SWGDE Test Method for Skimmer Forensics – Digital 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 – Digital Devices

Table of Figures

SWGDE Test Method for Skimmer Forensics – Digital Devices

1 Purpose

Skimmers are magnetic card readers used to capture account information. Able to be implanted into or overlaid onto 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 digital skimming devices.

2 Scope

This document is intended for computer forensic practitioners conducting forensic analysis on skimming devices using a digital storage format. It does not provide best practices regarding skimmer examinations, chip/SD card extractions, analog 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 Chip Off;*
- *SWGDE Best Practices for Computer Forensics Acquisitions;*
- *SWGDE Test Method for Skimmer Analysis – Analog Devices;*
- *SWGDE Test Method for Bluetooth Module Extractions and Analysis;* and
- *SWGDE Embedded Device Core Competencies.*

3 Limitations

Skimmers present unique examination challenges due to:

- Rapid changes in technology;
- Countermeasures;
- Multiple data encoding/ciphering formats; 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 digital skimmer column, post imaging.

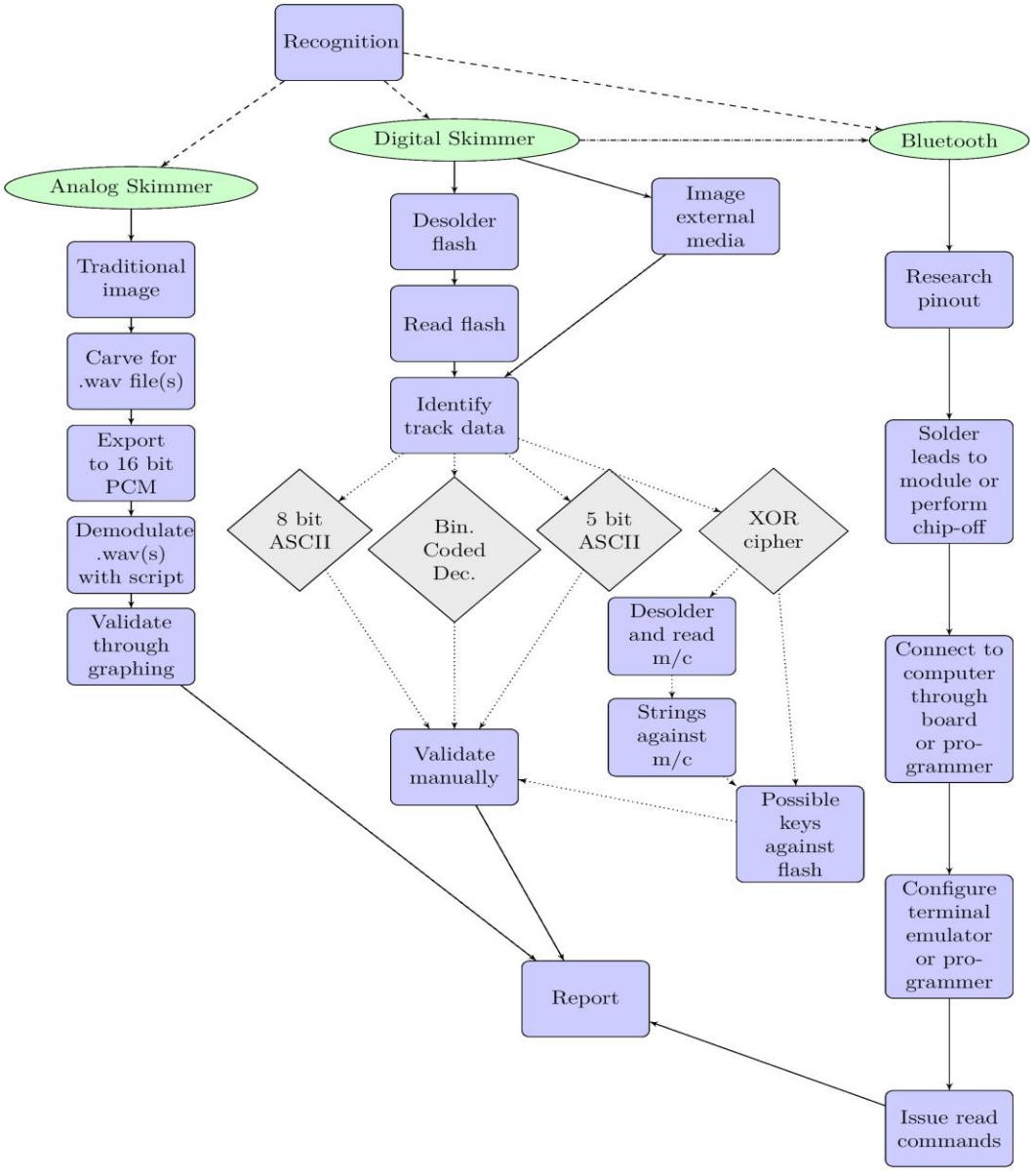


Figure 1- Process map

5 Analysis – Digital Skimmer

5.1 8-bit ASCII

The following is the process to analyze a skimmer that is storing account data in an 8-bit ASCII format:

- The majority of hex viewers have the ability to display data by default in both base 16 hexadecimal and 8-bit ASCII formats. A skimmer designed to store account information in 8-bit ASCII makes data identification simple, e.g. 4308906496460329
- Below, one can see that a primary account number is highlighted between 3B and 3D in hex or; and = on the ASCII side:

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x00	3B34	3330	3839	3036	3439	3634	3630	3332	;430890649646032
0x10	393D	3135	3037	3130	3131	3030	3030	3132	9=150710110000012
0x20	3334	3F00	3B34	3333	3133	3438	3932	3434	34?.;43313489244
0x30	3835	3932	383D	3133	3033	3130	3131	3030	85928=1303101100
0x40	3030	3132	3334	3F00	3B34	3339	3233	3230	0012342.;4392320
0x50	3337	3539	3433	3936	363D	3132	3031	3130	375943966=120110
0x60	3131	3030	3030	3132	3334	3F00	3B34	3338	11000012342.;438
0x70	3234	3337	3635	3734	3838	3930	383D	3136	2437657488908=16

Figure 2 - 8-bit hex and ASCII

5.1.1 Carve the account numbers by running Strings against the file, i.e., strings filename.bin.

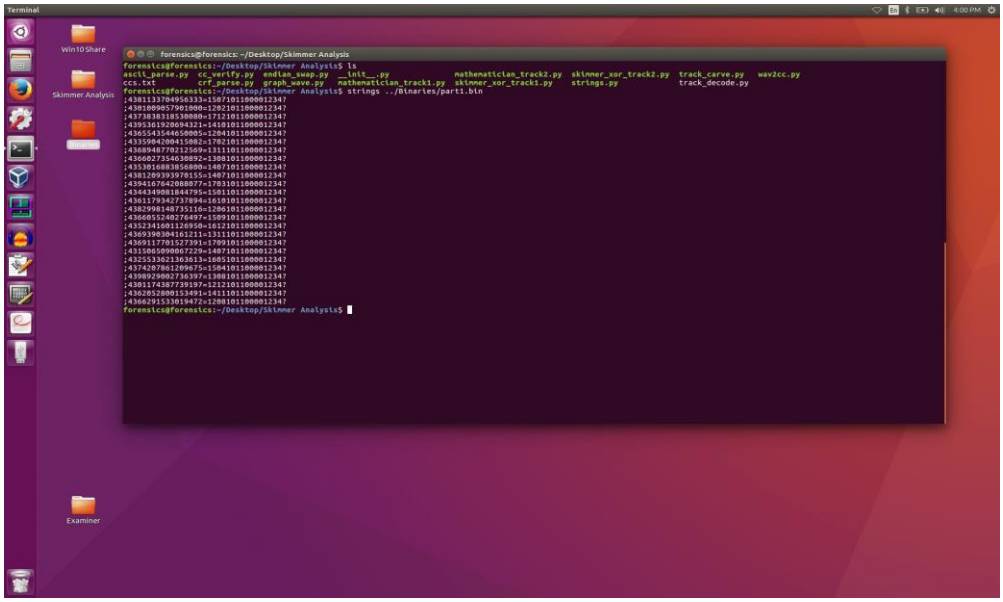


Figure 3 - Carving with Strings

- One can send the output to a text file by adding the ">" and a filename to the argument, e.g. "strings filename.bin > resultsfilename.txt"
- In addition to account data, Strings will also display other track data, e.g. account holder names if the skimmer was capturing and recording track 1 data (for skimmers using 8-bit ASCII)

5.1.2 Validate the Strings carving by viewing the file in a hex editor and matching at least one of the numbers from the output to a number present in the 8-bit ASCII view of the hex editor, running any possible account numbers against the Luhn algorithm.
Using the number displayed above: 4308906496460329
8 3 0 8 9 0 3 4 9 6 8 6 0 3 4 9 = 80 = valid
8 0 18 12 18 8 0 4
4 3 0 8 9 0 6 4 9 6 4 6 0 3 2 9

5.2 5-bit Binary

Another encoding scheme used with skimmers is 5-bit binary. The process to analyze skimmers encoded in this format is as follows:

5.2.1 In order to view and export image data in binary form, from an Ubuntu Linux terminal window, run:

xxd -b -c 256 part3.bin > ../test.txt

(this command will write the binary data to a file without column breaks in a file one directory higher then where the bin file resides);

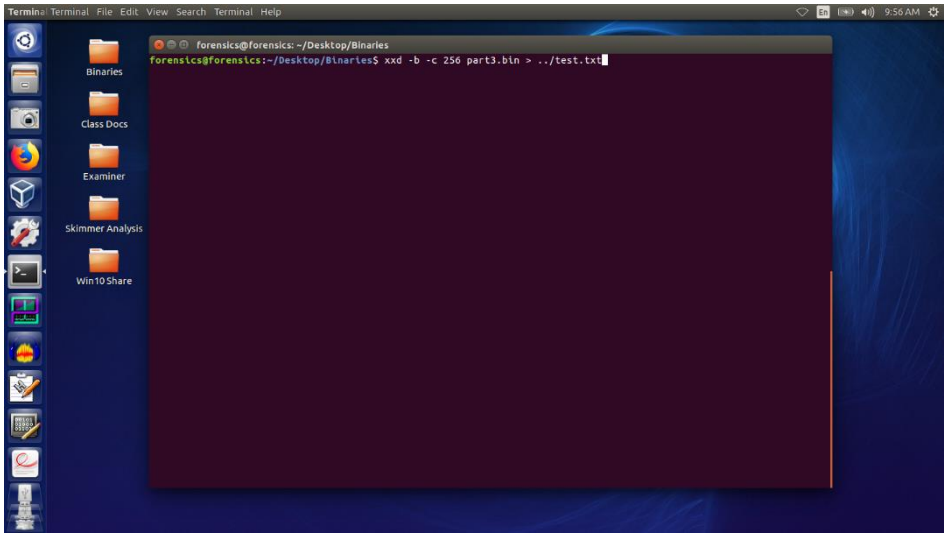


Figure 4 - View 5-bit as Binary

5.2.2 Open in an editor (Gedit is shown here) by double clicking on the new file;

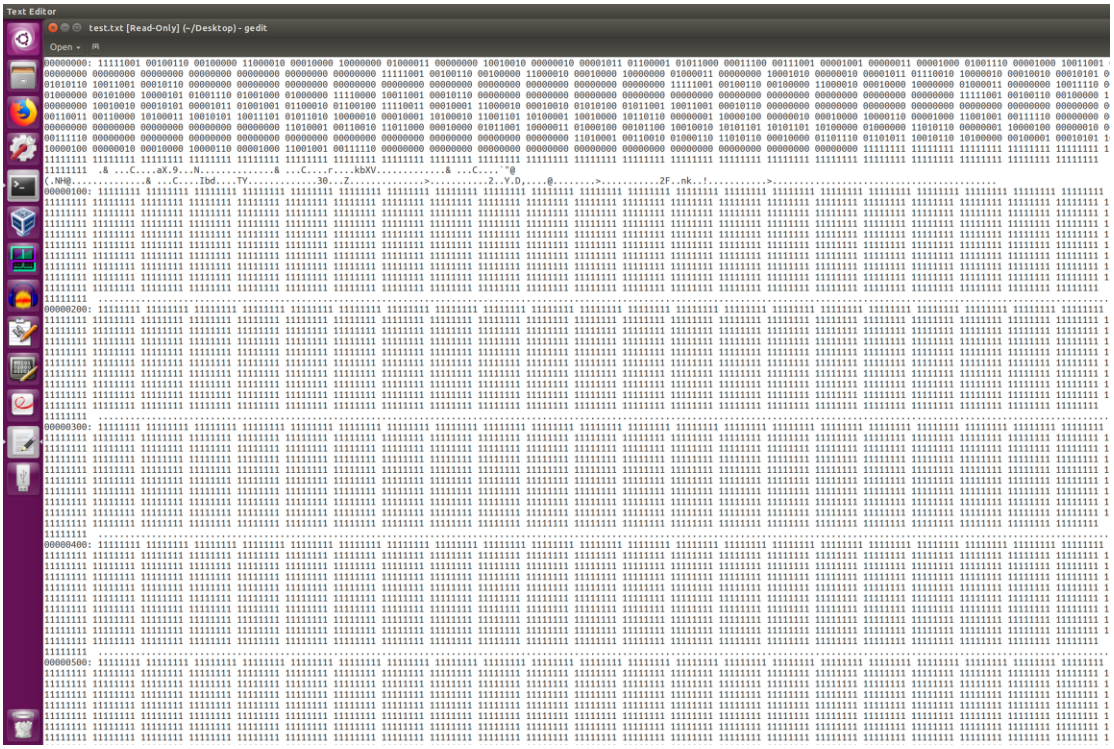


