LSB Wave File Steganography

Vega Carlson
Computer Engineering
University of Nebraska - Lincoln
vegac@protonmail.com

Noah McCashland
Computer Science
University of Nebraska - Lincoln
noahmccashland77@gmail.com

Abstract

The objective of this paper is to study and implement a method of Steganography that is undetectable to a user by hiding the binary payload of a file into a .wav file and determine to what extent the payload may occupy the least significant bits (LSB) of the waveform file before a change in the audio is perceptible. Steganography with digital files is often done by simply appending data of one format to that of another format, such as hiding an executable on the end of an image file. This method suffers from two significant problems. The first being the file is often easy to identify due to its unreasonable size. The second being that some programs will look for the appropriate file header, finding the hidden file more easily than intended. We propose a method for storing the secret message in the least significant bits of a wave file. This method requires a dedicated decoder as the data is distributed through the original file instead of being continuous.

Index Terms

Steganography, Wave File, .wav, Concealed Writing

I. Introduction

Steganography is being used whenever a message is being hidden inside of another media or file. Some examples include an image being rendered in a frequency spectrum view of a radio transmission or ink that is only visible when exposed to heat. Usually, Steganography involves there being an ‘obvious’ message that the normal viewer would see before the real message is found.

Steganography is not the first thing that comes to mind when a user thinks about file security. For the most part, the concept that is on everybody’s mind is encryption. This is not without reason: normal steganography does not actually provide any mathematical assurance of security like encryption does. Instead, steganography, a word that at its root literally means ‘concealed writing’[1], is not about providing assurance that some data is secure, but rather it is used to send messages where the concern is that the message exchange is being monitored and where a third party, knowing there is encrypted data, may be a problem in the first place. This might be the case for any of a number of reasons. One of the most notable is the possibility that a user lives under an oppressive government, where all communications are monitored and encryption is illegal.

Unfortunately, commonly digital steganography methods are easily detectable, tipping off the fact that there is something suspicious about a given file. If a user sends an .mp3 file that when played only contains a few minute song, but the file is ten gigabytes, it is not exactly covert anymore. This is particularly bad when you consider the simplest form of digital steganography, wherein it is often done by appending data of one format to that of another format, such as hiding an executable on the end of an image file. This method just assumes that a user’s computer will only read the beginning file as the file extension (such as .jpg) will inform the system to open a photo viewer, then the photo viewer will only interpret the bytes between the image format headers. This is the absolute worst case for the file growth problem. This is made an even bigger problem by particular applications ignoring the file extension outright. If, for example, an image file had an .mp3 appended to it a music playing program may be able to see the .mp3 header outright even if the file is stored as a .jpg, meaning the ‘security’ of the file is dependent on the user’s application preferences.

To bypass this concern, we have implemented a system for encoding a payload file into the least significant bits of a target .wav file. '.wav' is an uncompressed audio file format, this means that changing the least significant bits of a given sample will only affect that sample, instead of having the possibility of corrupting the entire file by only changing a few bits. By only encoding the data into the least significant bits, the actual values of any given sample point will only change very slightly. Assuming that the data being appended is random, this functionally equates to adding a small amount of noise to the file. While we had the idea without prior knowledge, upon researching the field we found this idea is anything but novel [2]. Despite this, we still deemed it to be valuable to explore implementing this ourselves.

This method produces an output file of exactly the same size as the input file, eliminating both concerns, file size inflation and the continuous hidden data, that were expressed above about the simple steganography method.

While this paper focuses on the Wave (.wav) file format specifically, this method should be applicable to many file formats which contain uncompressed blocks of data, such as bitmap images, so long as the changes to the least significant bits do not cause a perceptible change in the original file. [3]

It is worth noting, that any stenography method can store encrypted data as the payload; however, many encryption algorithms themselves add a file header, so if the goal is to send a message without a third party knowing that the message contains encrypted data, our method (or any that does not store the payload bytes continuously) is necessary if the user wishes for the message to be able to bypass simple tools which may identify these foreign headers within the file. Some steganography tools, such as "SilentEye" [4] do exactly this, allowing for LSB steganography with strong encryption.

II. Aside on Polyglots

Above, it was mentioned that one way of doing steganography results in a file that may be opened in different ways depending only on its file extension. This is generally referred to as a "polyglot" file. Generally, polyglot files are any file which may be interpreted as something else depending on how they are opened, compiled, or viewed [5]. One example to demonstrate this idea is a code polyglot as in Figure 1 where the code is valid in PHP, C, and Bash. The author of this is unknown, but the code, written for the Wikipedia entry on Polyglots, was released to the public domain. [6]

As this code is very obviously strange to anyone that knows any of these programming languages, it is not really making the effort of concealment that is implied by steganography. There is however some relevance in applications of this idea in which the file types are more hidden and the relation less obvious. For example, for the freely distributed PoC||GTFO hacker culture magazine, the .pdfs are notoriously much more than meets the eye. In each successive issue the hidden filetypes and process to do so is revealed, so, in that sense, the process is one of steganography. Examples include PoC||GTFO issues two and eleven, which along with their obvious nature as PDFs could be interpreted as a ZIP file and a bootable Operating System or ruby code and HTML respectively [7]. The efforts to make this work often require incredible trickery to make valid file headers and to prevent the reading of excess information.

