

10

Data Hiding in Text

Today, in the digital age, any type of data, such as text, images, and audio, can be digitized, stored indefinitely, and transmitted at high speeds. Notwithstanding these advantages, digital data also have a downside. They are easy to access illegally, tamper with, and copy for purposes of copyright violation.

There is therefore a need to hide secret identification inside certain types of digital data. This information can be used to prove copyright ownership, to identify attempts to tamper with sensitive data, and to embed annotations. Storing, hiding, or embedding secret information in all types of digital data is one of the tasks of the field of steganography.

Steganography is the art and science of data hiding. In contrast with cryptography, which secures data by transforming it into another, unreadable format, steganography makes data invisible by hiding (or embedding) them in another piece of data, known alternatively as the cover, the host, or the carrier. The modified cover, including the hidden data, is referred to as a *stego object*. It can be stored or transmitted as a message. We can think of cryptography as overt secret writing and of steganography as covert secret writing.

Secret data can be embedded in various types of cover. If the data are embedded in a text file (coverttext), the result is a stego-text (or stegotext) object. Thus, it is possible to have coverimage and stegoimage, coveraudio and stegoaudio, covervideo and stegovideo, etc. This terminology was agreed upon at the First International Workshop on Information Hiding [Pfitzmann 96].

The word *steganography* is derived from the Greek *στεγανος γραφειν*, meaning “covered writing.” The term was coined by Johannes Trithemius, whose *Steganographia* [Trithemius 06], the first treatise on this subject, was published in 1606, long after his death. Another old book on steganography and cryptography is *Schola Steganographica* by Gaspari Schotti (1665) [Schotti 65]. Four hundred pages in this book are devoted to

steganography.

Notice that the term *steganography* (spelled with “stega,” meaning “covered”) is not related to *stegosaurus* (spelled with “stego,” meaning “roof”), although one may claim, as a pun, that “roof” and “cover” are semantically related.

Steganography is useful even in cases where cryptographic tools are available and provide adequate security. The reason is psychological. When a file is examined or intercepted and is found to be encrypted, it may raise suspicion in the mind of the interceptor, who may presume that the sender is performing or planning malicious or illegal acts. Someone who does not want to risk raising suspicion and prefers not to attract attention, may opt to use steganography to hide sensitive data (perhaps after encrypting it, to feel even safer) in an innocuous cover.

Embedding data in a cover is a technological challenge. The embedded data should not increase the size of the cover, because this would be noticeable to an attacker familiar with the original cover. Secret data should therefore be embedded in “holes” in the cover (places where the cover data have redundancies). Unfortunately for the steganographer, lossy compression techniques operate by removing redundancy from the cover, thereby destroying any data hidden in such holes. Thus, steganography faces the additional challenge of embedding the secret data in a robust way to make it impervious to lossy compression and other operations that may modify the cover.

Figure 10.1 shows the main steps in a typical steganographic method. The encoding algorithm receives three inputs, the secret data to be embedded, the cover data, and an optional steganographic key. The algorithm then produces a stego cover that can be stored and/or transmitted. The decoding algorithm receives the stegocover and the (optional) stego-key, and extracts the secret data. In some algorithms, the decoder cannot actually extract the data and can only answer the question, “Are these data really embedded in the file being examined?” This makes sense in cases where the hidden data are a watermark, originally placed in the cover to prove ownership or simply for pride of ownership.

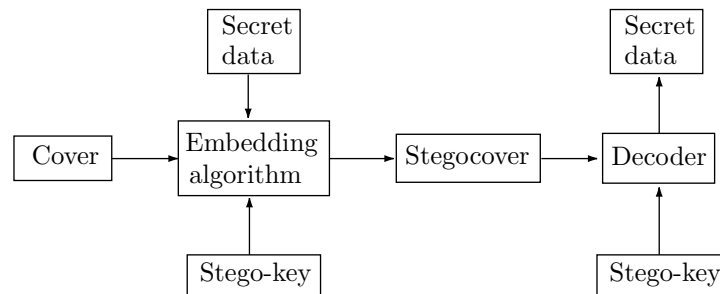


Figure 10.1: The Main Steps of Data Hiding and Extracting.

Fabien Petitcolas maintains a data hiding Web page that includes an annotated bibliography [Petitcolas 01] on the field of information hiding and digital watermarking. Some popular steganography programs are available at <ftp://ftp.csua.berkeley.edu/pub/cypherpunks/steganography/>.

Watermarking World [WatermarkingWorld 01] is an international meeting point for scientists, researchers, and organizations active in digital watermarking (one of the areas of steganography). It is the first nonprofit organization dedicated to digital watermarking. It provides means and services to the digital watermarking community. Its goal is to facilitate communication among those active in this field and to contribute to research and implementation efforts.

The organization of this chapter is as follows. Following a discussion of basic features, applications of steganography, the concepts of watermarking and fingerprinting, and intuitive methods, the remainder of the chapter is devoted to text steganography (data hiding in text files) starting at Section 10.6.

10.1 Basic Features

The many steganographic methods described in this chapter have various strengths and weaknesses and this short section discusses the main features of a data hiding algorithm.

Embedding Capacity: Data are hidden (or embedded) in a larger volume of data called a cover or a carrier. The cover is a computer file, such as text, image, audio, or video. Embedding capacity (also known as *payload*) is the amount of data that can be hidden in a cover, compared to the size of the cover. This feature can be measured numerically in units of bit-per-bit (bpb). A steganographic algorithm with small embedding capacity may have other good features such as robustness, so it may be the ideal choice when only a small amount of data, such as a short message, has to be hidden.

Invisibility: Any data hidden in a cover causes it to be modified. Invisibility (also termed perceptual transparency, or algorithm quality) is a measure of the amount of distortion to the cover. A large embedding capacity is useless if it causes large distortions to the cover. Invisibility is a qualitative feature. It cannot be measured numerically and the best way to measure it is to present several observers with the cover before and after the embedding. If no one can tell the difference between the covers, the steganographic algorithm is judged highly invisible. Invisibility is therefore tied to human visual or auditory perception.

Undetectability: An attacker may be able to detect the presence of hidden data in a given file by computing certain statistical properties of the file and comparing them to what is expected in that type of file. For example, the errors in the predictions of pixels in a color or grayscale image are many times distributed according to the Laplace distribution (this is explained in the next paragraph). If a particular image is examined and is found to have a significantly different pixel distribution, it may raise suspicion and lead to further scrutiny. Thus, a good steganographic method should not change the statistical properties of the cover file. This property is termed *undetectability* and is different from invisibility because it does not depend on human perception.

