**Encryption/Decryption in Bitmaps**

Dylan Corriveau

February 4, 2014

**Scope**

This document is intended for anyone interested in learning about the development, implementation, and upkeep of a C# based encryption tool, using Bytes to save the information within the bitmap

This application can take an image, and when given an image, get a message from the user. We then take this message, and embed the message inside of the bitmap. This message can also be decrypted and displayed to the user

This document is valid until the technology has become outdated.

**Encoding/Decoding Algorithms**

In terms of encoding/decoding the data, a number of elements were used. The first part of the decoding was an entirely new protocol was developed to ensure validity of the information being passed. The protocol is described in much more detail [here](#protocolDesciption). In conjunction with the protocol, a hash algorithm was used to help mask the data. More information about the process of selecting the hash function is found [here](#SelectHash) in the footnotes.

In terms of the protocol used, it is pretty extensive in terms of the rules that must be adhered to.

The first step to describing this protocol would be to describe what the protocol actually does. It provides us with a means of hiding information within a bitmap, all the while making sure the information hidden is not visible (if not entirely, to a very small degree) to the naked eye.

The next step in describing this protocol would be to mention which OSI layer it falls in. This protocol would be found in the Presentation layer. This layer represents converting data from the application layer, and altering it so it is readable by the other layers.

After this, we would describe who can actually write the data. The only application that can write the data would be the ScanProgram V3.

Next, we describe who can read the data. With this section, there is a very special case. Any program capable of viewing images can read the data, but there are two types of readers. There are intentional readers (people who actually know about the data and know how to handle it), and unintentional readers (people who don’t know the data is there or don’t know how to access it).

The next step involves the actual structure of the packets. The packet sections are accurately described in this table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Start | Date | Name | Message | End |
| Two characters to signify the start of the packet. The characters used are always S- | This is the current date and time used when creating the packed. | This is the name who created this file (this data can be arbitrary, but must fall in the size constraint) | This is the message that the person wants to add (this data can be arbitrary, but must fall in the size constraint) | Two characters to signify the end of the packet. The characters used are always E\a |

The last step in describing the protocol before getting onto the error detection/correction (which will be described in the fool proofing section) would be to go over the specific rule sets found within the protocol. These rules are very strict in their design and implementation, so the data can be hid properly and reasonably. The rule set is a follows:

* The very first characters sent must be the length of the packet to follow
* The first two characters found must be the letter S followed by a –
* The last two characters found must be the letter E followed by the BEL (\a)
* The date section must be followed by the terminator(\0)
* The name section of the package must be 20 characters long, with the last character being the terminator(\0)
* The text section of the package must be 32 characters long, with the last character being the terminator(\0)
* If the name or the text section do not meet the length, they must be padded until they reach it (preferably with \n, but other values are accepted)
* If \a or \0 are found anywhere in the actual text or name, it must be changed to another value

The second last step would be to describe the error detection in place. There are a number of methods in place in order to check for errors. As specified by the multitude of rules above, if any are broken, the packet is considered invalid, and the data is not read or transmitted.

The final step would be to describe the error correction in place. In terms of correcting any errors on-the-fly, there is no specific technique in place. It simply identifies the packet as being invalid. Although this is a flaw in terms of usability, the amount of rules and encryption set in place is made so the data, if tempered with, would not be useable. So in this sense, the protocol is very good at sending secure data through bitmaps.

If anyone were to follow these steps properly and coherently, they would have no problem implementing the concept of embedding messages into the bitmap.

**Ensuring “foolproof” decoding**

In this kind of application, fool proofing is mandatory in order to verify integrity of the data. With a very specific data set in terms of which information goes where, it is very hard to actually start messing with the data to the point where it would not be readable. There are ways to check if the data has been altered, however. Those ways include the following.

* If the amount of data received does not specify the length as its first element, or if the number is wrong, then the data has either been added to or subtracted from, and would make the package invalid.
* If the first two characters aren’t S- or the last two characters aren’t E\a, the format of the package is invalid, and it would not be able to be read
* If there is not a \0 on the end of every data packet section(the date, the name, and the text), the format of the package has been damaged, or not followed, and the data is invalid.

Hopefully with these different parts of error detection in place, it will be much easier to keep secure information encoding within bitmaps. A specific example of a properly built package is found [here](#BuldingString). In terms of implementation, the concept of “Noise bits” had to be taken advantage of.

