Enhancing a Steganography Program With Audio and Video

Scott Cameron
camerons46@mymacewan.ca
Patrick Martel
martelp4@mymacewan.ca
MacEwan University
Computer Science Department

Abstract
The need for increased privacy is always present in the modern age of the internet. Steganography is an effective method not only to hide a message being sent, but also hiding the fact that a message is being sent at all. There are plenty of individual projects on image based steganography, and less so for audio and video steganography. However, there is a lack of hubs that hold a collection of steganography methods for use in different data mediums and file formats. We decided to expand upon the open source OpenStego project to allow for data hiding in audio and video files. In this paper we discuss the implementation of our expansion of OpenStego to work with WAV and MP3 audio files as well as various video formats.

Keywords
Steganography, Audio, Video, LSB, OpenStego, Open Source

I. Introduction

The source code and materials mentioned throughout this report can be found at https://github.com/Camerons46/openstego_capstone

With the advent of the internet people became increasingly connected and were able to share ideas with each other across the world. However, you may not want to share everything you post online to the world. There's several methods to do this but one way involves sending a message to one or more people in such a way where no one but the intended recipients can tell there was a message sent at all. This method is called steganography, and involves hiding some sort of message inside a file called a cover file and distributing that resulting file called a stego file. Then only the intended recipients will know that there's a hidden message and know how to extract the message. One of the more popular open source steganography programs is OpenStego, a project by Samir Vaidya that's currently capable of hiding and extracting messages inside images. OpenStego is made with a plugin based nature to easily add modules to handle different methods of steganography. We thought OpenStego would be the best fit for our project to create an audio and video steganography program. A common steganography method that's commonly used, including by is LSB, or Least Significant Bit. LSB consists of taking each discrete data point from the cover file and modifying the bit that produces the smallest impact on the result to be a bit from your message. Move on to the next data point and repeat until you have modified the same number of bits as your original message. Ideally the resulting file will look indistinguishable from the original while still holding your secret message inside.

A. Problem

This brings us to OpenStego and steganography in general. The majority of steganography software available is only capable of handling images, if not they're usually limited to one or two media formats. We didn't want to contribute to this by making a brand new steganography program for handling the formats we wanted so we decided to expand upon pre-existing software to create sort of a "hub" of steganography. Another issue is because images are the most common host of steganography data, it is also the most likely to be scrutinized. We thought less used steganography formats would be far less scrutinized which is a massive benefit over traditional image steganography. Our goal with this project was to produce a program that handles images, audio, and video formats so the users need only one program to hide their data. By adding in the functionality to handle audio and video into OpenStego, we would improve the versatility of the program and provide a better user experience and those hiding and retrieving the messages only need one program to exchange information.

B. Use Case and Threat

A possible use case for steganography, OpenStego in particular, that would benefit from the addition of audio and video files would be a blog being used to pass secret messages. Why would a person send a secret message? We've all seen the rising internet censorship practiced by several countries. There's also an ever growing want for increased privacy among internet users as corporations gather people's personal data for profit. This software, with its expanded functionality, will further allow people to bypass unethical censorship and give them another tool for private communication. While we can all hope that events won't escalate to the point where we need to hide our true communications this is a very real possibility moving into the future. Primarily, by uploading to a website that the user controls such as a self-hosted blog, they can have audio or video formats of any type they wish and do not have to worry about unexpected compression or formatting changes upon upload. This makes it unnecessary to account for every video or audio format and allows the ability to focus our time on robustly developing steganography methods for a few formats. Additionally, this blog would appear innocuous sharing images, songs, podcasts, and videos of a person's day to day life or a hobby such as bird watching or food preparation. This creates a level of security and flexibility in the techniques used to hide the files. The blog is unlikely to garner a wide audience if it doesn't cover a widely interesting topic lowering the bar for how likely the uploaded content will be scrutinized. This will naturally lead to fewer technically minded individuals who have the knowledge to spot and identify modified files finding the content in the first place. This allows for less advanced hiding techniques required as the intersection of people interested in your blog about blue jays and chickadees in Edmonton and people who know about steganography is likely very low. Furthermore, a homemade sound clip, video, or pictures can have issues with the sound or picture quality that do not stand out as much due to their unprofessional nature. Issues with sound and video are common in homemade media filmed on a smartphone and this provides flexibility as issues in the audio are harder to detect when the exact quality of the initial media is expected to be poor. Of course, a certain level of data concealment is still required considering that we must always assume the uploaded content is being scrutinized.

The potential threats in the proposed use case are any parties invested in the information contained in the hidden messages. This may be governments who may use automated algorithms that search through websites analyzing audio and video content. The internet is too vast for any group to manually analyze content en masse. There's also the more tech-savvy passersby who may investigate and report the blog owner, or attempt to extract the message and sell it to whoever may be interested. Some governments are censoring the internet significantly more than others, but luckily it seems to almost always be based around messages on the open internet that get attention. There doesn't seem to be significant resources dedicated to censoring every possible message they don't like no matter how hidden. At the very least there's insignificant crackdown on these sort of hidden messages that we've found. The use case itself presents a scenario that is meant to allay suspicion by making the media publicly available and apparently innocent. For the average user there should be no indication that any secret messages and covert communications are contained in the blog. To our knowledge current software that scans website content is aimed at finding copyrighted material such as music or movies making the likelihood of the audio and video being closely examined for hidden messages exceptionally low. That being said, better security and concealment is always a plus and will reduce risk so an effort should be made to make sure the data is well hidden.

C. Preliminary Research

We looked at several open source steganography projects and published papers to get a good grasp on what's been done, the complexity and time required for each technique, and what we should use as inspiration for our implementation. First we found a relatively simple project called *Enshroud* by vibhusehra on GitHub which embeds and extracts data in uncompressed WAV data using the LSB method with every byte. Unfortunately the results were not stellar and had an odd hissing noise in the resulting stego file. We found there was just one simple flaw in their implementation to do with embedding the message in every single byte sequentially. WAV files don't necessarily store sound in discrete one byte chunks, instead the number of bytes per data point can change depending on the audio file. With this in mind we decided to use Enshroud as an inspiration for our first endeavour into steganography with our own WAV steganography implementation.