(The pixels of an image are not independent. When we select a pixel at random, we normally find that it is similar in color to its near neighbors. Thus, it makes sense to *predict* the value of a pixel by computing an average of its near neighbors. This should be a weighted average with smaller weights assigned to nonimmediate neighbors. When

the prediction is subtracted from the actual value of the pixel, the result is the error of the prediction. This is normally a small number, but may sometimes be as large as the maximum pixel value, and may also be negative. The prediction errors in the entire image are normally distributed according to the well-known Laplace statistical distribution, whose shape resembles the normal, Gaussian distribution somewhat.)

Robustness: This is a measure of the ability of the algorithm to retain the data embedded in the cover even after the cover has been subjected to various changes as a result of lossy compression and decompression or of certain types of processing such as conversion to analog and back to digital. Most steganographic algorithms embed data in an image, and images may be subject to image processing operations such as filtering, color changes, rotating, cropping, resampling, and sharpening. Robustness is especially important when the hidden data consist of copyright or ownership information (the so-called watermark). A user may compress such an image with a lossy compression method, then decompress it in an attempt to destroy any hidden watermarks.

- ◇ **Exercise 10.1:** There is a close relationship between compression and steganography. The two are often mentioned together here and elsewhere. Explain why.

Tamper Resistance: An attacker may try to alter the data embedded in a cover, rather than destroy them. A tamper resistant steganographic algorithm makes it extremely hard to alter the hidden data or to erase them and embed a different message. Especially vulnerable is copyright information embedded in a cover. Such information should stay intact for years (70 years after the author's death for text and 50 years for audio) and should resist attempts to modify it using future technologies.

Long experience in other areas of the computing field has shown that there is always a tradeoff. Improving a feature of an algorithm normally involves some downgrading of other features. In steganography, there is a tradeoff between embedding capacity and robustness (and also tamper resistance). The more robust an algorithm is, the less data it can embed in a given cover.

Notice that the data should be hidden in the *body* of the cover, not in a header, trailer, or footer. The principle is that the hidden data should stay intact even after the cover is converted to a different file format and its headers changed or removed.

However, long experience shows that anything done by a person may be undone (if not forbidden by a law of nature) by another person. It is therefore obvious that even the most sophisticated methods cannot always defeat attacks by knowledgeable, determined persons. In fact, such a person may regard sophisticated protection as a personal challenge and spend much time and effort attempting to break it.

Signal-to-Noise Ratio (SNR): This quantity serves as a measure of invisibility (or its opposite, detectability). In general, high SNR is desirable in communications systems, but a low SNR is ideal for steganography. This is because in steganography, the cover is the noise while the embedded data are the signal. As a result, low SNR corresponds to low perceptibility.

Cover Escrow vs. Blind Cover: In some steganographic methods, the original cover is needed in order to retrieve the embedded data. Such a method is said to use *escrow cover*. Where the embedded data can be extracted without the need for the original cover, the cover is said to be *blind* or *oblivious*.

Some of these requirements conflict, so any specific algorithm can satisfy only one or two of them. In particular, embedding capacity, robustness, and undetectability are mutually conflicting and cannot all be achieved by one algorithm. Figure 10.2 is a graphical description of the relationships between these three requirements. It shows that naive steganographic methods can achieve large embedding capacity, but at the expense of robustness and undetectability. Advanced algorithms can achieve a high degree of undetectability, but offer small embedding capacity and insufficient robustness. Methods for embedding a watermark are normally designed to be robust, but result in small embedding capacity and questionable undetectability.

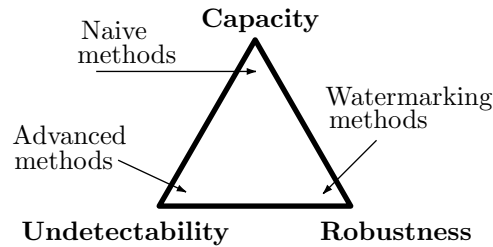


Figure 10.2: Conflicting Requirements for Data Hiding.

Other Features: Like any other discipline that uses computers, steganography has many other “digital” features, a few of which are listed here.

1. Much as encryption methods use an encryption key, steganographic algorithms can use a stego-key to increase security.

Kerckhoffs’ principle, discussed in the Introduction, applies to steganography as well. When applied to steganography, this principle, due to the Dutch linguist Auguste Kerckhoffs von Nieuwenhoff [Kerckhoffs 83], states that the security of a hidden message must depend on keeping the stego-key secret. It must not depend on keeping the hiding algorithm secret.

2. The complexity of the algorithm is also an important feature. A computationally intensive method may result in greater invisibility and robustness, and these may justify the extra time needed to embed and retrieve the hidden data. The execution time of an algorithm may often be less important than its other features. An example is a music CD with copyright information embedded. The copyright has to be embedded into the music just once, following which the digital information (music plus copyright) can be recorded on as many CDs as needed without any extra computations.

3. Asymmetric hiding of data may be a desirable feature. A slow encoding algorithm may many times be acceptable if the decoding algorithm is simple and fast.

4. The use of error-correcting codes is an advantage. Many types of processing may modify the cover and thus corrupt the hidden data. An ideal steganographic algorithm may be able to detect and even correct the embedded data after such a process.

5. Certain attacks on hidden data may destroy part of the cover. Ideally, it should be possible to retrieve that part of the embedded data that’s hidden in the remaining part of the cover.

6. A parallel steganographic algorithm is one that can be executed by several processors simultaneously, each embedding (or retrieving) part of the data. Such an algorithm may be important to steganalysts trying to break a steganographic algorithm.

10.2 Applications of Data Hiding

The first and most obvious application of data hiding is simply to hide private, sensitive data in a cover. The data are embedded either in their original, raw format or encrypted first. Since the amount of data to be hidden may be large, it makes sense to compress it prior to its encryption and embedding. This enables several parties to exchange messages without communicating directly. Sender and receiver do not even have to exchange email, use instant messaging, or log into specific computers or accounts. All that a sender has to do is post a cover (text, image, or audio file) in a public forum under a pseudonym known to the receiver. The cover is then downloaded by the receiver (as well as by any other interested party) and the secret data extracted with the use of a stego-key. A text file with a hidden message may, for example, be placed in a newsgroup site. An image file may be placed in an Internet auction site by the sender pretending to be a seller.

Another important application of data hiding is to place a digital identification or signature, commonly known as a *watermark*, in the cover. The watermark is normally a small amount of data that indicates ownership, authorship, or another kind of relationship between the cover and a person or an organization. A suitable algorithm for this application can have low embedding capacity, but has to be immune to removing or damaging the watermark. A watermark is usually placed in a cover so the owner could answer the question, “Is this cover mine?” Sometimes the aim is for anyone to be able to answer the question, “Whose file is this?” A common example is an artist who tries to sell pictures by placing them in a Web page. The pictures should be watermarked and part of the watermark should be visible, to deter pirating. The hidden part of the watermark should not be affected even when the visible part is removed.

