 1, 1, Article 1 (December 2017), 16 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

0730-0301/2017/12-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Information embedding, the technique of embedding a message into host data, has numerous applications: Digital photographs have metadata embedded to record such information as capture date, exposure time, focal length, and camera's GPS location. Watermarks embedded in images, videos, and audios have been one of the most important means in digital production to claim copyright against piracies [Bloom et al. 1999]. And indeed, the idea of embedding information in light signals has grown into an emerging field of visual light communication (e.g., see [Jo et al. 2016]).

In all these areas, information embedding techniques meet two desiderata: (i) the host medium is minimally perturbed, implying that the embedded message must be minimally intrusive; and (ii) the embedded message can be robustly recovered by the intended decoder even in the presence of some decoding errors.

Remaining reclusive is the information embedding technique for text documents, in both digital and physical form. While explored by many previous works on digital text steganography, information embedding for text documents is considered more challenging to

meet the aforementioned desiderata than its counterparts for images, videos, and audios [Agarwal 2013]. This is because the "pixel" of a text document is individual letters, which, unlike an image pixel, cannot be changed into other letters without causing noticeable differences. Consequently, existing techniques have limited information capacity or work only for specific digital file formats (such as PDF *or* Microsoft Word).

We propose *FontCode*, a new information embedding technique for text documents. Instead of changing text letters into different ones, we alter the *glyphs* (i.e., the particular shape designs) of their fonts to encode information, leveraging the recently developed concept of font manifold [Campbell and Kautz 2014] in computer graphics. Thereby, the readability of the original document is fully retained. We carefully choose the glyph perturbation such that it has a minimal effect on the typeface appearance of the text document, while ensuring that glyph perturbation can be recognized through Convolutional Neural Networks (CNNs). To recover the embedded information, we develop a decoding algorithm that recovers the information from an input encoded document—whether it is represented as a vector graphics file (such as a PDF) or a rasterized pixel image (such as a photograph).

Exploiting the features specific to our message embedding and retrieval problems, we further devise an error-correction coding scheme that is able to fully recover embedded information up to a certain number of recognition errors, making a smartphone into a robust FontCode reader (see Fig. 1).

Applications. As a result, FontCode is not only an information embedding technique for text documents but also an unobtrusive tagging mechanism, finding a wide array of applications. We demonstrate four of them. (i) It serves as a metadata holder in a text document, which can be freely converted to different file formats or printed on paper without loss of the metadata—across various digital and physical forms, the metadata is always preserved. (ii) It enables to embed in a text unobtrusive optical codes, ones that can replace optical barcodes (such as QR codes) in artistic designs such as posters and flyers to minimize visual distraction caused by the barcodes. (iii) By construction, it offers a basic cryptographic scheme that not only embeds but also encrypts messages, without resorting to any additional cryptosystem. And (iv) it offers a new text signature mechanism, one that allows to verify document authentication and integrity, regardless of its digital format or physical form.

Technical contributions. We propose an algorithm to construct a glyph codebook, a lookup table that maps a message into perturbed glyphs and ensures the perturbation is unobtrusive to our eyes. We then devise a recognition method that recovers the embedded message from an encoded document. Both the codebook construction and glyph recognition leverage CNNs. Additionally, we propose an error-correction coding scheme that rectifies recognition errors due to factors such as camera distortion. Built on a 1700-year old number theorem, the Chinese Remainder Theorem, and a probabilistic decoding model, our coding scheme is able to correct more errors than what block codes based on Hamming distance can correct, outperforming their theoretical error-correction upper bound.

2 RELATED WORK

We begin by clarifying a few typographic terminologies [Campbell and Kautz 2014]: the *typeface* of a character refers to a set of *fonts* each composed of *glyphs* that represent the specific design and features of the character. With this terminology, our method embeds messages by perturbing the glyphs of the fonts of text letters.

Font manipulation. While our method perturbs glyphs using the generative model by Campbell and Kautz [2014], other methods create fonts and glyphs with either computer-aided tools [Rugglcs 1983] or automatic generation. The early system by [Knuth 1986] creates parametric fonts and was used to create most of the Computer Modern typeface family. Later, Shamir and Rappoport [1998] proposed a system that generate fonts using high-level parametric features and constraints to adjust glyphs. This idea was extended to parameterize glyph shape components [Hu and Hersch 2001]. Other approaches generate fonts by deriving from examples and templates [Lau 2009; Suveeranont and Igarashi 2010], similarity [Loviscach 2010] or crowdsourced attributes [O'Donovan et al. 2014]. Recently, Phan et al. [2015] utilize a machine learning method trained through a small set of glyphs in order to synthesize typefaces that have a consistent style.

Font recognition. Automatic font recognition from a photo or image has been studied [Avilés-Cruz et al. 2005; Jung et al. 1999; Ramanathan et al. 2009]. These methods identify fonts by extracting statistical and/or typographical features of the document. Recently in [Chen et al. 2014], the authors proposed a scalable solution leveraging supervised learning. Then, Wang et al. [2015] improved font recognition using Convolutional Neural Networks. Their algorithm can run without resorting to character segmentation and optical character recognition methods. In our work, we use existing algorithms to recognize text fonts of the input document, but further devise an algorithm to recognize glyph perturbation for recovering the embedded information. Unlike existing font recognition methods that identify fonts from a text of many letters, our algorithm aims to identify glyph perturbation for individual letters.

Text steganography. Our work is related to digital steganography (such as digital watermarks for copyright protection), which has been studied for decades, mostly focusing on videos, images, and audios—for example, we refer to [Cheddad et al. 2010] for a comprehensive overview of digital imaging steganography. However, digital text steganography is much more challenging [Agarwal 2013], and thus much less developed. We categorize existing methods based on their features (see Table 1):

Methods based on cover text generation (**CTG**) hide a secret message by generating an *ad-hoc* cover text which looks lexically and syntactically convincing [Wayner 1992, 2009]. However, this type of steganography is unable to embed messages in existing text documents. Thus, they fail to meet the attribute that we call cover text preservation (**CP**), and have limited applications.

The second type of methods exploits format-specific features (**FSF**). For example, specifically for Microsoft Word document, Bhaya et al. [2013] assign each text character a different but visually similar font available in Word to conceal messages. Others hide messages by changing the character scale and color or adding underline styles

Category	Previous work	Attribute CP	FGC	FI	PP
CTG	[Agarwal 2013; Wayner 2009]		✓	✓	✓
SP	[Alattar and Alattar 2004; Brassil et al. 1995; Gutub and Fattani 2007; Kim et al. 2003]	✓		✓	✓
FSF	[Bhaya et al. 2013; Chaudhary et al. 2016; Panda et al. 2015; Rizzo et al. 2016]	✓	✓		
Our work		✓	✓	✓	✓

Table 1. **A summary** of related text steganographic methods. CP indicates cover text preservation; FGC indicates fine granularity coding; FI indicates format-independent; PP indicates printed paper.

in a document [Panda et al. 2015; Stojanov et al. 2014], although those changes are generally noticeable. More recent methods exploit special ASCII codes and Unicodes that are displayed as an empty space in a PDF viewer [Chaudhary et al. 2016; Rizzo et al. 2016]. These methods are not format independent (**FI**); they are bounded to a specific file format (such as Word or PDF) and text viewer. The concealed messages would be lost if the document was converted in a different format or even opened with a different version of the same viewer. If also fails to preserve the concealed message when the document is printed on paper (**PP**) and photographed later.

More relevant to our work is the family of methods that embed messages via what we call structural perturbations (**SP**). Line-shift methods [Alattar and Alattar 2004; Brassil et al. 1995] hide information by perturbing the space between text lines: reducing or increasing the space represents a 0 or 1 bit. Similarly, word-shift methods [Brassil et al. 1995; Kim et al. 2003] perturb the spaces between words. Others perturb the shape of specific characters, such as raising or dropping the positions of the dots of "i" and "j" [Brassil et al. 1995]. These methods cannot support fine granularity coding (**FGC**), in the sense that they encode 1 bit in every line break, word break or special character that appears sparsely. Our method provides fine granularity coding by embedding information in individual letters, and thereby has a much larger information capacity. In addition, all these methods demonstrate retrieval of hidden messages from *digital document files only*. It is unclear to what extent they can decode from real photos of text.

Table 1 summarizes these related work and their main attributes. Like most of these work, our method aims to perturb the appearance of the text in an unobtrusive, albeit not fully imperceptible, way. But our method is advantageous by providing more desired attributes.

3 OVERVIEW

Our FontCode system embeds in a text document any type of information as a bit string. For example, an arbitrary text message can be

Fig. 2. **Embedding and extraction.** Here we sample 5 points around the Times New Roman on the manifold (left), generating the perturbed glyphs to embed integers (top-right). We embed "1" in letter "a" using the second glyph (in orange) in the perturbation list. In the retrieval step, we evaluate a probability value (inverse of distance) by our CNNs (bottom-right), and extract the integer whose glyph results in the highest probability.

coded into a bit string using the standard ASCII code or Unicode[1]. We refer to such a bit string as a *plain message*.

In a text document, the basic elements of embedding a plain message are the *letters*, appearing in a particular font. Our idea is to perturb the glyph of each letter to embed a plain message. To this end, we leverage the concept of *font manifold* proposed by Campbell and Kautz [2014]. Taking as input a collection of existing font files, their method creates a low-dimensional font manifold for every character—including both alphabets and digits—such that every location on this manifold generates a particular glyph of that character. This novel generative model is precomputed once for each character. Then, it allows us to alter the glyph of each text letter in a subtle yet systematic way, and thereby embed messages.

It is worth noting that our method does not depend on specifically the method of [Campbell and Kautz 2014], and other font generative models can also be used to generate glyph perturbations in our work. In this paper, when there is no confusion, we refer to a location $\boldsymbol{u}$ on a font manifold and its resulting glyph interchangeably.

3.1 Message Embedding

Our message embedding method consists of two steps, (i) precomputation of a codebook for processing all documents and (ii) runtime embedding of a plain message in a given document.

During precomputation, we construct a codebook of perturbed glyphs for typically used fonts. Consider a font such as Times New Roman. Each character in this font corresponds to a specific location, $\bar{\boldsymbol{u}}$, on the font manifold. We identify a set of locations on the manifold as the *perturbed glyphs* of $\bar{\boldsymbol{u}}$ and denote them as $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots\}$ (see Fig. 2). Our goal in this step is to select the perturbed glyphs such that their differences from the glyph $\bar{\boldsymbol{u}}$ of the original font is almost unnoticeable to our naked eyes but recognizable to a computer algorithm (detailed in §4). The sets of perturbed glyphs for all characters with typically used fonts form our codebook.