We looked into other audio steganography methods with different file types because we were unsure if we would be satisfied with just a WAV implementation. We looked towards MP3, the most popular audio file type. The resource that gave us the most insight into MP3 steganography and what was possible in our time frame was a recording of a talk done at the Illinois Institute of Technology titled *Mp3 Steganography - Presented at Forensecure: Cyber Forensics & Security Conference 2016* by Gentiana Desipojci, Adrijan Seferi, and Michael Theis.

For a straightforward method of video steganography we looked at *Video-Steganography* by Amritaryal44 on GitHub which specializes in using small videos as the message data and embedding them into larger videos using a fairly basic LSB method to insert the message data into the 2 least significant bits in the uncompressed video cover file.

II. Audio Steganography

We developed methods for embedding messages in audio files for OpenStego. We took advantage of existing functions in the program to compress and encrypt the message file we want to hide and then call our methods to take care of the hiding procedures. With our limited time we decided to focus on WAV and MP3 files as we wanted to hide data in compressed and uncompressed audio formats for the additional flexibility it provides. We succeeded in the implementation of both. However, there are some limitations found within the methods.

WAV files are handled relatively simply as their uncompressed format makes them easier to work with. The bits from the message are hidden in the least significant bit of every sample in the WAV file. A sample is a single data point and a WAV file uses many samples to represent a sound wave in digital form. The amount of data that represents each sample can vary but the most common is 2 bytes per sample. Our plugin automatically detects the bytes used for each sample and always uses the least significant bit of the sample no matter the bytes per sample. Even modifying each bit in every 2 byte sample, we noticed no change in the resulting audio quality. That being said, we still spread the changed bits as much as possible to minimize potential artifacts. We found that modifications could be obvious when looking at the file with a hex editor and the beginning of the file often starts with several 0 value bytes so we set the minimum threshold that the byte could be modified to 2. A threshold of 1 cannot be used as a modification to the least significant bit can turn it into a 0, putting it below the threshold and corrupting the extracted data. If our plugin finds that there's not enough bytes above this threshold then it will set the threshold to 0 and inform the user that the message could only be inserted at the cost of higher risk of hidden data discovery. Additionally, the modified bytes are spaced out using a random number generator to decide how many bytes should be between the previously modified byte and the next. This should make it substantially more difficult to detect that there is a hidden message in the file when analyzing the data. The encryption password is used as the RNG seed to ensure the same bytes are chosen on embedding and extracting the message. We then proceed through the cover file and hide every bit from the secret message. This method is good in that it is capable of hiding relatively large messages in the cover file. This provides more flexibility for what can be hidden within the cover file. Moreover, the method used to conceal the message results in a fairly scrambled and spread out distribution of bits. This will make it harder for anyone who notices irregularities in the file to find the actual message and decode it. There are few drawbacks with this method but one would be the embedding of the data affects the actual audio data of the audio file very slightly distorting it. Although the changes are so little that we believe it to be practically impossible for anyone to discern the difference, it is always a risk and a possibility that it can be noticed.

MP3 files are more complicated due to their compressed nature. They are composed of many data frames that can vary in length and quality within the file. This creates a problem for hiding data as small bit changes within the data of the frame can cause noticeable audio distortions and hiding messages of any meaningful length heavily degrades the sound quality. To this end we looked into the header of the frames. Each frame has a header that contains information about the data that follows. More importantly the header contains 3 bits that do not affect the playback of audio namely the copyright, private, and original bits. As such those bits can be modified without impacting the audio at all. Thus, to even an extremely sensitive listener, the hidden message cannot be detected at all. The drawback of this method is twofold. First, a deeper examination of a MP3's headers will show that the bits do not match frame to frame, some will show the original bit set in the frame and some will not. This could lead to suspicion and further examination of the file as it generally isn't common for an audio file to be different in copyright and originality in different portions of the same file. The second major drawback is the limited size of the message that can be hidden with this method as we are using a handful of bits per thousands of bytes. However, within the time frame we had to complete the project this seems an adequate method of hiding the message and will hopefully inspire further improvement in the algorithm after the completion of this project. We found the quality of the MP3 had an impact on the number of bits we could hide. With higher quality files there would be more audio data per frame meaning that more of the total file size is made up of audio data as opposed to frame headers. Likewise lower quality files had more frames relative to their file size. Therefore lower bitrate audio allows for a higher density of message data able to be embedded.

A. Future Improvements in WAV Steganography

We had quite limited time to research and implement our solution and we have some ideas that would have enhanced our data hiding with WAV files. First is choosing the best locations to insert our bits such that the average user would have the least chance of noticing an oddity in the audio. We have conceptualized a method of doing this that requires a bit of background. The human ear interprets sound volume logarithmically. In essence, this means that a small change in volume when the sound is quiet is significantly more noticeable than a small change in volume when the sound is loud. To get the volume from our WAV file we must split the file into many small chunks. It is important to note that a single data point in a WAV file cannot have a volume; a chunk of data must at least be one full wavelength as the volume is the amplitude of the wave. Once we have a chunk, we can calculate the volume with the decibel formula:

$\text{db} = 10\log_{10}(\text{average amplitude} / 32768)$

Once this is calculated for every chunk, you can pick the chunks with the highest values to store your bits in. As for which bit to change within a chunk, you have the choice of choosing the highest and lowest bits, or the bits in between. Changing the highest and lowest bits will very slightly change the amplitude and therefore the volume. Changing the in between bits will very slightly change how it sounds. We were unable to reach testing for this part, so we do not know which is superior. We believe changing the bits that affect amplitude might be best for the logarithmic hearing reasons above.

We have also conceptualized a similarity check for comparing the original audio file to the newly created steganography file. Our method consists of splitting the files into chunks and checking the loudness as described previously. Once you have split both files in the same chunks as each other, get the loudness of both chunks in decibels and take the absolute value of both values subtracted from each other. This will give you how different in loudness the chunks are as a positive value. Because of the logarithmic nature of the decibel scale, the same change will produce a smaller difference when in a louder portion of the audio file as opposed to a quieter portion. This is just like the volume scale of the human ear. Because of this, getting the absolute difference of loudness produces the result of how much a human will be able to detect the change. Summing this resulting value for every chunk in the audio file will produce a score of how different the produced steganography file is from the original. Testing would need to be done to determine what is a "good" and "bad" score.