Watermarking is an important application for the publishing and broadcasting industries, because any advance in multimedia technologies brings with it new markets but also new opportunities for illicit copying and pirating.

Fingerprinting is a variant of watermarking, where each watermark is different. This is useful in cases where many copies of the same product have to be tagged. When a fingerprint is hidden in a music CD, any illegal copies discovered and seized can be traced back to the original from which they were made.

Another variant of watermarking is *traitor tracing*. Before an important document is disseminated to those in need to know it, each is fingerprinted differently. If the document finds its way to the general public, the person responsible for the leak can easily be identified.

A third application is tamper-proofing the cover. Data can be hidden in such a way that any modification to the cover would be reflected in the hidden data. The purpose of this application is to be able to answer the question, “Has this file been modified?” If the cover is an audio file, tamper-proofing should be able to detect whether some words

Rumor has it that when Margaret Thatcher was the British prime minister in the 1980s, leaks of secret cabinet meetings to the press were so rife that she ordered the word processors of her cabinet ministers reprogrammed such that each generated slightly different spaces, to identify the origin of a document. No word as to whether the culprit had been unmasked this way.

have been erased or modified, but should forgive modifications such as stereo left–right balance. If the cover is an image, tamper-proofing should be able to detect whether an object has been removed from the image, but should ignore small modifications such as change of color space or gamma correction.

Sealing a file is a special case of tamper-proofing. A checksum of the file is prepared and is hidden in the file. The file creator can then easily test the file for a broken seal.

Laws enacted by governments to restrict the encryption capabilities of private citizens are the source of the fourth important application of data hiding. Anyone concerned with privacy and unable to legally use strong encryption should consider data hiding. Also, as more lawmakers come to appreciate the power of steganography, they may relent and remove any useless, unproductive restrictions on cryptography.

Methods of hiding data in audio files may have their own applications. Caller id information can be embedded in a telephone conversation, enabling the receiver to identify the caller. A client contracting with a radio station to broadcast commercials can hide identifying information (a signature) in the commercial. A computer can then listen to the radio station continuously, looking for the signature. This can tell the client when, how often, and how many times the commercial had been broadcast. It can also identify cases where the commercial was only partly broadcast.

The last important application of steganography is feature location (or feature tagging). It may be desirable to hide in the cover items such as captions, annotations, names of persons or places associated with the cover, and dates (of creation and modification of parts of the cover). This application may require a large embedding capacity but not as much robustness as watermarking.

10.3 Watermarking

The term *watermarking* refers to any technique used to embed ownership information in important digital data. The watermark can be a signature of the author, placed in the data for pride of authorship, the same reason that painters sign their work. The watermark can also be used to mark an important digital creation (mostly image, video, audio, but even text) before it is released externally (posted on the Web, sent to a magazine or a museum, or sold to a collector). In such cases, the watermark can later be used to identify digital data that have been illegally copied, stolen, or altered in any way. A typical example is a map. It takes time, money, and effort to create a road map of a city. Once this map is digitized, it becomes easy for someone to copy it, make

slight modifications, and sell it as an original product. A watermark hidden in the map can help the original developer of the map identify any attempts to steal the work. The watermark itself should not increase the size of the original data significantly, and must also be robust, so it does not get destroyed by operations such as filtering, compressing, and cropping.

Watermarking can be achieved by steganographic methods. The watermark may be a string with, for example, the name of the owner, repeated as many times as needed. This string is hidden in the original data (image or audio) and can be extracted by the original owner by means of a secret key.

There is, however, a fundamental difference between steganography and watermarking. In the former, the hidden data are important, while the cover image is not. The embedding capacity of the algorithm is important, but the hidden data may be fragile (they may be destroyed by transforming or compressing the cover image). In watermarking, the cover image is valuable, while the hidden data are not (they can be any identifying data). The embedding capacity is unimportant, but the hidden data have to be robust. As a result, watermarking should use special steganographic techniques.

The related concept of *fingerprinting* should also be mentioned. Fingerprinting refers to embedding a secret serial number in each copy of some important digital data. A commercially sold computer program, for example, is easy to copy illegally. By embedding a secret serial number in each copy of the program sold, the manufacturer can identify each buyer with a serial number, and so identify pirated copies.

[Cox 02] is a general reference for watermarking.

10.4 Intuitive Methods

We start with a few elementary and tentative methods, most of which are implemented manually, although some can benefit from a computer. These are mostly of historical interest.

1. Write a message on a wooden tablet and cover it with a coat of wax on which another, innocuous message is written. This method is related by Greek historians. Modern variations include hiding a message in a hollow heel or in a false bottom of a suitcase.

2. Choose a messenger, shave his head, tattoo the message on his head, wait for the hair to grow, and send him to his destination, where his head is shaved again. This method is attributed to Histiaeus.

3. Use invisible ink, made of milk, vinegar, fruit juice, or even urine, and hide the message between the lines of a seemingly innocent letter. When the paper is heated up, the text of the hidden message slowly appears.

4. The letters constituting the data are concealed in the second letter of every word of a specially constructed *cover text*, or in the third letter of the first word of each sentence. An example is the data “coverblown” that can be hidden in the specially-contrived cover text “Accepted you over Neil Brown. About ill Bob Ewing, encountered difficulties.” A built-in computer dictionary can help in selecting the words, but such

specially-constructed text often looks contrived and may raise suspicion. A variation on this method starts by writing the words of the secret message vertically. Each word becomes the first word of a line and the steganographer (writer or originator of the message) has to complete each line by adding more text. The following is an example of how the message “Come to our place at midnight” can be hidden in this way.

Come see how the swallows fly
to the south island helped by
our fine weather. They locate the
place in the immense ocean by navigating
at night, following the clouds which form around
midnight above the shore.

5. An ancient method to hide data uses a large piece of text where small dots are placed under the letters that are to be hidden. For example, this paragraph has dots placed under certain letters that together spell the message “i am hidden.” A variation of this method slightly perturbs certain letters from their original positions to indicate the hidden data.

6. A series of lists of alternative phrases from which a paragraph can be built can be used, where each choice of a phrase from a list conceals one letter of a message. This method was published in 1518 in *Polygraphiae* by Johannes Trithemius, and was still used during World War II.

◇ **Exercise 10.2:** Show examples of such phrases.

7. Check digits. This idea is used mostly to verify the validity of important data, such as bank accounts and credit card numbers, but can also be considered a method for hiding validation information in a number. A simple example is the check digit used in the well-known *international standard book number* (ISBN), assigned to every book published. This number has four parts, a country code, a publisher code, a book number assigned by the publisher, and a check digit, for a total of 10 digits. For example, ISBN 0-387-95045-1 has country code 0, publisher code 387, book number 95045, and check digit 1. The check digit is computed by multiplying the leftmost digit by 10, the next digit by 9, and so on, up to the ninth digit from the left, which is multiplied by 2. The products are then added, and the check digit is determined as the smallest integer that when added to the sum will make it a multiple of 11. The check digit is therefore in the range $[0, 10]$. If it happens to be 10, it is replaced by the Roman numeral X in order to make it a single symbol.

