# A Simple Yet Efficient Solution For Secured Data Exchange - Hiding Encrypted Text In An Image File

**Lawrence Wu[1] Penn Wu[2]**

[1] Diamond Bar High School
[2] Cypress College

## Abstract

Hiding encrypted text in an image file can possibly be a feasible solution to small-office-home-office (SOHO) and personal use for data and file transmission across digital networks. In this paper, the authors propose a simple, low-cost, but efficient algorithm to (a) convert a text (short or lengthy) to a byte array with a simple encryption mechanism, (b) append the byte array to an existing image file as hidden content without changing the original viewable image, and (c) retrieve and decode the encrypted text with a reverse algorithm and required keys. The authors also discuss the algorithm with Windows application created based on the proposed algorithm.

**Keywords**: Cipher in bitmap, enciphered text in bitmap, bitmap for enciphering

## 1. Problem Statement

Living in a highly transparent life, encrypting content of a file, or simply a string literal is often inevitable to all of us. In spite that there are many technologies available for encrypting data stored on end user devices as well as network-based data transmission, their algorithms are either too complicated for programmers without strong knowledge and skills in Cryptography to code or too difficult to implement for SOHO and personal use. Creating a simple but efficient solution for secured data exchange is a pressing issue that warrants attentions.

## 2. Objectives And Expected Outcomes

Results of a preliminary research have pointed to a direction -- it seems to be a feasible and optimal solution to store ciphertext or encrypted string as part of the content of a “Bitmap” file, and keep the “key” separately to decrypt at a later time. The authors thus initiated the first step to design an algorithm with an attempt to provide a simple, light-weighted, easy-to-implement but efficient solution of encryption.

The objective of the algorithm is to (a) read the bitmap content to an array (the “origin”) of the Byte type; (b) convert a plaintext (or a string) of any human language supported by Unicode to a UTF-32 encoded array (the “data”) of Byte type; (c) add a randomly selected integer from a large range (e.g. 0 to 100000) as a “key” to every element of the “data” array as a basic encryption to convert the plaintext to ciphertext; (d) apply other encryption algorithms presented by the authors to make the encryption mechanism more difficult to crack; (e) combine the “origin” and “data” arrays to make a new array (the “encrypted”); and finally (f) generate a new bitmap image file using the "encrypted" which should contain the bitmap contents and the ciphertext; however, only the bitmap content can be displayed to any graphic editing tools (e.g. Microsoft Paint, PhotoShop, etc.). The ciphertext is not visible to viewers without the use of a special decoding tool. In other words, the bitmap image looks the same on screen to general viewers, with and without ciphertext in it, as shown in Figure 1.

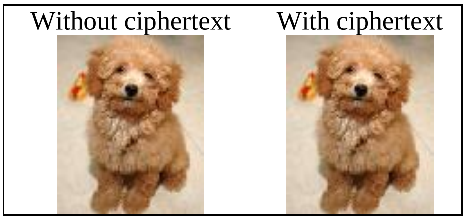


Fig. 1: Without vs. with ciphertext

The "key" may function as password(s) of the basic encryption; therefore, the text stored in the bitmap is not retrievable unless a matching "key" is given. This project also designed a reversed algorithm to retrieve the text or the entire file stored in the image file back to their origins.

The outcome of this project is a set of applications, written in Visual C++ and Visual C# with the .NET Framework, as demonstration tools of the proposed algorithm. The authors anticipate to advocate an feasible way to encrypt text or string literal in a bitmap file for SOHO and personal use, without the need of using symmetric encryption algorithms such as SHA-2, MD5, DES, TripleDES, AES, Rijndael, etc..

## 3. Significance

This paper distinguishes itself in the attempt to shorten the gap between an academic algorithm and its practical usage in transmitting text or a file within a digital network. The authors also attempt to provide an easy-to-use algorithm for programmers to build custom-make applications and optionally choose simple encryption method(s) from a list of available ones.

## 4. BITMAP And PNG

Many studies confirmed that the design of file structure of a bitmap allows room for stuffing plaintext as additional content to these required sections (Xochellis, 2006; Anonymous, 2013; Wikipedia, 2013). Therefore, it is feasible to hide text in a bitmap file with a good understanding of bitmap structure.