At runtime, provided a text document (or a text region or paragraphs), we perturb the glyph of the letter in the document to embed a given plain message. Consider a letter in an original glyph $\bar{\boldsymbol{u}}$ in the given document. Suppose in the precomputed codebook, this letter has N perturbed glyphs, namely $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$. We embed in the letter an integer i in the range of $[0, N)$ by changing its glyph from $\bar{\boldsymbol{u}}$ to $\boldsymbol{u}_i$ (see Fig. 2-top). A key algorithmic component of this

[1]See the character set coding standards by the Internet Assigned Numbers Authority (http://www.iana.org/assignments/character-sets/character-sets.xhtml)

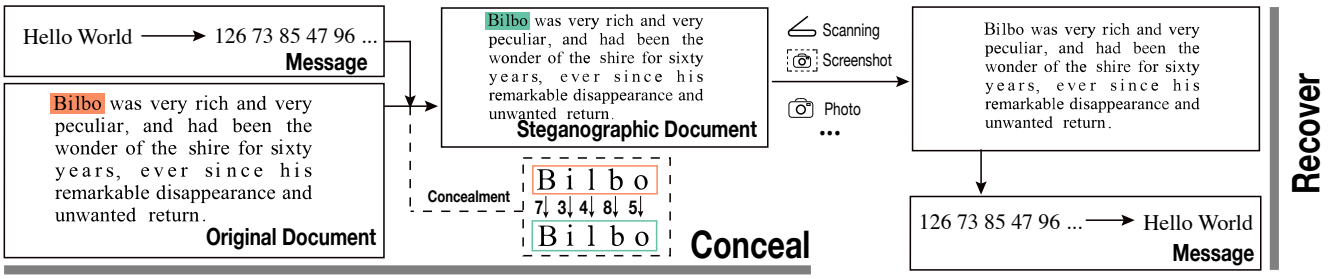


Fig. 3. **Overview.** (left) Our embedding method takes an input message and a text document. It encodes the message into a series of integers and divides the letters into blocks. The integers are assigned to each block and embedded in individual letters. (right) To recover the message, we extract integers by recognizing the glyphs of individual letters. Then, the integers are decoded into the original plain message.

step is to determine the embedded integers for all letters such that together they encode the plain message.

In addition, we propose a new error-correcting coding scheme to encode the plain message. This coding scheme adds certain redundancy (i.e., some extra data) to the coded message, which, at decoding time, can be used to check for consistency of the coded message and recover it from errors (see §5).

3.2 Message Retrieval

To retrieve information from a coded text document, the first step is to recover an integer from each letter. For each letter in the document, suppose again this letter has N perturbed glyphs in the codebook, whose manifold locations are $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$. We recognize its glyph $\boldsymbol{u}'$ in the current document as one of the N perturbed glyphs. We extract an integer i if $\boldsymbol{u}'$ is recognized as $\boldsymbol{u}_i$ (see Fig. 2-bottom). Our recognition algorithm works with not only vector graphics documents (such as those stored as PDFs) but also rasterized documents stored as pixel images. For the latter, the recognition leverages convolutional neural networks.

The retrieved integers are then fed into our error correction coding scheme to reconstruct the plain message. Because of the data redundancy, even if some glyphs are mistakenly recognized (e.g., due to poor image quality), those errors will be rectified and we will still be able to recover the message correctly (see §5). The embedding and retrieval process is summarized in Fig. 3.

4 GLYPH RECOGNITION

We start by focusing on our basic message embedding blocks: individual letters in a text document. Our goal in this section is to embed an integer number in a single letter by perturbing its glyph, and later retrieve that integer from a vector graphics or pixel image of that letter. In the next section, we will address what integers to assign to the letters in order to encode a message.

As introduced in §3.1, we embed an integer in a letter through glyph perturbation, by looking up a precomputed codebook. Later, when extracting an integer from a letter, we compute a "distance" metric between the extracted glyph and each perturbed glyph in $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$ in the codebook; we obtain integer i if the "distance" of glyph $\boldsymbol{u}_i$ is the smallest one.

Our recognition algorithm supports input documents stored as vector graphics and pixel images. While it is straightforward to recognize vector graphic glyphs, pixel images pose significant challenges due to camera perspectives, rasterization noise, blurriness, and so forth. Our initial attempt used various template matching methods (e.g., [Dogan et al. 2015]), but they easily become error-prone when image quality and camera perspective are not ideal.

In this section, we first describe our algorithm that decodes integers from rasterized (pixel) glyphs. This algorithm leverages convolutional neural networks (CNNs), which also allow us to systematically construct the codebook of perturbed glyphs. Next, we describe the details of embedding and extracting integers, as well as a simple algorithm for recognizing vector graphic glyphs.

4.1 Pixel Image Preprocessing

When a text document is provided as a pixel image, we use the off-the-shelf optical character recognition (OCR) library to detect individual letters. Our approach does not depend on any particular OCR library. In practice, we choose to use Tesseract [Smith 2007], one of the most popular open source OCR engines. In addition to detecting letters, OCR also identifies a bounding box of every letter on the pixel image.

To recognize the glyph perturbation of each letter using CNNs, we first preprocess the image. We crop the region of each letter using its bounding box detected by the OCR. We then binarize the image region using the classic algorithm by Otsu [1975]. This step helps to eliminate the influence caused by the variations of lighting conditions and background colors. Lastly, we resize the image region to have 200×200 pixels. This 200×200, black-and-white image for each letter is the input to our CNNs (Fig. 4-left).

4.2 Network Structure

We treat glyph recognition as an image classification problem: provided an image region of a letter which has a list of perturbed glyphs $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots\}$ in the codebook, our goal is to classify the input glyph of that letter as one from the list. Therefore, we train a CNN for each letter in a particular font.

Thanks to the image preprocessing, we propose to use a simple CNN structure (as illustrated in Fig. 4), which can be quickly trained and evaluated. The input is a 200×200, black-and-white image containing a letter. The CNN consists of three convolutional layers, each followed by a ReLU activation layer (i.e., $f : x \in \mathbb{R} \mapsto \max(0, x)$) and a 2×2 max pooling layer. The kernel size of the three convolutional layers are 8×8×32, 5×5×64, and 3×3×32, respectively. The

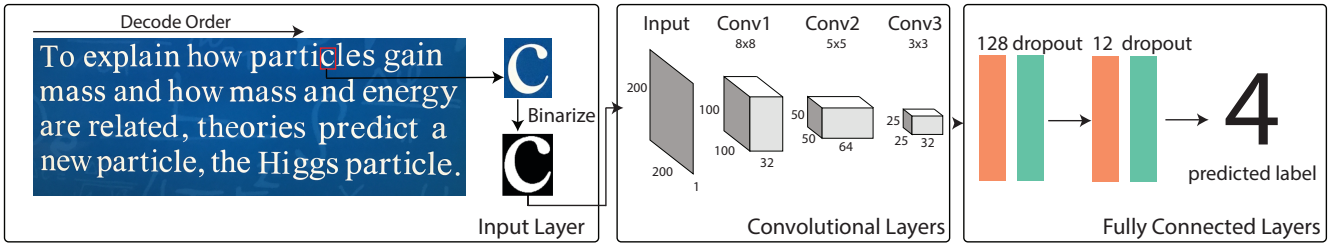


Fig. 4. **Network architecture.** Our CNN takes as input a 200×200, black-and-white image containing a single letter (left). It consists 3 convolutional layers (middle) and 2 fully connected layers (right). It outputs a vector, in which each element is the estimated probability of recognizing the glyph in the input image as a perturbed glyph in the codebook.

convolutional layers are connected with two fully connected (FC) layers, arranged in the following form (see Fig. 4):

$$\rightarrow \text{FC}(128) \rightarrow \text{Dropout} \rightarrow \text{FC}(N) \rightarrow \text{Softmax} \rightarrow \text{output}.$$

Here "Dropout" indicates a 0.5-dropout layer, and FC(m) is an m-dimensional FC layer. In the second FC layer, N is the number of perturbed glyphs in the codebook for the letter in a certain font. After passing through a softmax layer, the output is an N-dimensional vector indicating the probabilities (or the inverse of "distance") of classifying the glyph of the input letter as one of the perturbed glyphs in the list. The glyph of the letter is recognized as $\boldsymbol{u}_i$ if its corresponding probability in the output vector is the largest one.

4.3 Network Training

Training data. Our CNNs will be used for recognizing text document images that are either directly synthesized or captured by digital cameras. Correspondingly, the training data of the CNNs consist of synthetic images and real photos. Consider a letter whose perturbed glyphs from a standard font are $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$. We print all the N glyphs on a paper and take 10 photos with different lighting conditions and slightly different camera angle. While taking the photos, the camera is almost front facing the paper, mimicking the scenarios of how the embedded information in a document would be read by a camera. In addition, we include synthetic images by rasterizing each glyph (whose vector graphics path is generated using [Campbell and Kautz 2014]) into a 200×200 image.

Data augmentation. To train the CNNs, each training iteration randomly draws a certain number of images from the training data. We ensure that half of the images are synthetic and the other half are from real photos—empirically we found that the mix of real and synthetic data leads to better CNN recognition performance. To reduce overfitting and improve the robustness, we augment the selected images before feeding them into the training process. Each image is processed by the following operations:

- Adding a small Gaussian noise with zero mean and standard deviation 3.
- Applying a Gaussian blur whose standard deviation is uniformly distributed between 0 (no blur) and 3px.
- Applying a randomly-parameterized perspective transformation.
- Adding a constant border with a width randomly chosen between 0 and 30px.

We note that similar data augmentations have been used in [Chen et al. 2014; Wang et al. 2015] to enrich their training data. We therefore refer to those papers for more details of data augmentation.

Lastly, we apply the preprocessing routine described in §4.1, obtaining a binary image whose pixel values are either 0 or 1.

Implementation details. In practice, our CNNs are optimized using the Adam algorithm [Kingma and Ba 2015] with the parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and a mini-batch size of 15. The network was trained with 10^5 update iterations at a learning rate of 10^{-3}. Our implementation also leverages existing libraries, Tensorflow [Abadi et al. 2015] and Keras [Chollet et al. 2015].

4.4 Constructing The Glyph Codebook

The CNNs not only help to recognize glyphs at runtime, but also enable us to systematically construct the glyph codebook, a lookup table that includes a set of perturbed glyphs for every character in commonly used fonts (such as Times New Roman, Helvetica, and Calibri). Our codebook construction aims to satisfy three criteria: (i) The included glyph perturbation must be perceptually similar; their differences from the original fonts should be hardly noticeable to our eyes. (ii) The CNNs must be able to distinguish the perturbed glyphs reliably. (iii) Meanwhile, we wish to include as many perturbed glyphs as possible in the codebook for each character, in order to increase the capacity of embedding information (see §5).

To illustrate our algorithm, consider a single character in a given font. We use its location $\bar{\boldsymbol{u}}$ on the character's font manifold to refer to the original glyph of the font. Our construction algorithm first finds a large set of glyph candidates that are perceptually similar to $\bar{\boldsymbol{u}}$. This step uses crowdsourced perceptual studies following a standard user-study protocol. We therefore defer its details until §6, while in this section we denote the resulting set of glyphs as $\mathcal{C}$, which typically has hundreds of glyphs. Next, we reduce the size of $\mathcal{C}$ by discarding glyphs that may confuse our CNNs in recognition tests. This is an iterative process, wherein each iteration takes the following two steps (see Algorithm 1 in Appendix):

Confusion test. We randomly select M pairs of glyphs in $\mathcal{C}$ (M =100 in practice). For each pair, we check if our CNN structure (introduced in §4.2) can reliably distinguish the two glyphs. In this case, since only two glyphs are considered, the last FC layer (recall §4.2) has two dimensions. We train this network with only synthetic images processed by the data augmentations. We use synthetic images generated by different augmentations to test the accuracy of the resulting CNN. If the accuracy is less than 95%, then we record this pair of glyphs in a set $\mathcal{D}$, one that contains glyph pairs that cannot be distinguished by our CNN.

Updating glyph candidate set. Next, we update the glyph candidates in $\mathcal{C}$ to avoid the recorded glyph pairs that may confuse the

CNNs while retaining as many glyph candidates as possible in C. To this end, we construct a graph where every node i represents a glyph in C. Initially, this graph is completely connected. Then, the edge between node i and j is removed if the glyph pair i j is listed in $\mathcal{D}$, meaning that i and j are hardly distinguishable by our CNN. With this graph, we find the maximum set of glyph that can be distinguished from each other. This amounts to the classic maximum clique problem on graphs. Although the maximum clique problem has been proven NP-hard (see page 97 of [Karp 1972]), our graph size is small (with up to 200 nodes), and the edges are sparse. Consequently, many existing algorithms suffice to find the maximum clique efficiently. In practice, we choose to use the method of [Konc and Janezic 2007]. Lastly, the glyphs corresponding to the maximum clique nodes form the updated glyph candidate set C, and we move on to the next iteration.