◇ **Exercise 10.3:** What is the check digit for the 9-digit ISBN 0-387-98682?

8. It is possible to assign 3-digit codes to the letters of the alphabet as shown in Table 10.3. Once this is done, each letter can be converted into three words of cover text according to its 3-digit code. A code digit of 1 is converted into a word with 1, 4, or 7 syllables, a code digit of 2 is converted into a word with 2 or 5 syllables, and a code digit of 3 is converted into a word with 3 or 6 syllables. This way, there is a large selection of words for each word in the cover text.

254 10 Data Hiding in Text

A 111	D 121	G 131	J 211	M 221	P 231	S 311	V 321	Y 331
B 112	E 122	H 132	K 212	N 222	Q 232	T 312	W 322	Z 332
C 113	F 123	I 133	L 213	O 223	R 233	U 313	X 323	

Table 10.3: Three-Digit Codes for the Letters.

◊ **Exercise 10.4:** Convert the two letters MO to innocuous text in this way.

9. Similarly, each of the 26 letters can be assigned a 5-bit code (there are 32 such codes, so six more symbols can be added to the 26 letters and coded). Thus, any message to be hidden becomes a binary string. To hide the string, select some innocuous text and hide each bit in a letter of this text. A bit of zero, for example, may be hidden by changing the letter to lowercase or to italics. Thus, the bits 011000011 may be hidden in the text **to be or not** by changing it to **t0 Be or n0T**. This type of information hiding is sometimes referred to as Bacon's biliteral cipher.

10. Select a keyword, such as **blah**. Find the serial numbers, 2, 12, 1, and 8 of its letters. Construct a grid with 12 columns and several rows, and mark columns 1, 2, 8, and 12 of each row. The letters of the message are inserted in those positions, four letters per row, and are hidden by filling the grid up with other letters, trying to come up with meaningful text.

11. Assign a letter to each of 26 musical notes, then write your message in the form of musical notes and hope that no one will try to play the music. This idea is due to Gaspari Schotti who published it in his *Schola Steganographica* [Schotti 65].

12. Write or print the data on paper, then photograph it and shrink it to the size of a dot, like the one at the end of this sentence. Send an innocuous letter to the receiver and paste the dot at the end of one of the sentences. The receiver can identify the microdot by holding the letter to the light, looking for a shiny dot.

Microphotography. You see, if there's anything really urgent that you can't put in a telegram, London wants us to communicate direct and save all the time it takes to Kingston. We can send a microphotograph in an ordinary letter. You stick it on as a full stop and they float the letter in water until the dot comes unstuck. I suppose you do write letters home sometimes. Business letters. . . ?

—Graham Greene, *Our Man In Havana* (1958)

◊ **Exercise 10.5:** Try to guess the data hidden in the sentence “I’m feeling really stuffy. Emily’s medicine wasn’t strong enough without another nembotil.”

10.5 Simple Digital Methods

Modern steganography methods are more sophisticated than the ones presented so far and are based on the use of computers and on the binary nature of computer data. They can be classified into naive steganography (methods where no key is used), secret steganography (methods where a secret key is used), and public-key steganography (methods where an asymmetric key is used, similar to public key in cryptography). All three classes of steganographic methods are based on the fact that every communications process is accompanied by some natural randomness. A steganographic method replaces this randomness by the hidden data without changing the nature of the process. The hidden data end up being *embedded* in the process.

A modern personal computer may have tens of thousands of files on one hard disk. Many files are part of the operating system, and are unfamiliar to the computer owner or even to an expert user. Each time a large application is installed on the computer, it may install several system files, such as libraries, extensions, and preference files. A data file may therefore be hidden by making it look like a library or a system extension. Its name may be changed to something like `MsProLibMod.DLL` and its icon modified. When placed in a folder with hundreds of similar-looking files, it may be hard to identify as something special.

Camouflage is the name of a steganography method that hides a data file D in a cover file A by scrambling D , then appending it to A . The original file A can be of any type. The camouflaged A looks and behaves like a normal file, and can be stored or emailed without attracting attention. Camouflage is not very safe, since the large size of A may raise suspicion.

When files are written on a disk (floppy, zip, or other types), the operating system modifies the disk directory, which also includes information about the free space on the disk. Special software can write a file on the disk, then reset the directory to its previous state. The file is now hidden in space that's declared free, and only special software can read it. This method is risky, because any data written on the disk may destroy the hidden file. Section 12.10.3 describes an example of such a method.

10.6 Data Hiding in Text

Text has less noise than an image, so hiding data in text normally results in low embedding capacity. Nevertheless, there are several methods, some of them ingenious, for hiding bits of data in a text file. This section discusses three general approaches and is followed by several detailed algorithms that hide data in text.

Modifying Spaces: Data can be hidden in a cover text by modifying blank spaces. A word processor can modify (1) the interword spaces in a sentence, (2) the spaces at the end of each line, and (3) the spaces following punctuation marks. Normally, spaces are automatically adjusted by the word processor in order to justify the right margin; they cannot be explicitly controlled by the user. Such a word processor should be rewritten to (1) allow the user a certain degree of control over spaces and (2) list the precise sizes of the blank spaces in a document, so that the hidden bits could be retrieved.

In a primitive word processor where spaces have fixed size, a bit can be hidden at the end of each sentence by appending one or two spaces to the sentence, where one space indicates a hidden 0 and two spaces indicate a hidden 1. Since a sentence ends with a period, every period in the text, even those in a context such as “Mr. Smith,” hides one data bit and must be followed by one or two spaces.

Appending one or two spaces to the end of each line is also a simple data hiding method. Such spaces do not show up when the text is printed, but can be easily identified by the word processor. A potential problem may arise when the text is processed by programs that remove extra blank spaces.

A word processor using fixed-size spaces justifies the right margin of the text by placing more than one space between certain words, and this can be exploited for hiding data bits. Either one or two spaces are placed between successive words, to hide a 0 or a 1, respectively. A potential problem with this method may be a case where the last two words on a line must have just one space between them (to right-justify the line), but the current bit to be hidden is a 1, requiring two spaces. A better algorithm hides bits by interpreting spaces as follows.

1. A single space followed by a word followed by a double space is interpreted as a bit of 0.
2. A double space followed by a word followed by one space is interpreted as a 1.
3. A single space followed by a word followed by a single space is interpreted as no data hidden.
4. A double space followed by a word followed by a double space is also interpreted as no data hidden.