This is not directly related to our work with .wav file steganography, but the idea is a good example of how far the concepts of steganography can be taken.

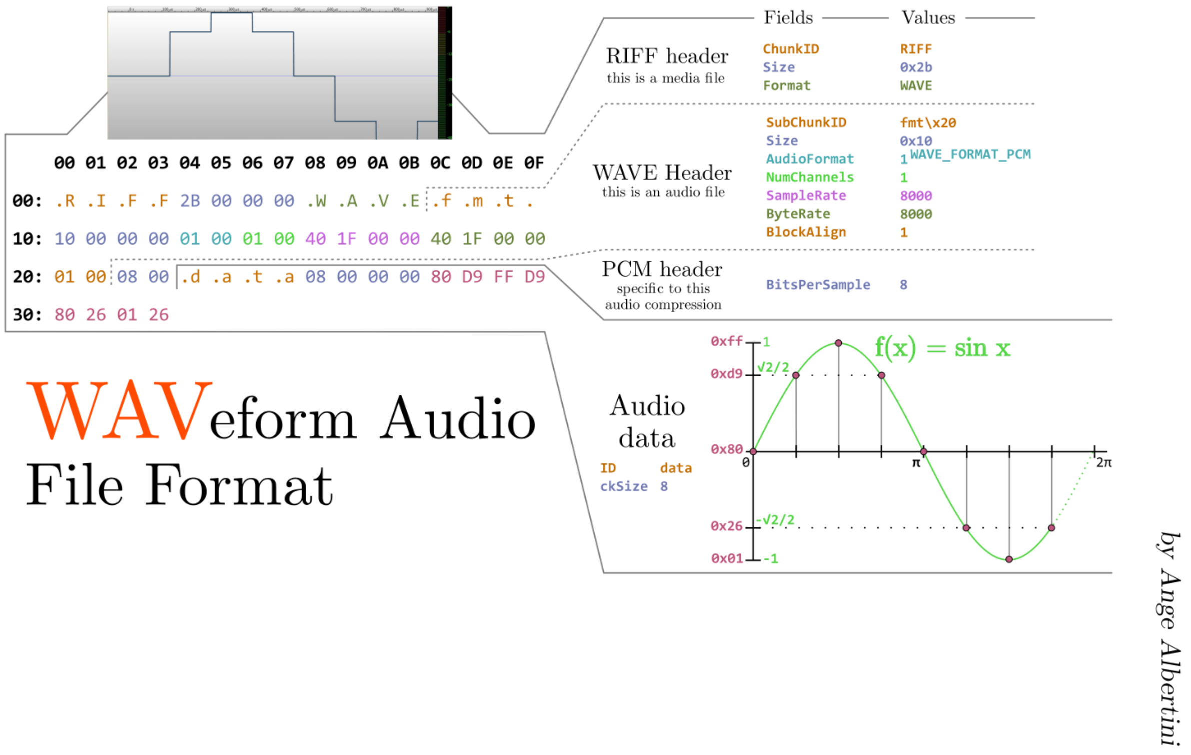
III. Methodology

The Waveform audio file format (.wav) stores individual sample points without compression, then interpolates a waveform from these points during playback, as can be seen in Figure 2. In a typical audio .wav file, a new sample point is stored over forty-thousand times per second, and with a bit depth of either sixteen or twenty-four bits per sample. Even at the lower sixteen bits per sample, there are 2^{16} or 65536 levels for a waveform to occupy. If only the the two least significant bits are used to store our payload, 2^{14} or 16384 levels will still remain for the audio. Furthermore, these least significant bits are exactly as they sound, not very significant to the waveform. Each consecutive bit in the sample matters

```
#define a /*
#<?php
echo "\010Hello, world!\n";// 2> /dev/null > /dev/null \ ;
// 2> /dev/null; x=a;
$x=5; // 2> /dev/null \ ;
if (($x))
// 2> /dev/null; then
return 0;
// 2> /dev/null; fi
#define e ?>
#define b */
#include <stdio.h>
#define main() int main(void)
#define printf printf(
#define true )
#define function
function main()
{
printf "Hello, world!\n"true/* 2> /dev/null | grep -v true*/;
return 0;
}
#define c /*
main
#*/
```

