SWGDE Test Method for Skimmer Forensics – Analog Devices

Disclaimer:

As a condition to the use of this document and the information contained therein, the SWGDE requests notification by e-mail before or contemporaneous to the introduction of this document, or any portion thereof, as a marked exhibit offered for or moved into evidence in any judicial, administrative, legislative or adjudicatory hearing or other proceeding (including discovery proceedings) in the United States or any Foreign country. Such notification shall include: 1) the formal name of the proceeding, including docket number or similar identifier; 2) the name and location of the body conducting the hearing or proceeding; 3) subsequent to the use of this document in a formal proceeding please notify SWGDE as to its use and outcome; 4) the name, mailing address (if available) and contact information of the party offering or moving the document into evidence. Notifications should be sent to secretary@swgde.org.

It is the reader's responsibility to ensure they have the most current version of this document. It is recommended that previous versions be archived.

Redistribution Policy:

SWGDE grants permission for redistribution and use of all publicly posted documents created by SWGDE, provided that the following conditions are met:

1. Redistribution of documents or parts of documents must retain the SWGDE cover page containing the disclaimer.
2. Neither the name of SWGDE nor the names of contributors may be used to endorse or promote products derived from its documents.
3. Any reference or quote from a SWGDE document must include the version number (or create date) of the document and mention if the document is in a draft status.

Requests for Modification:

SWGDE encourages stakeholder participation in the preparation of documents. Suggestions for modifications are welcome and must be forwarded to the Secretary in writing at secretary@swgde.org. The following information is required as a part of the response:

a) Submitter's name
b) Affiliation (agency/organization)
c) Address
d) Telephone number and email address
e) Document title and version number
f) Change from (note document section number)
g) Change to (provide suggested text where appropriate; comments not including suggested text will not be considered)
h) Basis for change

Intellectual Property:

Unauthorized use of the SWGDE logo or documents without written permission from SWGDE is a violation of our intellectual property rights.

Individuals may not misstate or over represent duties and responsibilities of SWGDE work. This includes claiming oneself as a contributing member without actively participating in SWGDE meetings; claiming oneself as an officer of SWGDE without serving as such; claiming sole authorship of a document; use the SWGDE logo on any material or curriculum vitae.

Any mention of specific products within SWGDE documents is for informational purposes only; it does not imply a recommendation or endorsement by SWGDE.

Table of Contents

SWGDE Test Method for Skimmer Forensics – Analog Devices

Table of Figures

SWGDE Test Method for Skimmer Forensics – Analog Devices

1 Purpose

Skimmers are magnetic card readers used to steal account information. Implanted on or into a device or carried on a person, skimmers collect personal (typically banking) information in an unauthorized manner. The purpose of this test method is to provide specific procedures required to analyze data contained on analog skimming devices.

2 Scope

This document is intended for computer forensic practitioners conducting skimming device forensic analysis on devices using an analog storage format. It does not provide best practices regarding skimmer examinations, chip/SD card extractions, digital skimmer analysis, Bluetooth® module extraction / analysis, or device repair as that data is available, at a minimum, in the following publications:

- S*WGDE Best Practices for Examining Magnet Card Readers;*
- *ASTM E3017-19 Standard Practice for Examining Magnetic Card Readers;*
- *SWGDE Best Practices for Computer Forensics Acquisitions;*
- *SWGDE Test Method for Bluetooth® Module Extraction and Analysis;*
- *SWGDE Test Method for Skimmer Forensics – Digital Devices;* and
- *SWGDE Embedded Device Core Competencies.*

3 Limitations

Skimmers present unique examination challenges due to:

- Rapid changes in technology;
- Countermeasures;
- Examiner unfamiliarity with data modulation; and
- Lack of commercially available software to analyze data extracted from skimmers.

4 Process Map

As an overview, the following map depicts the complete process for examining a skimming device to include any wireless components used for the exfiltration of data. This document focuses on the analog skimmer analysis column, post extraction.

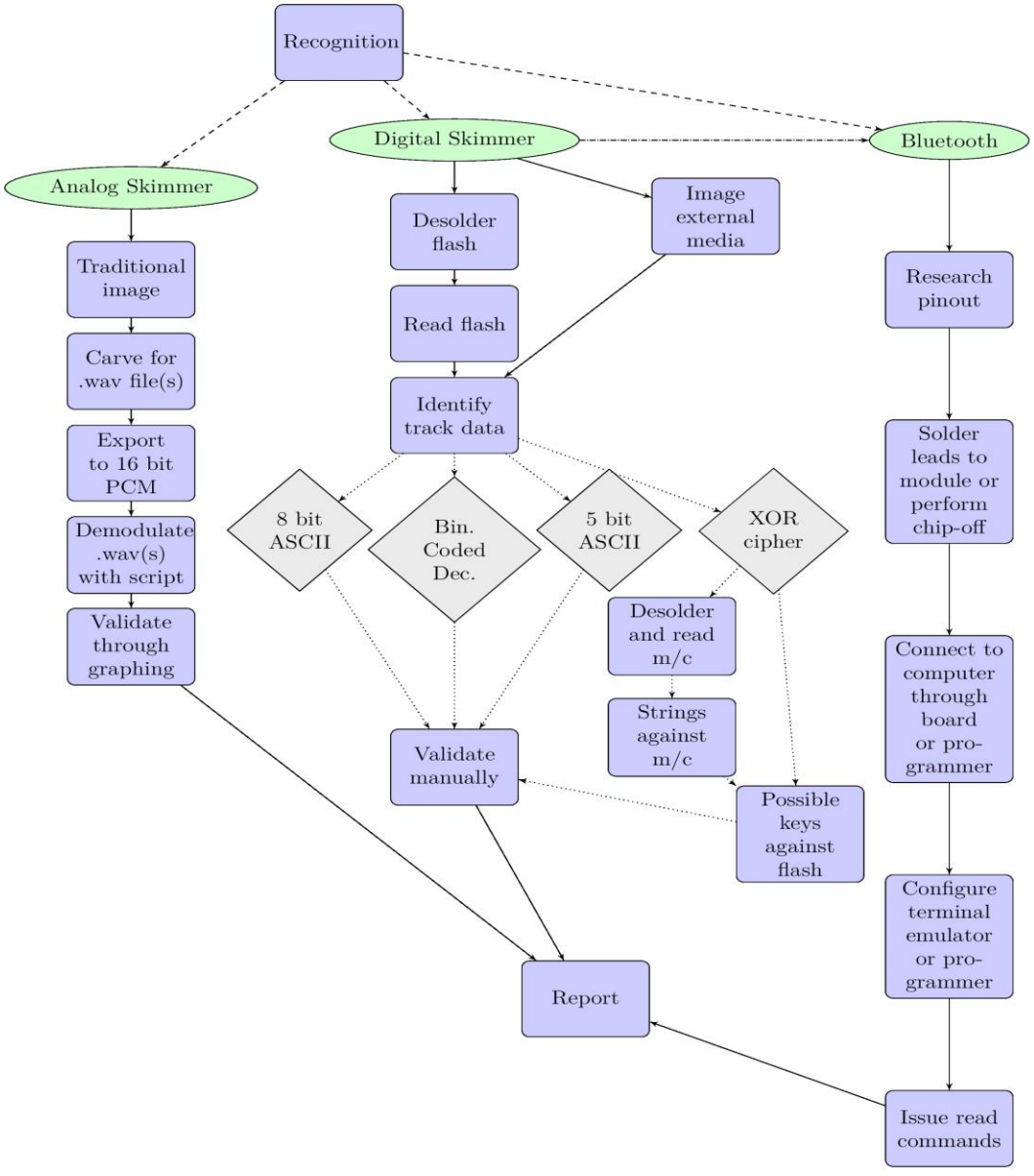

Figure 1- Process map

5 Analysis – Analog Skimmer

5.1 Image

Begin with an image extracted through USB. The device may be recovered without a USB header. However, as analog skimmers are typically cannibalized from digital audio players that were originally designed to connect to a computer via USB, re-attaching a USB header is typically all that is required to facilitate connection to an examination computer. Due to wear levelling concerns, do not attempt to extract data via chip-off processes.