Figure 5 - 5-bit in editor

5.2.3 Use the search/replace function of the editor to remove spaces; i.e. search for " ", replace with (leave blank)

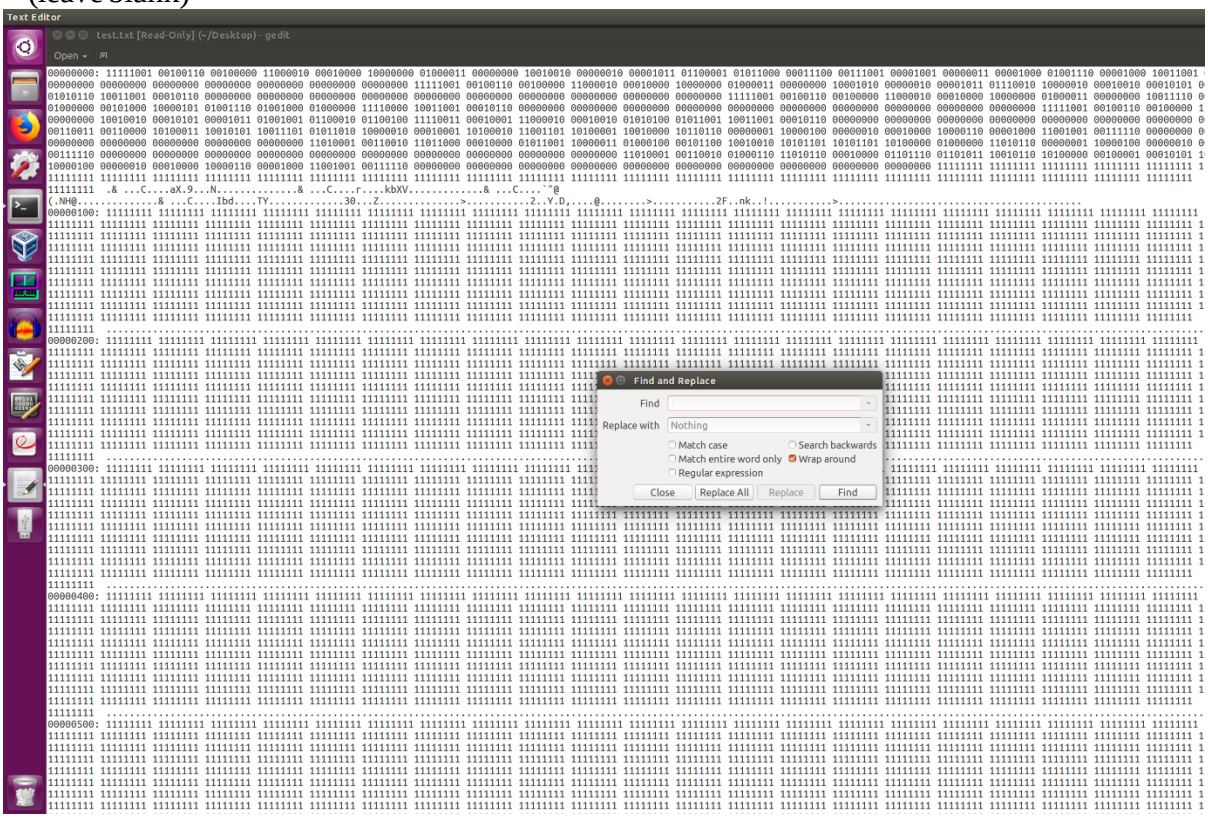


Figure 6 - Formatting 5-bit

5.2.4 Locate 5-bit binary start sentinels (right to left and left to right for reverse swipes) and manually decode the following (or preceding) bits according to 5-bit binary, running any possible account numbers against the Luhn algorithm;