A bitmap file (BMP) contains an exact pixel by pixel mapping of an image used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk (MSDN, 2013; Microsoft, 2012; Lancaster, 2003). A bitmap graphic is made of pixels (short for "picture element") in a grid. Each pixel can be considered as a "cell" in a two-dimensional plane similar to the Figure 2 which, when zoomed in, is a capital letter "A."

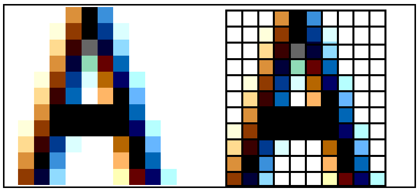


Fig. 2: Pixel

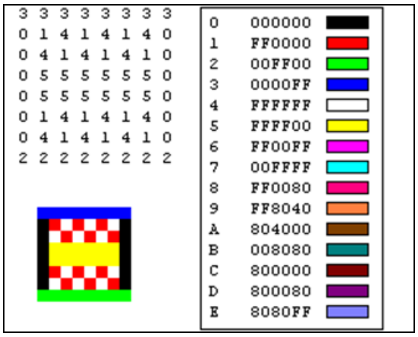
The structure of a bitmap image file contains three main sections: headers, color table, and other data (Pachghare, 2005). The rendering application reads these information and contracture the image on the display surface of an output device. Due to decades of evolution, content of a modern bitmap image file (e.g. Windows-based bitmap) may start with 54 bytes of headers in a predetermined sequence. The first 14 bytes are the "bitmap file header" for information describing the bitmap. The next 40 bytes are the "device independent bitmap (DIB) header" for information such as width, height, file size, and number of colors used. Green (2002) uses Table 1 to illustrate the file structure of a bitmap.

Table 1: Bitmap file structure

| Byte # | Information |
|---|---|
| 0 | Signature |
| 2 | File size |
| 18 | Width (number of columns) |
| 22 | Height (number of rows) |
| 28 | Bits/pixel |
| 46 | Number of colors used |
| 54 | Start of color table |

Source: Green (2002)

Next to the DIB header is the "color table" which maps numbers in the bitmap to specific colors. Microsoft (2012) uses Figure 3 to illustrate how the color table and its bitmap work in an image. Each pixel is represented by a 4-bit number, so there are 2^4 = 16 colors in the color table. Each color in the table is represented by a 24-bit number: 8 bits for red, 8 bits for green, and 8 bits for blue. The numbers are shown in hexadecimal (base 16).



```
3 3 3 3 3 3 3 3
0 1 4 1 4 1 4 0
0 4 1 4 1 4 1 0
0 5 5 5 5 5 5 0
0 5 5 5 5 5 5 0
0 1 4 1 4 1 4 0
0 4 1 4 1 4 1 0
2 2 2 2 2 2 2 2
```

| | | |
|---|---|---|
| 0 | 000000 | ⬛ |
| 1 | FF0000 | 🟥 |
| 2 | 00FF00 | 🟩 |
| 3 | 0000FF | 🟦 |
| 4 | FFFFFF | ⬜ |
| 5 | FFFF00 | 🟨 |
| 6 | FF00FF | 🟪 |
| 7 | 00FFFF | 🟦 |
| 8 | FF0080 | 🟥 |
| 9 | FF8040 | 🟧 |
| A | 804000 | 🟫 |
| B | 008080 | 🟦 |
| C | 800000 | 🟥 |
| D | 800080 | 🟪 |
| E | 8080FF | 🟦 |

Fig. 3: Color table and bitmap

The structure of a modern bitmap file may be further categorized as: (a) 14 bytes of bitmap file header, (b) DIB header, (c) extra bit masks, (d) color table, (e) Gap1, (f) pixel array, (g) Gap2, and (h) International Color Consortium (ICC) color profile. The "extra bit masks" defines the pixel format. Gap1 and Gap2 are for structure alignments. The ICC color profile defines the color profile for color management. The pixel array must begin at a memory address that is a multiple of 4 bytes.