5.2 Carve

Carve audio files (*.wav) from the image using a software tool that identifies and extracts individual types of files, e.g. a Waveform audio file, based on header and footer bytes that correspond with that type of file. Below is a screenshot of the configuration file in Scalpel that shows the wav file header that the tool will use for carving.

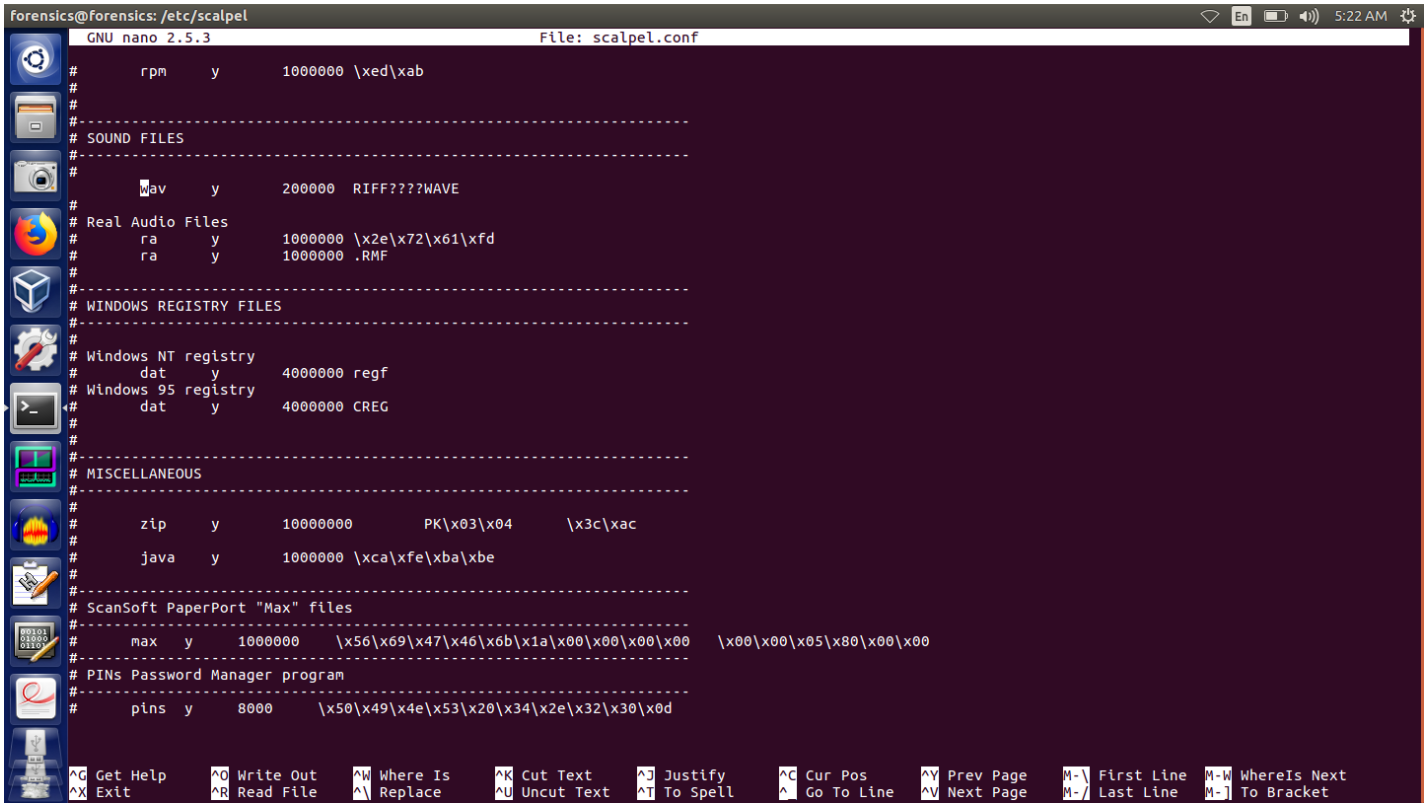

Figure 2- File carving

5.3 Visualize

Open the carved file(s) in audio editing software, taking note of the number of swipes. Below is a screenshot of swipes carved from a file extracted from an analog skimmer.