III. Video Steganography

When starting our research on video steganography we first looked into common ways video is stored and shared over the internet. We started research into the mp4 file format as it seemed like the most common format for videos. It turns out that mp4 is simply a container that's capable of holding various forms of encoded video. We learned the basics of how video is encoded using lossy compression to drastically save space and found the 2 most universal video codecs are H.264 and H.265. Looking into how we could insert a message into video we settled on 3 possible approaches. Our first idea was to find bits in an already encoded video that would have little impact if changed, thus allowing us to hide a message. Alas, the data stream for both H.264 and H.265 proved to be too complex for us to completely understand and exploit for steganography purposes in our time frame. The second approach was the idea of embedding the message during the encoding process. This was inspired by methods we found were very promising when researching mp3 steganography but required a massive effort in rewriting the encoder to hide imperceptible messages in audio. We thought perhaps there may be a simple way to do this with the video encoder, but after finding no previous research on this method we decided it was not feasible. Finally, we thought of methods of hiding a message in a video before running it through the encoding process and have the message still be retrievable. Uncompressed video is simply a series of images, and the data stream consists of the values needed to represent each pixel in such a way that an LSB method could be applied to it. Afterwards, a form of lossless compression would need to be applied to make sure the message stays intact. It turns out H.264 and H.265 both support true lossless encoding. We decided this is the method we shall use for our video steganography implementation. We needed a video encoding program which had a license that allowed us to run it in our program. We found FFmpeg which is a suite of libraries and programs for handling and converting all popular video formats and codecs. We call FFmpeg from our program to handle the deconstruction of a video into a raw file and the encoding into a lossless video file.

This has some drawbacks and benefits. The density it is able to hold message data is very high at 1 bit for every byte of uncompressed data. Depending on the compression ratio of the lossless encoded video we found that the maximum amount of message data able to be embedded could vary from 25% of the size of the lossless cover video to well over 100%. It's important to note that the size of the lossless compressed file could be significantly larger than the lossy original.

We found that the file size of a video with an embedded message was often larger than the lossless encoded video cover file. It turns out that changing the least significant bit of some pixels in a pseudorandom way caused the compression ratio to be a bit worse. The size differential seemed to vary in proportion with the size of the message being hidden and the compressibility of that message. Different video cover files used also had an impact on this size differential. We tested several videos with several different messages with varying file sizes and measured odd results including some messages having far less impact than others and inconsistent changes in resulting file sizes with different message sizes. We found our testing methodology was flawed because we were using static message sizes over vastly different video sizes and OpenStego automatically compressed each message so we changed our approach. We turned off OpenStego's compression and generated 10 message files for each video, each containing random bytes and set to a specific fraction of the max possible message data each video was capable of holding. We chose the messages to be in orders of 10% of the holding capacity. After seeing promising preliminary results we created four methods for spreading the message data across the cover files to see if that had any effect on output message size. To ensure consistency we set each test video to a resolution of 640x360 with lossless H.264 encoding. The exact test files used along with the video source locations are on the capstone's github page under the folder videoCoverExamples

These are the 4 embedding methods we tested:

- Frontloaded: Embedding the message in the data sequentially from the beginning. This confined all of the data to only the earliest frames in the video and none in the later frames if possible. We thought maybe confining the changes to as few frames as possible would allow for a better compression ratio.

- Random: Embedding the message with random spacing between each bit embedded. The same algorithm as our WAV embedding method was used. It's important to note that at 60% message saturation and above there was no room to randomly spread the data so the embedding method became effectively the same as frontloaded.

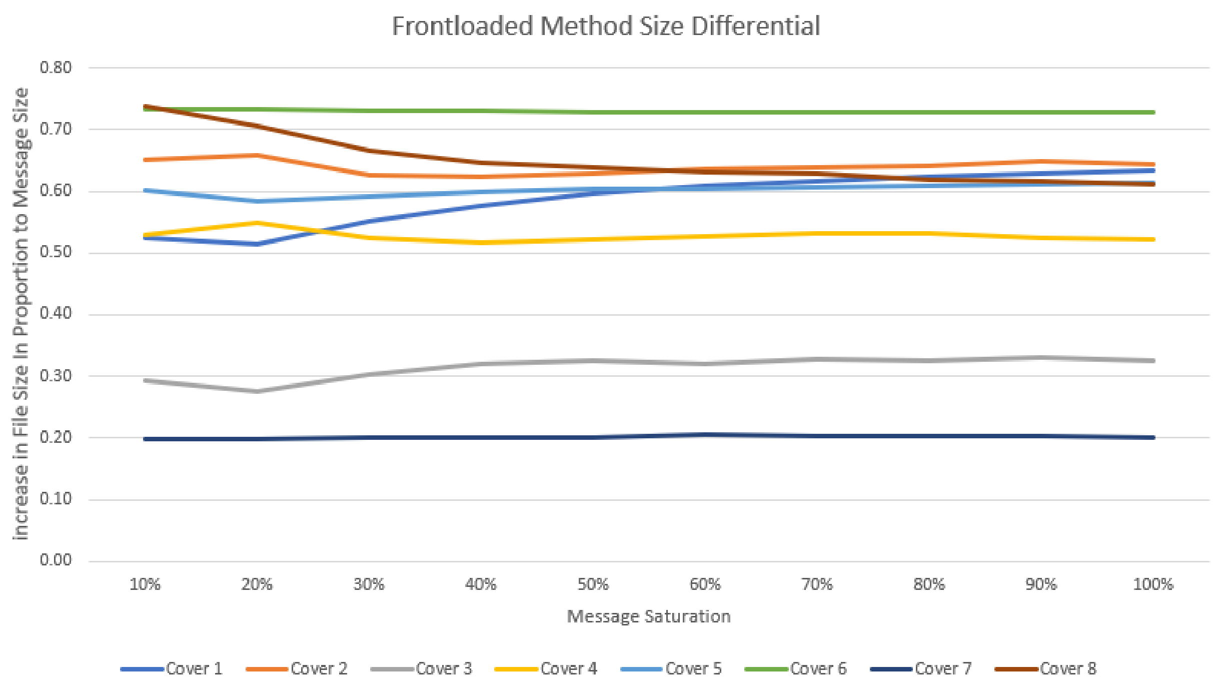
- Evenly Spread: Embedding the message with a static number of bytes between each bit embedded. For the same reasons as random, at 60% message saturation and above there was no room to spread the data and it became effectively the same as frontloaded.