Fig. 1. Polyglot code, valid C, PHP, and Bash

less to the amplitude of that sample point. As such, these final bits being occupied by the payload should only introduce minimal noise, if any.



```
     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00:  .R .I .F .F 2B 00 00 00 .W .A .V .E .f .m .t .
10:  10 00 00 00 01 00 01 00 40 1F 00 00 40 1F 00 00
20:  01 00 08 00 .d .a .t .a 08 00 00 00 80 D9 FF D9
30:  80 26 01 26
```

	Fields	Values
RIFF header this is a media file	ChunkID	RIFF
	Size	0x2b
	Format	WAVE
WAVE Header this is an audio file	SubChunkID	fmt\x20
	Size	0x10
	AudioFormat	1 WAVE_FORMAT_PCM
	NumChannels	1
	SampleRate	8000
	ByteRate	8000
	BlockAlign	1
PCM header specific to this audio compression	BitsPerSample	8
Audio data	ID	data
	ckSize	8

Fig. 2. .wav file structure [8]

TABLE I
MAPPING OF PAYLOAD AND LENGTH INDICATOR BITS

Input Wave	Payload	Length Indicator Bits	Result
0xFFFF	0b00		0xFFFC
0x0000	0b01		0x0001
0xFFFC	0b10	N/A	0xFFFE
0x0000	0b11		0x0002
0xABCD	0b11		0xABCF
...			
0xFFFF		0b0	0xFFFC
0x0000	N/A	0b1	0x0001
0xFFFF		0b1	0xFFFD

This project requires two primary parts: an encoder and a decoder. The purpose of each is relatively self explanatory, the encoder takes the input .wav file and payload, stores the binary of the payload into the 'n' least significant bits, and then writes out the new .wav file. The decoder simply recovers the binary from these least significant bits.

As with all things though, the devil is in the details. It would be relatively trivial to make an encoder decoder if some variables were kept constant, but for this paper where the effects of the noise are to be explored, the number of least significant bits in which the data was encoded as well as the bit depth of the input .wav file are both variable. This makes writing the encoder and decoder a much more interesting task.

While it may seem like this problem would be well suited to a lower level programming language such as C++ or Rust, we chose to implement our proof of concept in Python to facilitate rapid development and to take advantage of readily available libraries for reading and writing the .wav files and their metadata.

To start, we'll examine the encoder. The encoder, which can be found in this project's GitHub repository or in the source code appendix on this paper, begins with 'hardcoded' variables to specify the input .wav file, payload file, and the number of least significant bits used store the payload. To run the file, these should be changed accordingly. Then, the scipy.io wavefile library is used to read in the wave and get its sample rate. Next, the numpy library is used to get the payload data's bytes into a matrix. From there, the array of payload bytes is split into a larger array such that each array index only contains as many bits of the input byte as will be stored into the least significant bits of each sample.

Now, with the payload processed, a loop is ran to replace the specified number of least significant bits in the payload file with the bits of the payload. In our method, this process also results in any excess space in the wave file's least significant bits being filled with zeros. This is one potential short coming of our proof of concept implementation, as it may be noticeable that only the beginning of a wave file has non-zero values in the least significant bits. This problem will be expanded on shortly.

With the payload now stored into the least significant bits of the wave file, there is the problem of how to store the file such that the decoder can know how many of the least significant bits were used to store the payload. To solve this problem, we simply assume their payload will not be the entire length of the input wav, and use the end of the wav to store the number of least significant bits as a sequence of 1's followed by a 0, read backwards from the end of the .wav. For example, when storing the payload with two of the least significant bits into a 16 bit .wav the first 14 bits (masked by 0xfffc) will hold the unaltered audio data, the least significant two bits hold the payload, but, at the end of the file, the singular least significant bits in the last three samples hold this count of least significant bits (two for the count, one to indicate the stop). This can be seen in Table I.