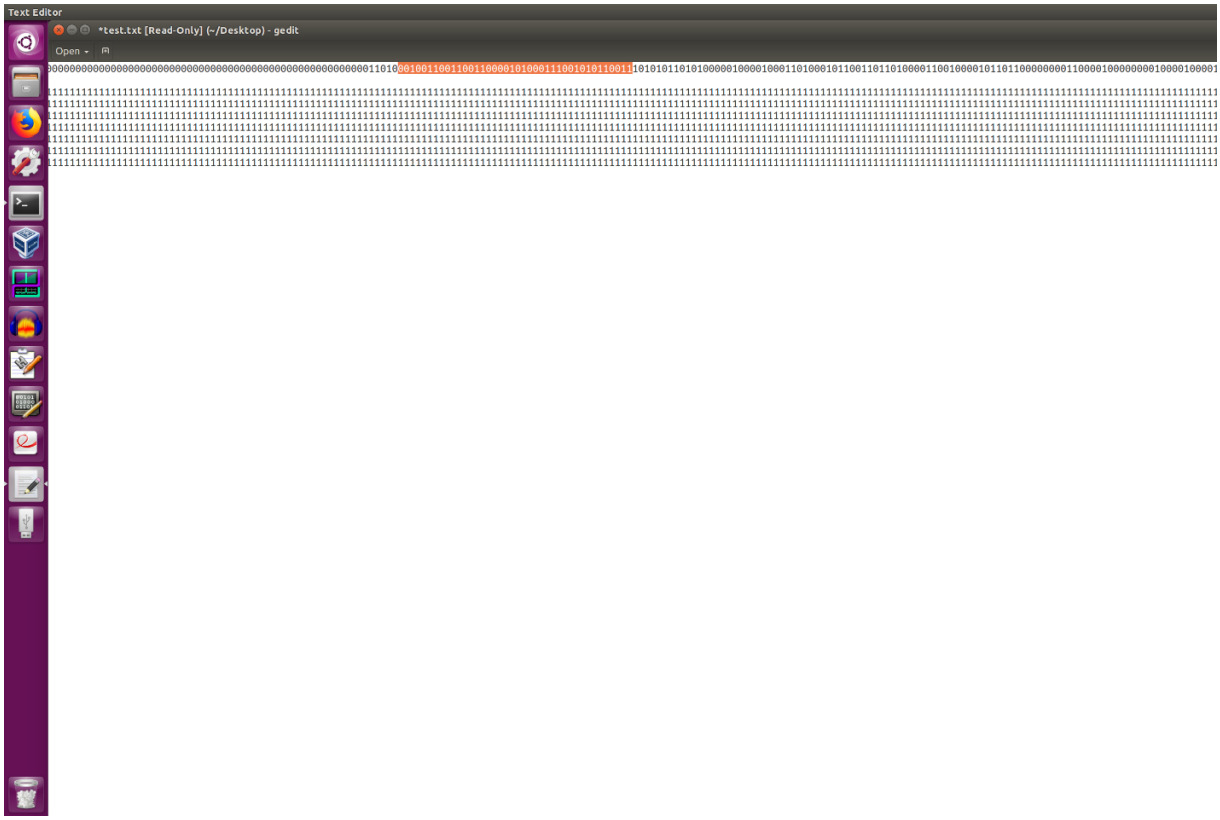


Figure 7 - 5-bit stream

- Bit stream from above:

0010011001100110000101000111001010110011101010110101000001000010001101000101100 1
0010|0 – 4
1100|1 – 3
1001|1 – 9
0000|1 – 0
0100|0 – 2
1110|0 – 7
1010|1 – 5
1001|1 – 9
1010|1 – 5
0110|1 – 6
0100|0 – 2
0010|0 – 4
0010|0 – 4
0110|1 – 6
0001|0 – 8
1100|1 – 3

4390 2759 5624 4683
Luhn:
8 3 9 0 4 7 1 9 1 6 4 4 8 6 7 3 = 80 = Valid
8 18 4 10 10 4 8 16
4 3 9 0 2 7 5 9 5 6 2 4 4 6 8 3

5.3 Additional 5-bit encoding

There are other ways in which 5-bit binary encoding can be implemented. In the previous example, the recovered account number was encoded as a forward swipe, 5-bit binary; however, there are more encoding options within 5-bit binary, one of which is bit-order in byte reverse endian. The following is the process to analyze 5-bit binary bit-order in byte reverse endian encoding:

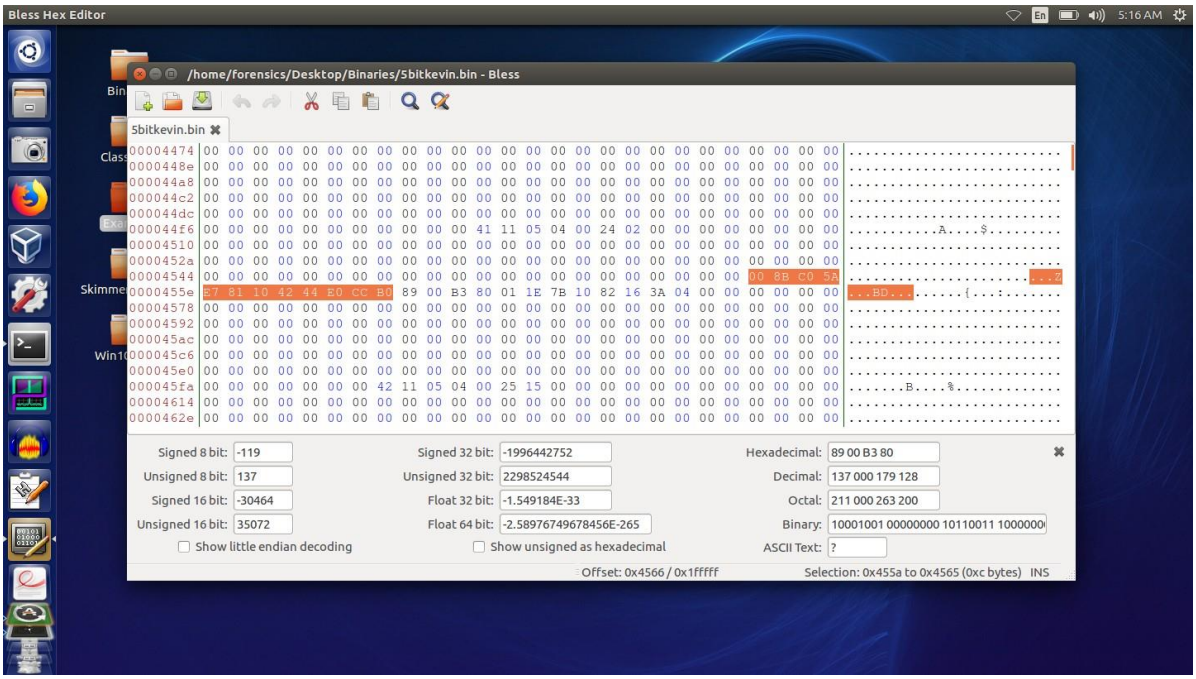


Figure 8 - 5-bit reversed

5.3.1 Identify possible account data. One can see from the above image, there appears to be a repeated 7 byte header followed eventually by what may be track data;

5.3.2 Copy base 16 hexadecimal values for what appears to be track data;

- Note: at this point one would not know exactly how many bytes to copy. As such, copying 11 bytes is a good starting point and as the decoding process continues, more/less bytes may be required.
- From the above file: 0x 8b c0 5a e7 81 10 42 44 e0 cc b0

5.3.3 Translate from hexadecimal to binary:

0x8B – 1000 1011
0xC0 – 1100 0000
0x5A – 0101 1010
0xE7 – 1110 0111
0x81 – 1000 0001
0x10 – 0001 0000
0x42 – 0100 0010
0x44 – 0100 0100
0xE0 – 1110 0000
0xCC – 1100 1100
0xB0 – 1011 0000

- Combined as a single binary string:

10001011110000000101101011100111100000010001000001000010010001001110000011001100101100000

- One can immediately recognize with all the repeated zeros and the lack of proper 5-bit parity, a different process is required

5.3.4 Separate the binary string into 8-bit chunks, then perform the reversal operation:

Raw data:	Reversed bytes:
10001011	11010001
11000000	00000011
01011010	01011010
11100111	11100111
10000001	10000001
00010000	00001000
01000010	01000010
01000100	00100010
11100000	00000111
11001100	00110011
10110000	00001101

- Reversed values from above:

11010001 00000011 01011010 11100111 10000001 00001000 01000010 00100010 00000111 00110011 00001101

5.3.5 Remove the spaces:

11010001000000110101101011100111100000010000100001000010000100010000001110011001100001101

5.3.6 Separate into 5-bit chunks:

11010 00100 00001 10101 10101 11001 11100 00001 00001 00001 00001 00010 00100 00001 11001 10011 00001 101

5.3.7 Decode the little endian* 5-bit binary to an account number

11010 – Start Sentinel (it is now known the decoding started at a correct byte)
00100 – 4
00001 – 0
10101 – 5
10101 – 5
11001 – 3
11100 – 7
00001 – 0
00001 – 0
00001 – 0
00001 – 0
00010 – 8
00100 – 4
00001 – 0
11001 – 3
10011 – 9
00001 – 0
101 <== Cut-off field separator

The 5-bit binary, reverse endian bit order in bytes, value is 4055 3700 0084 0390.

* Note: when working the decoding, it will not be immediately apparent if the data is big endian or little endian. One would work both until a valid account number is resolved. It is seen from above that little endian resolved the account number correctly.