This iterative process stops when there is no change of C. In our experiments, this takes up to 5 iterations. Because only synthetic images are used as training data, this process is fully automatic and fast. But it narrows down the glyph candidates conservatively—two glyphs in C might still be hardly distinguishable in real photos. Therefore, in the last step, we train a CNN (as described in §4.3) using both synthetic images and real photos to verify if the CNN can reliably recognize all glyphs in C. Here the last FC layer of CNN has a dimension of $|C|$ (i.e., the size of C). At this point, $|C|$ is small (typically around 25). Thus, the CNN can be easily trained. Afterwards, we evaluate the recognition accuracy of the CNN for each glyph in C, and discard the glyphs whose recognition accuracy is below 90%. The remaining glyphs in C are added to the codebook as the perturbed glyphs of the character. A fragment of our codebook for Times New Roman is shown in Fig. 5. Note that different characters have a different number of perturbed glyphs in the codebook. This difference poses a technical challenge when designing the message decoding algorithm in §5.

4.5 Embedding and Retrieving Integers

Embedding. Now we can embed integers into a text document in either vector graphic or pixel image format. First, we extract from the input document the text content and the layout of the letters. Our method also needs to know the original font $\bar{\boldsymbol{u}}$ of the text. This can be specified by the user or automatically obtained from the metadata of a vector graphic document (such as a PDF). If the document is provided as a pixel image, recent methods [Chen et al. 2014; Wang et al. 2015] can be used to recognize the text font.

In order to embed an integer i in a letter of font $\bar{\boldsymbol{u}}$, we look up its perturbed glyph list $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$ in the precomputed codebook, and generate the letter shaped by the glyph $\boldsymbol{u}_i$ [Campbell and Kautz 2014]. We then scale the glyph to fit it into the bounding box of the original letter (which is detected by the OCR library for pixel images), and use it to replace the original letter of the input document.

Retrieval. To retrieve integers from a pixel image, we extract the text content and identify regions of individual letters using the OCR tool. After we crop the regions of the letters, they are processed in parallel by the recognition algorithm: they are preprocessed as described in §4.1 and then fed into the CNNs. An integer i is extracted from a letter if the output vector from its CNN has the i-th element as the largest (recall §4.2).

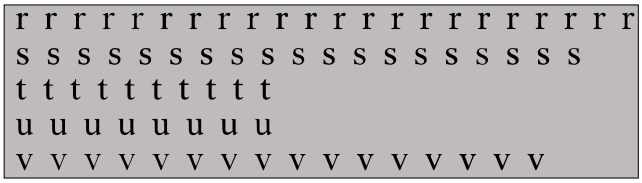


Fig. 5. **A fragment of codebook.** A complete codebook for all alphabetic characters in lower case for Times New Roman can be found in the supplemental document.

If the input document is provided as a vector graphics image (e.g., as a PDF file), glyph recognition of a letter is straightforward. In our codebook, we store the outlines of the perturbed glyphs as polylines. In the integer embedding stage, these polylines are scaled to replace the paths of the original glyphs. In the retrieval stage, we compute the "distance" between the letter's glyph polyline, denoted as $\boldsymbol{f}$, in the document and every glyph $\boldsymbol{u}_i$ in the perturbed glyph list of the codebook: we first scale the polylines of $\boldsymbol{u}_i$ to match the bounding box of $\boldsymbol{f}$, and then compute the L_2 distance between the polyline vertices of $\boldsymbol{u}_i$ and $\boldsymbol{f}$. We recognize $\boldsymbol{f}$ as $\boldsymbol{u}_j$ (and thus extract the integer j) if the distance between $\boldsymbol{f}$ and $\boldsymbol{u}_j$ is the minimum.

5 ERROR CORRECTION CODING

After presenting the embedding and retrieval of integer values over individual letters, we now focus on a plain message represented as a bit string and describe our coding scheme that encodes the bit string into a series of integers, which will be in turn embedded in individual letters. We also introduce an error-correction decoding algorithm that decodes the bit string from a series of integers, even when some integers are incorrect.

Algorithms presented in this section are built on the coding theory for noise channels, founded by Shannon's landmark work [Shannon 1948]. We refer the reader to the textbook [Lin and Costello 2004] for a comprehensive introduction. Here, we start with a brief introduction of traditional error-correcting codes, to point out their fundamental differences from our coding problem.

5.1 Challenges of the Coding Problem

The most common error-correction coding scheme is *block coding*, in which an information sequence is divided into blocks of k symbols each. We denote a block using a k-tuple, $\boldsymbol{r} = (r_1, r_2, ..., r_k)$. For example, a 128-bit binary string can be divided into 16 blocks of 8-bit binary strings. In this case, $k = 8$, and r_i is either 0 or 1. In general, r_i can vary in other ranges of discrete symbols. A fundamental requirement of error-correction coding is that (i) the number of possible symbols for each r_i must be the same, and (ii) this number must be a prime power p^m, where p is a prime number and m is a positive integer. In abstract algebraic language, it requires the blocks to be in a *Galois Field*, GF(p^m). For example, the aforementioned 8-bit strings are in GF(2). Most of the current error-correction coding schemes (e.g., Reed-Solomon codes [Reed and Solomon 1960]) are built on the algebraic operations in the polynomial ring of GF(p^m). At encoding time, they transform k-tuple blocks into n-tuple blocks in the same Galois field GF(p^m), where n is larger than k. At decoding

time, some symbols in the n-tuple blocks can be incorrect. But as long as the total number of incorrect symbols in a block is no more than $\lfloor \frac{n-k}{2} \rfloor$ regardless of the locations of incorrections, the coding scheme can fully recover the original k-tuple block.

In our problem, a text document consists of a sequence of letters. At first glance, we can divide the letter sequence into blocks of k letters each. Unfortunately, these blocks are ill-suited for traditional block coding schemes. For example, consider a five-letter block, $(C_1, C_2, ..., C_5)$, with an original font $\bar{\boldsymbol{u}}$. Every letter has a different capacity for embedding integers: in the codebook, C_i has s_i glyphs, so it can embed integers in the range $[0, s_i)$. However, in order to use block coding, we need to find a prime power p^m that is no more than any $s_i, i = 1...5$ (to construct a Galois field GF(p^m)). Then every letter can only embed p^m integers, which can be significantly smaller than s_i. Perhaps a seemingly better approach is to find $t_i = \lfloor \log_2 s_i \rfloor$ for every s_i, and use the 5-letter block to represent a T-bit binary string, where $T = \sum_{i=1}^{5} t_i$. In this case, this binary string is in GF(2), valid for block coding. Still, this approach wastes much of the letters' embedding capacity. For example, if a letter has 30 perturbed glyphs, this approach can only embed integers in $[0, 16)$. As experimentally shown in §8.2, it significantly reduces the amount of information that can be embedded. In addition, traditional block coding method is often concerned with a noisy communication channel, where an error occurs at individual *bits*. In contrast, our recognition error occurs at individual *letters*. When the glyph of a letter is mistakenly recognized, a chunk of bits becomes incorrect, and the number of incorrect bits depends on specific letters. Thus, it is harder to set a proper relative redundancy (i.e., $(n-k)/n$ in block coding) to guarantee successful error correction.

5.2 Chinese Remainder Codes

To address these challenges, we introduce a new coding scheme based on a 1700-year old number theorem, the *Chinese remainder theorem* (CRT) [Katz and Imhausen 2007]. Error-correction coding based on Chinese remainder theorem has been studied in the field of theoretical computing [Boneh 2000; Goldreich et al. 1999], known as the *Chinese Remainder Codes* (CRC). We adopt CRC and further extend it to improve its error-correction ability (§5.3).

Problem definition. Given a text document, we divide its letter sequence into blocks of n letters each. Consider a block, denoted as $\mathbb{C} = (C_1, C_2, ..., C_n)$ in an original font $\bar{\boldsymbol{u}}$. Our goal for now is to embed an integer m in this n-letter block, where m is in the range $[0, M)$. We will map the plain message into a series of integers later in this section. Formally, we seek an encoding function $\phi : m \to \boldsymbol{r}$, where $\boldsymbol{r} = (r_1, ..., r_n)$ is an n-vector of integers. Following the coding theory terminology, we refer $\boldsymbol{r}$ as a *codeword*. The function ϕ needs to ensure that every r_i can be embedded in the letter C_i. Meanwhile, ϕ must be injective, so $\boldsymbol{r}$ can be uniquely mapped to m through a decoding function $\phi^+ : \boldsymbol{r} \to m$. Additionally, the computation of ϕ and ϕ^+ must be fast, to encode and decode in a timely fashion.

5.2.1 *Hamming Distance Decoding.*

We now introduce *Hamming Distance decoding*, a general decoding framework for block codes, to pave the way for our coding scheme.

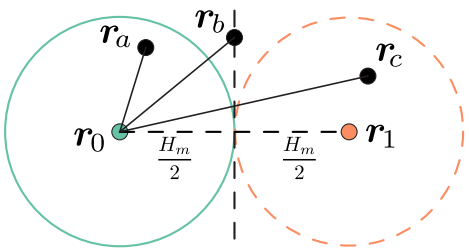


Fig. 6. **Analogy** of Euclidean distance on 2D plane to gain intuition of Theorem 5.2. Here $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$ represent two codewords having the minimum Hamming distance H_m. Suppose $\boldsymbol{r}_0$ is recognized as a code vector $\boldsymbol{r}$. If the distance $|\boldsymbol{r} - \boldsymbol{r}_0| < H_m/2$ (e.g., $\boldsymbol{r}_a$), it is safe to decode $\boldsymbol{r}$ into $\boldsymbol{r}_0$, as no other codeword is closer to $\boldsymbol{r}$. If $|\boldsymbol{r} - \boldsymbol{r}_0| > H_m/2$, there might be multiple codewords having the same distance to $\boldsymbol{r}$ (e.g., when $\boldsymbol{r}$ is $\boldsymbol{r}_b$), and $\boldsymbol{r}$ may be mistakenly decoded as $\boldsymbol{r}_1$ (e.g., when $\boldsymbol{r}$ is $\boldsymbol{r}_c$).

Definition 5.1 (Hamming Distance). Given two codewords $\boldsymbol{u}$ and $\boldsymbol{v}$, the Hamming Distance $H(\boldsymbol{u}, \boldsymbol{v})$ measures the number of pair-wise elements in which they differ.

For example, given two codewords, $\boldsymbol{u} = (2, 3, 1, 0, 6)$ and $\boldsymbol{v} = (2, 0, 1, 0, 7)$, $H(\boldsymbol{u}, \boldsymbol{v}) = 2$.

Now we consider glyph recognition errors. If the integer retrieval from a letter's glyph is incorrect, then the input $\tilde{\boldsymbol{r}}$ of the decoding function ϕ^+ may not be a valid codeword. In other words, because of the recognition errors, it is possible that no m satisfies $\phi(m) = \tilde{\boldsymbol{r}}$. To distinguish from a codeword, we refer to $\tilde{\boldsymbol{r}}$ as a *code vector*.