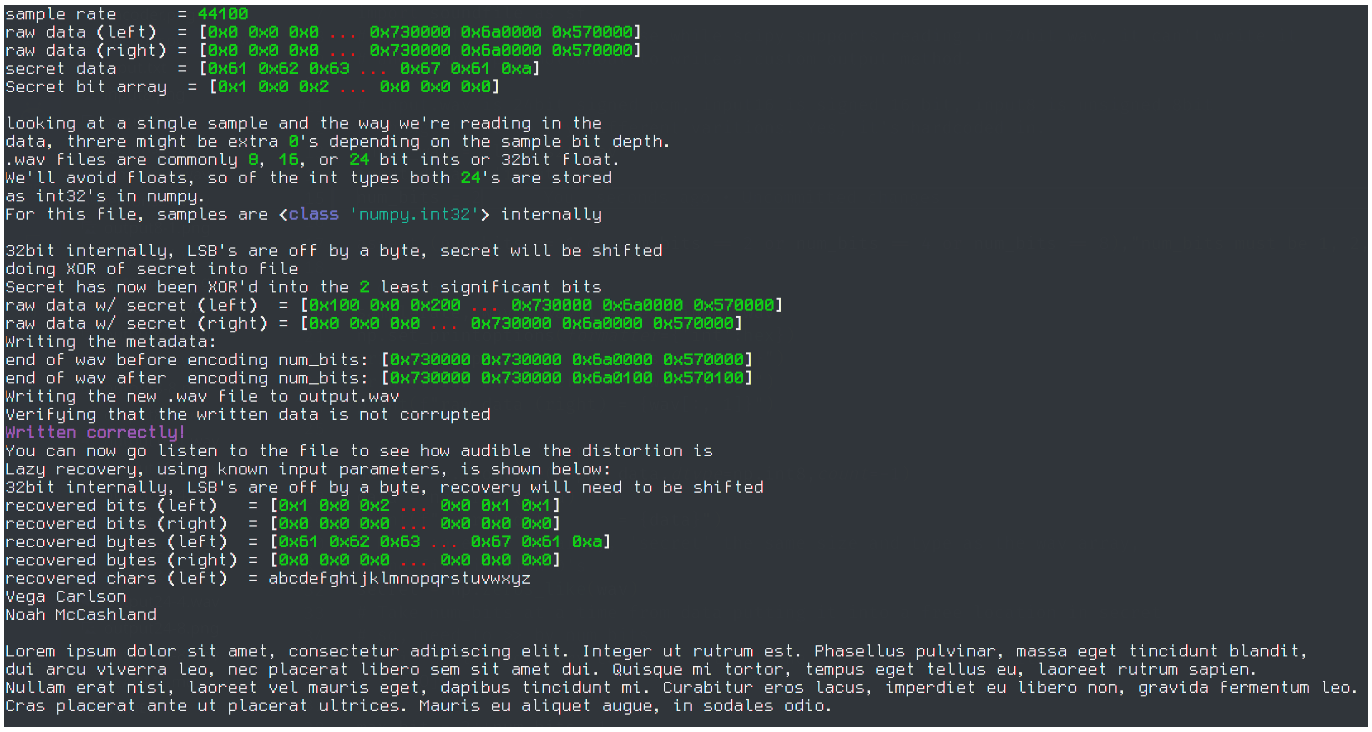
Now, the new wave file can be written out to a different file to be passed to the decoder. However, our code assumes the payload is an ascii file, and proceeds to run the decoding process on the internal data structure, as a check that the data was written correctly.

A few issues still remain with this proof of concept encoder. The most obvious is that while the number of least significant bits is being stored for use by the encoder, the length of the payload is not. The decoder simply has to assume a long chain of 0 values indicates the end of the file. This should be able to be implemented similarly to this 'number of bits' method, but for simply showing this project as a proof of concept and with limited time, this was not done. The encoder takes a few other short cuts as well. First of all, there's the previously mentioned issue of the large zero region acting as a 'red flag' of sorts to indicate the file has been tampered with, and second, our implementation only uses the left channel of the stereo wave file. This cuts the amount of room in which we could store data in half. Regardless, the payload is successfully injected.

From here, we can examine the decoder.

The decoder first reads the very least significant bits from each sample at the end of the wave file, iterating backwards, and keeping a tally of 1's until a zero is encountered. This count is the number of least significant bits in which the payload is stored. Now, using this, the file is read forward, retrieving the bits of the payload, storing them into an array. Then, this array of chopped up payload bytes is reconstructed by shifting these bits back into the true payload bytes, before finally being echoed back out and decoded as ASCII text to verify the payload was recovered.

This entire process is shown with intermediate values exposed when the encoder is ran by invoking `python encoder.py` with the payload file path, input wave file path, and number of least significant bits to utilize all defined in the encoder.py file appropriately.



```
sample rate      = 44100
raw data (left)  = [0x0 0x0 0x0 ... 0x730000 0x6a0000 0x570000]
raw data (right) = [0x0 0x0 0x0 ... 0x730000 0x6a0000 0x570000]
secret data      = [0x61 0x62 0x63 ... 0x67 0x61 0xa]
Secret bit array  = [0x1 0x0 0x2 ... 0x0 0x0 0x0]

looking at a single sample and the way we're reading in the
data, threre might be extra 0's depending on the sample bit depth.
.wav files are commonly 8, 16, or 24 bit ints or 32bit float.
We'll avoid floats, so of the int types both 24's are stored
as int32's in numpy.
For this file, samples are <class 'numpy.int32'> internally

32bit internally, LSB's are off by a byte, secret will be shifted
doing XOR of secret into file
Secret has now been XOR'd into the 2 least significant bits
raw data w/ secret (left)  = [0x100 0x0 0x200 ... 0x730000 0x6a0000 0x570000]
raw data w/ secret (right) = [0x0 0x0 0x0 ... 0x730000 0x6a0000 0x570000]
Writing the metadata:
end of wav before encoding num_bits: [0x730000 0x730000 0x6a0000 0x570000]
end of wav after  encoding num_bits: [0x730000 0x730000 0x6a0100 0x570100]
Writing the new .wav file to output.wav
Verifying that the written data is not corrupted
Written correctly!
You can now go listen to the file to see how audible the distortion is
Lazy recovery, using known input parameters, is shown below:
32bit internally, LSB's are off by a byte, recovery will need to be shifted
recovered bits (left)   = [0x1 0x0 0x2 ... 0x0 0x1 0x1]
recovered bits (right)  = [0x0 0x0 0x0 ... 0x0 0x0 0x0]
recovered bytes (left)  = [0x61 0x62 0x63 ... 0x67 0x61 0xa]
recovered bytes (right) = [0x0 0x0 0x0 ... 0x0 0x0 0x0]
recovered chars (left)  = abcdefghijklmnopqrstuvwxyz
Vega Carlson
Noah McCashland

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer ut rutrum est. Phasellus pulvinar, massa eget tincidunt blandit,
dui arcu viverra leo, nec placerat libero sem sit amet dui. Quisque mi tortor, tempus eget tellus eu, laoreet rutrum sapien.
Nullam erat nisi, laoreet vel mauris eget, dapibus tincidunt mi. Curabitur eros lacus, imperdiet eu libero non, gravida fermentum leo.
Cras placerat ante ut placerat ultrices. Mauris eu aliquet augue, in sodales odio.
```