- 1st Pixel in Frame: Embedding the message in the bytes the make up the first (top left) pixel in each video frame and once every frame is used, embed the message in the second pixel, then third, and so on. This was to test if confining the changes in the same pixel allowed for a better compression ratio.

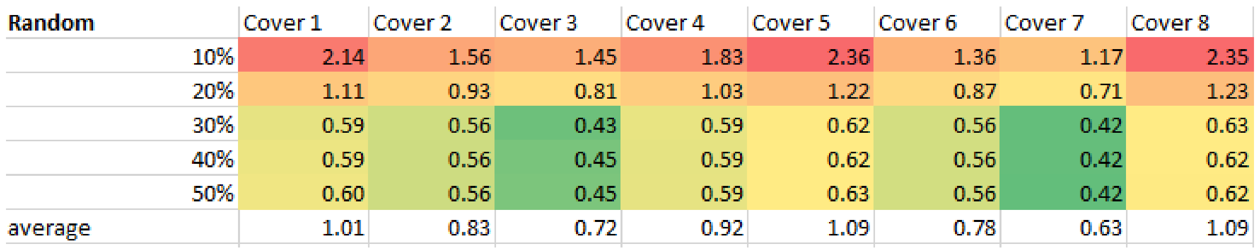
The results are formatted as an increase in file size after embedding in multiples of the message size. Therefore a value of 0.5 means the video increased by ½ of the embedded message size and a value of 2 means the video increased by 2 times the embedded message size.

Frontloaded:

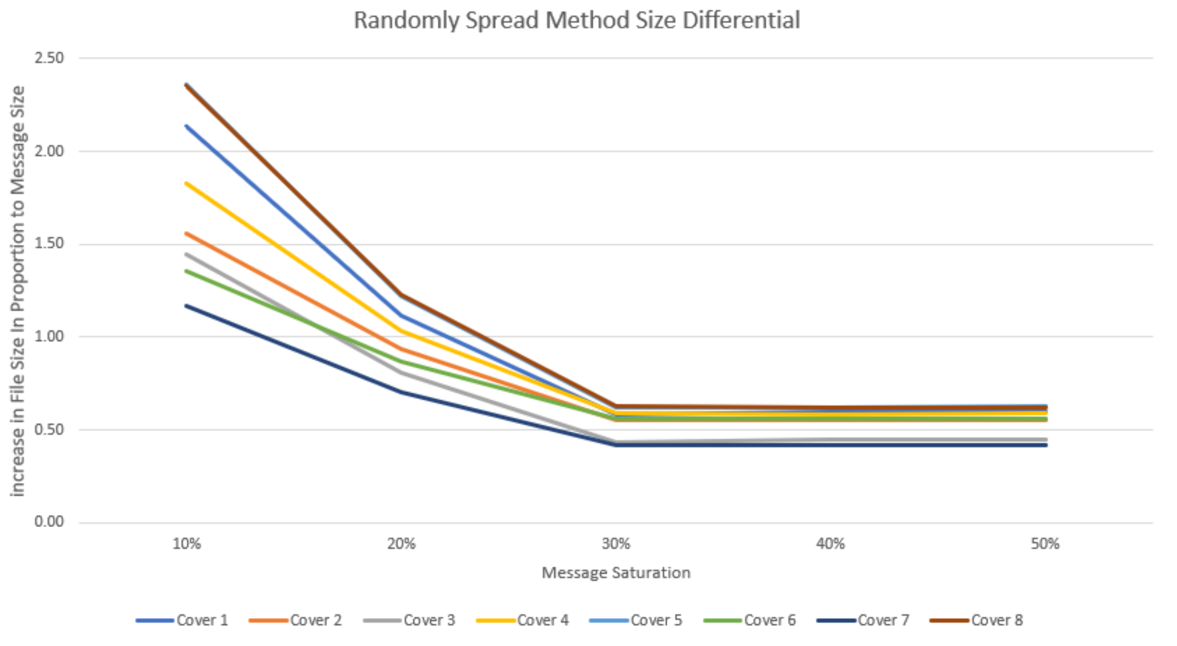
Frontloaded								
Message Saturation	Cover 1	Cover 2	Cover 3	Cover 4	Cover 5	Cover 6	Cover 7	Cover 8
10%	0.52	0.65	0.29	0.53	0.60	0.73	0.20	0.74
20%	0.51	0.66	0.28	0.55	0.58	0.73	0.20	0.71
30%	0.55	0.63	0.30	0.52	0.59	0.73	0.20	0.67
40%	0.58	0.62	0.32	0.52	0.60	0.73	0.20	0.65
50%	0.60	0.63	0.32	0.52	0.60	0.73	0.20	0.64
60%	0.61	0.64	0.32	0.53	0.60	0.73	0.20	0.63
70%	0.62	0.64	0.33	0.53	0.61	0.73	0.20	0.63
80%	0.62	0.64	0.32	0.53	0.61	0.73	0.20	0.62
90%	0.63	0.65	0.33	0.52	0.61	0.73	0.20	0.62
100%	0.63	0.64	0.32	0.52	0.61	0.73	0.20	0.61
average	0.59	0.64	0.31	0.53	0.60	0.73	0.20	0.65



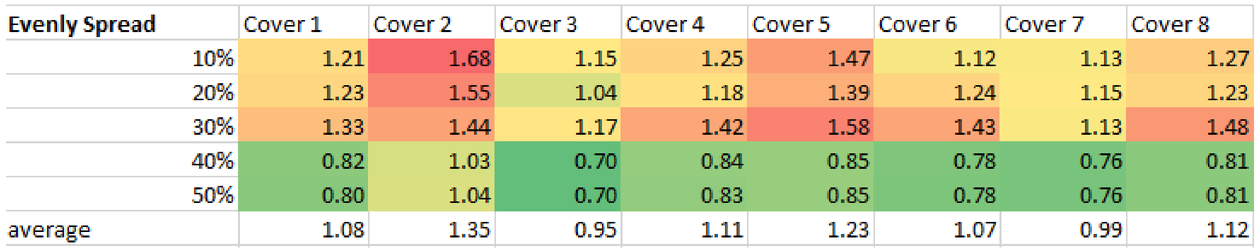
Random:



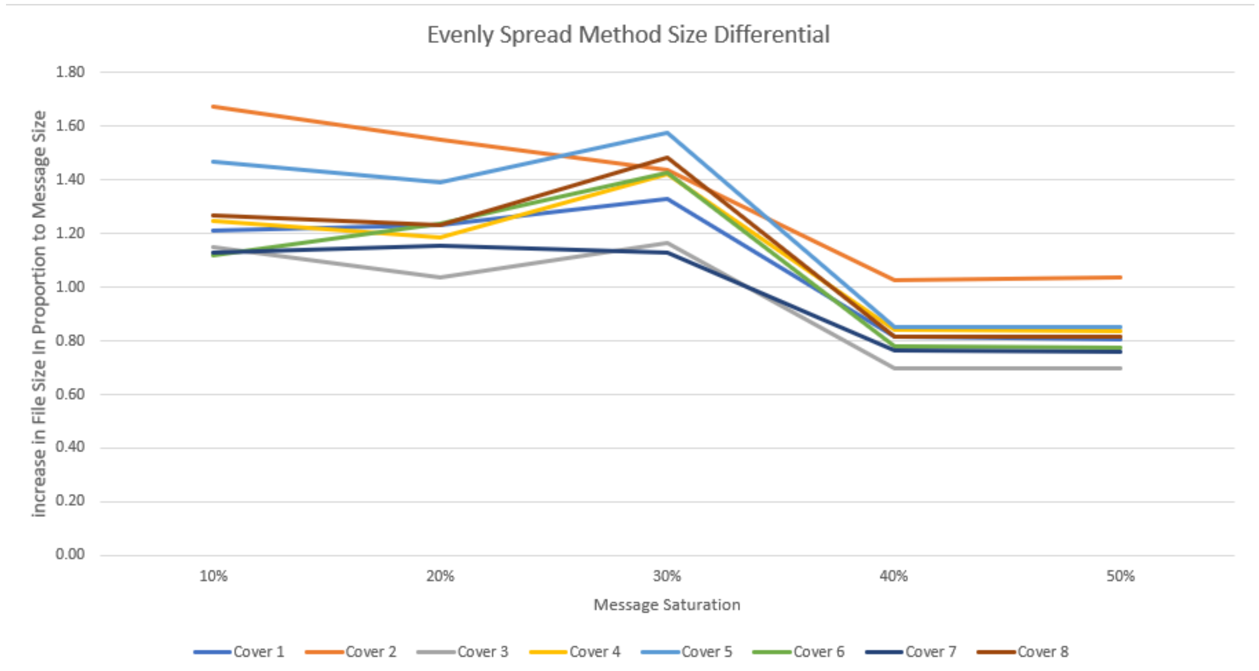
Random	Cover 1	Cover 2	Cover 3	Cover 4	Cover 5	Cover 6	Cover 7	Cover 8
10%	2.14	1.56	1.45	1.83	2.36	1.36	1.17	2.35
20%	1.11	0.93	0.81	1.03	1.22	0.87	0.71	1.23
30%	0.59	0.56	0.43	0.59	0.62	0.56	0.42	0.63
40%	0.59	0.56	0.45	0.59	0.62	0.56	0.42	0.62
50%	0.60	0.56	0.45	0.59	0.63	0.56	0.42	0.62
average	1.01	0.83	0.72	0.92	1.09	0.78	0.63	1.09



Evenly Spread:

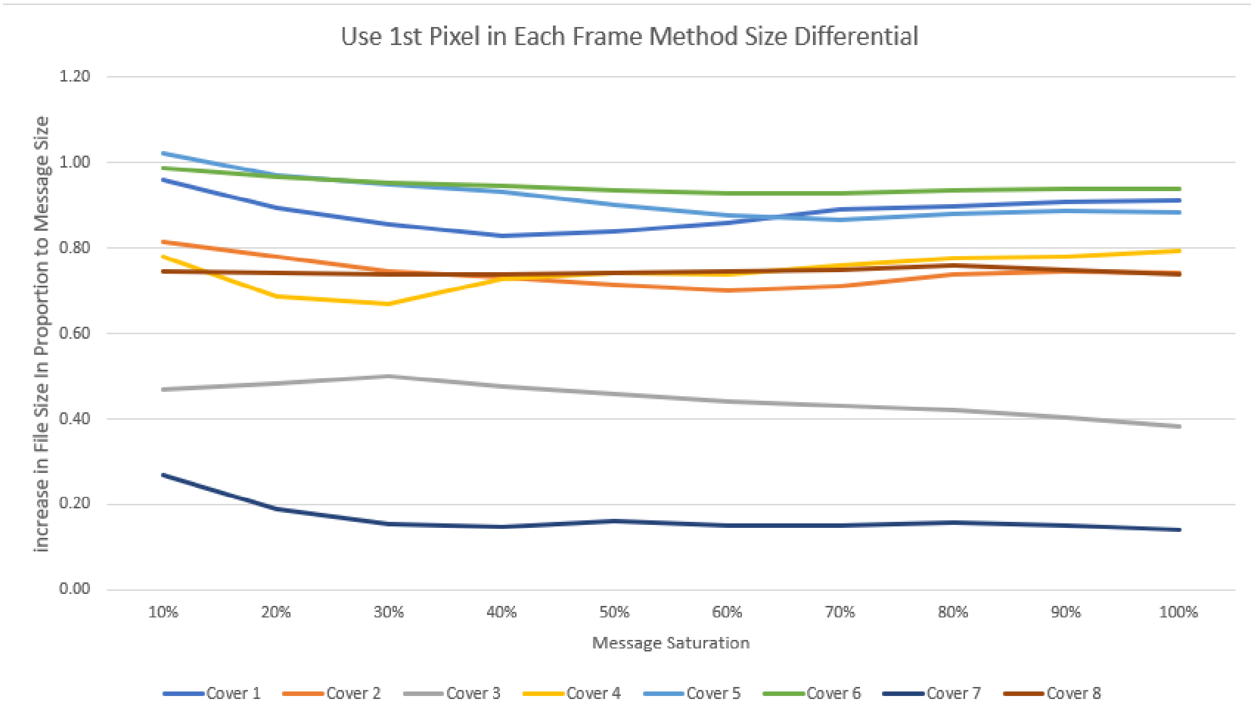


Evenly Spread	Cover 1	Cover 2	Cover 3	Cover 4	Cover 5	Cover 6	Cover 7	Cover 8
10%	1.21	1.68	1.15	1.25	1.47	1.12	1.13	1.27
20%	1.23	1.55	1.04	1.18	1.39	1.24	1.15	1.23
30%	1.33	1.44	1.17	1.42	1.58	1.43	1.13	1.48
40%	0.82	1.03	0.70	0.84	0.85	0.78	0.76	0.81
50%	0.80	1.04	0.70	0.83	0.85	0.78	0.76	0.81
average	1.08	1.35	0.95	1.11	1.23	1.07	0.99	1.12



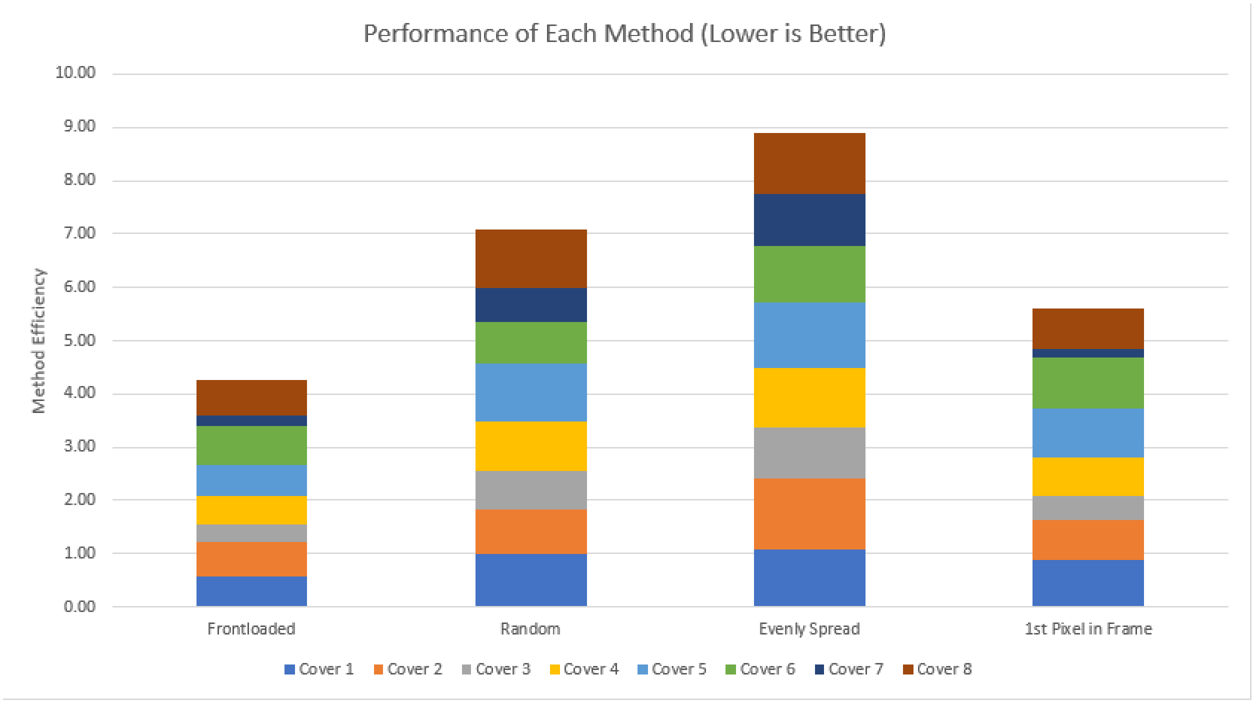
1st Pixel in Frame:

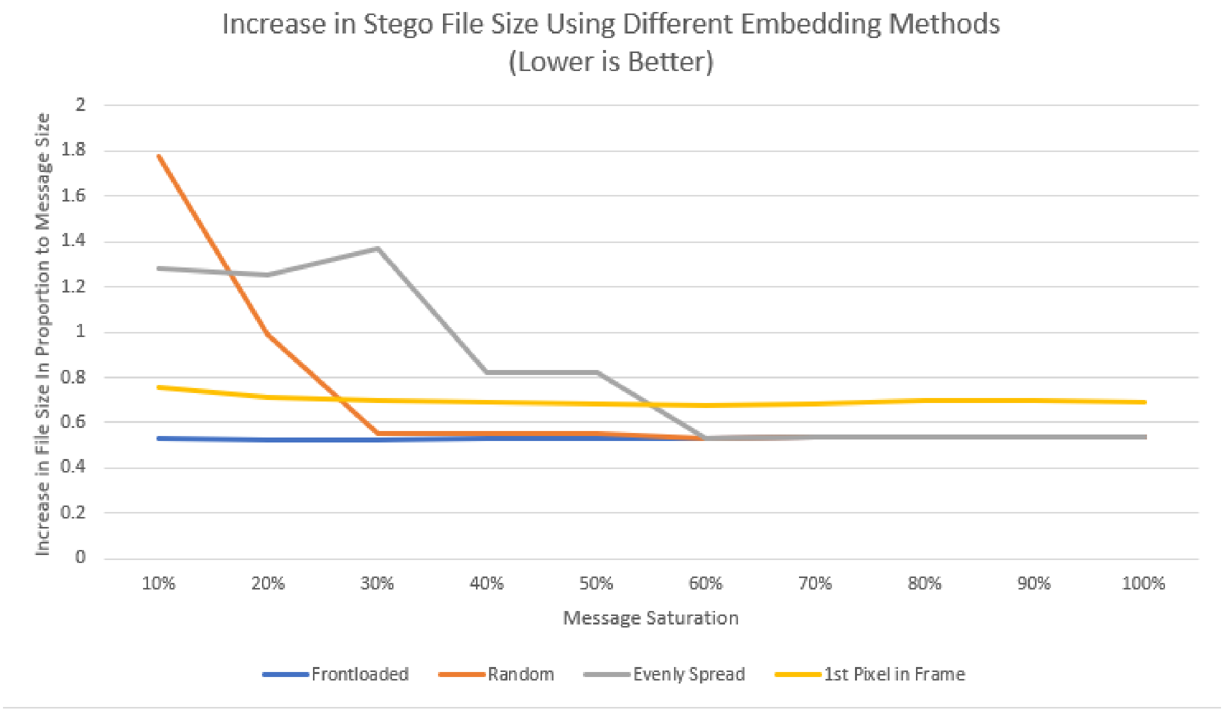
1st Pixel in Frame	Cover 1	Cover 2	Cover 3	Cover 4	Cover 5	Cover 6	Cover 7	Cover 8
10%	0.96	0.81	0.47	0.78	1.02	0.99	0.27	0.74
20%	0.89	0.78	0.48	0.69	0.97	0.96	0.19	0.74
30%	0.86	0.75	0.50	0.67	0.95	0.95	0.15	0.74
40%	0.83	0.73	0.48	0.73	0.93	0.94	0.15	0.74
50%	0.84	0.72	0.46	0.74	0.90	0.94	0.16	0.74
60%	0.86	0.70	0.44	0.74	0.88	0.93	0.15	0.74
70%	0.89	0.71	0.43	0.76	0.87	0.93	0.15	0.75
80%	0.90	0.74	0.42	0.78	0.88	0.94	0.16	0.76
90%	0.91	0.75	0.40	0.78	0.89	0.94	0.15	0.75
100%	0.91	0.74	0.38	0.79	0.88	0.94	0.14	0.74
average	0.88	0.74	0.45	0.75	0.92	0.95	0.17	0.74