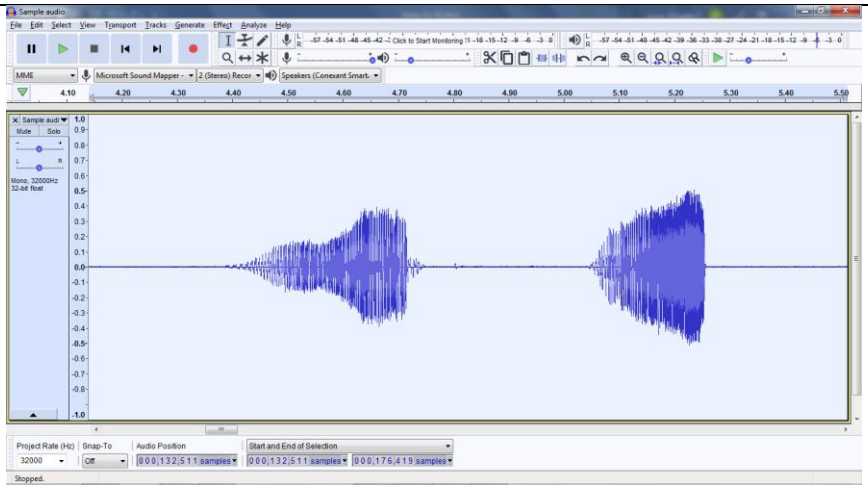

Figure 3- Analog swipes

5.4 Demodulate

5.4.1 Visualize the frequency key shifting, also known as Atkins Biphase;

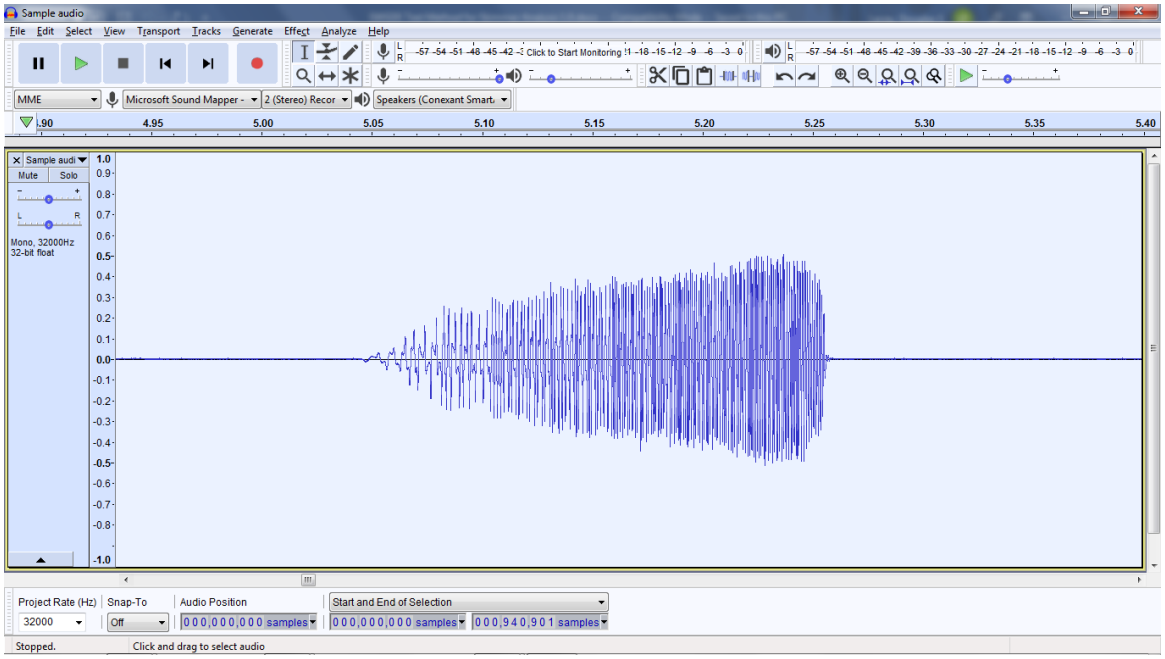

Figure 4 - Frequency shifting

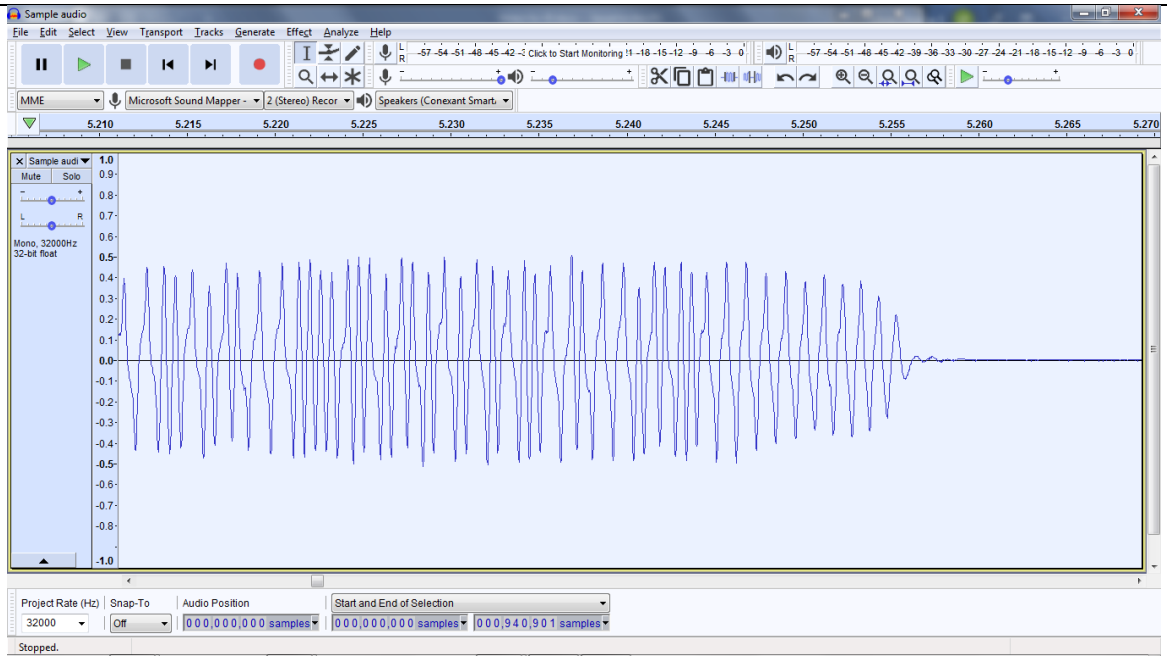

Figure 5 - Shifting, zoomed-in

5.4.1.1 The above (Figure 5) represents zoomed-in views of one of the carved swipes. The frequency of the wave changes as 1s and 0s are encountered. Regardless of the density of data and the speed at which the mag stripe is swept (assuming a linear speed) passed the read head, 1s are represented at twice the frequency as the 0s. On a financial card mag stripe, the data will represent a series of zeros followed by a start sentinel, 0b11010. One can start from the right (card readers work in either direction, so in this situation the card

was swiped in a manner that saved the data form right to left vs left to right) and begin to visualize the frequency changes as they pass;

5.4.1.2 Zooming further into the start sentinel and the beginning of the fictitious account number:

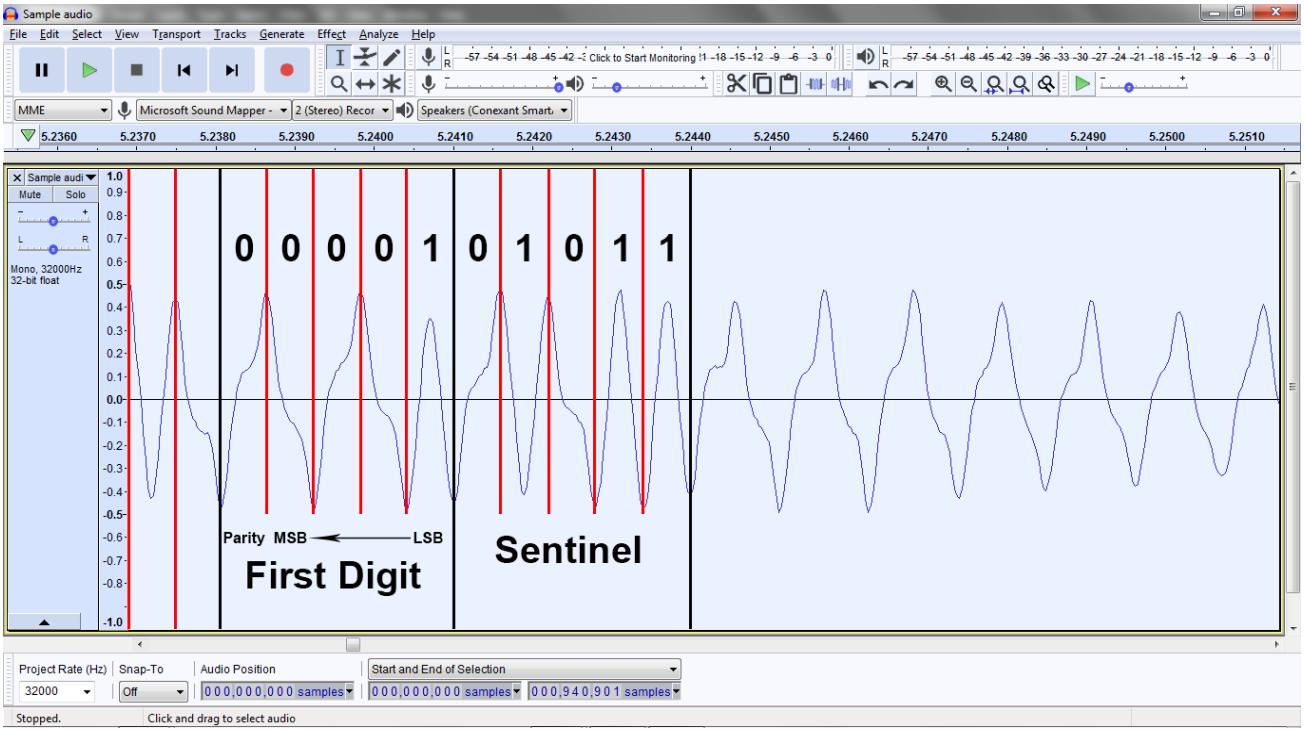

Figure 6 - Wave with binary values

A rectified (turns negative sample values below the x axis to positive) view to assist in visual recognition of the bits:

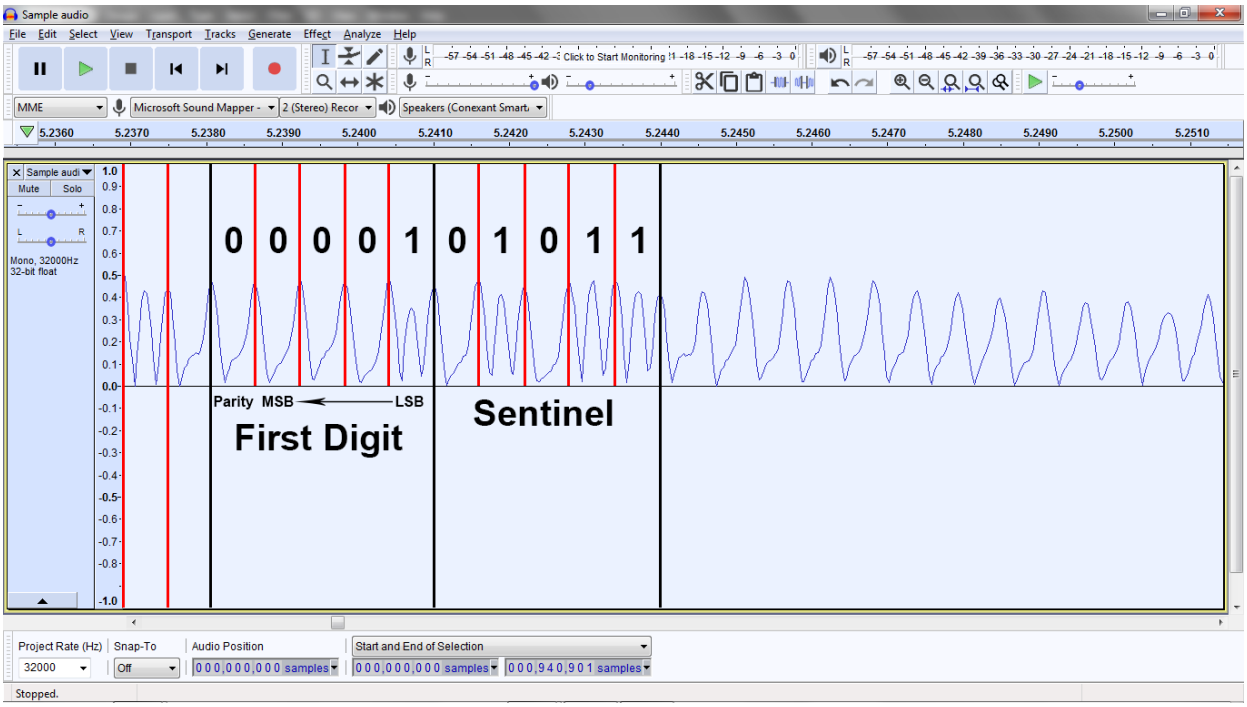

Figure 7 - Rectified View

5.4.1.3 From this point one would make note of all the binary values for all of the swipes.

5.5 Resolve

With the wav file now demodulated, resolve the bits according to 5-bit binary (four bits to account for the numerical data plus one parity bit) to recover the account number;

5.5.1 The encoding scheme for account numbers is 5-bit binary, the resolution is as follows:

5.5.1.1 Big endian: 8s 4s 2s 1s | Parity

5.5.1.2 Little endian: 1s 2s 4s 8s | Parity

5.5.2 Using the fictitious account number from above: …000110101000001000110010010000:

5.5.2.1 11010 – Start Sentinel
10000 – 1
01000 – 2
11001 – 3
00100 – 4
0…

- As such, this account number is encoded in 5-bit binary little endian, and the account number begins with 1234.

5.6 Automate

Once the data is found to conform to Atkins Biphase and the wav file demodulated, one may choose to automate the resolution of data to account numbers by scripting. Appendix 1 is a Python 2.7 script suite used to resolve the data.