IV. Results and Findings

As for our results, it can be seen that, as expected, as the number of bits used for carrying the payload is increased, so to does the distortion. Because the right channel audio (which is on the bottom of each of the following figures) is unused for the payload but does still have the bits the payload would occupy set to zero, the effect of the bit-depth reduction without the payload can be seen as well.

TABLE II
AVAILABLE STORAGE BY PAYLOAD DEPTH AND FILE BIT DEPTH

Input Bit Depth	Original File Size	Payload Bit Depth	Amount Of Storage For Payload	Percent Payload
8	5.04MB	1	645KB	12.5%
		2	1.26Mb	25%
		4	2.52Mb	50%
		8	5.04MB	**100%**
16	10.09Mb	1	645KB	6.25%
		2	1.26Mb	12.5%
		4	2.52Mb	25%
		8	5.04MB	50%
24	15.14Mb	1	645KB	4.16%
		2	1.26Mb	8.33%
		4	2.52Mb	16.66%
		8	5.04MB	33.33%

The waveform difference is difficult to see at the 1-bit payload depth in figure 4, but at two bits as in figure 5, the start of the wave appears thicker, which is reflective of the distortion caused by the payload. Similarly, at 4 bits used for the payload as in figure 6 the .wav begins to reach max or minimum values of 0 or 255 (indicated by the vertical red lines). This is known as ‘clipping’ and causes a very audible click. This may tip off a listener that something is wrong. Finally, in figure 7 all of the bits are used to carry the pay load. This may seem totally useless, but it may be able to pass some tools which simply look for a valid file header, see the .wav, and move on; however, it does bring back the continuous data issue as mentioned previously in this paper. It is essentially just wrapping the data in a .wav file header.

Audibly, all of the eight bit files have noticeable distortion. Interestingly, the 1 and 2 bit payload outputs both seem to be dominated by a single high frequency whine, the 4 bit payload sounds more like a distortion effect that would be applied to an electric guitar.

All of these eight bit results are a a bit academic in nature to begin with, as eight bit audio is already of such poor quality that even the input wave file is of very poor quality.

The same experiment was run for all combinations of 8, 16, and 24-bit input wav files with 1, 2, 4, and 8 bits used to store the payload. Not all the waveforms are shown here, as generally they are very similar graphically. To show a more real world scenario though the input 24-bit wave file is shown in figure 8 and the output with 8-bit dedicated to the payload in figure 9. Even at 8-bits of payload data in the 24-bit wave, the effect is barely very subtle in the waveform view, and - at least for the two of us writing this paper - completely equivalent to our ears. Vega Carlson was able to listen via a studio monitor headphones and a high end audio interface and still could not tell the difference in a blind test.

While it may seem that these qualitative results are a sub-optimal way to present our findings, measuring audio distortion in a way that is actually reflective of human hearing is a research paper in of itself. To keep this research reasonable in the time frame provided, these qualitative results are the best we can provide.

Naturally, it is hard to convey acoustic results via text and graphs. We strongly encourage the reader to listen to the input and output waveforms that can be found in the GitHub repository as well as to run the code on their own files to hear how various payload bit depths and input file resolutions sound.

Finally, for the sake of comparison, Table II is provided showing how large of payload can be carried in one minute of audio at various file bit depth and payload bit depths. These are theoretical values, and ignore the bits occupied by the file header as well as those used for storing the payload metadata; however, these bits are so few compared to those of the samples as to be insignificant. All values assume a 44100Hz sample rate and that both stereo channels may be used, which is not the case with our proof of concept implementation.