The Hamming distance decoding uses a decoding function ϕ^+ based on the Hamming distance: $\phi^+(\tilde{\boldsymbol{r}})$ returns an integer m such that the Hamming distance $H(\phi(m), \tilde{\boldsymbol{r}})$ is minimized over all $m \in [0, M)$. Intuitively, although $\tilde{\boldsymbol{r}}$ may not be a valid codeword, we decode it as the integer whose codeword is closest to $\tilde{\boldsymbol{r}}$ under the measure of Hamming distance. Let H_m denote the *minimum* Hamming distance among all pairs of valid codewords, namely,

$$H_m = \min_{\substack{i,j \in [0, M-1] \\ i \neq j}} H(\phi(i), \phi(j)). \tag{1}$$

The error-correction ability of Hamming distance decoding is bounded by the following theorem [Lin and Costello 2004],

Theorem 5.2. *Let E denote the number of incorrect symbols in a code vector $\tilde{\boldsymbol{r}}$. Then, the Hamming distance decoding can correctly recover the integer m if $E \leq \lfloor \frac{H_m - 1}{2} \rfloor$; it can detect errors in $\tilde{\boldsymbol{r}}$ but not recover the integer correctly if $\lfloor \frac{H_m - 1}{2} \rfloor < E < H_m$.*

An illustrative understanding of this theorem using an analogy of Euclidean distance is shown in Fig. 6. This theorem holds regardless of the encoding function ϕ, as long as it is a valid injective function.

5.2.2 *Chinese Remainder Codes.*

We now introduce the following version of the Chinese Remainder Theorem, whose proof can be found in [Rosen 2011].

Theorem 5.3 (Chinese Remainder Theorem). *Let $p_1, p_2, ...p_k$ denote positive integers which are mutually prime and $M = \prod_{i=1}^{k} p_i$. Then, there exists an injective function,*

$$\phi : [0, M) \to [0, p_1) \times [0, p_2) \times ...[0, p_k),$$

defined as $\phi(m) = (r_1, r_2, ..., r_n)$, *such that for all* $m \in [0, M)$, $r_i = m \bmod p_i$.

This theorem indicates that given k pairs of integers (r_i, p_i) with all p_i being mutually prime, there exists a unique non-negative integer $m < \prod_{i=1}^{k} p_i$ satisfying $r_i = m \bmod p_i$ for all $i = 1...k$. Indeed, m can be computed using the formula,

$$m = CRT(\boldsymbol{r}, \boldsymbol{p}) = r_1 b_1 \frac{P}{p_1} + ... + r_n b_n \frac{P}{p_n}, \tag{2}$$

where $P = \prod_{i=1}^{k} p_i$, and b_i is computed by solving a system of modular equations,

$$b_i \frac{P}{p_i} \equiv 1 \pmod{p_i}, \quad i = 1...k, \tag{3}$$

using the classic Euclidean algorithm [Rosen 2011].

If we extend the list of p_i to n mutually prime numbers ($n > k$) such that the $n - k$ additional numbers are all larger than p_i up to $i = k$ (i.e., $p_j > p_i$ for any $j = k{+}1...n$ and $i = 1...k$), then we have an encoding function ϕ for non-negative integer $m < \prod_{i=1}^{k} p_i$:

$$\phi(m) = (m \bmod p_1, m \bmod p_2, ..., m \bmod p_n). \tag{4}$$

This encoding function already adds redundancy: because m is smaller than the product of any k numbers chosen from p_i, we can compute m from any k of the n pairs of (r_i, p_i), according to the Chinese Remainder Theorem. Indeed, as proved in [Goldreich et al. 1999], the minimum Hamming distance of the encoding function (4) for all $0 \le m < \prod_{i=1}^{k} p_i$ is $n - k + 1$. Thus, the Hamming decoding function of ϕ can correct up to $\left\lfloor \frac{n-k}{2} \right\rfloor$ errors by Theorem 5.2.

Encoding. We now describe our encoding algorithm based on the encoding function $\phi(m)$ in (4).

- **Computing p_i.** Suppose that the letter sequence of a document has been divided into N blocks, denoted as $\mathbb{C}_1, ..., \mathbb{C}_N$, each with n letters. Consider a block $\mathbb{C}_t = (C_1, ..., C_n)$, where C_i indicates a letter with its original font $\boldsymbol{u}_i$, whose integer embedding capacity is s_i (i.e., C_i's font $\boldsymbol{u}_i$ has s_i perturbed glyphs in the codebook). We depth-first search n mutually prime numbers $p_i, i = 1...n$, such that $p_i \le s_i$ and the product of k minimal p_i is maximized. At the end, we obtain $p_i, i = 1...n$ and the product of k minimal p_i denoted as M_t for each block $\mathbb{C}_t$. Note that if we could not find mutually prime numbers p_i for block $\mathbb{C}_t$, we simply ignore the letter whose embedding capacity is smallest among all s_i and include C_{n+1} to this block. We repeat this process until we find a valid set of mutually prime numbers.
- **Determining m_t.** Given the plain message represented as a bit string $\mathbb{M}$, we now split the bits into a sequence of chunks, each of which is converted into an integer and assigned to a block $\mathbb{C}_t$. We assign to each block $\mathbb{C}_t$ an integer m_t with $\lfloor \log_2 M_t \rfloor$ bits, which is sequentially cut from the bit string $\mathbb{M}$ (see Fig. 7).
- **Embedding.** For every block $\mathbb{C}_t$, we compute the codeword using the CRT encoding function (4), obtaining $\boldsymbol{r} = (r_1, ..., r_n)$. Each r_i is then embedded in the glyph of the letter C_i in the block $\mathbb{C}_t$ as described in §4.5.

Decoding. At decoding time, we recognize the glyphs of the letters in a document and extract integers from them, as detailed in §4.5. Next, we divide the letter sequence into blocks, and repeat the algorithm of computing p_i and M_t as in the encoding step, for every block. Given a block $\mathbb{C}_t$, the extracted integers from its letters form a code vector $\tilde{\boldsymbol{r}}_t = (\tilde{r}_1, ..., \tilde{r}_n)$. We refer to it as a code vector because some of the $\tilde{r}_i$ may be incorrectly recognized. To decode $\tilde{\boldsymbol{r}}_t$, we first compute $\tilde{m}_t = CRT(\tilde{\boldsymbol{r}}_t, \boldsymbol{p}_t)$ where $\boldsymbol{p}_t$ stacks all p_i in the block. If $\tilde{m}_t < M_t$, then $\tilde{m}_t$ is the decoding result $\phi^+(\tilde{\boldsymbol{r}}_t)$, because the Hamming distance $H(\phi(\tilde{m}_t), \tilde{\boldsymbol{r}}) = 0$. Otherwise, we decode $\tilde{\boldsymbol{r}}_t$ using the Hamming decoding function: concretely, since we know the current block can encode an integer in the range $[0, M_t)$, we decode $\tilde{\boldsymbol{r}}_t$ into the integer m_t by finding

$$m_t = \phi^+(\tilde{\boldsymbol{r}}) = \arg \min_{m \in [0, M_t)} H(\phi(m), \tilde{\boldsymbol{r}}). \tag{5}$$

As discussed above, this decoding function can correct up to $\left\lfloor \frac{n-k}{2} \right\rfloor$ incorrectly recognized glyphs in each block. Lastly, we convert m_t into a bit string and concatenate m_t from all blocks sequentially to recover the plain message.

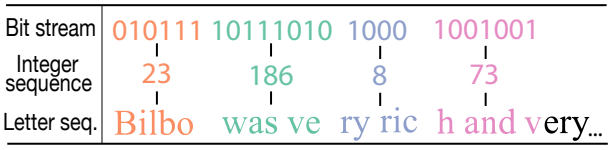


Fig. 7. **An example** of determining m_t: (top) The bit string representation of the plain message; (bottom) Each block of letters is highlighted by a color; (middle) m_t values assigned to each block.

Implementation details. A few implementation details are worth noting. First, oftentimes letters in a document can carry a bit string much longer than the given plain message. To indicate the end of the message, we attach a special chunk of bits (end-of-message bits) at the end of each plain message, very much akin to the end-of-line (newline) character used in digital text systems. Second, in practice, blocks should be relatively short (i.e., n is small). If n is large, it becomes much harder to find n mutually prime numbers that are no more than each letter's embedding capacity. In practice, we choose $n = 5$ and $k = 3$ which allows one mistaken letter in every 5-letter block. The small n and k also enable brute-force search of m_t in (5) sufficiently fast, although there also exists a method solving (5) in polynomial time [Boneh 2000; Goldreich et al. 1999].

5.3 Improved Error Correction Capability

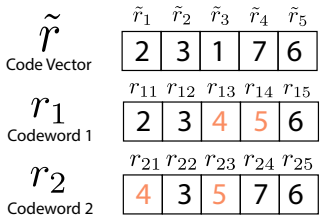
Using the Chinese Remainder Codes, the error-correction capacity is upper bounded. Theorem 5.2 indicates that at most $\left\lfloor \frac{n-k}{2} \right\rfloor$ mistakenly recognized glyphs are allowed in every letter block. We now break this theoretical upper bound to further improve our error-correction ability, by exploiting specific properties of our coding problem. To this end, we propose a new algorithm based on the maximum likelihood decoding [Lin and Costello 2004].

In coding theory, maximum likelihood decoding is not an algorithm. Rather, it is a decoding philosophy, a framework that models the decoding process from a probabilistic rather than an algebraic point of view. Consider a letter block $\mathbb{C}$ and the code vector $\tilde{\boldsymbol{r}}$ formed by the extracted integers from $\mathbb{C}$. We treat the true codeword $\boldsymbol{r}$ encoded by $\mathbb{C}$ as a *latent variable* (in statistic language), and model the probability of $\boldsymbol{r}$ given the extracted code vector $\tilde{\boldsymbol{r}}$, namely $\mathbb{P}(\boldsymbol{r}|\tilde{\boldsymbol{r}})$. With this likelihood model, our decoding process finds a codeword $\boldsymbol{r}$

that maximizes the probability $\mathbb{P}(\boldsymbol{r}|\tilde{\boldsymbol{r}})$, and decodes $\boldsymbol{r}$ into an integer using the Chinese Remainder Theorem formula (2).

When there are at most $\left\lfloor \frac{n-k}{2} \right\rfloor$ errors in a block, the Hamming decoding function (5) is able to decode. In fact, it can be proved that when the number of errors is under this bound, there exists a unique $m \in [0, M_t)$ that minimizes $H(\phi(m), \tilde{\boldsymbol{r}})$ [Lin and Costello 2004], and that when the number of errors becomes $\left\lfloor \frac{n-k}{2} \right\rfloor + 1$, there may be multiple $m \in [0, M_t]$ reaching the minimum (see Fig. 6 for an intuitive explanation). The key idea of breaking this bound is to use a likelihood model to choose an m when ambiguity occurs for the Hamming decoding function (5).

Consider a block with $\left\lfloor \frac{n-k}{2} \right\rfloor + 1$ errors, and suppose in (5) we find N_c different integers m_i, all of which lead to the same minimal Hamming distance to the code vector $\tilde{\boldsymbol{r}}$. Let $\boldsymbol{r}_i = \phi(m_i), i = 1 \ldots N_c$, denote the corresponding codewords. We use another subscript j to index the integer elements in code vectors and codewords. For every $\boldsymbol{r}_i$, some of its integer elements r_{ij} differs from the corresponding integer elements $\tilde{r}_j$. Here r_{ij} can be interpreted as the index of the perturbed glyphs of the j-th letter. We denote this glyph as $\boldsymbol{u}_{ij}$ and the letter's glyph extracted from the input image as $\boldsymbol{f}$. If r_{ij} is indeed the embedded integer, then $\boldsymbol{f}$ is mistakenly recognized, resulting in a different glyph number $\tilde{r}_j$.