6 Appendix 1 - Python Scripts

6.1 Main Script

```
#!/usr/bin/env python2

import re, sys
from struct import unpack
from cc_verify import check_luhn
from track_decode import print_report_track1
from track_decode import print_report_track2
from track_carve import process_track1_data
from track_carve import process_track2_data

DEFAULT_WINDOW_SIZE = 4
DEFAULT_NOISE_THRESHOLD = 0.1
DEFAULT_TIME_THRESHOLD  = 400
DEFAULT_STARTING_FRAME = 0
DEFAULT_NUMBER_OF_FRAMES = -1
DEFAULT_TRACK_NUMBER = 2

__date__          = '5-10-2011'
__version__       = '1.1'
__doc__           = "\n\
  Usage:\n\
    %s [actions] [options] wave_file_name\n\
    \n\
  Actions:\n\
    -h, --help   Display this help message\n\
    -V           Show version information\n\
    \n\
  Options:\n\
    -s frame_number  Specify starting frame [default all]\n\
    -n number        Specify number of frames to graph [default all]\n\
    -g              Show a graph of the data\n\
    -l threshold    Specify a noise threshold between 0 and 1 [default %f]\n\
    -r report_name  Specify a file to write the report to\n\
    -t threshold    Specify a time threshold [default %d]\n\
    -T track_number  Specify which track format to search (1 or 2) [default %d]\n\
    -w window       Specify a window size [default %d]" % (sys.argv[0],
DEFAULT_NOISE_THRESHOLD, DEFAULT_TIME_THRESHOLD, DEFAULT_TRACK_NUMBER,
DEFAULT_WINDOW_SIZE)


def extract_bits(minmax_times, window_size):
    # convert to distance between min/max
    minmax_times = [abs(minmax_times[lcv+1] - minmax_times[lcv]) for lcv in
range(len(minmax_times)-1)]
    #print minmax_times
    window = []
    bits = []
```

```
    skip = 0
    for lcv in range(len(minmax_times)):
        if skip:
            skip = 0
        else:
            if (len(window) < window_size):
                # fill up the window with enough samples first
                window.insert(0, minmax_times[lcv])
                bits.append(0)
            else:
                window_avg = float(sum(window))/len(window)
                #print 'Window:', window
                #print 'Window avg: ', window_avg
                #print 'Current %0.2f (%0.2f)' % (minmax_times[lcv],
float(minmax_times[lcv]) / window_avg)
                if minmax_times[lcv] < 4 * window_avg:
                    if minmax_times[lcv] > 0.80 * window_avg:
                        # strong zero
                        #print '======> 0'
                        window.insert(0, minmax_times[lcv])
                        bits.append(0)
                    elif minmax_times[lcv] < 0.70 * window_avg:
                        # strong one
                        #print '======> 1'
                        if lcv != len(minmax_times) - 1:
                            window.insert(0, minmax_times[lcv] +
minmax_times[lcv+1])
                            #window.insert(0,
(minmax_times[lcv]+minmax_times[lcv+1])/2)
                        bits.append(1)
                        skip = 1
                    else:
                        if lcv >= len(minmax_times) - 1:
                            break
                        # we need to look at the next center crossing
                        # to help us determine this one
                        next_window_avg =
float(sum([minmax_times[lcv+1],]+window[:-1]))/len(window)
                        if minmax_times[lcv+1] > 0.75 * next_window_avg:
                            # next is whole freq, then this is a zero
                            #print '======> 0?'
                            bits.append(0)
                            window.insert(0, minmax_times[lcv])
                        else:
                            # next is half freq... we don't really know
                            # go with the best guess for this center crossing
                            if minmax_times[lcv] > 0.75 * window_avg:
                                #print '======> 0??'
                                window.insert(0, minmax_times[lcv])
                                bits.append(0)
                            else:
                                #print '======> 1??'
                                if lcv != len(minmax_times) - 1:
```

```
                                        window.insert(0, minmax_times[lcv] +
minmax_times[lcv+1])
                                    bits.append(1)
                                    skip = 1
                        window.pop()
    return bits

def five_bits_to_ascii(bits):
    if bits == '00001': return '0' # (0H) Data
    if bits == '10000': return '1' # (1H)
    if bits == '01000': return '2' # (2H)
    if bits == '11001': return '3' # (3H)
    if bits == '00100': return '4' # (4H)
    if bits == '10101': return '5' # (5H)
    if bits == '01101': return '6' # (6H)
    if bits == '11100': return '7' # (7H)
    if bits == '00010': return '8' # (8H)
    if bits == '10011': return '9' # (9H)
    if bits == '01011': return ':' # (AH) Control
    if bits == '11010': return ';' # (BH) Start Sentinel
    if bits == '00111': return '<' # (CH) Control
    if bits == '10110': return '=' # (DH) Field Separator
    if bits == '01110': return '>' # (EH) Control
    if bits == '11111': return '?' # (FH) End Sentinel<>
    return ''

def decode_bits(bits):
    """ Returns a list of possible decoded track 2 data
    """
    bits = ''.join(['%d'%x for x in bits])
    data = ''
    start = bits.find('11010')
    datas = []
    while start != -1:
        data = ''
        for lcv in range(start, len(bits), 5):
            char = five_bits_to_ascii(bits[lcv:lcv+5])
            if char == '':
                datas.append(data)
                break
            data += char
        start = bits.find('11010', start + 1)
    return datas

def force_decode_bits(bits):
    """ Returns a list of decoded track 2 data with '?' as
        invalid characters
    """
    bits = ''.join(['%d'%x for x in bits])
    data = ''
    for lcv in range(0, len(bits), 5):
        char = five_bits_to_ascii(bits[lcv:lcv+5])
        if char == '': char = '?'
```

```
        data += char
    return data

def carve_cc(wave_stream,
             starting_frame = DEFAULT_STARTING_FRAME,
             number_of_frames = DEFAULT_NUMBER_OF_FRAMES,
             noise_threshold = DEFAULT_NOISE_THRESHOLD,
             time_threshold = DEFAULT_TIME_THRESHOLD,
             window_size = DEFAULT_WINDOW_SIZE,
             track_number = DEFAULT_TRACK_NUMBER,
             graph = None,
             report_stream = None):
    samples = []
    minmax_times = []

    wave_stream.rewind()
    if (number_of_frames == -1) or (number_of_frames > wave_stream.getnframes() -
starting_frame):
        number_of_frames = wave_stream.getnframes()
    wave_stream.setpos(starting_frame)
    # extract the times between mininums and maximums (or maximums and minimums)
    if wave_stream.getsampwidth() == 1:
        center = 128
        maximum = 127
    elif wave_stream.getsampwidth() == 2:
        center = 0
        maximum = 32767

    prev_data = center + 1

    # while we have unread wave data
    while wave_stream.tell() != (number_of_frames + starting_frame):
        # search for crossing of center
        data = center
        while abs(data - center) < noise_threshold * maximum and
wave_stream.tell() != (number_of_frames + starting_frame):
            if wave_stream.getsampwidth() == 1:
                data = unpack('B', wave_stream.readframes(1))[0]
            elif wave_stream.getsampwidth() == 2:
                data = unpack('h', wave_stream.readframes(1))[0]

        data = [data, data,]
        if wave_stream.tell() != (number_of_frames + starting_frame):
            while (data[-1]-center)*(data[-2]-center) > 0 and wave_stream.tell()
!= (number_of_frames + starting_frame):
                if wave_stream.getsampwidth() == 1:
                    data.append(unpack('B', wave_stream.readframes(1))[0])
                elif wave_stream.getsampwidth() == 2:
                    data.append(unpack('h', wave_stream.readframes(1))[0])

            # find the min/max
            if data[0] < center:
                # min found
```

```
                #print wave_stream.tell() - len(data) + data.index(min(data)),
                #print min(data)
                samples.append(wave_stream.tell() - len(data) +
data.index(min(data)))
            else:
                # max found
                #print wave_stream.tell() - len(data) + data.index(min(data)),
                #print max(data)
                samples.append(wave_stream.tell() - len(data) +
data.index(max(data)))

    # separate bit groups by threshold time
    group = []
    for sample in samples:
        if group == []:
            group = []
            group.append(sample)
        elif sample - group[-1] < time_threshold:
            group.append(sample)
        else:
            # too long between peaks, start new bit group
            if group != []:
                minmax_times.append(group)