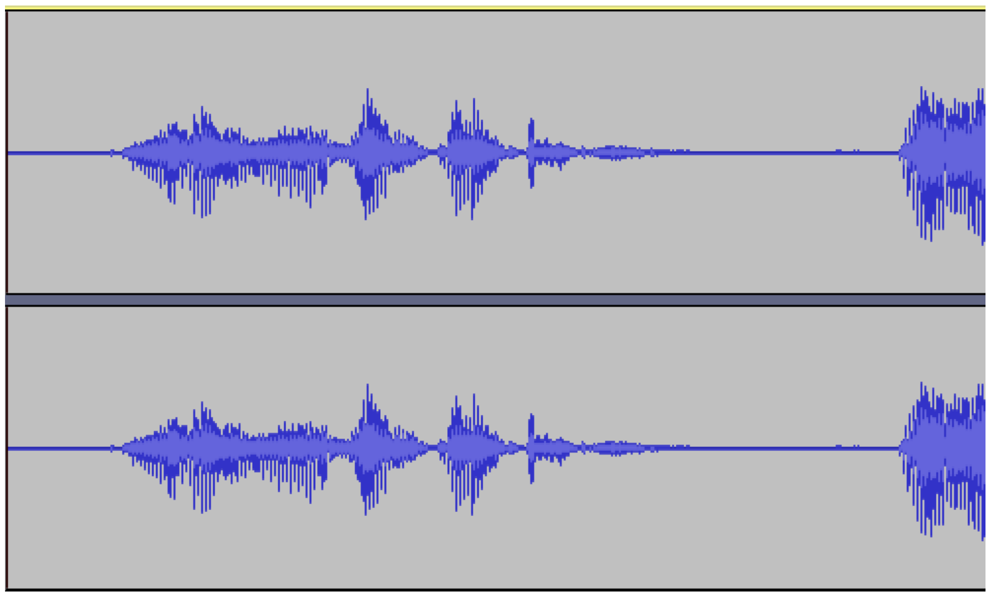


Fig. 3. Input 8-bit .wav file

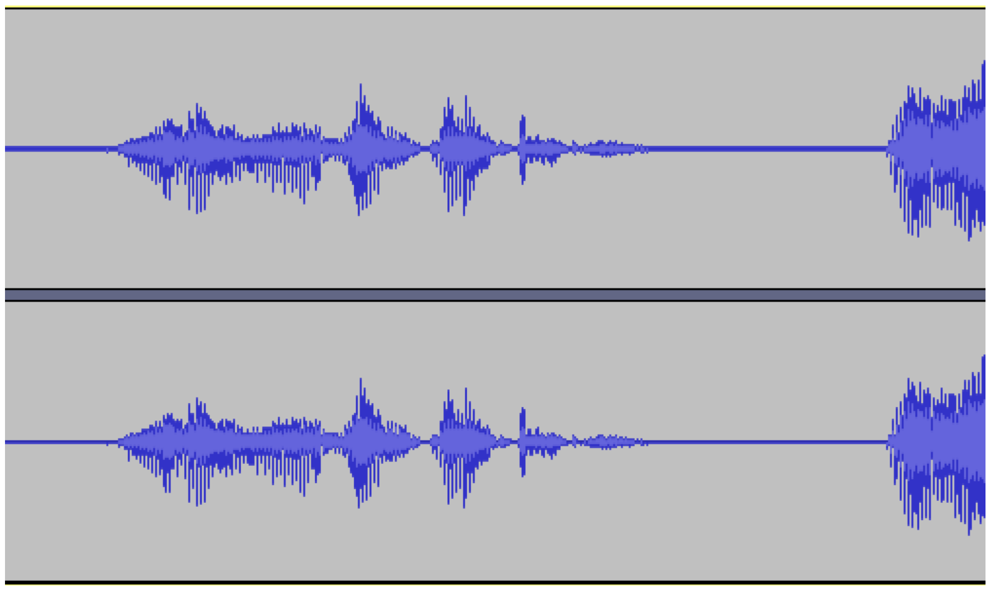


Fig. 4. Output 8-bit .wav file with 1 bit used for the payload

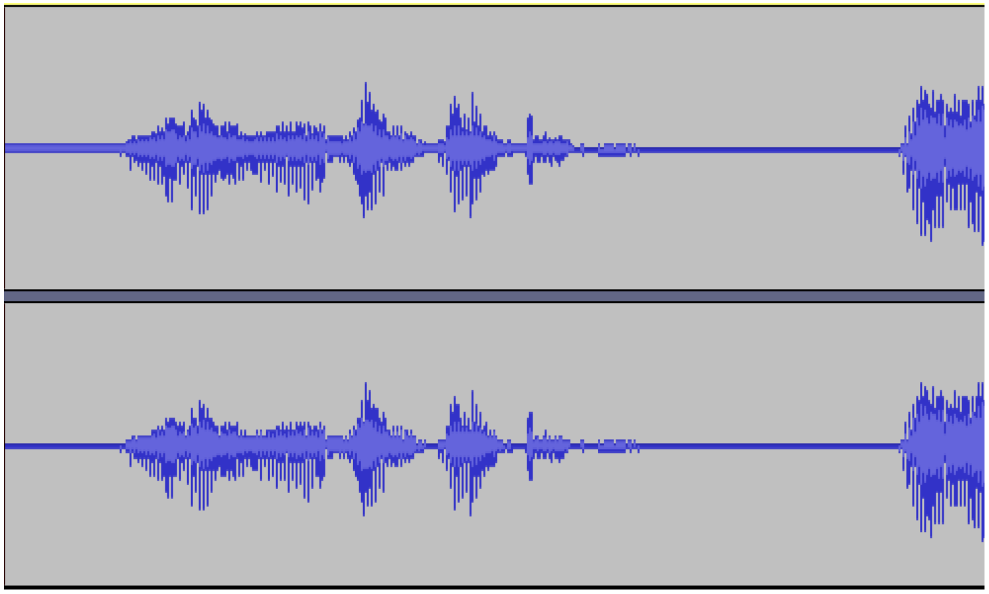


Fig. 5. Output 8-bit .wav file with 2 bits used for the payload

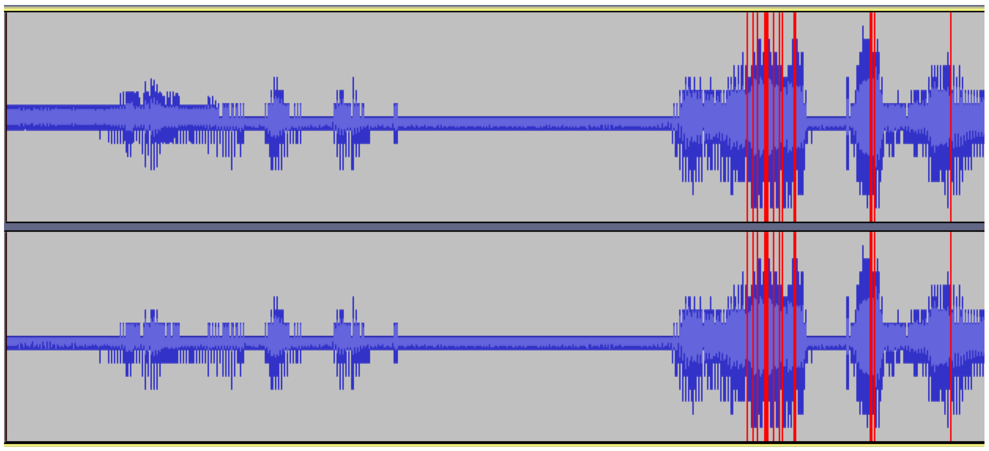


Fig. 6. Output 8-bit .wav file with 4 bits used for the payload

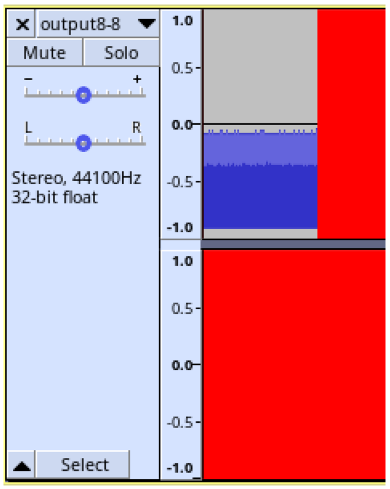


Fig. 7. Output 8-bit .wav file with 8 bits used for the payload

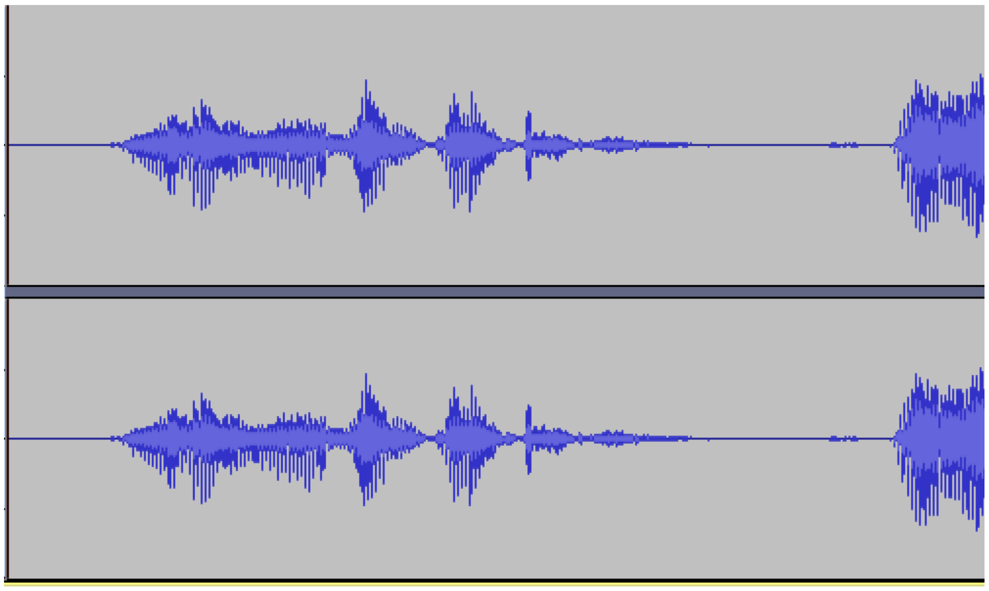


Fig. 8. Input 24-bit .wav file

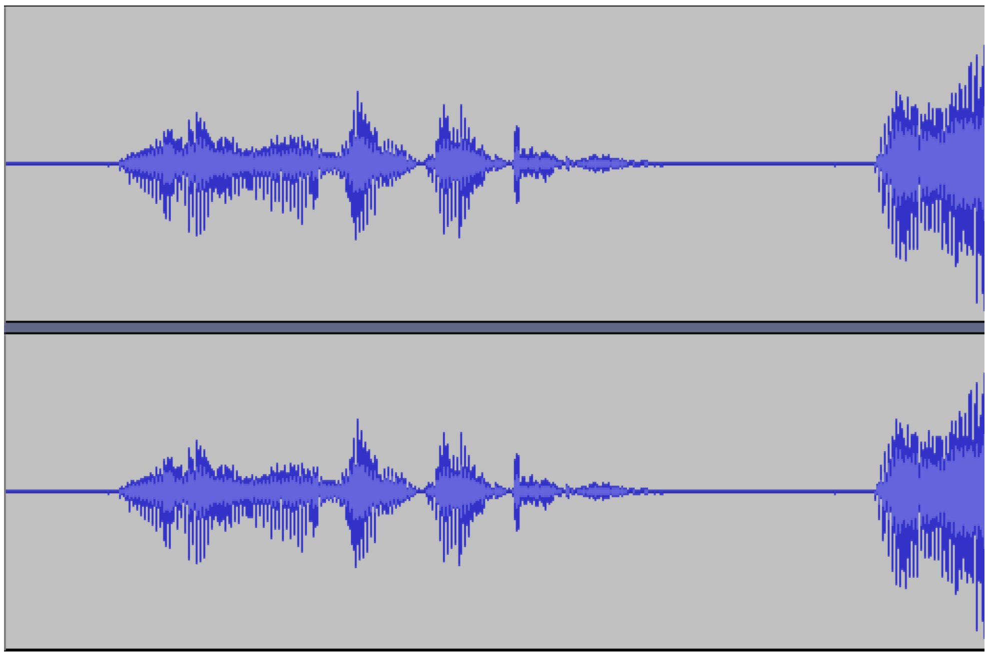


Fig. 9. Output 24-bit .wav file with 8 bits used for the payload

V. Conclusion

We found that this method of Steganography produced better results on audio that was a higher bit depth to begin with. While using 2 bits for the payload in an 8-bit file and using 8-bits for the payload in a 24-bit file may