COSC 4P98 Final Project
May 7, 2012

Steganography Application

Stu Gritter
sg03tn

Table of Contents

Author's Comments

I was happy with this project. It was quite interesting, and functioned as well as I expected it to. Although I didn't specialize the program, it shows clearly how we can easily transfer information using a virtual sub-channel. I also can't believe I cited a Facebook page, but it actually is the original posting, and it's valuable to see.

In retrospect, I – of course – wish I had started sooner. Then I could've put a couple more bells and whistles on, and it would have felt less like a prototype.

I enjoyed it, learned a bit… it was good.

I am living locally so if a demo is required please feel free to send an email my way at either sg03tn@brocku.ca or sgritter@gmail.com.

Cheers,
Stu

Data Hider

This small data hiding application takes a 32-bit wave file and buries information in it. This information can be of any sort, although in the program's current state it is best suited toward textual information. The program is able to encode a 32-bit wave file in order to also contain an extractable 16-bit 1-channel wave, or an extremely large text message. Modifications made to the program could easily scale this ability to handle larger data, as will be discussed later.

What is its Shtick?

This system manipulates the fact that 32-bit floating point sample representation can easily be modified without doing any substantial or audible damage to the original wave file. The data representation is – in all honesty – a little bit unusual. Maintaining the value of a decimal for a data type which can range from negative to positive tens of thousands seems to be fruitless. This program effectively hijacks these decimal values and uses the space of this data for another purpose.

This system currently only utilizes one channel of the 32-bit wave – it could be restructured to properly use every channel reliably. This would improve the efficiency of the primary-to-sub-channel data ratio, allowing a wave file of equal time length to be hidden, rather than the current approximation of five seconds of parent data to one second of hidden data. The current structure was designed due to its relative simplicity, and was never changed as it still offers itself as a proof of concept.

But We Want to Hide Text

Examining the system, it obviously is not best designed to manage inserting wave files into other wave files – it is an unusual request, and is handled largely as a side-effect capability. The primary feature is the system's capability of hiding text based information. Given a wave file with 41000hz sampling rate, five seconds gives us 205000 samples, and a typical five minute pop song would give us 12,300,000 samples. If we wanted to smuggle a 10,000 word paper averaging 8 letter words, we need to hide approximately 90,000 characters (including spaces). This means for every sample we use, there will be 135 samples untouched. We can use this space to better hide our information with clever utilization of a three integer key.

The first digit of the key will indicate which sample is the first to contain data. This first data sample holds the length of the message. Without this first number, then, a snoop would not know the length of the message hidden within the sound file.

The second digit of the key indicates the number of unused samples between data samples – no sense in putting all the ducks in a row. This still leave us with an unusual problem, however. If we are manipulating only 1/136 samples there is still a worry that these samples could be isolated by evidence of their being tampered with, effectively removing our numeric data from the shelter and safety of the wave file's surrounding, natural samples. In order to prevent this, the system offers the capability of tampering with every sample in the wave – this means a random ASCII character will be generated and inserted into any sample not being used by the original message. Now, digital evidence of tampering will be "visible" in every sample, making it seem either natural, or at least disguising which samples are used in the actual message.

The third digit of the key is a simple offset for each of our data characters. Using the bounds of standard ASCII characters, we perform a simple shift on the data we wish to hide, and need to know the proper shift in order to decipher the message properly.

Resulting Security?

All of these features combine into an absolute chaos of characters should a snoop decide to try to decipher the message. A snoop would find literally thousands (or hundreds of thousands) of dummy characters on top of not knowing how many characters belong in the final message. As a kicker, the characters of the message are not even stored in this mash of data. The three digit key, then, is required to make any sense of the data.

Ha! Now I Know I Need Three Key Numbers!

Correct. And they can have millions of permutations. In our previous example we are limited by the standard ASCII table up to 127 as one number, and our character spacing is limited to 100 in order to provide complete dead-space over one quarter of the wave file, leaving our starting sample to range very easily into the area of 1,000, resulting in 12,700,000 combinations. Applying 12,700,000 combinations of keys to 12,300,000 samples would take a disastrously long time to brute force. Given the haste with which a music album can be transferred digitally, this allows for vast amounts of information to change hands on an invisible sub-channel. The Harry Potter audio books could very well house schematics to a nuclear power plant, and with the use of a third party text scrambling application, it would take months of computing time to crack, but mere minutes to unlock.

Practicality & Discussion

There certainly are other ways to transfer information. Reliably and even secretly, perhaps. But this way is fun. And it shows more a proof of concept that this kind of steganography can be used for other means. One great example is how the developers of the video game “Monaco” have shown their level data is easily encapsulated and hidden within an image file. Salting and hashing data, or storing it behind password protection is all well and good, but we all know nothing is entirely hack-proof. Further encoding secure or private information creatively is one of the best methods of security we will ever have.

