.Wav file Steganography

Vega Carlson and Noah McCashland

What is Steganography?

-Steganography is the practice of concealing messages or information within other non secret text or data.

- 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.

- Steganography usually involves an “obvious” message that a user would see before finding the actual “hidden” message.

- Steganography can be done by appending data of one format to another, like hiding an executable on the end of an image file, but can be much more subtle

Source [1]: https://www.merriam-webster.com/dictionary/steganography
Source [2]: Abbas Cheddad, Joan Condell, Kevin Curran, Paul Mc Kevitt, Digital image steganography: Survey and analysis of current methods

Best Practice of Steganography

Simple Steganography usually has two main issues:

1) The file is easy to identify from its large size
2) When the hidden file is continuous, some programs will ignore the file extension and look for the file header, finding the hidden file easily.
    a) This can actually be useful, as we'll see on the next slide.

- Our solution to these issues is to store the secret message in the least significant bits of a wave file, keeping the file size identical.
- This same technique can be applied to uncompressed media files other than .wav files as long as the changes gives little to no perceptible change to the file.

An Aside From Our Research: Polyglot files

Polyglot files are files that can act as two different formats depending on how they're opened or compiled.

- This isn't *inherently* steganography as it may not be hidden
- Often, polyglots exist as multi-language source code

This code is ANSI C, PHP, & Bash:

```
#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
#*/
```

Code is public domain, but the source is unknown

Polyglots continued

POC||GTFO (Proof of Concept or [Please Kindly Leave]) has included a plethora of polyglot files into the .pdf

- Issue #2 is a PDF, ZIP, and a Bootable Operating System in the QEMU emulator.
- Issue #11 is a PDF, Ruby Code, and HTML

These are usually left for the reader to discover until the next issue, thus acting as a kind of steganography.

While we find these immensely interesting, they are tangential to our goal, as we want our output to need a specialized decoder to be able to get the output, not be able to open by simply changing the file extension.

Source[4]: https://www.sultanik.com/pocorgtfo/

Our Goal

- We want to hide the binary code of a file into a .wav file, then determine to what extent the payload can occupy the least significant bits of the waveform file before a noticeable change in the audio quality.
- We chose to program this project in python to use its libraries, specifically scipy and numpy.

Example sample point of `0xABCD` with payload byte of `0xFD`, into the 2 least significant bits:

Input: `0xABCF` = `0b1010101111001111`

Zero'd out 2 least significant bits = `0b1010101111001100`

`0xFD = 0b11111101`, grabbing just the last two bits, 0b01 for this sample:

Putting 2 bits of payload into input = `0b1010101111001101`

In the next sample, the next two bits of the payload would be inserted. Assuming the sample had the same value:

Putting 2 bits of payload into input = `0b1010101111001111`

How a Wav File Works

- A wav file is a raw audio format. This means that it is not compressed like an .mp3 would be.
- The wav file is made up of multiple individual samples as shown in the image below.

Bit depth deals with the range of amplitude values

that can be expressed.

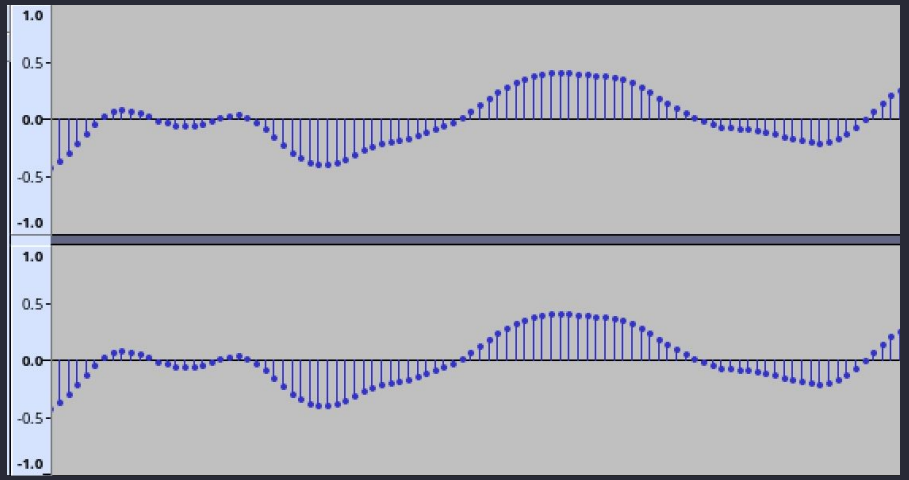
- 8-bit files have 256 amplitude levels
- 16-bit files have 65536 levels.