Summary:

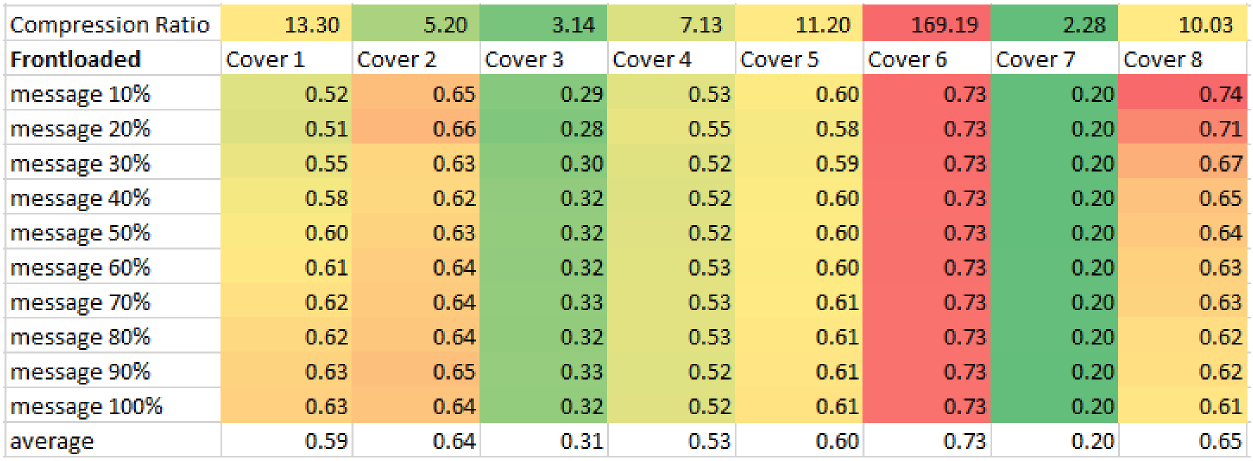
Average Performance	Cover 1	Cover 2	Cover 3	Cover 4	Cover 5	Cover 6	Cover 7	Cover 8	SUM
Frontloaded	0.59	0.64	0.31	0.53	0.60	0.73	0.20	0.65	4.25
Random	1.01	0.83	0.72	0.92	1.09	0.78	0.63	1.09	7.07
Evenly Spread	1.08	1.35	0.95	1.11	1.23	1.07	0.99	1.12	8.88
1st Pixel in Frame	0.88	0.74	0.45	0.75	0.92	0.95	0.17	0.74	5.59





We found with our results that confining the modified bits to as few video frames as possible with our frontloaded method consistently resulted in the smallest increase in file size. The magnitude that it increased also didn't seem to change as we approached the saturation limit of message data that the cover videos could hold. The 1st Pixel in Frame method performed almost as well as front loaded and also stayed consistent as message saturation increased. Randomly distributed message data produces very poor results in low message saturation and approaches the success of frontloaded as the density of changed bits increases. We suspect this is because as more of the least significant bits are changed, the encoder notices that inconsistency and changes it's strategy for how it compresses the data. We suspect this is likewise for the evenly spread method which somewhat holds true as it's relative file size decreases as message saturation increases; however, there is an odd bump at 30% message saturation. Perhaps this is at a point where the encoder's compression strategies struggle to deal with both mostly consistent pixel data that it expects, but also a large amount of odd and noisy data which is the message we inserted. We decided to use the frontloaded method for our OpenStego implementation.

Looking at the increase in size for each cover file reveals that how large the file size increases changes drastically with each video. For example cover file 7 has much smaller increases and cover file 6 has the largest. After watching each video it's clear that the videos we suspect are harder to compress with more things rapidly moving and changing color have much smaller increases than easily compressible videos with few colors and slow moving things. Evaluating the compression ratios for each cover file shows a clear correlation between high compression ratio and larger increases in file size.