            group = []
            group.append(sample)
    if group != []:
        minmax_times.append(group)

    # extract the bits from the minimum/maximum times
    # reverse for a reverse swipe
    bits = []
    for each in minmax_times:
        bits.append((extract_bits(each, window_size), extract_bits(each[::-1],
window_size))) # original
        #bits.append((extract_bits(each, window_size), extract_bits(each,
window_size)[::-1]))
        #print extract_bits(each, window_size)
        #print extract_bits(each, window_size)[::-1]
        #print ''

    # decode the information
    info = []
    for each in bits:
        if track_number == 1:
            info.extend(process_track1_data(''.join(['%d'%x for x in each[0]]),
convert_to_binary=False))
        else:
            info.extend(process_track2_data(''.join(['%d'%x for x in each[0]]),
convert_to_binary=False))
    if report_stream:
        print 'Generating Report...'
    if track_number == 1:
        print_report_track1(info, print_stream=report_stream)
```

```
    else:
        print_report_track2(info, print_stream=report_stream)
    #else:
    #    if track_number == 1:
    #        print_report_track1(info)
    #    else:
    #        print_report_track2(info)
    info = [x[0] for x in info]

    #print 'Peaks:'
    #print minmax_times
    #print 'Bits:'
    #print bits
    #print 'Hex:'
    #print [[''.join([chr(int(''.join([str(x) for x in (seq[8*i:8*(i+1)]) +
['0',]*(8-len(seq[8*i:8*(i+1)]))]), 2)) for i in range(len(seq)/8 +
(7+len(seq)%8)/8)]) for seq in pair] for pair in bits]
    #print 'Decoded info:'
    #print info

    '''
    for shift in range(5):
        for b in bits:
            if len(b[0]) > shift:
                print 'Raw decoded forward bits (shift %d):' % shift
                print force_decode_bits(b[0][shift:])
            if len(b[1]) > shift:
                print 'Raw decoded reverse bits (shift %d):' % shift
                print force_decode_bits(b[1][shift:])
    '''

    if graph:
        print("Generating Graph...")
        from graph_wave import graph_wave, graph_minmax, graph_bits
        import pylab as p
        ax = p.subplot(111)
        #ax = p.subplot(211)
        p1 = graph_wave(wave_stream,
                        number_of_frames = number_of_frames,
                        starting_frame = starting_frame,
                        noise_threshold = noise_threshold,
                        p = ax)
        if wave_stream.getsampwidth() == 1: p1.set_yticks([0,255])
        elif wave_stream.getsampwidth() == 2: p1.set_yticks([-32768,32767])
        p1.set_yticklabels(['-1.0', '1.0'])
        p1.set_xticks([starting_frame, starting_frame+number_of_frames])
        p1.set_xticklabels([str(0), str(number_of_frames)])
        p1.set_xlim((starting_frame, starting_frame+number_of_frames))
        p1.set_title('Waveform and Demodulated Bit Stream')
        #p1.set_xlabel('Sample Number')
        p1.set_ylabel('Magnitude')

        # [y for x in bits for y in x]
```

```
        #ax = p.subplot(212)
        #p2 = ax
        for lcv in range(len(minmax_times)):
            p = graph_minmax(wave_stream, minmax_times[lcv], frames =
wave_stream.getnframes(), p = p)
            p = graph_bits(wave_stream, bits[lcv][0], minmax_times[lcv],
number_of_frames = number_of_frames, starting_frame = starting_frame, p = p)
            #p2 = graph_bits(wave_stream, bits[lcv][0], minmax_times[lcv],
number_of_frames = number_of_frames, starting_frame = starting_frame, p = p2)
            #p = graph_bits(wave_stream, bits[lcv][1][::-1], minmax_times[lcv],
frames = wave_stream.getnframes(), p = p)

        #if wave_stream.getsampwidth() == 1: p2.set_yticks([0,255])
        #elif wave_stream.getsampwidth() == 2: p2.set_yticks([-32768,32767])
        #p2.set_yticklabels(['0', '1'])
        #p2.set_xticks([starting_frame, starting_frame+number_of_frames])
        #p2.set_xticklabels([str(0), str(number_of_frames)])
        #p2.set_xlim((starting_frame, starting_frame+number_of_frames))

        #p2.set_title('Demodulated Waveform Bit Stream')

        #p2.set_xlabel('Sample Number')
        #p2.set_ylabel('Bit Value')

        #p.axis('tight')
        #p.legend()
        p.show()

def main(args):
    import getopt
    try:
        wave_file_name = args[-1]
        # note: args[1:] is not typical because first arg is always a file_name
        opts, args = getopt.getopt(args[0:], 'ghl:n:o:r:s:t:T:Vw:', ['help',])
    except getopt.GetoptError, ex:
        print >> sys.stderr, 'Invalid argument: ' + str(ex)
        print >> sys.stderr, __doc__
        sys.exit(1)
    except IndexError:
        print >> sys.stderr, 'Please specify an image file to analyze...'
        print >> sys.stderr, __doc__
        sys.exit(1)

    def badArguments():
        # print out usage if bad arguments are supplied
        print >> sys.stderr, 'Bad command line argument formation'
        print >> sys.stderr, __doc__
        sys.exit(1)

    window_size      = DEFAULT_WINDOW_SIZE
    starting_frame   = DEFAULT_STARTING_FRAME
    number_of_frames = DEFAULT_NUMBER_OF_FRAMES
    noise_threshold  = DEFAULT_NOISE_THRESHOLD
```

```
    time_threshold   = DEFAULT_TIME_THRESHOLD
    track_number     = DEFAULT_TRACK_NUMBER
    graph = None
    report_stream = None

    # action parsing
    for opt, arg in opts:
        # Short options
        if opt in ['-h', '--help']:
            print __doc__
            sys.exit(1)
        elif opt == '-V':
            print sys.argv[0] + ', Version ' + __version__
            sys.exit(1)

    # option parsing
    for opt, arg in opts:
        if opt == '-g':
                graph = True
        elif opt == '-l':
            try:
                noise_threshold = float(arg)
                assert 0. < noise_threshold < 1.
            except:
                print >> sys.stderr, 'Noise threshold must be a floating point
number between 0 and 1 (got "%s")...' % arg
                print >> sys.stderr, __doc__
                sys.exit(1)
        elif opt == '-n':
            try:
                number_of_frames = int(arg)
            except:
                print >> sys.stderr, 'Number of frames must be an integer (got
"%s")...' % arg
                print >> sys.stderr, __doc__
                sys.exit(1)
        elif opt == '-r':
            try:
                report_stream = open(arg, 'w')
            except:
                print >> sys.stderr, 'Could not open report output file "%s"...' %
arg
                print >> sys.stderr, __doc__
                sys.exit(1)
        elif opt == '-s':
            try:
                starting_frame = int(arg)
            except:
                print >> sys.stderr, 'Starting frame number must be an integer
(got "%s")...' % arg
                print >> sys.stderr, __doc__
                sys.exit(1)
        elif opt == '-t':
```

```
            try:
                time_threshold = int(arg)
            except:
                print >> sys.stderr, 'Time threshold must be an integer (got
"%s")...' % arg
                print >> sys.stderr, __doc__
                sys.exit(1)
        elif opt == '-T':
            try:
                track_number = int(arg)
                if not(track_number == 1 or track_number == 2):
                    print >> sys.stderr, 'Track number must be equal to either 1
or 2 (got "%s")...' % arg
                    print >> sys.stderr, __doc__
                    sys.exit(1)
            except SystemExit:  # needed due to how sys.exit() works
                sys.exit(1)
            except:
                print >> sys.stderr, 'Track number must be an integer (got
"%s")...' % arg
                print >> sys.stderr, __doc__
                sys.exit(1)
        elif opt == '-w':
            try:
                window_size = int(arg)
                assert window_size > 0
            except:
                print >> sys.stderr, 'Window size must be a positive integer (got
"%s")...' % arg
                print >> sys.stderr, __doc__
                sys.exit(1)

    try:
        import wave
        wave_stream = wave.open(wave_file_name, 'rb')
    except IOError:
        print >> sys.stderr, 'Could not open wave file "%s"...' % wave_file_name
        print >> sys.stderr, __doc__
        sys.exit(1)

    # let's do it
    carve_cc(wave_stream,
             starting_frame = starting_frame,
             number_of_frames = number_of_frames,
             noise_threshold = noise_threshold,
             time_threshold = time_threshold,
             window_size = window_size,
             track_number = track_number,
             graph = graph,
             report_stream = report_stream)

    if report_stream:
        report_stream.close()
```

```
if __name__ == '__main__':
    try:
        import psyco
        psyco.full()
    except ImportError:
        pass
        #print 'Psyco not installed, the program will just run slower...'

    main(sys.argv[1:])
```