We first model the probability of this occurrence, denoted as $\mathbb{P}(\tilde{r}_j|r_{ij})$, using our "distance" metric. Intuitively, the closer the two glyphs are, the more likely one is mistakenly recognized as the other. Then the probability of recognizing a codeword $\boldsymbol{r}_i$ as a code vector $\tilde{\boldsymbol{r}}$ accounts for all inconsistent element pairs in $\boldsymbol{r}_i$ and $\tilde{\boldsymbol{r}}$, namely,

$$\mathbb{P}(\tilde{\boldsymbol{r}}|\boldsymbol{r}_i) = \prod_{j, r_{ij} \neq \tilde{r}_j} \frac{g(\boldsymbol{u}_{ij}, \boldsymbol{f})}{\sum_{k=1}^{s_j} g(\bar{\boldsymbol{u}}_j^k, \boldsymbol{f})}. \tag{6}$$

Here $g(\cdot, \cdot)$ is (inversely) related to our distance metric between two glyphs: for pixel images, $g(\boldsymbol{u}_{ij}, \boldsymbol{f})$ is the probability of recognizing $\boldsymbol{f}$ as $\boldsymbol{u}_{ij}$, which is the r_{ij}-th softmax output from the CNN of the i-th letter, given the input glyph image $\boldsymbol{f}$ (recall §4.2); for vector graphics, $g(\boldsymbol{u}_{ij}, \boldsymbol{f}) = 1/d(\boldsymbol{u}_{ij}, \boldsymbol{f})$, the inverse of the vector graphic glyph distance (defined in §4.5). The denominator is for normalization, where $\bar{\boldsymbol{u}}_j^k$ iterates through all perturbed glyphs of the j-th letter in the codebook. For pixel images, the denominator is always 1 because of the unitary property of the softmax function. Lastly, the likelihood $\mathbb{P}(\boldsymbol{r}_i|\tilde{\boldsymbol{r}})$ needed for decoding is computed by Bayes Theorem,

$$\mathbb{P}(\boldsymbol{r}_i|\tilde{\boldsymbol{r}}) = \frac{\mathbb{P}(\tilde{\boldsymbol{r}}|\boldsymbol{r}_i)\mathbb{P}(\boldsymbol{r}_i)}{\mathbb{P}(\tilde{\boldsymbol{r}})}. \tag{7}$$

Here, $\mathbb{P}(\tilde{\boldsymbol{r}})$ is fixed, and so is $\mathbb{P}(\boldsymbol{r}_i)$ as all codewords are equally likely to be used. As a result, we find $\boldsymbol{r}_i$ that maximizes (7) among all N_c ambiguous codewords, and decode $\tilde{\boldsymbol{r}}$ using $\boldsymbol{r}_i$.

In our experiments (§8.3), we show that the proposed decoding scheme indeed improves the error-correction ability. For our implementation wherein $n = 5$ and $k = 3$, in each block we are able to correct up to **2** errors, as opposed to 1 indicated by Theorem 5.2, thereby increasing the error tolerance from 20% to 40%.

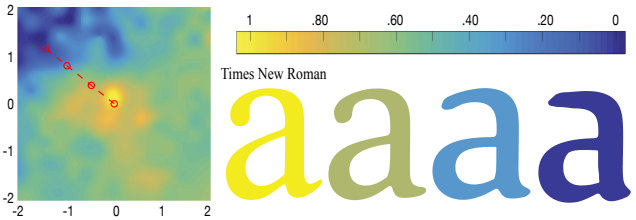


Fig. 8. **MTurk result.** (left) The font manifold of the letter "a" color-mapped using the perceptual similarity value of each point to the standard Time New Roman font. (right) Sampled glyphs that correspond to the manifold locations indicated by red circles. The colors indicate their perceptual similarity to the Times New Roman.

6 PERCEPTUAL EVALUTATION

We now describe our crowd-sourced perceptual studies on Mechanical Turk (MTurk). The goal of the studies is, for each character in a particular font, to find a set of glyphs that are perceptually similar to its original glyph $\bar{\boldsymbol{u}}$. When constructing the glyph codebook, we use these sets of glyphs to initialize the candidates of perturbed glyphs (recall §4.4).

The user studies adopt a well-established protocol: we present the MTurk raters multiple questions, and each question uses a two-alternative forced choice (2AFC) scheme, i.e., the MTurk raters must choose one from two options. We model the rater's response using a logistic function (known as the Bradley-Terry model [1952]), which allows us to learn a perceptual metric (in this case the perceptual similarity of glyphs) from the raters' response. We note that this protocol has been used previously in other perceptual studies (e.g., [O'Donovan et al. 2014; Um et al. 2017]), and that we will later refer back to this protocol in §8.3 when we evaluate the perceptual quality of our results. Next, we describe the details of the studies.

Setup. We first choose on the font manifold a large region centered at $\bar{\boldsymbol{u}}$, and sample locations densely in this region. In practice, all font manifolds are in 2D, and we choose 400 locations uniformly in a squared region centered at $\bar{\boldsymbol{u}}$. Let $\mathcal{F}$ denote the set of glyphs corresponding to the sampled manifold locations. Then, in each MTurk question, we present the rater a pair of glyphs randomly selected from $\mathcal{F}$, and ask which one of the two glyphs looks closer to the glyph of the original font $\bar{\boldsymbol{u}}$. An example is shown in Fig. 9. We assign 20 questions to each MTurk rater. Four of them are control questions, which are designed to detect untrustworthy raters. The four questions ask the rater to compare the same pair of glyphs presented in different order. We reject raters whose answers have more than one inconsistencies among the control questions. At the end, about 200 raters participated the studies for each character.

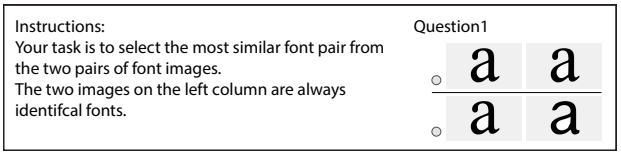


Fig. 9. MTurk user interface.

Analysis. From the raters' response, we estimate a scalar value $s_i = s(\boldsymbol{u}_i; \bar{\boldsymbol{u}})$, the perceptual similarity value of the glyph $\boldsymbol{u}_i$ to the original glyph $\bar{\boldsymbol{u}}$, for every glyph $\boldsymbol{u}_i \in \mathcal{F}$. Let a set of tuples $\mathcal{D} = \{(\boldsymbol{u}_i, \boldsymbol{u}_j, u, q)\}$ record the user study responses, where $\boldsymbol{u}_i$ and $\boldsymbol{u}_j$ are randomly selected glyphs being compared, u is the MTurk rater's ID, and the binary value q is the rater's choice: $q = 1$ indicates the rater judges $\boldsymbol{u}_i$ perceptually closer to $\bar{\boldsymbol{u}}$, and $q = 0$ otherwise. We model the likelihood of the rater's choice using a logistic function,

$$p(q = 1 | \boldsymbol{u}_i, \boldsymbol{u}_j, u) = \frac{1}{1 + \exp\left(r_u (s_i - s_j)\right)}, \tag{8}$$

where the scalar r_u indicates the rater's reliability. With this model, all the similarity values s_i and user reliability values r_u are obtained by minimizing a negative log-likelihood objective function,

$$\begin{aligned} E(\boldsymbol{s}, \boldsymbol{r}) = -\sum_k \Big[& q^k \ln p(q = 1 | \boldsymbol{u}_i^k, \boldsymbol{u}_j^k, u^k) + \\ & (1 - q^k) \ln p(q = 0 | \boldsymbol{u}_i^k, \boldsymbol{u}_j^k, u^k) \Big], \end{aligned} \tag{9}$$

where k indices the MTurk response in $\mathcal{D}$, $\boldsymbol{r}$ stacks the raters' reliability values, and $\boldsymbol{s}$ stacks the similarity values for all $\boldsymbol{u}_i \in \mathcal{F}$. The s_i values are then normalized in the range of $[0, 1]$.

We learn the similarity values for the glyphs of every character with a font $\bar{\boldsymbol{u}}$ independently. Fig. 8 visualizes the perceptual similarity of glyphs near the standard Times New Roman for the character "a". Lastly, We iterate through all the pre-sampled glyphs $\boldsymbol{u}_i \in \mathcal{F}$, and add those whose similarity value s_i is larger than a threshold (0.85 in practice) into a set C, forming the set of perceptually similar glyphs for constructing the codebook (in §4.4).

7 APPLICATIONS

Our method finds many applications. In this section, we discuss four of them, while referring to the supplemental video for their demonstrations and to §8.1 for implementation summaries.

7.1 Application I: Format-Independent Metadata

Many digital productions carry *metadata* that provide additional information, resources, and digital identification [Greenberg 2005]. Perhaps most well-known is the metadata embedded in photographs, providing information such as camera parameters and copyright. PDF files can also contain metadata[2]. In fact, metadata has been widely used by numerous tools to edit and organize digital files.

Currently, the storage of metadata is *ad hoc*, depending on specific file format. For instance, a JPEG image stores metadata in its EXIF header, while an Adobe PDF stores its metadata in XML format with Adobe's XMP framework [Adobe 2001]. Consequently, metadata is lost whenever one converts an image from JPEG to PNG format, or rasterizes a vector graphic document into a pixel image. Although it is possible to develop a careful converter that painstakingly preserves the metadata across all file formats, the metadata is still lost whenever the image or document is printed on paper.

Our FontCode technique can serve as a means to host text document metadata. More remarkably, the metadata storage in our technique is *format-independent*. Once information is embedded in a document, one can freely convert it to a different file format, or rasterize it into a pixel image (as long as the image is not severely downsampled), or print it on a piece of paper. Throughout, the glyphs of letters are preserved, and thereby metadata is retained (see video).

7.2 Application II: Imperceptible Optical Codes

Our FontCode technique can also be used as optical barcodes embedded in a text document, akin to QR codes [Denso 2011]. Barcodes have numerous applications in advertising, sales, inventory tracking, robotics, augmented reality, and so forth. Similar to QR codes that embed certain level of redundancy to correct decoding error, FontCode also supports error-correction decoding. However, all existing barcodes require to print black-and-white blocks and bars, which can be visually distracting and aesthetically imperfect. Our technique, in contrast, enables not only an optical code but an unobtrusive optical code, as it only introduces subtle changes to the text appearance. Our retrieval algorithm is sufficiently fast to provide point-and-shoot kind of message decoding. It can be particularly suitable for use as a replacement of QR codes in an artistic work such as a poster or flyer design, where visual distraction needs to be minimized. As a demonstration, we have implemented an iPhone application to read a hidden message from coded text (see video).

7.3 Application III: Encrypted Message Embedding

Our technique can further encrypt a message when embedding it in a document, even if the entire embedding and retrieval algorithms are made public. Recall that when embedding an integer i in a letter c of a glyph $\bar{\boldsymbol{u}}$, we replace $\bar{\boldsymbol{u}}$ with a glyph chosen from a list of perturbed glyphs in the codebook. Let $\mathbb{L}_c = \{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N_c - 1}\}$ denote this list. Even though the perturbed glyphs for every character in a particular font are precomputed, the order of the glyphs in each list $\mathbb{L}_c$ can be arbitrarily user-specified. The particular orders of all $\mathbb{L}_c$ together can serve as an encryption key.