◊ **Exercise 10.6:** Figure out the bits hidden by this method in the text

```
happy_families_are_all_alike_every_unhappy_family_is_unhappy
in_its_own_way_everything_was_in_confusion_in_the_oblonskys
house_the_wife_had_discovered_that_the_husband_was_carrying
on_an_intrigue_with_a_french_girl
```

The T_EX typesetting software [Knuth 84] permits very fine control over the interword spaces and the spaces following certain punctuation marks. The smallest dimension that T_EX can use is called a *scaled point* (sp). One inch equals 72.27 printers’ points (pt), and one pt equals 65,536 scaled points. Thus, the value of a sp is about the wavelength of visible light, and changing the normal interword space by 1 sp is invisible. T_EX can also list the precise values of all the components of text (dimensions of letters and spaces). Because of these features, T_EX may be an ideal tool for hiding data in spaces, although it was originally designed as a high-quality typesetting system for the production of books.

Syntactic Methods: These methods are based on ambiguous punctuation or on modifying the text such that its meaning is preserved. The former approach is vulnerable to attack, because inconsistent use of punctuation is noticeable, especially to an observer predisposed to being suspicious. The latter approach is safer but harder to implement, because computers are notoriously bad at “understanding.”

As an example of ambiguous punctuation consider the phrases “a common, boring, but responsible task” and “a common, boring but responsible task.” The latter phrase

omits one comma and may be considered syntactically wrong by certain editors and linguists. The point is that one bit may be hidden in each of the two types of phrases. This method has low embedding capacity, since only one bit can be hidden for each phrase of the form “A, B, and/or/but not C” in the cover text.

Another example of applying ambiguous punctuation is slight modification of abbreviations. We normally write “see e.g., page 100” but the phrase “see e.g. page 100” is only slightly different. These two forms of “e.g.” may be used to hide bits.

Modifying text while preserving its meaning is a subtle way to hide data bits in text. It is easy to hide one bit in each of the phrases “when you finish this, you can go” and “you can go when you finish this.” Such a method has low embedding capacity and must involve at least some manual adjustments to the text, but it is reliable, because it is difficult to detect the two types of phrases mechanically (by computer). An attacker would have to actually read the message and manually identify the relevant phrases and extract the data bits.

Semantic Methods: Data are embedded in text by special word usage. The sender and receiver using such a method may agree on the use of a certain online thesaurus. This is a data set that contains synonyms for many words. The decoder reads the cover text word by word and searches the thesaurus for the first occurrence of each word as a synonym. If a word, such as **godchild**, is not the synonym of any other word, the decoder assumes that no data are hidden in it. If a word such as **child** is input, whose first occurrence in the thesaurus as a synonym is in the list **bud**, **chick**, **child**, **kid**, **minor** (perhaps five synonyms for **youngster**), then this list is considered to hide two bits and **child** (being the third word in the list, where word count starts from 0) is interpreted as hiding the 2-bit number 2 (01 in binary).

Encoding is not trivial but can be done mostly mechanically. The encoder inputs the next word of the cover text. Suppose it is **child**. It searches the thesaurus and finds that the first occurrence of **child** as a synonym is as the third word (i.e., word number 2) of the list of synonyms for **youngster**. If the next two bits of data to be hidden happen to be 01, then **child** can be used as is. Otherwise, the encoder tries to replace **child** with **bud**. It searches the thesaurus for the first occurrence of **bud** as a synonym and proceeds as with **child**. If **bud** cannot be used, the encoder tries to replace **child** with **chick**, and so on. If none of the five synonyms in the list can be used, the encoder may ask for a person’s help in finding a replacement for **child**.

Another approach to semantic data hiding in text is to define a function that reduces a sentence to one bit. A possible choice is the parity (odd or even) of all the ASCII codes of the characters in the sentence. In a practical implementation of this method, the word processor should be modified as follows.

1. It first inputs the secret data and stores them as a string of bits.
2. It computes the function each time a period is typed in the artificial text and compares its result to the next bit to be hidden.
3. If the parity of all the ASCII codes of the current sentence corresponds to a bit opposite that to be hidden, the word processor beeps and refuses to take any more input. The only choice for the user is to back up and rewrite the sentence.

In the future, meaningful artificial text may be generated and revised by computers. At present, a person should do this task.

10.7 Innocuous Text

A government authority such as the NSA that intercepts a huge number of messages every day must use computers to analyze them and identify a small number of suspect messages, to be further checked by humans. A computer program looking for suspect messages checks first for randomness. If the message consists of random data, it may be enciphered and is therefore potentially suspect, but it may also be innocent and look random simply because it is compressed. The next test may be for a plain text message. Such a test starts by computing letter and digram frequencies. If these seem normal, then the program looks for words that are not in a dictionary (while doing this, the program may flag suspect words, such as “smuggling,” “FBI,” and “bomb”). If most of the words are in a dictionary, the program tries to identify invalid syntax (sentence structure). If a text file passes all these tests, it will be considered “clean” by the computer. Thus, one approach to steganography is to hide information (binary data) by creating a text message that has only valid words (i.e., words drawn from some dictionary) and where the words are generated in a special order so they constitute syntactically valid sentences. Such a text message contains nonsense text, but may fool any computer algorithm designed to detect suspicious messages. The method used to transform pieces of binary data to words must, of course, be reversible, so a receiver will be able to read any hidden messages received.

This approach to steganography is different from hiding data in an image or an audio file. The data are not hidden in a cover image. Instead, pieces of data are replaced by words. Some may consider this approach a form of cryptography rather than steganography.

This section is based on two actual implementations, the *NICETEXT* program, by Mark Chapman and George Davida [Nicetext 01] and *Steganosaurus* by John Walker [Steganosaurus 01]. The former reads binary data and uses a dictionary and a style source to generate an innocuous text file that can later be converted back, with the help of the dictionary, to the original data. The dictionary contains valid words, classified according to type. The latter is similar but does not use syntax rules.

The example dictionary shown here has five grammatical types of words.

```
Article (4): the, a, this, your.
Noun    (16): dog, ball, woman, spring, John, Mary, house, man,
              car, soil, brush, animal, place, sentence, cup, food.
Verb    (16): hit, took, saw, spring, longed, sees, runs, played,
              hates, hear, knew, guesses, liked, walked, believes, work.
Adj.    (8): big, little, blue, green, long, wide, bright, dull.
Prep.   (4): to, in, by, which.
```

Notice that a word may have more than one meaning and may therefore appear more than once in the dictionary. To avoid ambiguity, such a word should always appear in the same position in all its types and those types should have the same sizes. In our example dictionary, the word **spring** is both a noun and a verb, but appears in position 3 in both lists (positions are numbered from 0). Also, both lists are 16 words long. The word **long** is both a verb and an adjective, but these types have different lengths, so

the verb **long** had to be changed (to **longed**). Because of this feature, the decoder does not need the style rules.

Another solution to the problem of multiple dictionary words is to merge types. If the word **spring** is a noun and a verb, a new type **Noun-Verb** may be created, with **spring** (and possibly other words) included in it.