Compression Ratio	13.30	5.20	3.14	7.13	11.20	169.19	2.28	10.03
Frontloaded	Cover 1	Cover 2	Cover 3	Cover 4	Cover 5	Cover 6	Cover 7	Cover 8
message 10%	0.52	0.65	0.29	0.53	0.60	0.73	0.20	0.74
message 20%	0.51	0.66	0.28	0.55	0.58	0.73	0.20	0.71
message 30%	0.55	0.63	0.30	0.52	0.59	0.73	0.20	0.67
message 40%	0.58	0.62	0.32	0.52	0.60	0.73	0.20	0.65
message 50%	0.60	0.63	0.32	0.52	0.60	0.73	0.20	0.64
message 60%	0.61	0.64	0.32	0.53	0.60	0.73	0.20	0.63
message 70%	0.62	0.64	0.33	0.53	0.61	0.73	0.20	0.63
message 80%	0.62	0.64	0.32	0.53	0.61	0.73	0.20	0.62
message 90%	0.63	0.65	0.33	0.52	0.61	0.73	0.20	0.62
message 100%	0.63	0.64	0.32	0.52	0.61	0.73	0.20	0.61
average	0.59	0.64	0.31	0.53	0.60	0.73	0.20	0.65

This creates a sort of tradeoff between having a smaller increase in the output stego file at the cost of having a harder to compress and therefore larger video file. However there is an important piece of information in the data of cover file 8 that shows larger relative increases when the message is utilizing the first 10% of the file as opposed to a larger message utilizing 100% of usable bits for our message. After viewing cover file 8 we noticed that the video starts off easily compressible with a static background, but as it goes on has a lot of rapidly moving objects. This implies that the message size increase is smaller when the message is inserted into harder to compress parts of the video. We could take advantage of this by finding the most noisy/hard to compress part of the video and interesting the message only in there.

Next we experimented with different encoding parameters to minimize the resulting file size. FFmpeg uses a default chroma subsampling of YUV420 which means each pixel is represented by 12 bits of data with 8 bits for the luminance information for each pixel and 16 bits of data for the color information for each square group of 4 pixels. Neither of us encountered this before and wondered why RGB wasn't used to represent a pixel with 3 8 bit values for red, green, and blue. FFmpeg also supports RGB so we tested both and found that our 8 test cover files encoded with H.264 had video file sizes that were on average 56% larger when using RGB compared to using YUV420. It turns out YUV420 is the most common chroma subsampling used for storing and streaming video, it's also very easily compressible. We decided to use YUV420 the default chroma subsampling for our implementation. Next is the encoder; earlier we identified H.264 and H.265 as the best candidates. A quick test on our 8 cover files shows that there are negligible differences in lossless file sizes; however H.265 encoding times were significantly longer. Both encoders allow for different presets that allow for a tradeoff between file size and time needed to encode. The more time taken, the smaller the file size. Using the 'veryslow' preset managed to give a good reduction in file size but only H.264 was able to encode each video in a reasonable amount of time. H.265 took roughly 10 times as long. Because of this we chose to have H.264 as our default codec used for our implementation. We do allow for the user to enter their own custom FFmpeg commands that can be used for encoding using the -fc or --ffmpegCommand flag if they choose to not use our default parameters.

A. Video Results:

The results proved our implementation successful. With OpenStego, messages can be hidden within video files, supporting a large density of message data if needed. This opens up interesting possibilities for the messages that can be transferred as a user could potentially hide larger images, audio, and more text within an innocuous video file thus improving their ability to secretly communicate information. By default we output the video as an mp4 encoded in lossless H.264 but we support the ability to use any codec and file types that FFmpeg supports, as long as the encoding parameters the user chooses doesn't corrupt the message.

The key downside is the resulting video size. At the base level the results are that lossless compressed video files are quite large and not commonly used. Despite this most media players we tested seemed to support it perfectly. Furthermore the increase in size from the hidden message is not ideal. These large video file sizes can potentially make transference of the message more difficult for those living in places with poor internet connections. Uploading a video to a website may take a long time.

IV. Conclusion

In summary, we broadly achieved our goals of enhancing OpenStego to handle audio and video formats. However, there are several improvements that could be made to the software that would further improve the concealment and the variety of formats that can be used. Sadly, this project must conclude due to time constraints, but we can hope that we will have more time at some point to work on it or perhaps other future programmers may take this project and further refine and improve its functionality.

A. Overall Project Status:

OpenStego now handles the embedding of message files into WAV and MP3 audio. While our embedding processes are imperfect and certainly have room for improvement, a person can safely share audio files with their contacts and have it look and sound the same to an outside observer. Video is much the same;

while the resulting video files are quite large and also can be larger when compared to the original when compressed losslessly, if an outside observer did not already know the size of the original they would be unable to detect that anything was amiss. If embedded into a website and the user's internet speeds were sufficient, they wouldn't notice anything wrong at all and if they had slower speeds, they would just notice some buffering. To address our use case, the project is a success. If a person is using homemade audio and video files, suspicious parties have no way of knowing in what state they were in before the embedding process and so the size of video will not arouse suspicion as a well known file would.

B. Future Work:

There are several things we wish we could've improved upon if we had the time. Firstly, we would want to include more audio formats for embedding messages in. While WAV and MP3 are quite common, having a greater variety of audio to embed messages in would further help to obscure just what files moving around the internet might contain hidden messages. Furthermore, improving the method of embedding data in MP3 files so that it can hide more data and to make that data better obscured is a priority. As for video, the main improvement that would be to decrease the resulting file size of the stego file. This likely means supporting lossy encoding which means a new embedding and extracting algorithm would need to be tailored for it. Another important issue is the increased file size on embedding a message. The increased file size reduces the portability of those videos and may hurt users ability to share or embed them. We conceptualized a method for minimizing the increase in file size by embedding the message in only hard to compress parts of the video but sadly we were not able to implement this in time. A further exploration would be to test if both the visual and audio components of a video can be used to hide data, essentially double dipping on the hidden content and adding an extra step when examining files for tampering. In addition to these primary focuses adding more data mediums such as GIF files would further improve the program. Our use case was for the more tech literate to use so we chose to only support using our plugins via command line but OpenStego does have a graphical user interface and integrating our plugins with that would benefit the user experience and lower the tech knowledge needed for use.

V. References

Sehra, V (2020). *Enshroud*, GitHub repository, https://github.com/vibhusehra/Enshroud

Desipojci, G., Seferi, A., Theis, M. (2016). *Mp3 Steganography,* Forensecure: Cyber Forensics & Security Conference 2016, https://appliedtech.iit.edu/school-applied-technology/projects/mp3-steganography

Amritaryal44, (2020). *Video-Steganography,* GitHub repository, https://github.com/Amritaryal44/Video-Steganography