**How “noise bits” were taken advantage of**

In terms of embedding bytes into a bitmap, the concept of “noise bits” had to be taken advantage of. This “noise bits” concept is also called stenography. Stenography is the art of “hiding in plain sight,” and is the exact art we are trying to recreate/implement. In order to do this for a bitmap, we would need to hide the data in a place where the user wouldn’t expect; where they cannot see it. In a bitmap, this concept involves changing the lower bits of a color in order to do necessary storage/implementation/etc. As an example, the bits in the RGB value 0, 0, 0 are identical (to the untrained eye) to 0, 0, 1, or 0, 1, 0. With this in mind, if you change values within the ARGB value, it is possible to use the space to actually store information! In the case of this program, we stored the message values inside of the inner bitmap values, and alternating through the different RGB values for each. Since the RGB values are found in every bitmap, it is imperative that this information is not touched in an image damaging way. The Algorithm consisted of changing the value found within the differing R, G, or B values to what was needed, or the specific value needed. A snippet showing where I write the value, followed by reading the value is found [here](#ReadingWriting).

**Algorithm Code Samples for:**

1. **Building Encrypted String**

The code to generate the encrypted string is found here:

****

1. **Reading/Writing the Encrypted Value**

The format used to write the value in place of the alpha value is as follows:

****

1. **Hash Algorithm Implementation**

The format used for encryption of the message is found here:



**Conclusion**

In conclusion, this assignment taught me a little bit in bitmap manipulation, but it actually taught me more in terms of encryption techniques. The various cryptographic algorithms I had found had actually taught me about encryption. Even though I had tried a bunch of differing methods in terms of encrypting the data, it not only exposed me to these differing areas, but also when appropriate situations would call for these algorithms, and how to implement them. I also found that this project would have been done much better in C++. This is because in C++, it is much easier to access the color information (or the “noise”) within the bitmap. In C# it is much harder, but through the other assignments, it provides a good basis for this assignment so implementation is much simpler. This assignment has provided a good lead in (for me anyways) into the realm of cryptography, and actually makes me want to read/implement more in terms of cryptographic methods. It also taught me about the realm of stenography, and how it can help in situations. In the end, bitmaps, when done correctly, can not only be a storage of image information, but also any other form of information that can be contained in the byte region (such as cryptographic messages, or format codes).

**Footnotes**

1. **Selection of the Hashing Algorithm**

While selecting the algorithm, there were a number of algorithms that were attempted. The first attempt was to use the Rijandael algorithm (also known as AES). This algorithm supersedes the algorithm known as DES, and it is very similar to the algorithm being used currently (it uses a common key to encrypt and decrypt the data), but the initial implementations proved that the Rijandael method fell on one simple fact. The Rijandael method uses randomly generated value, which would cause the program to fall down if a user was to encrypt the message, and decrypt the message at different times. The next method considered was MD5. MD5 is a commonly used cryptographic has function that produces a 128 bit (sometimes 160 bit) hash value. It is nearly impossible to have duplicate keys from the hash function (even if the files differ by a byte), as well as repeatable (that meaning every time a hash is done on one file, it is the same). There is one major falling point when implementation took place; the values are non-reversible. If you hash a value, you can’t get the value back from it (very easily). This makes it impossible for this kind of application to be used. The final method chosen and implemented was Triple DES. Triple DES is a variation on the DES algorithm. In fact, it uses a system involving 3 DES keys put together to make the Triple DES key (hence the name). It was implemented in the following fashion for [Encryption](#EncryptionAlgorithm) and Decryption. This algorithm was chosen due to not only being applicable to the application (being able to generate hash values that are reversible with the same key), but also in a secure manner. With this in mind however, it does not mean that fool proofing cannot occur in the decoding process.  **References**

AES encryption using c#. (n.d.). *MSDN â€“ Explore Windows, Web, Cloud, and Windows Phone Software Development*. Retrieved March 6, 2012, from <http://social.msdn.microsoft.com/forums/en-US/vssmartdevicesvbcs/thread/8d92d9f6-a84b-4b2b-aaa1-f1da053d5720/>

.NET Encryption Simplified - CodeProjectÂ®. (n.d.). *CodeProjectÂ® - Your Development Resource*. Retrieved March 6, 2012, from <http://www.codeproject.com/Articles/10154/NET-Encryption-Simplified>

Slama, J. (n.d.). InputBox [C#]. *C# Examples*. Retrieved March 6, 2012, from <http://www.csharp-examples.net/inputbox/>

What is the MD5 hash?. (n.d.). *MD5 Checksum Checker for Windows*. Retrieved March 6, 2012, from <http://www.fastsum.com/support/md5-checksum-utility-faq/md5-hash.php>

tes. (n.d.). Encrypt/Decrypt String using DES in C# - CodeProjectÂ®. *CodeProjectÂ® - Your Development Resource*. Retrieved March 6, 2012, from <http://www.codeproject.com/Articles/19538/Encrypt-Decrypt-String-using-DES-in-C>

binary universe : Steganography - Page 1. (n.d.). *binary universe : Steganography in C# by Corinna John.* Retrieved March 6, 2012, from <http://www.binary-universe.de/index.php?article=1&language=e>

*APA formatting by BibMe.org*