For example, when Alice and Bob[3] communicate through encoded documents, they can use a publicly available codebook, but agree on a private key, which specifies the glyph permutation of each list $\mathbb{L}_c$. If an original list $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, \ldots\}$ of a letter is permuted into $\{\boldsymbol{u}_{p_0}, \boldsymbol{u}_{p_1}, \boldsymbol{u}_{p_2}, \ldots\}$ by the key, then Alice uses the glyph $\boldsymbol{u}_{p_i}$, rather than $\boldsymbol{u}_i$, to embed an integer i in the letter, and Bob deciphers the message using the same permuted codebook. For a codebook that we precomputed for Times New Roman (see the supplemental document), if we only consider lowercase English alphabet, there exist 1.39×10^{252} different glyph permutations in the codebook; if we also include uppercase English alphabet, there exist 5.73×10^{442} glyph permutations. Thus, without resorting to any existing cryptographic algorithm, our method already offers a basic encryption scheme. Even if others can carefully examine the text glyphs and discover that a document is indeed embedding a message, the message can still be protected from leaking.

[2] If you are reading this paper with Adobe Acrobat Reader, you can view its metadata by choosing "File→Properties" and clicking the "Additional Metadata" under the "Description" tab.

[3] Here we follow the convention in cryptography, using Alice and Bob as placeholder names for the convenience of presenting algorithms.

7.4 Application IV: Text Document Signature

Leveraging existing cryptographic techniques, we augment FontCode to propose a new digital signature technique, one that can authenticate the source of a text document and guarantee its integrity (thereby protecting from tampering). This technique has two variations, working as follows:

Scheme I. When Alice creates a digital document, she maps the document content (e.g., including letters, digits, and punctuation) into a bit string through a cryptographic hash function such as the MD5 [Rivest 1992] and SHA [Eastlake and Jones 2001]. We call this bit string the document's *hash string*. Alice then chooses a private key to permute the codebook as described in §7.3, and uses the permuted codebook to embed the hash string into her document. When Bob tries to tamper this document, any change leads to a different hash string. Without knowing Alice's private key, he cannot embed the new hash string in the document, and thus cannot tamper the document successfully. Later, Alice can check the integrity of her document, by extracting the embedded hash string and comparing it against the hash string of the current document.

Scheme II. The above algorithm allows only Alice to check the integrity of her document, as only she knows her private key. By combining with *asymmetric cryptography* such as the RSA algorithm [Rivest et al. 1978], we allow everyone to check the integrity of a document but not to tamper it. Now, the codebook is public but not permuted. After Alice generates the hash string of her document, she encrypts the hash string using her private key by an asymmetric cryptosystem, and obtains an *encrypted string*. Then, she embeds the encrypted string in the document using the FontCode method, while making her public key available to everyone. In this case, Bob cannot tamper the document, as he does not have Alice's private key to encrypt the hash string of an altered document. But everyone in the world can extract the encrypted string using the FontCode method, and decipher the hash string using Alice's public key. If the deciphered hash string matches the hash string of the current document, it proves that (i) the document is indeed sourced from Alice, and (ii) the document has not been modified by others.

Advantages. In comparison to existing digital signatures such as those in Adobe PDFs, our method is *format-independent*. In contrast to PDF files whose signatures are lost when the files are rasterized or printed on physical papers, our FontCode signature is preserved regardless of file format conversion, rasterization, and physical printing.

Further extension. In digital text forensics, it is often desired to not only detect tampering but also locate where the tampering occurs. In this regard, our method can be extended for more detailed tampering detection. As shown in our analysis (§8.2), in a typical English document, we only need about 80 letters to embed (with error correction) a string of 128 bits, which is the length of a hash string resulted from a typical cryptographic hash function (e.g., MD5). Given a document, we divide its text into a set of segments, each with at least 80 letters. We then compute a hash string for each segment and embed the encrypted strings in individual segments. This creates a fine granularity of text signatures, allowing the user to check which text segment is modified, and thereby locating tampering occurrences more precisely. For example, in the current text format of this paper, every two-column line consists of around 100 letters, meaning that our method can identify tampering locations up to two-column lines in this paper. To our knowledge, digital text protection with such a fine granularity has not been realized.

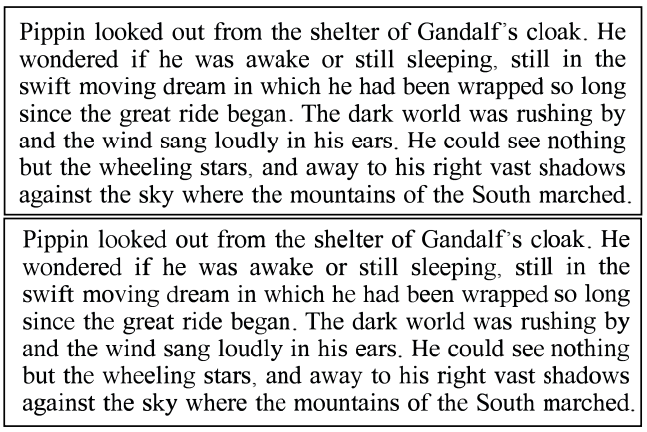


Fig. 10. **(top)** an input text document. **(bottom)** the output document that embeds a randomly generated message.

Discussion about the storage. We close this section with a remark on the memory footprint of the documents carrying messages. If a document is stored as a pixel image, then it consumes no additional memory. If it is in vector graphics format, our current implementation stores the glyph contour polylines of all letters. A document with 376 letters with 639 bits encoded will consume 1.2M memory in a compressed SVG form and 371K in a compressed JPEG format. PDF files can embed glyph shapes in the file and refer to those glyphs in the text. Using this feature, we can also embed the entire codebook in a PDF, introducing about 1.3M storage overhead, regardless of the text length. In the future, if all glyphs in the codebook are pre-installed on the operating system, like the current standardized fonts, then the memory footprint of vector graphic documents can be further reduced, as the PDF and other vector graphic file format are able to directly refer to those pre-installed fonts.

8 RESULTS AND VALIDATION

We now present the results and experiments to analyze the performance of our technique and validate our algorithmic choices. Here we consider text documents with English alphabet, including both lower- and upper-case letters, while the exact method can be directly applied to digits and other special characters. We first present our main results (§8.1), followed by the numerical (§8.2) and perceptual (§8.3) evaluation of our method.

8.1 Main Results

We implemented the core coding scheme on an Intel Xeon E5-1620 8 core 3.60GHz CPU with 32GB of memory. The CNNs are trained with an NVidia Geforce TITAN X GPU. Please see our accompanying video for the main results. A side-by-side comparison of an original document with a coded document is shown in Fig. 10.

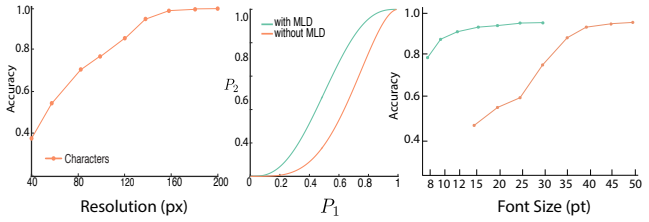


Fig. 11. **(left)** The accuracy of our CNN decoder changes as the resolution in height of the letters increases. **(middle)** Theoretical improvement of maximum likelihood decoding (green) over the Chinese Remainder coding (orange). Please see the main text for the definition of P_1 and P_2. **(right)** The accuracy of our CNN decoder trained with two different font sizes (green curve for 15pt fonts and orange curve for 30pt fonts) printed on paper. Please see the main text for the details of experiment setup.

Metadata viewer. We implemented a simple text document viewer that loads a coded document in vector graphics or pixel image format. The viewer displays the document. Meanwhile, it extracts the embedded metadata with our decoding algorithm and presents it in a side panel.

Unobtrusive optical codes. We also implemented an iPhone application (see Fig. 1), by which the user can take a photo of an encoded text displayed on a computer screen or printed on paper. The iPhone application interface allows the user to select a text region to capture. The captured image is sent through the network to a decoding server, which recovers the embedded message and sends it back to the smartphone.

Embedding encrypted message. Our implementation allows the user to load an encryption key file that specifies the permutation for all the lists of perturbed glyphs in the codebook. The permutation can be manually edited, or randomly generated—given a glyph list of length n, one can randomly sample a permutation from the permutation group $\mathbb{S}_n$ [Seress 2003] and attach it to the key.

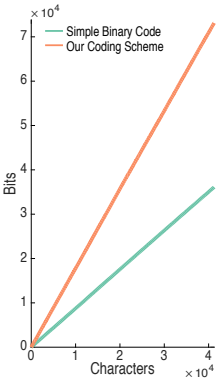
Text document signature. We use our technique to generate a MD5 signature as described in §7. Since the MD5 checksum has only 128 bits, we always embed it in letters from the beginning of the text. Our text document viewer can check the signature and alert the user if the document shows as tampered.

8.2 Validation

Information capacity. As described in §5, we use $n = 5$ and $k = 3$ in our error-correction coding scheme. In every 5-letter block, if the mutually prime numbers are $p_i, i = 1...5$, then this block can encode integers in $[0, p_1p_2p_3)$, where p_1, p_2, and p_3 are the smallest three numbers among $p_i, i = 1...5$. Thus, this block can encode at most $\lfloor \log_2(p_1p_2p_3) \rfloor$ bits of information.

To estimate the information capacity of our scheme for English text, we randomly sample 5 characters from the alphabet to forming a block. The characters are sampled based on the widely known English letter frequencies (e.g., "e" is the most frequently used while "z" is the least used) [Ferguson and Schneier 2003]. We compute the average number of bits that can be encoded by a character. The result is **1.77**, suggesting that on average we need 73 letters to encode a 128-bit MD5 checksum for the application of digital signature (§7).

Next, we compare our coding scheme with the simple approach discussed in §5.1. Recall that our method can correct at least one error in a block of 5 letters, which amounts to correcting $\log_2(\max s_i)$ bits out of the total $\sum_{i=1}^{5} \log_2 s_i$ bits. The simple approach in §5.1 can store $\sum_{i=1}^{5} \lfloor \log_2 s_i \rfloor$ bits of information. But in order to correct one recognition error of the letter with the standard linear block codes, it needs to spend $2\lceil \log_2(\max s_i) \rceil$ bits for adding redundancy, leaving $\sum_{i=1}^{5} \lfloor \log_2 s_i \rfloor - 2\lceil \log_2(\max s_i) \rceil$ bits to store information. We compare it with our method using *The Lord of The Rings, Chapter 1* as our input text, which contains in total 41682 useful letters (9851 words). As shown in the adjacent figure, as the number of letters increases, our method can embed significantly more information.



Decoding accuracy. We first evaluate the glyph recognition accuracy of our CNNs. For every character, we print it repeatedly on a paper with randomly chosen glyphs from the codebook and take five photos under different lighting conditions. Each photo has regions of around 220px×220px containing a character. We use these photos to test CNN recognition accuracy, and for all characters, the accuracy is above 90%.

We also evaluate the decoding accuracy of our method. We observed that decoding errors are mainly caused by image rasterization. If the input document is in vector graphics format, the decoding result is fully accurate, as we know the glyph outlines precisely. Thus, we evaluate the decoding accuracy with pixel images. We again use *The Lord of The Rings, Chapter 1* to encode a random bit string. We rasterize the resulting document into images with different resolutions, and measure how many letters and blocks can be decoded correctly. Figure 11-left shows the testing result.

We also theoretically estimate the decoding robustness of our maximum likelihood decoding method (§5.3). Suppose the probability of correctly recognizing the glyph of a single letter is a constant P_1. The probability P_2 of correctly decoding a single 5-letter block can be derived analytically: if we only use Chinese Remainder Decoding algorithm (§5.2), P_2 is $\binom{5}{1}P_1^4(1-P_1)$. With the maximum likelihood decoding (§5.3), P_2 becomes $\binom{5}{2}P_1^3(1-P_1)^2$. The improvement is visualized in Fig. 11-middle.