5.3.8 Luhn validation

8 0 1 5 6 7 0 0 0 0 7 4 0 3 9 0 = 50 = valid
8 10 6 0 0 16 0 18
4 0 5 5 3 7 0 0 0 0 8 4 0 3 9 0

5.4 Binary Decimal

5.4.1 Binary Decimal is a non-ASCII encoding. The second nibble of a range of bytes represents a digit in the account number.

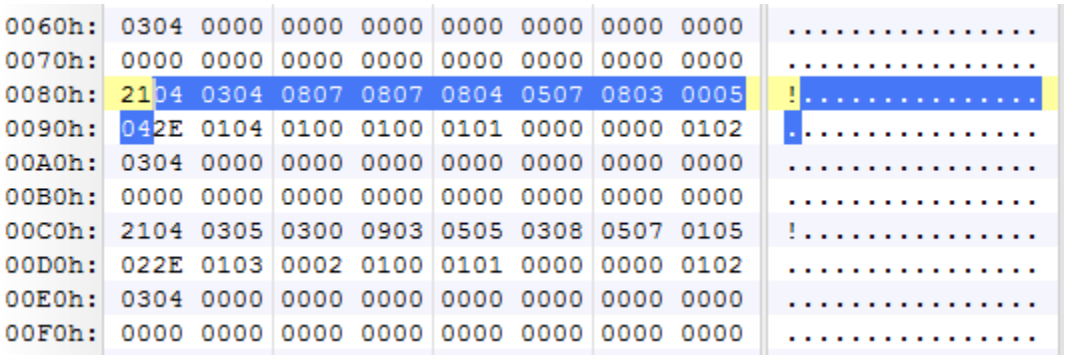


Figure 9 - Binary decimal

- Note in the above image, there is a hexadecimal header that equals 0x21, after which the account number immediately begins. The account number highlighted above is 4348 7878 4578 3054

5.4.2 Luhn validation

8 3 8 8 5 8 5 8 8 5 5 8 6 0 1 4 = 90 = valid
8 8 14 14 8 14 6 10
4 3 4 8 7 8 7 8 4 5 7 8 3 0 5 4

5.5 Encoded Skimmer Scripting

Whether seeking to automate the above digital encoding analysis tasks one may turn to scripting solutions. Scripts can take advantage of certain knowns about digital encoding formats plus the knowledge of the format of valid account data, e.g. valid account numbers must resolve to the Luhn algorithm. For Python 2.7 based decoding scripts See Appendix 1.

5.6 XOR Ciphers

5.6.1 XOR Basics

XOR cipher is defined as a simple repeated-key. An XOR cipher applies a key to a message to produce a cipher text:

$$plain\ text \oplus key = cipher\ text$$

XOR functions in a way that if any two values are known, the third can be solved, e.g.:

$$cipher\ text \oplus key = plain\ text$$

$$plain\ text \oplus cipher\ text = key$$

In simple form, data may be XOR ciphered using a one byte hexadecimal key, for example:

$$\begin{array}{rr} A & B \\ \oplus\ C & D \\ \hline 6 & 6 \end{array}$$

The above cipher is completed by converting from hexadecimal to binary and XOR the matching nibble.
0xA = 0b1010 0xB = 0b1011
0xC = 0b1100 0xD = 0b1101

For the left nibble:
$$\begin{array}{rrrrr} & 1 & 0 & 1 & 0 \\ \oplus & 1 & 1 & 0 & 0 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

For the right nibble:
$$\begin{array}{rrrrr} & 1 & 0 & 1 & 1 \\ \oplus & 1 & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

The resulting cipher text is 0b 0110 0110, 0x66

As was mentioned, the XOR function is easily reversible:

$$\begin{array}{rr} 6 & 6 \\ \oplus\ C & D \\ \hline A & B \end{array}$$

0x6 = 0b0110 0x6 = 0b0110
0xC = 0b1100 0xD = 0b1101

For the left nibble:

	0	1	1	0
⊕	1	1	0	0
	1	0	1	0

For the right nibble:

	0	1	1	0
⊕	1	1	0	1
	1	0	1	1

The plain text is 0b 1010 1011, 0xAB.

5.6.2 XOR In Practice

The following is offered to show how one may test logical guesses in order to determine a key.
USSS Tulsa Cell Phone Laboratory challenge coin
Example cipher text: 0x8B EB EB EB EA 88 18 18 DB 58
28 38 88 BA 28 F9 88 38 E9 48
E8 BA C8 E8 48 18 48 99 49 7A
9A EB FA 7A 7A

Since there is only cipher text available, one can attempt to solve by using educated guesses of the plain text in order to solve for a possible key. Once that possible key is found, one then tests it against other probable known values to determine if the key is correct.