The style source provides syntax rules for several types of sentences. A basic sentence may have the syntax **Article, Noun, Verb, Article, Noun**, whereas a more complex sentence may be expressed by the rule

Article, (Adj), Noun, Verb, Article, Noun, (Prep, Article, Noun)

where the parentheses indicate repetition. More complex style rules may include options. For example, the square brackets in

Article, (Adj), Noun, [Verb, Noun-Verb], Article, Noun, (Prep, Article, Noun)

indicate either a verb or a noun-verb.

Imagine the 17-bit binary data 01011101001010101. The Nicetext encoder randomly selects one of the style rules and uses the input bits to select appropriate words from the dictionary for a sentence. If the first syntax rule is selected, the first word the encoder has to generate is of type **Article**. There are four such words, so the encoder uses the first two bits 01 of the input as a pointer and selects word 1 (the second word) of type **Article**. This is the word **a**. The next syntactic element of this syntax rule is **Noun**. There are 16 nouns, so the encoder uses the next four bits 0111 of the input as a pointer to select word 7 (**man**) of this type. The next four input bits 0100 are used to select **longed** (in type **Verb**), the following two bits 10 select the article **this**, and the next four bits 1010 select the noun **brush**. The 16-bit input string 01|0111|0100|10|1010 ends up being encoded as the sentence **a man longed this brush**. The encoder may generate a period and a space, then randomly select another style rule and encode more input bits. The random numbers used to select style rules may be uniformly distributed, or may have a distribution that prefers certain rules, thereby generating text dominated by certain types of sentences.

Since fragments of the input are used as pointers, the number of words included in each dictionary type must be a power of 2 (2^n for $n \geq 1$).

A problem arises if too few input bits remain, such that no style rule can be used. In our example, only one input bit remains, namely the bit 1. Our dictionary does not have a type with just two rules, so this single bit cannot be used as a pointer. A possible solution is to measure the length of the binary data before the encoding process starts and to prepare a string consisting of (1) the length of the data as a fixed-size number, (2) the binary data themselves, and (3) many random bits. In our example, the binary data are 17 bits long, so we can prepare the string

0000010001|01011101001010101|1100...

where the length 17 is written as a 10-bit number and the 17-bit data are followed by some random bits. When the encoder arrives at the end of the data, it uses as many of the random bits as needed to select the last word. The decoder starts by decoding the fixed-size length, so it knows how many bits remain to be decoded.

◇ **Exercise 10.7:** Suggest another solution.

Decoding is simple because the decoder does not need the style rules. When the decoder inputs the first word **a**, it searches the dictionary and finds **a** at position 1 (second position) of type **Article**. Since this type consists of four words, the **a** is decoded to the two bits 01. If the word **spring** is input by the decoder, it is found either in the word-list for type **Noun** or in the list for type **Verb**. In either case, it is decoded to the four bits 0011.

The types in the dictionary may be more than just grammatical. There may be, for example, the types **MaleName**, **FemaleName**, and **SportVerb**. The latter type should include verbs such as **run**, **kicks**, and **jumped**. If the user wants to generate output text that seems to discuss sports, then instead of the style rule **Article, Noun, Verb, Article, Noun**, there may be the rule **Article, MaleName, SportVerb, Article, Noun**.

A syntax rule in the style source may also include punctuation marks, such as commas and question marks. Those are inserted by the encoder into the text and are ignored by the decoder. It is also possible to have several dictionaries and style sources and any compatible pair may be used. A style source and a dictionary are compatible if every type mentioned in the syntax rules exists in the dictionary and any punctuation marks in the style are not words in the dictionary.

The syntax rule **do-verb name verb article noun prepos name?** is an example of a rule that may generate nonsense but seemingly correct sentences such as **will Henry gave my brush to Janet?**.

This basic method may be extended and generalized in various ways to provide greater security. In one such variant, the encoder and decoder use different dictionaries. In the encoder's dictionary, several extra words have been appended to each type. From time to time, the encoder selects one of those words at random, and embeds it, as an extra word, in the text being generated. The decoder cannot find those words in its dictionary, so it simply ignores them and decodes the rest of the words. The advantage of this variant is that an eavesdropper who has access to the encoder's dictionary will not be able to use it to decode messages.

Another variant of the basic method embeds chunks of the innocuous text in real text. It is relatively easy to embed a few innocuous sentences (identified by special key words preceding and following them) in every other paragraph of some real text. This may fool even a person looking for suspicious messages. The decoder reads the text, looking for the key words to identify the important parts, and ignoring the rest.

We next describe ways to create a large dictionary with types. Collecting words is very easy because there are large collections on the Internet. One such file of English words can be found at [FreeBSD Words 01]. Another is the Gutenberg collection of books, located at [Gutenberg 01]. It is trivial to write a program that will delete all duplicate words, but the main problem is to associate a type with each word. There are several approaches.

1. Do it manually (preferably by an experienced user). This should be done as a last resort, after types have been assigned to most words automatically (i.e., by a computer program).

2. Use an online dictionary that has a type (such as noun, verb, or adjective) already assigned to each word. A program may be written that goes over the dictionary word by word and extracts each word and its type.

3. Use morphological analysis on root words. The root word **compartment** can be the source of derived forms such as **compartmented**, **compartmentalize**, **compartmentalization**, **recompartmentalize**, **noncompartmentalize**, and others. This is a complex process that does not work on many words.

Figure 10.4 shows a possible dictionary organization. Each type is assigned a number (article, noun, verb, adjective, and prep are assigned the numbers 1–5), so a syntax rule is a set of numbers (1, 2, 3, 1, 2 in the figure). The numbers point to an array of five structures (one of each type) where each structure contains the number of dictionary words for this type and a pointer to the start of the type in the dictionary array. The dictionary array itself is a dense array of characters, where each word is terminated by a special ASCII code).

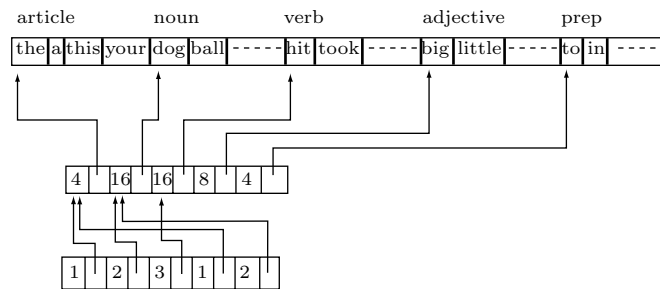


Figure 10.4: Dictionary Organization for Nicetext.

A full implementation of this approach to steganography should contain tools to create custom dictionaries from a variety of sources, to simulate many different writing styles by example, and to alternatively use context-free grammars to control writing style.