PNG, short for Portable Network Graphics, is another bit-mapped graphics format with built-in compression. PNG supports fifteen color options. PNG supports palette-based images (with palettes of 24-bit RGB or 32-bit RGBA colors), grayscale images (with or without alpha channel), and full-color non-palette-based RGGA images (with or without alpha channel). Unlike BMP, PNG uses 48 bits. Images can be created using color palettes or 8 bit grayscale. Pixel data with 8 bit values can index into palettes containing up to 256 colors, and with fewer colors pixel values can be 1, 2 or 4 bits. The "True color" 24-bit format and the "True color with alpha transparency" 32-bit format are highly compatible to the palette-based BMP.

The main difference between BMP and PNG is the compression. BMP is both uncompressed and lossless while PNG is compressed but lossless. A "lossless" compression algorithm discards no information. It looks for more efficient ways to represent an image, while making no compromises in accuracy. Interestingly, text hidden inside a bitmap can be easily converted to a BMP-compatible PNG. The authors found both BMP and PNG as two optimal options of encryption vehicle.

## 5. Why Using Unicode?

Text of English characters can be converted to integer-based ASCII codes, and then stored in a Byte array. However, the major disadvantage of using ASCII is its inability to process non-English characters, such as Chinese characters, because ASCII characters are limited to the lowest 128 Unicode characters, from U+0000 to U+007F. It is the authors' intention to support as many as human languages as possible. The Unicode is a standard that assigns a unique number to every character of all written languages (e.g. Chinese, Korean, Farsi, etc.) recognized by the United Nation. This standard has been adopted by most operating systems, programming languages, application programming interfaces (APIs), and encoding platforms. Table 2 lists three sample ranges and their languages.

Table 2: Three language ranges

| Range | Language |
|---|---|
| 0000 ~ 007F | Basic Latin |
| 0590 ~ 05FF | Hebrew |
| 0E00 ~ 0EFF | Thai |

Commonly used Unicode standards include UTF-8, UTF-16, and UTF-32. "UTF" stands for Unicode Transformation Format. UTF-8 uses 1 byte (8 bits), UTF-16 uses 2 bytes (16 bits), and UTF-32 uses 4 bytes (32 bits) to encode every individual character. UTF-8 is backwards compatible with ASCII while UTF-16 and UTF-32 are totally incompatible with ASCII. Table 3 compares these three standards (Microsoft, 2010).

Table 3: Unicode standards

| Option | Description |
|---|---|
| UTF-8 | Encoding using 8-bit data sizes and works well with many existing operating systems. It is identical to ASCII encoding and allows a broader set of characters. |
| UTF-16 | Characters as sequences of 16-bit integers. It is used natively in Windows operating system and the .Net Framework. It is the most popular Unicode code points take only 2 bytes. |
| UTF-32 | Characters as sequences of 32-bit integers. It avoids the surrogate code point behavior of UTF-16 |

## 6. The Algorithm

The authors presented the algorithm to meet the previously described objectives. Figure 4 illustrates the concept that serves as the guiding principles through the development of algorithm.

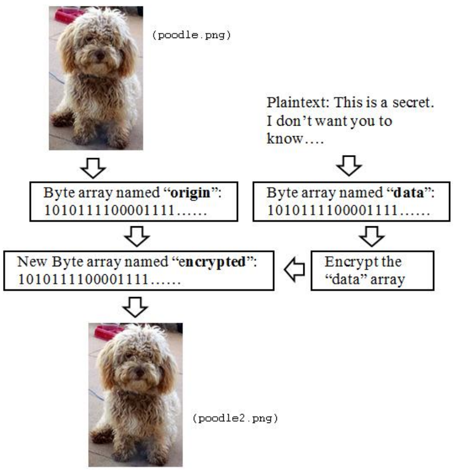

(poodle.png)
Plaintext: This is a secret. I don't want you to know….
Byte array named "origin": 1010111100001111……
Byte array named "data": 1010111100001111……
New Byte array named "encrypted": 1010111100001111……
Encrypt the "data" array
(poodle2.png)

Fig. 4: Algorithm

The algorithm starts with reading the content of a given bitmap (e.g. "poodle.bmp") and storing them in a byte array named "origin." It continues with converting a plaintext to Unicode equivalents with basic encryption mechanism presented by the authors, and storing the ciphertext in a byte array named "data." Then, it combines the "origin" and "data" arrays to another byte array named "encrypted," and converts the elements of the "encrypted" as bitmap content of a new image file (e.g. "poodle2.bmp").