6.2 Dependencies - Luhn Check

```
def check_luhn(ccnumber):
    num = [int(x) for x in str(ccnumber)]
    return sum(num[::-2] + [sum(divmod(d * 2, 10)) for d in num[-2::-2]]) % 10 ==
0
```

6.3 Dependencies - Report

6.3.1 Track 1

```
ef print_report_track1(unsorted_ccs, print_binary=False, print_stream=None):
    """Print out the reults from parsing the data for track 1 PANs.

    Arguments:
        unsorted_ccs: (list of tuples) the list of PANs found by the
            PAN parsing methods
        print_binary: (boolean) whether or not to print the raw binary
            of the verified PANs. Defaults to False
    """
    # printstream management
    temp_stream = None
    if print_stream:
        # save the print stream currently in use
        temp_stream = sys.stdout
        # overwrite the print stream to use
        sys.stdout = print_stream
    # if nothing found, print that and return to avoid errors
    if len(unsorted_ccs) == 0:
        print('\nNo track 1 PANs found.')
        return
    # initialize the array for validating PANs
    file_lines = cc_verify.read_file()
    # used for printing out overall stats
    all_count = 0
    all_verified_count = 0
    all_ccs = set()
    all_verified = set()
    # sort the PANs, then loop to print all PANs from each group
    sorted_ccs = sort_ccs(unsorted_ccs, 1)
```

```
    sorted_keys = sorted_ccs.keys()
    first_loop = True  # to make output look a bit cleaner
    for cc_key in sorted_keys:  # loop group-by-group
        sort_data = sorted_ccs[cc_key]
        # print out a line of *'s between each group for readability
        if not first_loop:
            print
'\n*********************************************************************************
***'
            print
'*********************************************************************************
*'
        first_loop = False
        # print out the string of transformations
        print '\n%s' % sort_data[0]
        ccs = sort_data[1:]
        all_count += len(ccs)
        # now check each PAN within the group
        print('\nPossible PANs found:')
        print('\t%20s\t%25s\t%5s' % ('PAN', 'Name', 'Expiration Date'))
        for cc in ccs:
            all_ccs.add(cc[0])
            print('\t%20s\t%25s\t%5s' % (cc[0], cc[1], cc[2]))
        print('\nUnique Luhn-verified PANs found:')
        if not print_binary:
            print('\t%20s\t%25s\t%25s\t%5s\t%5s' % ('PAN', 'Issuer', 'Name', 'Exp
Date', 'Count'))
        else:
            print('\t%20s\t%25s\t%25s\t%5s\t%5s\t%10s' % ('PAN', 'Issuer', 'Name',
'Exp Date', 'Count', 'Binary PAN'))
        unique_ccs = list(set(ccs))
        # Find the unique PAN count while working around duplicate
        # numbers with different binaries (which are not removed
        # by the previous line of code)
        unique_cc_count = len(set([cc[0] for cc in ccs]))
        total_verified = 0
        unique_verified = set()
        for cc in unique_ccs:
            card_data = cc_verify.verify_cc(cc[0], file_data=file_lines)
            if card_data:  # is not None
                all_verified_count += ccs.count(cc)
                all_verified.add(cc[0])
                iss = card_data["Scheme"]
                total_verified += ccs.count(cc)
                unique_verified.add(cc[0])
                if not print_binary:
                    #                                          PAN, issuer, name,
expire,    count
                    print('\t%20s\t%25s\t%25s\t%8s\t%5s' % (cc[0], iss, cc[1],
cc[2], ccs.count(cc)))
                else:
                    #                                               PAN, issuer,
name, expire,    count,     binary
```

```
                    print('\t%20s\t%25s\t%25s\t%8s\t%5s\t%-1s' % (cc[0], iss,
cc[1], cc[2], ccs.count(cc), cc[3]))
        print('\n')
        print('Total possible PANs found:                %s' % len(ccs))
        print('Total possible unique PANs found:         %s' % unique_cc_count)
        print('Total Luhn-verified PANs:                 %s' % total_verified)
        print('Total unique Luhn-verified PANs:          %s' %
len(unique_verified))
    # now print out the overall stats
    print
'\n*******************************************************************************
***'
    print
'*******************************************************************************
*'
    print '\nOverall statistics for all found PANs:'
    print('\nUnique possible PANs found:')
    for cc in all_ccs:
        print('\t' + cc)
    print('\nUnique Luhn-verified PANs found:')
    for cc in all_verified:
        print('\t' + cc)
    print('\n')
    print('Total possible PANs found:                %s' % all_count)
    print('Total possible unique PANs found:         %s' % len(all_ccs))
    print('Total Luhn-verified PANs:                 %s' % all_verified_count)
    print('Total unique Luhn-verified PANs:          %s' % len(all_verified))
    # restore the previous print stream if it was overwritten
    if print_stream:
        sys.stdout = temp_stream
```

6.3.2 Track 2

```
def print_report_track2(unsorted_ccs, print_binary=False, print_stream=None):
    """Print out the reults from parsing the data for track 1 PANs.

    Arguments:
        unsorted_ccs: (list of tuples) the list of PANs found by the
            PAN parsing methods
        print_binary: (boolean) whether or not to print the raw binary
            of the verified PANs. Defaults to False
    """
    # printstream management
    temp_stream = None
    if print_stream:
        # save the print stream currently in use
        temp_stream = sys.stdout
        # overwrite the print stream to use
        sys.stdout = print_stream
    # if nothing found, print that and return to avoid errors
```

```
    if len(unsorted_ccs) == 0:
        print('\nNo track 2 PANs found.')
        return
    # initialize the array for validating PANs
    file_lines = cc_verify.read_file()
    # sets used for printing out overall stats
    all_count = 0
    all_verified_count = 0
    all_ccs = set()
    all_verified = set()
    # sort the PANs, then loop to print all PANs from each group
    sorted_ccs = sort_ccs(unsorted_ccs, 2)
    sorted_keys = sorted_ccs.keys()
    first_loop = True  # to make output look a bit cleaner
    for cc_key in sorted_keys:  # loop group-by-group
        sort_data = sorted_ccs[cc_key]
        # print out a line of *'s between each group for readability
        if not first_loop:
            print
'\n*********************************************************************************
***'
            print
'*********************************************************************************
*'
        first_loop = False
        # print out the string of transformations
        print '\n%s' % sort_data[0]
        ccs = sort_data[1:]
        all_count += len(ccs)
        # now check each PAN within the group
        print('\nPossible PANs found:')
        for cc in ccs:
            all_ccs.add(cc[0])
            print('\t' + str(cc[0]))
        print('\nUnique Luhn-verified PANs found:')
        if not print_binary:
            print('\t%20s\t%25s\t%5s' % ('PAN', 'Issuer', 'Count'))
        else:
            print('\t%20s\t%25s\t%5s\t%10s' % ('PAN', 'Issuer', 'Count', 'Binary
PAN'))
        unique_ccs = list(set(ccs))
        # Find the unique PAN count while working around duplicate
        # numbers with different binaries (which are not removed
        # by the previous line of code)
        unique_cc_count = len(set([cc[0] for cc in ccs]))
        total_verified = 0
        unique_verified = set()
        for cc in unique_ccs:
            card_data = cc_verify.verify_cc(cc[0], file_data=file_lines)
            if card_data:  # is not None
                all_verified_count += ccs.count(cc)
                all_verified.add(cc[0])
                iss = card_data["Scheme"]
```

```
                total_verified += ccs.count(cc)
                unique_verified.add(cc[0])
                if not print_binary:
                    #                                PAN, issuer,   count
                    print('\t%20s\t%25s\t%5s' % (cc[0], iss, ccs.count(cc)))
                else:
                    #                                     PAN, issuer,  count,
binary
                    print('\t%20s\t%25s\t%5s\t%-1s' % (cc[0], iss, ccs.count(cc),
cc[1]))
        print('\n')
        print('Total possible PANs found:                %s' % len(ccs))
        print('Total possible unique PANs found:         %s' % unique_cc_count)
        print('Total Luhn-verified PANs:                 %s' % total_verified)
        print('Total unique Luhn-verified PANs:          %s' %
len(unique_verified))
```