Examples

The program has come with one file you can easily put messages into, and two examples of output with data in them already. “demo-text-863-99-4” can be deciphered using the key values 863, 99, and 4; it contains a quick sample of text. “demo-wave-1-5-0” can be deciphered using the key values 1, 5, and 0. The resulting message is a 16-bit version of the first second of the 32-bit wave file (which can be verified against “GetUsedToItSNAP”).

Moving Forward

This application is not as efficient as it could potentially be, nor does it support as many file types as it could. If one decided to continue with this project, the following are recommended avenues worth exploring:

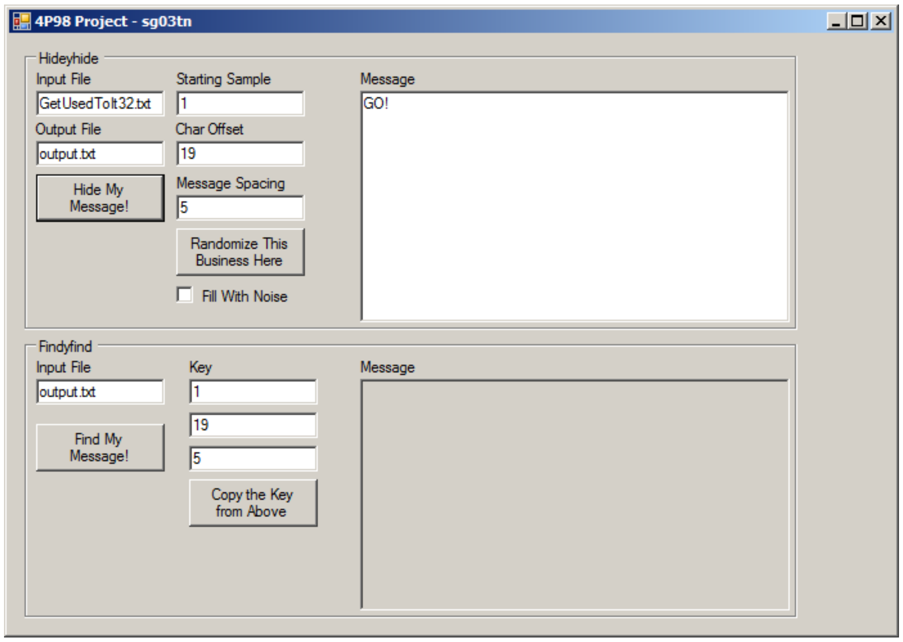
- Modifying the code to make the application hide information in each available channel
- Modifying the algorithm in order to hide data in a 16-bit wave file (utilizing the last character of each sample)
- Creating one or two more algorithm keys to further obscure our contained data
- Creating a “smart” key randomization which would examine the number of available samples in the primary file and the length of the hidden message

With a relatively small amount of work, one could fashion this application to hide a message in a 16-bit wave file, and hide that wave file within a 32-bit wave file, each with its own key bearing trillions of plausible combinations.

The pages following contain a user manual and a works cited page.

Data Hider Manual

So you have something to hide, eh? Well, look no further. This is the interface you will be using:



Hideyhide

Input File: GetUsedToIt32.txt
Output File: output.txt
Starting Sample: 1
Char Offset: 19
Message Spacing: 5
Fill With Noise: [ ]
Message: GO!

Findyfind

Input File: output.txt
Key: 1
19
5
Message:

We'll deal with the top half (marked "Hideyhide") first.

Input File is the initial, raw parent wave file you wish to have carry your message.

Output File will be the file created – audibly the same as your input file, but with a hidden agenda.

Starting Sample, **Char Offset**, and **Message Spacing** are the three key values. You set can these individually in order to help shape your message as you wish, or simply click the **Randomize This Business Here** button to generate random values for each of them.

If the **Fill With Noise** checkbox is checked, any samples not actually used to carry your message will still be given a random ASCII character in order to help mask your message.

The **Message** textbox is where you type your message, and is copy/paste friendly.

Click **Hide My Message!** to start the process and write your file.

Once you have created your file, it can be decoded using the bottom half of the interface (marked "Findyfind").

Input File is the file you have which contains the message you want to decipher.

The three **Key** values must be the same as the three values located on the "Hideyhide" panel. For onsite testing purposes, you are also given a **Copy the Key from Above** button, but if the program was used in two different locations, each of the values would have to be delivered some other way and entered manually.

Then the **Find My Message!** button can be clicked, and the hidden message will be displayed in the **Message** textbox. This is read-only, but is copy friendly.

Works Cited

Monaco, “Steganography – The Secret Data behind the Level Images”, October 4, 2010. https://www.facebook.com/note.php?note_id=437497056995