Data expansion is an inherent problem with this type of steganography. To get an idea of the numbers involved, we assume a typical word size of four bytes. A typical grammatical type in a large dictionary should contain at least 1000 words, so a pointer to such a type is about 10 bits (because $2^{10} = 1024$). Thus, the algorithm replaces 10 bits of original data with about five bytes (a 4-byte word followed by a space, a total of 40 bits), resulting in an expansion factor of 4, very large. The solution is to use compression. The original data should be compressed, then enciphered (for added security), encoded as innocuous text, and the resulting text file (which is nonrandom) compressed again before being transmitted. As an example, a 1 Mbyte file may be reduced to 500 Kbytes by compression. Encryption may increase this to, say, 600 Kbytes, and encoding may multiply this by four, resulting in a 2.4 Mbyte file. After further compression, this may become a 1.2 Mbyte file, only about 20% bigger than the original.

Figure 10.5 lists some innocuous text in the style of Shakespeare, generated by Nicetext.

Which subtext so cold that is not remounted here? Would import above bran once think it? Hansom, I will. You lid me mistake it generally and yeah, Gaining to the pension and the rhyme. You may not, my lord, disguise her inertial jute. HOW, My lord! The ame oneself doeth reek Before equivalent parody. Hark you, sir. The Formant House Girlish. ZOUNDS, the mud more occurs To rouse a baron than to sort a mare! INSURGENT Good, i faith! I have referred my father blame him. Precon- tently, niobium, imaginatively. Thence, pack! Now shine it like a comet above infringe, A packet to the shawl above all our shows! Sorghum, your Sovereignty istoomuch sad. Good sparrow, groceries. IN Rearming whichever was furthest, we shall part up another. A background valves! Strut my lace, Lillian, strum! Vestally, naturedly: Therefore acquiesce thee studiously of thy pin, For to deny each marble ere oath Cannot re- move nor stoke the prolong permission That I do moan unright. Pester Johnnie. Now remit down, now omit down; come, margin. Adieu; be happy! While I uoresce mushrooming it by topple. The raven himself is sparse That wednesdays the managerial entrance of Jamison Neath my. . .

Figure 10.5: A Shakespeare-Like Nicetext.

10.8 Mimic Functions

Hiding data in artificially generated text must result in text that can pass any mechanical tests. Such text has to satisfy at least the following requirements.

1. The letter frequencies in the text must resemble those of a natural language. If the text is supposed to resemble English, then E and T should be the most-common letters and Z and Q should be the rarest.
2. Most words in the text must be found in a good dictionary. Any text may include some words, such as proper names, slang, and scientific terms, that may not be found in a given dictionary, but if a computerized check finds too many such words, it should flag the text as suspicious.
3. The sentences in the text must be syntactically correct. If an automatic syntax checker finds, for example, two consecutive verbs in the text, it should become suspicious.

Just generating artificial text does not hide any data in it. One way to hide data in artificial text is to develop a method where a decision has to be made each time a word or a phrase is added to the newly generated text. Imagine a method where one of two options has to be chosen each time the next syntactic element is added to a sentence. The next data bit to be hidden can drive those decisions, resulting in text that can pass many computerized tests and also contain hidden data. An even better algorithm would have to select one of four options at each step, thereby hiding two data bits at a time.

The method described in this section does just that. It uses a context-free grammar to generate artificial text that mimics real text (i.e., has the same statistical properties), and uses several bits from the data being hidden in order to select one of several options at each step. The method is due to Peter Wayner [Wayner 92 and 02], who published the source code of his implementation, along with many examples, in [Wayner 02]. The method is based on the concept of *context-free grammars*, so we start with a short description of this important technique.

When I did him at this advantage take,
 An ass's nole I fixed on his head.
 Anon his Thisby must be answered,
 And forth my mimic comes.

Puck's monologue in Shakespeare's
A Midsummer Night's Dream,
 Act III, Scene ii

A context-free grammar (CFG) is a set of rewriting rules (also referred to as production rules or productions) that can be explicit or recursive. The rules are used to generate strings of various patterns. The set of all strings generated by a particular CFG is the *language* generated by the CFG. This set may be finite or (if the rules are recursive) infinite. The strings are considered sentences in the language. The concept of formal grammars was originated in the 1950s by Noam Chomsky [Chomsky and Miller 58].

A CFG consists of the following.

1. A set of *terminal* symbols. These are the characters and words (the alphabet) that constitute the sentences generated by the grammar.
2. A set of *nonterminal* symbols. These are placeholders for patterns of terminal and nonterminal symbols. In our examples, the nonterminals are typeset in boldface.
3. A set of *productions*. These are rules for replacing (or rewriting) nonterminal symbols in a string with other nonterminal or terminal symbols. A production has the form $L \rightarrow R$ where L is the nonterminal symbol that's replaced by the string R of nonterminal or terminal symbols.
4. A *start* symbol; a special nonterminal. The process of generating a string by the grammar should start with a production that has this symbol on its left-hand side.

The following rules specify how to generate a string (of terminal symbols) from a given CFG.

1. Use the start symbol as the initial nonterminal.
2. Select a production that has the start symbol on the left-hand side and use it to replace the start symbol with the right-hand side of the production. This is the text generated so far.
3. Select a nonterminal symbol in the text, find a production that has this nonterminal on the left side, and replace the nonterminal with the right hand side of the production.
4. Repeat Step 3 until the resulting text consists of just terminal symbols.

Here is a simple example of a CFG where the nonterminal symbols are typeset in boldface and the terminals are typeset in a Roman font.

Start \rightarrow noun verb
noun \rightarrow Alice | Bob
verb \rightarrow is sending | is receiving

The four rules above can be applied as follows.

1. Select **Start**.
2. Replace with **noun verb**.
3. Replace **noun** with “Alice” to obtain “Alice **verb**.”
4. Replace **verb** with “is sending” to obtain “Alice is sending.”

This generates the sentence “Alice is sending,” but it is obvious that the three sentences “Alice is receiving,” “Bob is sending,” and “Bob is receiving” can also be generated by this CFG if we select the productions and terminals in different orders. The four sentences that can be generated by this CFG constitute the language generated by the CFG.

The reason for the name “context-free” is that productions can be selected in any order. It is also possible to have context-sensitive grammars, where the choice of a production at each step is limited by certain rules.

◇ **Exercise 10.8:** Find the language generated by the following CFG.

Start → **noun verb**
noun → Alice | Bob
verb → is sending **what** | is receiving **what**
what → **type** | data | clothes
type → clean | dirty

The productions can also be recursive, which results in an infinite number of sentences that can be generated. A simple example is a CFG to generate simple arithmetic expressions. An expression consists of unsigned integers, the two arithmetic operations “+” and “−,” and parentheses.