- The greater the file's resolution the greater the realistic dynamic range of the file, for example Most CD's use .wavs with 16-bit samples

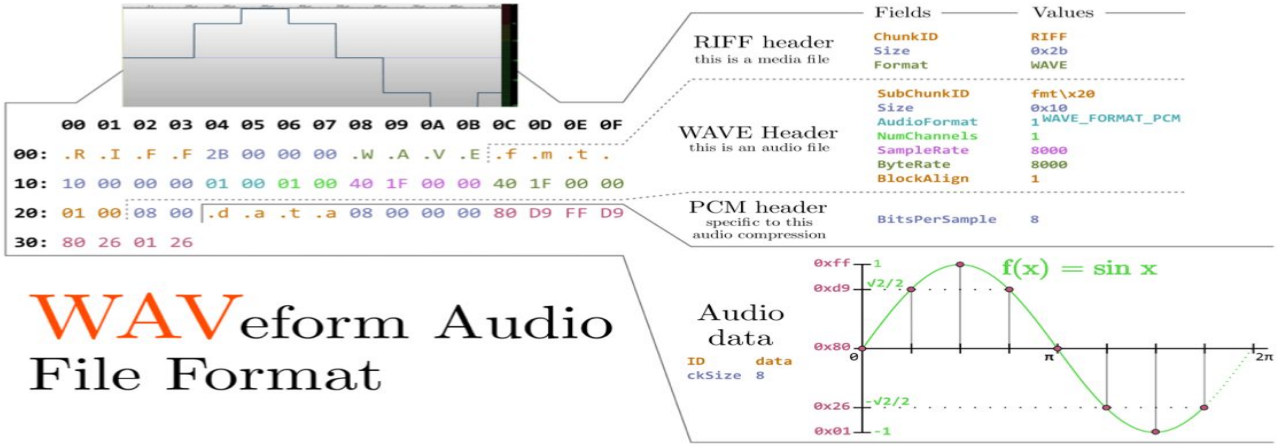
- Sample Rate is the number of samples per second.

- CDs have a sample rate of 44100 samples/second.

- For this project we only used a sample rate of 44100hz.



How a Wav File Works Continued



767

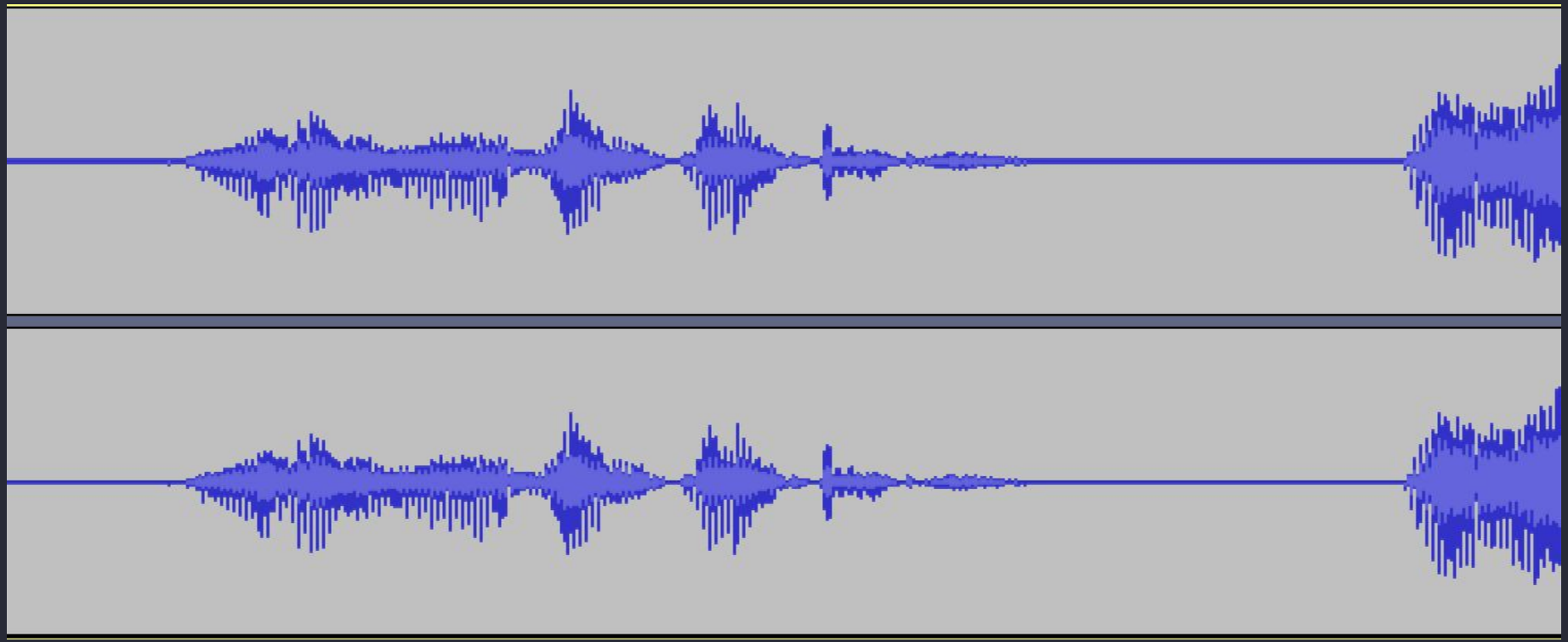
by Ange Albertini

Fig. 1. .wav format binary encoding [2]