The construction of the codebook (recall Fig. 5) needs to tradeoff recognition accuracy for the embedded information capacity. This is validated and illustrated in Fig. 11-right, where we show the recognition accuracy of a text document printed on a paper with fixed 600dpi. The orange and green curve correspond to two recognizers trained with different setups: for the orange curve, the recognizer is trained using characters printed with 30pt size on paper, and each character in the training data is captured with a resolution of 200px×200px; for the green curve, the recognizer is trained using characters printed with 15pt size, and each character is captured with a resolution of 100px×100px. Not surprisingly, the recognition accuracy drops as we reduce the printing font size. The latter recognizer (green curve) has higher accuracy in comparison to the former

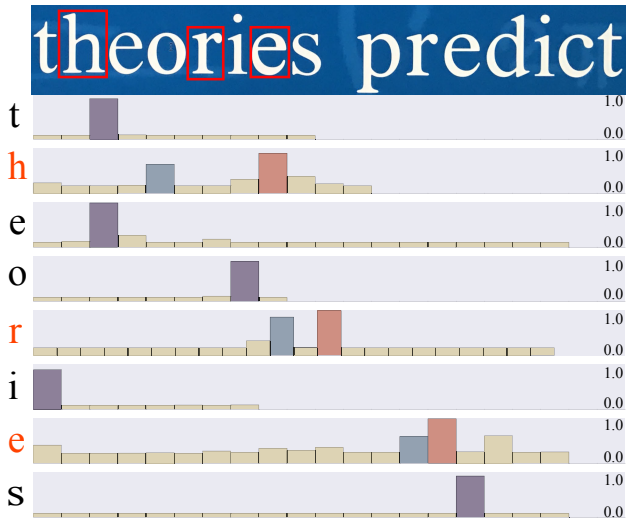


Fig. 12. **Decoding Probability. (top)** A small region of photo to be decoded, where red boxes indicate recognition errors. **(bottom)** Each row visualizes the probabilities of recognizing input glyphs (from the image) as the perturbed glyphs. Along the x-axis, every bar corresponds to a perturbed glyph in the codebook. The values on the y-axis are the output from our CNNs. When an error occurs, the blue and red bars indicate the discrepancy between the correctly and incorrectly recognized glyphs.

(orange curve). But this accuracy improvement comes with a price of the decreased codebook size. For example, for character "g", its number of perturbed glyphs in the codebook drops from 23 to 14. This indicates that if one needs to embed messages in a document that is expected to print in small fonts, then they needs to construct codebook more conservatively in order to retain the recognition accuracy.

Performance. We use tensorflow[Abadi et al. 2015] with GPU support to train and decode the input. It takes 0.89 seconds to decode a text sequence with 176 letters (30 blocks). Our encoding algorithm is running in a single thread CPU, taking 7.28 seconds for the same length of letters.

Error correction improvement. In §5.3, we hypothesize that the probability of recognizing an input pixel glyph $\boldsymbol{f}$ as a glyph $\boldsymbol{u}$ in the codebook is proportional to the softmax output of the CNNs. We validated this hypothesis experimentally. Let $g(\boldsymbol{u}, \boldsymbol{f})$ denote the softmax output value of recognizing $\boldsymbol{f}$ as a perturbed glyph $\boldsymbol{u}$. As an example shown in Fig. 12, when $\boldsymbol{f}$ is mistakenly recognized as $\boldsymbol{u}$ as opposed to its true glyph $\boldsymbol{u}^*$, the probability values $g(\boldsymbol{u}, \boldsymbol{f})$ and $g(\boldsymbol{u}^*, \boldsymbol{f})$ are both high (although $d(\boldsymbol{u}, \boldsymbol{f})$ is higher) and close to each other, indicating that $\boldsymbol{f}$ may be recognized as $\boldsymbol{u}$ or $\boldsymbol{u}^*$ with close probabilities. Thus, we conclude that using a likelihood model proportional to $g(\boldsymbol{u}^*, \boldsymbol{f})$ is reasonable.

We also extensively tested our decoding algorithm using text document photos under different lighting conditions and various camera perspectives. We verified that it can successfully decode all 5-letter blocks that have at most 2 errors. A small snapshot of our tests is shown in Fig. 13.

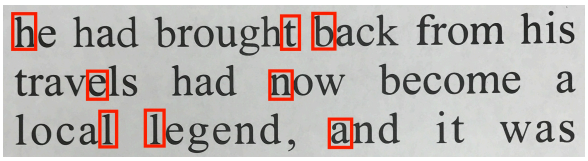


Fig. 13. **Error correction.** We decode information from an input photo. Red boxes indicate where recognition errors occur. While there exist one block contains two errors, our decoding algorithm still successfully decodes.

8.3 Perceptual Evaluation

To evaluate the subjective distortion introduced by perturbing the glyphs of a document, we conducted two user studies on MTurk. Both studies follow the standard 2AFC protocol which is described in §6 and has been used in other contexts.

Study A assesses the perceptual distortion of a perturbed glyph with respect to a standard font. We prepare a set of paragraphs, and the glyphs of each paragraph are from one of the six categories: (1) the standard Times New Roman; (2-5) the perturbed glyphs from four glyph codebooks, in which the thresholds used to select the perceptually similar glyph candidates are 0.95, 0.85 (i.e., the value we use in §6), 0.75, and 0.65, respectively; and (6) a different font (Helvetica). The font size of each paragraph ranges from 25pt to 60pt, so the number of letters in the paragraphs varies. In each 2AFC question, we present the MTurk rater three short paragraphs: one is in standard Times New Roman, the other two are randomly chosen from two of the six categories. We ask the rater to select from the latter two paragraphs the one whose font is closest to standard Times New Roman (shown in the first paragraph). We assign 16 questions of this type to each rater, and there were 169 raters participated. An example question is included in the supplemental document.

After collecting the response, we use the same model (8) to quantify the perceptual difference of the paragraphs in each of the six categories with respect to the one in standard Times New Roman. In this case, s_i in Eq. (8) is the perceptual difference of a category of paragraphs to the paragraphs in standard Times New Roman. As shown in Fig. 14, the results suggest that the glyphs in our codebook (generated with a threshold of 0.85) lead to paragraphs that are perceptually close to the paragraphs in original glyphs—much closer than the glyphs selected by a lower threshold but almost as close as the glyphs selected by a higher threshold (i.e., 0.95).

Study B assesses how much the use of perturbed glyphs in a paragraph affects the aesthetics of its typeface. We prepare a set of paragraphs whose glyphs are from one of the 12 categories: We consider four different fonts (see Fig. 15). For each font, we generate the glyphs in three ways, including (1) the unperturbed standard glyph; (2) the perturbed glyphs from our codebook using a perceptual threshold of 0.85; and (3) the perturbed glyphs from a codebook using a threshold of 0.7. In each 2AFC question, we present the MTurk rater two short paragraphs randomly chosen from two of the 12 categories, and ask the rater to select the paragraph whose typeface is aesthetically more pleasing to them. We assign 16 questions of this type to each rater, and there were 135 participants. An example question is also included in the supplemental document.

Again, using the logistic model (8), we quantify the aesthetics of the typeface for paragraphs in the aforementioned three categories

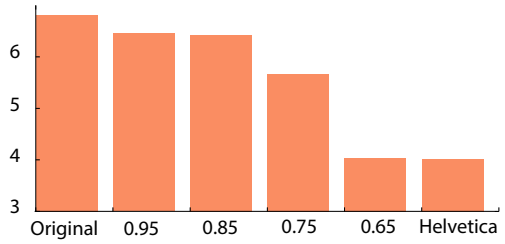


Fig. 14. **User Study A.** From left to right, the bars correspond to the user study result (see main text for details) for original font glyphs, 0.95, 0.85 (used in our examples), 0.75, 0.65, another different font, respectively.

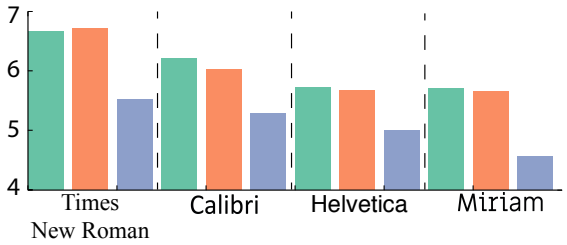


Fig. 15. **User Study B.** We conduct user studies for four different fonts simultaneously, and learn the scores of aesthetics when different types of glyphs are used: Green bars indicate the original glyphs, orange bars indicate the perturbed glyphs from our codebook, and purple bars indicate the perturbed glyphs chosen by a lower perceptual threshold (i.e., 0.7).

of glyphs. Fig. 15 shows the results, indicating that, while the aesthetics are different across the four fonts, paragraphs using glyphs from our codebook (a threshold of 0.85) are aesthetically comparable to the paragraphs in standard glyphs, whereas using glyphs selected by a lower perceptual threshold significantly depreciates the typeface aesthetics.

We note that in order to identify untrustworthy raters, in both user studies we include four control questions in each task assigned to the raters, in a way similar to those described in §6.

9 LIMITATIONS AND FUTURE WORK

We have introduced a new technique for embedding additional information in text documents. Provided a user-specified message, our method assigns each text letter an integer and embeds the integer by perturbing the glyph of each letter according to a precomputed codebook. Our method is able to correct a certain number of errors in the decoding stage, through a new error-correction coding scheme built on two algorithmic components: the Chinese Remainder coding and the maximum likelihood decoding. We have demonstrated our method with four applications, including text document metadata storage, unobtrusive optical codes on text, symmetric-key encryption, and format-independent digital signatures.

Limitations. Currently we only consider standard fonts such as regular Times New Roman, but not their variants such as Times New Roman Bold Italic. But we can treat those variants as independent standard fonts and include their perturbed glyphs in the codebook. Then, our method will work with those font variants.

When extracting messages from a rasterized text document (e.g., a photograph), letters in the document must have sufficient resolution. When printed on paper, the text document must have sufficient large font size (recall Fig. 11) for reliable message retrieval. We rely on the OCR library to detect and recognize characters. However, we cannot recover any OCR detection error. If a character is mistakenly recognized by the OCR, the integer embedded in that character is lost, and our error-correction scheme may not be able to recover the plain message since different characters may have different embedding capacities. Nevertheless, in our experiments the OCR library always recognizes characters correctly.

If a part of the text is completely occluded from the camera or contaminated by other inks, the embedded message is lost, as our decoding algorithm needs to know how the text is split into blocks. Similarly, if the document paper is heavily crumpled or attached to a highly curved surface, our method will fail, because our training data was collected on flat surface. Fig. 16 illustrates two cases wherein our method cannot recover the embedded message. In the future, we hope to improve our coding scheme so that it is able to recover from missing letters in the text as well.

In general, the idea of perturbing the glyphs for embedding messages can be applied to any symbolic system, such as other languages, mathematical equations, and music notes. It would be interesting to explore similar embedding methods and their applications for different languages and symbols. Particularly interesting is the extension to logographic languages (such as Chinese), in which the basic element for message embedding is a logogram (i.e., a written character). The number of logograms in a logographic language is much more than the size of English alphabet. For example, the number of commonly used Chinese characters is about 3500. Therefore, it would be expensive to straightforwardly extend our approach to build a codebook of thousands of characters.

Currently, our approach is not optimized to reduce the storage overhead introduced by embedding a hidden message and the codebook. How to compress a vector graphic document with embedded messages is an interesting venue for future work.