Start → **expression**
expression → **number** | (**expression**)
expression → **expression** + **expression** | **expression** − **expression**
number → **digit** | **number digit**
digit → 0 | 1 | ... | 9

The first rule states that a sentence is an expression. The second production says that an expression is a **number** (a nonterminal to be defined later) or any valid expression in parentheses (a recursive choice). The third production adds two recursive choices to **expression**. The fourth production defines **number** as either a single **digit** or (recursively) as any **number** with a **digit** appended to it. The last rule defines the nonterminal **digit** in terms of 10 terminals. The resulting sentences can have any length because the depth of the recursion is unlimited.

Once the basic concept of a CFG is clear, it is easy to see how data can be hidden in the sentences generated by the grammar. The idea is to associate each element (terminal or nonterminal) in the right-hand side of a production with a binary string. Thus, the four choices in the production **something** → A | B | C | D (notice that **C** is a nonterminal) correspond to the four bit strings 00, 01, 10, and 11. If the next pair of bits to be hidden is 01, the encoder should choose “B” at this point. An obvious conclusion is that each production should have 2^n choices in its right-hand side. If a production contains five choices, then the fifth one will never be used. Here is an example of a CFG used to hide the string 0100110.

Start → **adjective noun tense verb**

adjective → the **size** | a **size**

size → tiny | small | large | big

noun → saw | ladder | truth | boy

tense → is | was

verb → waiting | standing

The first nonterminal is **adjective**. The production for this nonterminal has two choices, so one bit can be hidden by picking the right choice. The first bit to be hidden is 0, so the first choice (the **size**) is selected. The terminal “the” is appended to the text and the nonterminal **size** is replaced next. There are four choices, so two bits can be hidden. The next two bits are 10, so the third choice, “large,” is selected. The next nonterminal is **noun**. Again, there are four choices and the next bit pair to be hidden is 01, so “ladder” is selected. The choice for **tense** is “was” (directed by the next bit, 1), and the choice for **verb** is “waiting,” since the last bit is 0. The resulting sentence is “the large ladder was waiting.” It may raise eyebrows if read by a human, but can easily pass many computerized tests. The encoder can easily end each sentence (i.e., follow the choice of **verb**) with a period and start each sentence with a capital letter, thereby adding more realism to the artificial text that’s generated.

It is clear that a large CFG with many choices can hide many bits and may produce realistic-looking sentences. The CFG listed in [Wayner 02] is based on baseball terminology, has hundreds of productions with thousands of choices, and produces mostly meaningful sentences (although a baseball expert may find the generated text somewhat limited, repetitive, and incomplete).

The decoder starts with the definition of the nonterminal **Start**. The first symbol in this definition is the nonterminal **adjective**, which has two choices. The decoder uses the input (the word “the”) to decode a zero, since “the **size**” is the first choice. The nonterminal **size** has four choices, so the decoder uses the next input “large” to generate the two bits 10. This process is called *parsing* and is straightforward if the CFG has been carefully constructed. Two principles should be followed when constructing a CFG for mimicry; it has to be unambiguous and it should be in *Greibach normal form*.

A CFG is ambiguous if the same sentence can be generated by selecting productions in different orders. A simple example makes this clear.

Start → **name action** | **who does**

name → Alice | Bob

action → is here | was there

who → Alice is | Bob was

does → here | there

The sentence “Alice is here” can be generated by replacing the nonterminals **name action** with “Alice” and “is here” but also by replacing **who does** by “Alice is” and “here.” Obviously, such a CFG generates text that can be decoded in more than one way. This CFG is therefore useless for hiding data.

A CFG is in Greibach normal form (GNF) if the nonterminals are always the last choices in each option of a production. Thus, the production **something** → A **B** | C **D**

is in GNF but “**blah** → the **size** sum | a **size** bell” is not. It can, however, be modified to GNF by adding more productions as follows.

adjective → the **sizesum** | a **sizebell**
sizesum → tiny sum | small sum | large sum | big sum
sizebell → tiny bell | small bell | large bell | big bell

Using a CFG in GNF simplifies the parsing which is the main job of the decoder, as the following example shows.

Start → **noun verb**
noun → Alice | Bob
verb → sent mail **to** | sent email **to**
to → to **rel recipient**
rel → all | some
recipient → friends | relatives

To hide the binary string 01010, the encoder selects “Alice” for the first bit (0) and “sent email **to**” for the second bit (1). Nothing is hidden when the fourth production is applied (because there are no choices), but the encoder generates the terminal “to,” uses the production for **rel** to select “all” for the third bit (0), then uses the production for **recipient** to select “relatives” for the fourth bit (1). To hide the fifth bit, the encoder starts the next sentence. The reader is encouraged to decode the sentence “Alice sent email to all relatives” manually to see how easy it is for the decoder to identify the syntactic elements of a sentence and determine the hidden bits.

Those who have followed the examples so far will realize that this steganographic method is not very efficient. The last example has hidden just four bits in the sentence `Alice sent email to all relatives` that’s 33 characters long (including spaces). The hiding capacity of this example is $4/(33 \cdot 8) \approx 0.015$ hidden bits per each bit of text generated. The method can be made much more efficient if each production has many options on the right. A production with 2^n options can hide n bits. If an option is a 4-letter (i.e., 32-bit) word and there are $1024 = 2^{10}$ options in a production, then 10 bits can be hidden in each 32 bits of generated text, leading to a hiding capacity of $10/32 = 0.3125$ bits per bit (bpb).

In principle, it is possible to construct a CFG with a hiding ratio of 1 bpb, although in practice, the text generated by such a CFG may be less convincing than the examples shown in this section. Here is the main idea.

1. The CFG has 256 terminals, T_0 through T_{255} , each a single character (i.e., 8 bits).
2. There are n nonterminals N_0 through N_n .
3. There are 256 productions for each nonterminal N_i . They are of the form $N_i \rightarrow T_j N_{a_1} \dots N_{a_k}$ for $j = 0, 1, \dots, 255$. Each production replaces the nonterminal N_i by one terminal T_j and by k nonterminals (k has different values for the various productions). The total number of productions is $256n$.

For each nonterminal N_i there are 256 productions, so it takes 8 bits to select one production. The production selected adds one terminal (i.e., 8 bits) to the text being generated, which leads to a hiding capacity of 1 bpb.

This method has been implemented and tested extensively by its creator, Peter Wayner, who also published its source code in [Wayner 92 and 02] and prepared a large set of production rules for testing purposes. Compared to other steganographic algorithms, this method is well documented and tested.

Hiding in the alternating patterns of digits, deep inside
the transcendental number, was a perfect circle, its
form traced out by unities in a field of noughts.

—Carl Sagan, *Contact*



<http://www.springer.com/978-0-387-00311-5>

Data Privacy and Security

Salomon, D.

2003, XIV, 465 p. 45 illus., Hardcover

ISBN: 978-0-387-00311-5