### 6.1. Implementation

The authors hand-coded the demo application as implementation of the algorithm in Visual C++, and then converted the source code to Visual C# to avoid the known “” problem of Visual C++ applications. The authors chose Visual C++, during the development stage, for the following reasons:

- C++ is still a dominant general purpose programming language.
- Visual C++ is one of the .NET Framework languages; therefore, the authors can shorten the coding time by using namespaces, classes, properties, and methods provided by the .NET Framework to focus more on conception formation and algorithm development.

The following code illustrates how the authors use the File.ReadAllBytes() method of .Net Framework to open a bitmap file of a given path, reads the contents of the file into a Byte array, and then closes the file. The variable “size1” stores the number of elements the “origin” array has, which is also the number of bytes of the bitmap content.

```
array<Byte>^ origin =
File::ReadAllBytes("poodle.bmp");
int size1 = origin->Length;
```

The algorithm continues with taking a string-based text (named “userInput”) from a given source, such as a textbox, converting every character of the text to their UTF-32 equivalents, and then storing each UTF32-encoded value as an element of a Byte array named “data”. The variable “size2” stores the number of elements of the “data” array, which is also the number of characters of the text. During this step, the algorithm also creates the “newSize” variable to perform a calculation, `size1 + size2*4`, to determine the size of the “encrypted” array. It is necessary to note that each character of the “userInput” takes four bytes due to the UTF-32 encoding. Since every element in a Byte array holds only a byte (or 8 bits), it takes four consecutive elements of the “data” array to store one single UTF-32 encoded character; therefore, the total number of elements of “data” must be at least `size2*4`.

```
String^ userInput = textBox3->Text;
int size2 = userInput->Length;

array <Byte>^ data = Encod-
ing::UTF32->GetBytes(userInput);
/* convert user's data into UTF32
array */

int newSize = size1 + size2*4;
/*4 ia number of bytes per charac-
ter */

newSize++;
/* 1 extra byte to store a hidden
key */
```

The last line of the above code snippet increases the total elements of the “encrypted” array by 1 because the authors attempt to store a one-byte-long “hidden key” to the last element of the “encrypted” array. Before proceeding to the encryption, the following code declares and instantiates the “encrypted” array with the size defined by the “newSize” variable. It also copies all elements of the “origin” array to “encrypted” in a verbatim manner.

```
array<Byte>^ encrypted = gcnew ar-
ray<Byte>(newSize);

for (int i=0; i < size1; i++) { en-
crypted[i] = origin[i]; }
```

The “hidden key” is a random number from 0 to 255 which will be stored in the last element of the “encrypted” array. The following code illustrates how the algorithm generates the value of “hidden key” and store the actually value as the last byte of the “encrypted” array. It is necessary to note that anyone who analyzes the bitmap content may retrieve the value of this “hidden key” although it is not disclosed to the human user. The “hidden key” is an approach to improve the complexity, not a measure of encryption.

```
int hidden = rn->Next(1, 256);
/* hidden key 1~255 (because a
byte)*/
encrypted[newSize-1] = (Byte) hid-
den;
```

### 6.2. The Encryption Mechanism

The encryption mechanism requires one or more keys in addition to the “hidden key.” The authors denote them as “key1,” “key2,” and so on. While the “hidden key” has a fixed range (0 ~ 255), the range of “key1” is variable and adjustable. The following code snippet illustrates how the authors add a random number to the existing value of every element of the “data” array, and then stores the computed results as new elements of the “encrypted” array. In the following example code, the range of “key1” is set to be 0 to 99999; however, the range is adjustable to increase the complexity and difficulty. In the “for” loop, the initial value is set to be the value of “size1”; therefore, the addition only applies to the text being stored in the bitmap file. It does not apply to the original bitmap content.

```
Random^ rn = gcnew Random;

int key1 = rn->Next(100000);
/* generate a random number 0 ~
99999 */

int j=0;

for (int i=size1; i<newSize-1; i++)
{
 encrypted[i] = data[j] + key1;
/* add the randomly number to every
element*/

 j++;
}
```