6.4 Dependencies - Process

6.4.1 Track 1

```
def process_track1_data(data, convert_to_binary=True):
    """Return all track 1 PANs found in a chunk of binary data.

    Arguments:
        data: (string) data to parse for track 1 encoded PANs
        convert_to_binary: (boolean) whether or not to convert data into
            a binary string. Added since wav2cc.py passes binary string
            for data argument. Default: True

    Returns:
        list of tuples
    """
    records = []
    if convert_to_binary:
        bin_data = track_decode.get_binary_string(data)
    else:
        bin_data = data

    # check forward-swipe and backward-swipe little endian 5 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
reverse_swipe=True))

    # check forward-swipe and backward-swipe little endian 4 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data, parity=False))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
parity=False, reverse_swipe=True))

    # check forward-swipe and backward-swipe big endian 5 bit ascii
```

```
    records.extend(track_decode.search_track1_bin_string(bin_data,
reverse_endian=True))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
reverse_endian=True, reverse_swipe=True))

    # check forward-swipe and backward-swipe big endian 4 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data, parity=False,
reverse_endian=True))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
parity=False, reverse_endian=True, reverse_swipe=True))

    # check for forward-swipe binary-coded decimal

records.extend(track_decode.binary_coded_decimal_track1(binascii.hexlify(data)))
    # check for backward-swipe binary-coded decimal

records.extend(track_decode.binary_coded_decimal_track1(binascii.hexlify(data[::-
1])))

    # checks for ascii encoded data big_endian
    records.extend(track_decode.ascii_parse_track1(data, little_endian=False))
    # checks for ascii encoded data little_endian
    records.extend(track_decode.ascii_parse_track1(data, little_endian=True))

    # reverse the bit order in each byte
    bin_data = track_decode.get_binary_string_reverse_endian(data)

    # check forward-swipe and backward-swipe little endian 5 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data,
data_reverse_endian=True))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
reverse_swipe=True, data_reverse_endian=True))

    # check forward-swipe and backward-swipe little endian 4 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data, parity=False,
data_reverse_endian=True))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
parity=False, reverse_swipe=True, data_reverse_endian=True))

    # check forward-swipe and backward-swipe big endian 5 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data,
reverse_endian=True, data_reverse_endian=True))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
reverse_endian=True, reverse_swipe=True, data_reverse_endian=True))

    # check forward-swipe and backward-swipe big endian 4 bit ascii
    records.extend(track_decode.search_track1_bin_string(bin_data, parity=False,
reverse_endian=True, data_reverse_endian=True))
    records.extend(track_decode.search_track1_bin_string(bin_data[::-1],
parity=False, reverse_endian=True, reverse_swipe=True, data_reverse_endian=True))

    # Currently no endianness reversal of the data happening here.
    # These calls will rturn the same results as before.
```

```
    # check for forward-swipe binary-coded decimal

#records.extend(track_decode.binary_coded_decimal_track1(binascii.hexlify(data)))
    # check for backward-swipe binary-coded decimal

#records.extend(track_decode.binary_coded_decimal_track1(binascii.hexlify(data[::-
1])))

    # checks for ascii encoded data big_endian
    #records.extend(track_decode.ascii_parse_track1(data, little_endian=False))
    # checks for ascii encoded data little_endian
    #records.extend(track_decode.ascii_parse_track1(data, little_endian=True))

    return records
```

6.4.2 Track 2

```
def process_track2_data(data, convert_to_binary=True):
    """Return all track 2 PANs found in a chunk of binary data.

    Arguments:
        data: (string) data to parse for track 2 encoded PANs
        convert_to_binary: (boolean) whether or not to convert data into
            a binary string. Added since wav2cc.py passes binary string
            for data argument. Default: True

    Returns:
        list of tuples
    """

    records = []
    if convert_to_binary:
        bin_data = track_decode.get_binary_string(data)
    else:
        bin_data = data

    # check forward-swipe and backward-swipe little endian 5 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
reverse_swipe=True))

    # check forward-swipe and backward-swipe little endian 4 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data, parity=False))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
parity=False, reverse_swipe=True))

    # check forward-swipe and backward-swipe big endian 5 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data,
reverse_endian=True))
```

```
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
reverse_endian=True, reverse_swipe=True))

    # check forward-swipe and backward-swipe big endian 4 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data, parity=False,
reverse_endian=True))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
parity=False, reverse_endian=True, reverse_swipe=True))

    # check for forward-swipe binary-coded decimal

records.extend(track_decode.binary_coded_decimal_track2(binascii.hexlify(data)))
    # check for backward-swipe binary-coded decimal

records.extend(track_decode.binary_coded_decimal_track2(binascii.hexlify(data[::-
1])))

    # checks for ascii encoded data big_endian
    records.extend(track_decode.ascii_parse_track2(data, little_endian=False))
    # checks for ascii encoded data little_endian
    records.extend(track_decode.ascii_parse_track2(data, little_endian=True))

    # reverse the bit order in each byte
    bin_data = track_decode.get_binary_string_reverse_endian(data)

    # check forward-swipe and backward-swipe little endian 5 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data,
data_reverse_endian=True))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
reverse_swipe=True, data_reverse_endian=True))

    # check forward-swipe and backward-swipe little endian 4 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data, parity=False,
data_reverse_endian=True))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
parity=False, reverse_swipe=True, data_reverse_endian=True))

    # check forward-swipe and backward-swipe big endian 5 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data,
reverse_endian=True, data_reverse_endian=True))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
reverse_endian=True, reverse_swipe=True, data_reverse_endian=True))

    # check forward-swipe and backward-swipe big endian 4 bit ascii
    records.extend(track_decode.search_track2_bin_string(bin_data, parity=False,
reverse_endian=True, data_reverse_endian=True))
    records.extend(track_decode.search_track2_bin_string(bin_data[::-1],
parity=False, reverse_endian=True, reverse_swipe=True, data_reverse_endian=True))

    # Currently no endianness reversal of the data happening here.
    # These calls will rturn the same results as before.
    # check for forward-swipe binary-coded decimal
```

```
#records.extend(track_decode.binary_coded_decimal_track2(binascii.hexlify(data)))
    # check for backward-swipe binary-coded decimal

#records.extend(track_decode.binary_coded_decimal_track2(binascii.hexlify(data[::-
1])))

    # checks for ascii encoded data big_endian
    #records.extend(track_decode.ascii_parse_track2(data, little_endian=False))
    # checks for ascii encoded data little_endian
    #records.extend(track_decode.ascii_parse_track2(data, little_endian=True))

    return records
```

History

Revision	Issue Date	Section	History
Draft	01/22/2020	All	Initial draft for public comment.
DRAFT 1.0	01/22/2020	All	Formatted and technical edit performed for release as a Draft for Public Comment.
1.0	09-17-2020	All	Voted for release as final publication