Lastly, while our method is robust to format conversion, rasterization, as well as photograph and scan of printed papers, it suffers from the same drawback that almost all text steganographic methods have: if a text document is completely retyped, the embedded information is destroyed.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We are grateful to Julie Dorsey for her suggestions of paper revision, Klint Qinami and Michael Falkenstein for video narration.

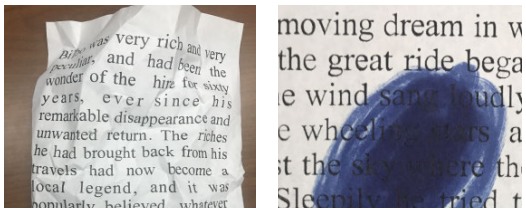


Fig. 16. **Failure cases.** If a printed text document is heavily crumpled (left) or contaminated by other colors (right), our method will fail to recover the embedded message.

This work was supported in part by the National Science Foundation (CAREER-1453101 and 1717178) and generous donations from SoftBank and Adobe. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies or others.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

A Adobe. 2001. Manager's Introduction to Adobe eXtensible Metadata Platform. *The Adobe XML Metadata Framework* (2001).

Monika Agarwal. 2013. Text steganographic approaches: a comparison. *International Journal of Network Security & Its Applications (IJNSA)* 5, 1 (2013).

Adnan M Alattar and Osama M Alattar. 2004. Watermarking electronic text documents containing justified paragraphs and irregular line spacing. In *Electronic Imaging 2004*. International Society for Optics and Photonics, 685–695.

Carlos Avilés-Cruz, Risto Rangel-Kuoppa, Mario Reyes-Ayala, A Andrade-Gonzalez, and Rafael Escarela-Perez. 2005. High-order statistical texture analysis—-font recognition applied. *Pattern Recognition Letters* 26, 2 (2005), 135–145.

Wesam Bhaya, Abdul Monem Rahma, and AL-Nasrawi Dhamyaa. 2013. Text steganography based on font type in ms-word documents. *Journal of Computer Science* 9, 7 (2013), 898.

Jeffrey A Bloom, Ingemar J Cox, Ton Kalker, J-PMG Linnartz, Matthew L Miller, and C Brendan S Traw. 1999. Copy protection for DVD video. *Proc. IEEE* 87, 7 (1999), 1267–1276.

Dan Boneh. 2000. Finding smooth integers in short intervals using CRT decoding. In *Proceedings of the 32nd ACM symposium on Theory of computing*. 265–272.

Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. The method of paired comparisons. *Biometrika* 39, 3/4 (1952), 324–345.

Jack T Brassil, Steven Low, Nicholas F. Maxemchuk, and Lawrence O'Gorman. 1995. Electronic marking and identification techniques to discourage document copying. *IEEE Journal on Selected Areas in Communications* 13, 8 (1995), 1495–1504.

Neill D. F. Campbell and Jan Kautz. 2014. Learning a Manifold of Fonts. *ACM Trans. Graph.* 33, 4 (July 2014), 91:1–91:11.

Sunita Chaudhary, Meenu Dave, and Amit Sanghi. 2016. Text Steganography Based on Feature Coding Method. In *Proceedings of the International Conference on Advances in Information Communication Technology & Computing*. ACM, 7.

Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. 2010. Digital image steganography: Survey and analysis of current methods. *Signal processing* 90, 3 (2010), 727–752.

Guang Chen, Jianchao Yang, Hailin Jin, Jonathan Brandt, Eli Shechtman, Aseem Agarwala, and Tony X Han. 2014. Large-scale visual font recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3598–3605.

François Chollet et al. 2015. Keras. https://github.com/fchollet/keras. (2015).

ADC Denso. 2011. Qr code essentials. *Denso Wave* 900 (2011).

Günay Dogan, Javier Bernal, and Charles R Hagwood. 2015. FFT-based Alignment of 2d Closed Curves with Application to Elastic Shape Analysis. In *Proceedings of the 1st International Workshop on Differential Geometry in Computer Vision for Analysis of Shape, Images and Trajectories*. 4222–4230.

D. Eastlake, 3rd and P. Jones. 2001. US Secure Hash Algorithm 1 (SHA1). (2001).

Niels Ferguson and Bruce Schneier. 2003. *Practical cryptography*. Vol. 23. Wiley New York.

Oded Goldreich, Dana Ron, and Madhu Sudan. 1999. Chinese remaindering with errors. In *Proceedings of the 31st ACM symposium on Theory of computing*. 225–234.

Jane Greenberg. 2005. Understanding metadata and metadata schemes. *Cataloging & classification quarterly* 40, 3-4 (2005), 17–36.

Adnan Gutub and Manal Fattani. 2007. A novel Arabic text steganography method using letter points and extensions. *World Academy of Science, Engineering and Technology* 27 (2007), 28–31.

Changyuan Hu and Roger D Hersch. 2001. Parameterizable fonts based on shape components. *IEEE Computer Graphics and Applications* 21, 3 (2001), 70–85.

Kensei Jo, Mohit Gupta, and Shree K. Nayar. 2016. DisCo: Display-Camera Communication Using Rolling Shutter Sensors. *ACM Trans. Graph.* 35, 5 (July 2016).

Min-Chul Jung, Yong-Chul Shin, and Sargur N Srihari. 1999. Multifont classification using typographical attributes. In *Document Analysis and Recognition, 1999. ICDAR'99. Proceedings of the Fifth International Conference on*. IEEE, 353–356.

Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.

Victor J Katz and Annette Imhausen. 2007. *The Mathematics of Egypt, Mesopotamia, China, India, and Islam: A Sourcebook*. Princeton University Press.

Young-Won Kim, Kyung-Ae Moon, and Il-Seok Oh. 2003. A Text Watermarking Algorithm based on Word Classification and Inter-word Space Statistics.. In *ICDAR*. 775–779.

Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *In International Conference on Learning Representations (ICLR)* (2015).

DE Knuth. 1986. The METAFONT book. Addison-Welsey. *Reading Mass* (1986).

Janez Konc and Dušanka Janezic. 2007. An improved branch and bound algorithm for the maximum clique problem. *proteins* 4, 5 (2007).

Vincent M. K. Lau. 2009. Learning by Example for Parametric Font Design. In *ACM SIGGRAPH ASIA 2009 Posters (SIGGRAPH ASIA '09)*. 5:1–5:1.

Shu Lin and Daniel J Costello. 2004. *Error control coding*. Pearson Education India.

Jörn Loviscach. 2010. The universe of fonts, charted by machine. In *ACM SIGGRAPH 2010 Talks*. ACM, 27.

Peter O'Donovan, Jānis Lībeks, Aseem Agarwala, and Aaron Hertzmann. 2014. Exploratory Font Selection Using Crowdsourced Attributes. *ACM Trans. Graph.* 33, 4 (July 2014), 92:1–92:9.

Nobuyuki Otsu. 1975. A threshold selection method from gray-level histograms. *Automatica* 11, 285-296 (1975), 23–27.

Jeebananda Panda, Nishant Gupta, Parag Saxena, Shubham Agrawal, Surabhi Jain, and Asok Bhattacharyya. 2015. Text Watermarking using Sinusoidal Greyscale Variations of Font based on Alphabet Count. (2015).

Huy Quoc Phan, Hongbo Fu, and Antoni B Chan. 2015. Flexyfont: Learning transferring rules for flexible typeface synthesis. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 245–256.

R Ramanathan, KP Soman, L Thaneshwaran, V Viknesh, T Arunkumar, and P Yuvaraj. 2009. A novel technique for english font recognition using support vector machines. In *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom'09. International Conference on*. IEEE, 766–769.

Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

R. Rivest. 1992. The MD5 Message-Digest Algorithm. (1992).

Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

Stefano Giovanni Rizzo, Flavio Bertini, and Danilo Montesi. 2016. Content-preserving Text Watermarking through Unicode Homoglyph Substitution. In *Proceedings of the 20th International Database Engineering & Applications Symposium*. ACM, 97–104.

Kenneth H Rosen. 2011. *Elementary number theory*. Pearson Education.

Lynn Rugglcs. 1983. *Letterform design systems*. Technical Report. Stanford University.

Ákos Seress. 2003. *Permutation group algorithms*. Vol. 152. Cambridge University Press.

Ariel Shamir and Ari Rappoport. 1998. Feature-based design of fonts using constraints. In *Electronic Publishing, Artistic Imaging, and Digital Typography*. Springer, 93–108.

Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27 (1948), 379–423, 623–656.

R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02 (ICDAR '07)*. IEEE Computer Society, Washington, DC, USA, 629–633.

Ivan Stojanov, Aleksandra Mileva, and Igor Stojanovic. 2014. A New Property Coding in Text Steganography of Microsoft Word Documents. (2014).

Rapee Suveeranont and Takeo Igarashi. 2010. Example-based automatic font generation. In *International Symposium on Smart Graphics*. Springer, 127–138.

Kiwon Um, Xiangyu Hu, and Nils Thuerey. 2017. Perceptual Evaluation of Liquid Simulation Methods. *ACM Trans. Graph.* 36, 4 (2017).

Zhangyang Wang, Jianchao Yang, Hailin Jin, Eli Shechtman, Aseem Agarwala, Jonathan Brandt, and Thomas S. Huang. 2015. DeepFont: Identify Your Font from An Image. In *Proceedings of the 23rd ACM International Conference on Multimedia (MM '15)*. ACM, New York, NY, USA, 451–459. https://doi.org/10.1145/2733373.2806219

Peter Wayner. 1992. Mimic functions. *Cryptologia* 16, 3 (1992), 193–214.

Peter Wayner. 2009. *Disappearing cryptography: information hiding: steganography & watermarking*. Morgan Kaufmann.

Received July 2017

ALGORITHM 1: Confusion Test

Initialize glyph candidates C ; ▷ §6
$G \leftarrow$ A complete graph whose nodes represent the elements of C;
while *true* **do**
  $Q \leftarrow$Random select M different pairs of nodes from C;
  foreach *pair* $(\boldsymbol{u}_i, \boldsymbol{u}_j)$ *in* Q **do**
    $a \leftarrow$ Training accuracy of CNN described using $(\boldsymbol{u}_i, \boldsymbol{u}_j)$; ▷ §4.4
    if $a < 0.95$ **then**
      remove edge $(\boldsymbol{u}_i, \boldsymbol{u}_j)$ from G;
    end
  end
  $G \leftarrow$ Maximum Clique(G);
  $C \leftarrow$ glyphs corresponding to nodes in G;
  if *No change to* C **then**
    break;
  end
end
Train CNN using all elements in C;
Remove elements whose recognition test accuracy is lower than 0.9;
return C;

ALGORITHM 2: Decoding

Input : code vector $\bar{\boldsymbol{r}} = (\bar{r}_1, \ldots \bar{r}_n)$, mutually prime integers $\boldsymbol{p} = (p_1, \ldots p_n)$, and M
Output : Decoded integer m

$\tilde{m} \leftarrow CRT(\bar{\boldsymbol{r}}, \boldsymbol{p})$;
if $\tilde{m} < M$ **then**
  $m \leftarrow \tilde{m}$;
  return success;
else
  Find codeword $\hat{\boldsymbol{r}}$, where $\hat{\boldsymbol{r}} = \min_{\hat{\boldsymbol{r}}} H(\hat{\boldsymbol{r}}, \bar{\boldsymbol{r}})$;
  if $\hat{\boldsymbol{r}}$ *is not unique* **then**
    return fail;
  else
    $m \leftarrow CRT(\hat{\boldsymbol{r}}, \boldsymbol{p})$;
    return success;
  end
end