The value of “key2” is another randomly picked integer from a variable and adjustable range (e.g. 0 ~ 9999 or 1000 ~ 99999). This is one of the options provided by the authors to increase the complexity of the encryption mechanism. There are other options the authors have presented. These options are furnished upon selection on a case-by-case basis to make the encryption mechanism more difficult to crack; however, details of these options are not discussed in this paper for the confidentiality.

All the keys when used, except the “hidden key,” are kept separately from the bitmap content by human users. The following illustrates how the authors write the elements of the “encrypted” array as content of a bitmap image file named “poodle.bmp.”

```
File::WriteAllBytes("poodle2.bmp",
encrypted);
```

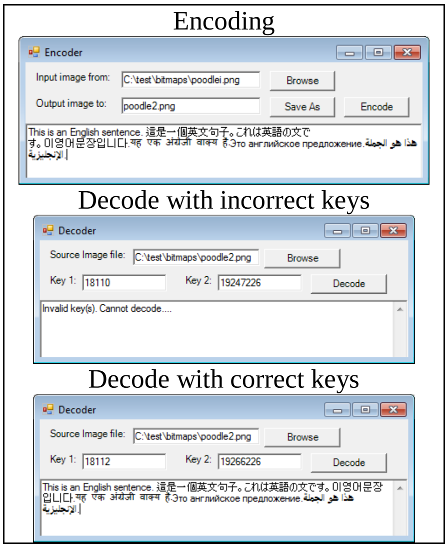
One problem is the number of bit a “Byte” type allows. Both “Byte” of C++ or “byte” of C# are defined by the System.Byte type of the .NET Framework type. They are unsigned 8-bit integers in the range of 0 to 255. In other words, a byte array cannot support any integer larger than 255. The authors chose to use the Encoding.UTF32.GetString() method to go around this limit (Microsoft, 2013). The following code illustrates the trick. UTF-32 is an encoding of Unicode in which each character is composed of 4 bytes.

```
String^ input = textBox1->Text;
int size = input->Length;

array <Byte>^ data = Encod-
ing::UTF32->GetBytes(test);

String^ str = Encoding::UTF32-
>GetString(data);
```

The decryption mechanism simply reverses the above described algorithm. It is necessary to note that the concepts and algorithms discussed in this paper apply to any general purpose language and are not specific to any programming language. Figure 5 demonstrates two Windows application, encode.exe and decode.exe, created based on the presented algorithm. The "encode.exe" stores (writes and encodes) a paragraph containing characters of English, Chinese, Japanese, Korean, Hindi, Russian, and Arabic characters within a PNG image file. The paragraph (plaintext) is also encrypted during the writing process. At least two keys are generated by the application. The "decode.ext" can only retrieve (read and decode) the paragraph by converting the "ciphertext" back to "plaintext" when the "decode.exe" is provided with a correct set of keys.



Input image from: C:\test\bitmaps\poodlei.png
Output image to: poodle2.png

This is an English sentence. 這是一個英文句子。これは英語の文です。이영어문장입니다.यह एक अंग्रेजी वाक्य है.Это английское предложение.هذا هو الجملة الإنجليزية.

Source Image file: C:\test\bitmaps\poodle2.png
Key 1: 18110 Key 2: 19247226

Invalid key(s). Cannot decode....

Source Image file: C:\test\bitmaps\poodle2.png
Key 1: 18112 Key 2: 19266226

This is an English sentence. 這是一個英文句子。これは英語の文です。이영어문장입니다.यह एक अंग्रेजी वाक्य है.Это английское предложение.هذا هو الجملة الإنجليزية.

Fig. 5: Encoder and Decoder Applications

### 6.3. Scalability of Encryption Mechanism

Even though the authors did not intend to adopt any existing symmetric, asymmetric or hash algorithms, such as those provided by the .NET Framework Cryptography Model, the algorithm has the scalability to work with these well-known encryption algorithms in two ways: built-in and external. Table 4 compares these two options.