M. Laphroaig, *PoC||GTFO*. No Starch Press, 2017.

Input 8-bit Wav

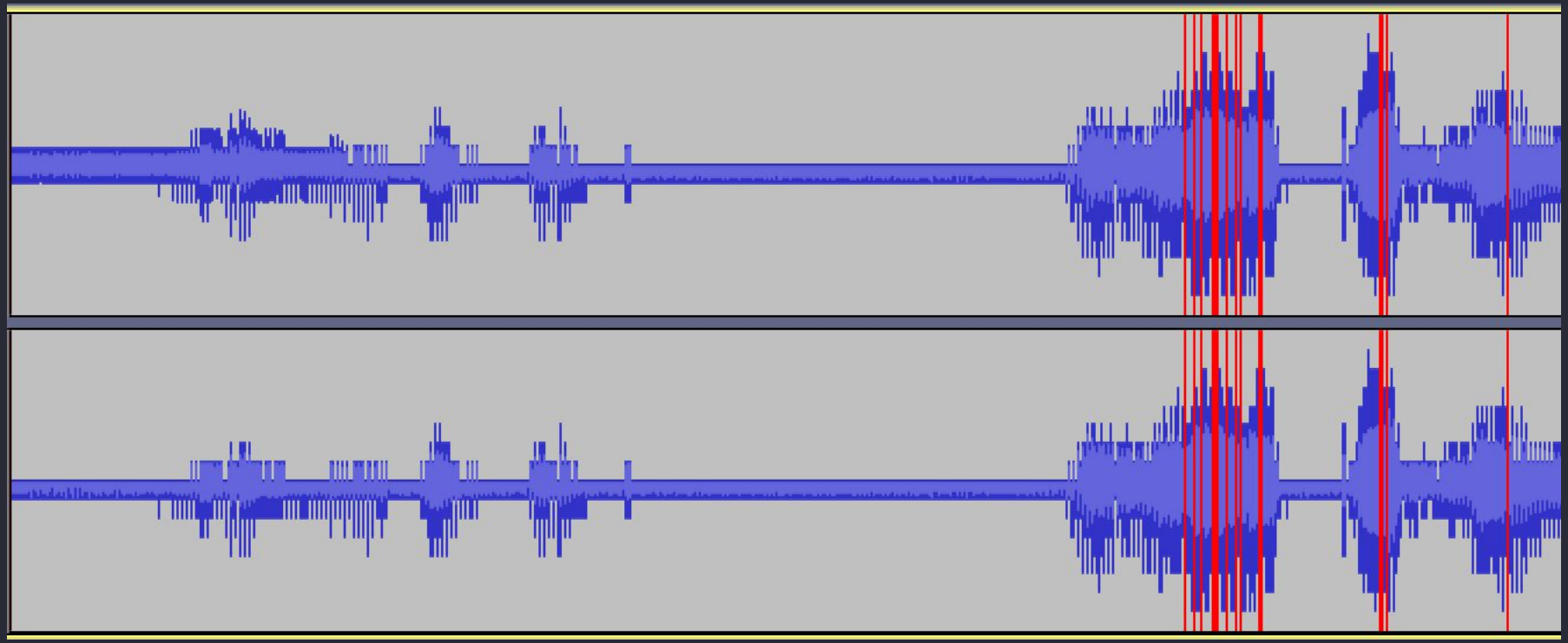
1-Bits for encoding, in 8-bit file



*It seems Google Slides may be tampering with our results, such that the defects sound different than in other software. Please listen to the files directly on your computer instead of the likely repeatedly re-compressed sound from these slides to hear the real results.