initially sound like they should bring about equivalent amounts of distortion, this was most certainly not the case. Instead, it seems that the least significant bits of a wave file are even less significant to the sound as the bit-depth is increased.

Overall, this method accomplished all of our goals. We found that we could insert a payload into a wave file in a way that a normal listener would not be able to hear and can even use up to one quarter of the initial file size of a 24 bit wave file for the payload. Because wave files are uncompressed and so large for even a small clip of audio, this means that a large amount of data can be hidden in even a short audio clip.

Our proof of concept is not code that should be utilized for doing Steganography in anything but an academic context, but we still found our results could point to this being a viable method for real world applications.

References

[1] "Steganography." [Online]. Available: https://www.merriam-webster.com/dictionary/steganography

[2] N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," *Computer*, vol. 31, no. 2, p. 26–34, 1998.

[3] A. Cheddad, J. Condell, K. Curran, and P. M. Kevitt, "Digital image steganography: Survey and analysis of current methods," *Signal Processing*, vol. 90, no. 3, p. 727–752, 2010.

[4] A. Chorein, "Silenteye." [Online]. Available: https://achorein.github.io/silenteye/

[5] V. Li, "Polyglot files: a hacker's best friend," Oct 2019. [Online]. Available: https://medium.com/swlh/polyglot-files-a-hackers-best-friend-850bf812dd8a

[6] "Polyglot (computing): Revision history," Jul 2003. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Polyglot_(computing)&dir=prev&action=history

[7] E. Sultanik, "International journal ofpoc||gtfo." [Online]. Available: https://www.sultanik.com/pocorgtfo/

[8] M. Laphroaig, *PoCGTFO*. No Starch Press, 2017.