## 7. Known Issues

All encryption algorithms can be cracked. It is just the matter of time. The presented encryption mechanism is still vulnerable to a "brute force" attack which basically tries every possible key combination until finding the one that decrypts. Furthermore, anyone with a good understanding of bitmap structure can manage to analyze the bitmap content and possibly shorten the time to crack the algorithm. However, with the options of encryption mechanism, the authors believe this vulnerability is not really applicable if the presented encryption is used for time-sensitive exchange of data over a network (e.g. within 24 hours). It would take significantly longer than days to crack the encryption.

Table 4: Options of Scalability

| Option | Details |
|---|---|
| Built-in | Using third-party encryption APIs (e.g. the .NET Framework Cryptography) as advanced options to encrypt text before using the presented encryption mechanism. |
| External | Use third-party encryption toolkit(s) to convert a plaintext to ciphertext, and then use the presented encryption mechanism to further encrypt the already-encrypted ciphertext. |

## 8. Future Development Plan

As of the time this paper is written, the authors have developed an algorithm to store an entire file of a variety of types in either a BMP or PNG file with basic encryption. The supported file types include binary (e.g. an .exe file), text (e.g. a .doc file), audio (e.g. an mp3 file), and video (e.g. an mp4 file). The authors are working on improving this algorithm with more reliable and trustworthy encryption mechanism. The authors also attempt to make this algorithm adaptive to third-party cryptograph APIs such as the .NET Framework Cryptography Model.

## 9. Conclusion

The proposed algorithm led to the development of demo applications which in turn confirms the algorithm could be a simple, low-cost, but efficient way to hide encrypted text in an image file. The authors believe this algorithm can possibly be a feasible solution to small-office-home-office (SOHO) and personal use for data and file transmission across digital networks. This algorithm is able to (a) convert a text (short or lengthy) to a byte array with a simple encryption mechanism, (b) append the byte array to an existing image file as hidden content without changing the original viewable image, and (c) retrieve and decode the encrypted text with a reverse algorithm and required keys. Although the quality of encryption algorithm mainly replies on how long it takes to break the encryption, the authors believe this vulnerability is not really applicable to this algorithm if the presented encryption is used for time-sensitive exchange of data over a network.

## 10. References

[1] Anonymous. (2013). *A Novel Use For Bitmap Files*. Retrieved on August 24, 2013 from http://www.simplecodeworks.com/New/tips/dataimages.html.

[2] Green, B. (2002). *Raster Data Tutorial*. Retrieved on August 24, 2013 from http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/raster.html.

[3] D. (Years Unknown). *Exploring the .BMP File Format*. Retrieved on September 28, 2013 from http://www.tinaja.com/glib/expbmp.pdf.

[4] Microsoft (2012). *Types of Bitmaps*. Retrieved on September 28, 2013 from http://msdn.microsoft.com/en-us/library/windows/desktop/ms536393(v=vs.85).aspx.

[5] Microsoft. (2010). *Using Unicode Encoding*. Retrieved on September 28, 2013 from http://msdn.microsoft.com/en-us/library/zs0350fy(v=vs.90).aspx.

[6] Microsoft. (2013). *Encoding.GetString Method (Byte[], Int32, Int32)*. Retrieved on August 24, 2013 from http://msdn.microsoft.com/en-us/library/05cts4c3(v=vs.110).aspx.

[7] MSDN. (2013). Bitmaps. Retrieved on September 28, 2013 from http://msdn.microsoft.com/en-us/library/windows/desktop/dd183377(v=vs.85).aspx.

[8] Pachghare, V. (2005). *Comprehensive Computer Graphics: Including C++*. New Delhi, India: Laxmi Publications.

[9] Wikipedia. (2013). *Cornelia*. Retrieved on September 28, 2013 from http://en.wikipedia.org/wiki/Cornelia.

[10] Xochellis, J. (2006). *A very simple solution for partial bitmap encryption*. Retrieved on September 28, 2013 from http://www.codeproject.com/Articles/13435/A-very-simple-solution-for-partial-bitmap-encrypti.