2-Bits for encoding, in 8-bit file

4-Bits for encoding, in 8-bit file



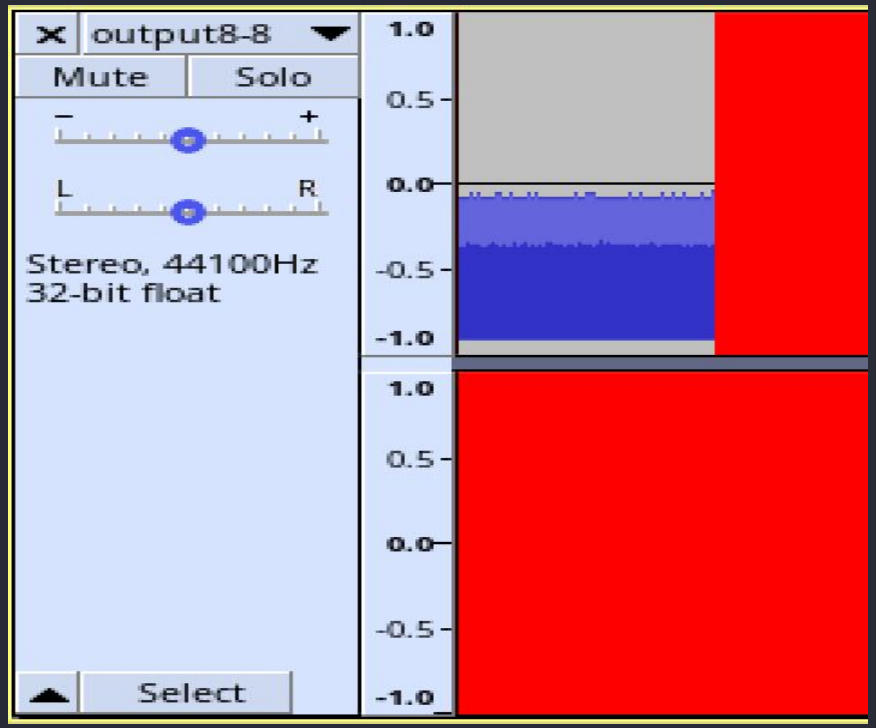
Higher Bit depths?

- We ran tests at every combination of 8, 16, and 24 bit input files with 1, 2, 4, and 8 bits used to store the payload.
- 24 bit performed **much** better than 8-bit, but the result can not be seen in the output waveform.

24 bit input .wav

8-bits for encoding, in 24-bit file

8-Bits for encoding, in 8-bit file



Obstacles in Writing Our Program

- Bitmath on a 24 bit data type is awkward, as they need to be 32-bit internally
  - Correct shifting depending on this bit depth
- Storing metadata causes a chicken-and-egg problem
  - How can you encode the number of bits that are used to store the secret data?
- Having to encode and decode

Code Samples, Featuring bit math.

```
if type(wav[0,0]) is np.int32:
    print("32bit internally, LSB's are off by a byte, recovery will need to be shifted")
    # First, we need to make a mask, then bitwise and out the bits we care about
    mask = np.empty_like(wav)
    mask.fill(0xffffffff)
    mask = np.left_shift(mask,num_bits+8) # +8 because of the 32 to 24 bit problem, if numbits 2 then 0x
    mask = np.invert(mask)                # make
    recovered_bits = np.bitwise_and(wav,mask)
    recovered_bits = np.right_shift(recovered_bits,8)
    recovered_bits = recovered_bits.astype(np.int8)
```

```
i = 0
k = 0
for false_byte in recovered_bits:
    # the bytes in recovered bits only contain a few bits, there's a bunch of 0's padding them.
    # this is where those sets of num_bits get turned back into bytes
    j = i%(8/num_bits)
    recovered_bytes[k,0] += int(false_byte[0]) << int(j*num_bits) #falsebyte[0] because only the left channel
    # If this mess doesn't make sense, uncomment this line and the print in the if below
    # print(f"false byte = {np.binary_repr(false_byte[0],width=2)} → recovered byte = {np.binary_repr(recovered_bytes[k,0],width=8)}")
    i+=1
    # if numbits is 2 we want j = 0,1,2,3
    if j == 8/num_bits - 1:
        k+=1
        i=0
```

```
vega@lyrae /mnt/data/School/Current/463_NetSec/project  <main*>
➤ python encoder.py
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.
```

Overall Project Conclusion

I. On a 24 bit waveform, even with 8 bits holding the secret data, the effect is inaudible and all but impossible to see in the waveform.
II. As the number of bits carrying the payload is increased so does the distortion.
III. The results are not very noticeable at 1-bit depth payload, at 2-bits the start of the wave gets thicker and so on with the 4-bit .wav reaching max or min values shown by the red lines on the wave in its previous image. These red lines give off audible clicks that may indicate something is wrong to the listener.
IV. Overall all of our goals were accomplished
    A. We were able to insert a payload into a wav file
    B. We could use up to one quarter of the initial file size of a 24 bit wav file for the payload
    C. Since wav files generally are so large even for small audio clips, a large amount of data can be hidden in short clips.

References

[1] https://www.merriam-webster.com/dictionary/steganography

[2] Abbas Cheddad, Joan Condell, Kevin Curran, Paul Mc Kevitt, Digital image steganography: Survey and analysis of current methods,

Signal Processing, Volume 90, Issue 3, 2010, Pages 727-752,

ISSN 0165-1684, https://doi.org/10.1016/j.sigpro.2009.08.010.

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

[4] https://www.sultanik.com/pocorgtfo/ (includes the list of polyglot files)

Any Questions?