Given that the first four bytes of the cipher text are one byte value followed by a second byte value repeated three times, one may theorize that the first four cipher text bytes equal plain text ASCII U, S, S, S and that the key is non-confused/non-diffused and only one byte in length. Given that there are two probable cipher text to plain text character guesses, it is rather simple to both work the decoding and test the theory.

Theory – If cipher text 0x8B equals plain text ASCII U then cipher text 0xEB equals plain text S. If the theory is true then a key will be discovered and can be applied against the remainder of the byte string.

5.6.2.1 Step 1:

Determine a potential key that will resolve the numbers as described immediately above. As ASCII U = 0x55, then

	8	B	cipher text
⊕	5	5	probable plain text
			probable key

0x8 = 0b1000 0xB = 0b1011
0x5 = 0b0101 0x5 = 0b0101

For the left nibble:

	1	0	0	0
⊕	0	1	0	1
	1	1	0	1

For the right nibble:

	1	0	1	1
⊕	0	1	0	1
	1	1	1	0

Probable key = 0b 1101 1110, 0xDE.

5.6.2.2 Step 2:

XOR second cipher text byte 0xEB with potential key 0xDE, expecting to see 0x53, plain text ASCII S.

```
    E   B   cipher text
⊕   D   E   key
            plain text
```

0xE = 0b1110 0xB = 0b1011
0xD = 0b1101 0xE = 0b1110

For the left nibble:

```
    1   1   1   0
⊕   1   1   0   1
    0   0   1   1
```

For the right nibble:

```
    1   0   1   1
⊕   1   1   1   0
    0   1   0   1
```

Result: 0b0011 0101, 0x35, ASCII 5....

So key 0xDE did not yield 0x53 (ASCII S) but it is interesting that it yielded 0x35, a nibble switched version of the expected result.

Theory1: Probable plain text 0x55 resolved using key 0xDE as expected due to the first nibble and the second nibble having the same value, i.e. a "5" and a "5". If the key 0xDE is used, will the nibble switched plain text byte value persist as expected?

5.6.2.3 Step1−1: XOR a third cipher text byte with key 0xDE nibble switching the end hex value, expecting to find a probable plain text value.

After the probable S's, the next cipher text byte is 0xEA. If one continues to use the coin and the facility title it represents as an indication of the possible plain text, the next value could be an ASCII "C" for "Cell". Following, if 0xEA is XOR by key 0xDE, the expected result will be the nibble switched value for ASCII C, i.e. 0x43.

```
    E   A
⊕   D   E
```

0xE = 0b1110 0xA = 0b1010
0xD = 0b1101 0xE = 0b1110

For the left nibble:

	1	1	1	0
⊕	1	1	0	1
	0	0	1	1

For the right nibble:

	1	0	1	0
⊕	1	1	1	0
	0	1	0	0

Result: 0b0011 0100, 0x34, nibble switched to 0x43 = ASCII C = Success!!

5.6.2.4 Step1−2: Given the success, one can now follow the pattern using key 0xDE and nibble switching the plain text byte before converting to ASCII. The process will result in solving this cipher, to include the non "guessable" bytes at the end.

5.6.3 XOR in the Wild

In the previous analysis examples, the track data processed was extracted directly from a PC board attached chip. There are also skimmers that save track data to SD cards. The SD cards will typically have both a file (or files) to analyze in addition to processing unallocated space. Skimmers using this type of architecture have found to use XOR ciphers. If a skimmer is programmed to save track data as eight bit ASCII with a one byte XOR cipher, it is possible to decipher the track data.

- If the data was saved in noncontiguous segments (allowing for a lot of plain text zeros (0) in between the track data), and all the data is ciphered (not just the track data) a simple frequency analysis should identify the most frequent hexadecimal byte.
- The most frequent ciphered byte should represent plain text ASCII 0, 0x30.
- From there, one can XOR that most frequent cipher text byte by 0x30 to determine the XOR key.
- One can then apply that key to the entire file to decipher the track data.
- If successful (Luhn validated account and other track data), one can run that key against data found in unallocated space on the SD card to potentially find more data.

Example cipher text:

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x00	7D1A	6C69	696F	6F6B	6B61	6068	696C	6160	}.liiookka`hila`
0x10	6F6C	060B	190C	1019	1617	1677	0B0D	0A19	ol.........w....
0x20	161F	066A	6968	6B6A	6869	6868	6868	6868	...jihkjhihhhhhh
0x30	6868	6860	6868	6868	686D	6860	6868	6868	hhh`hhhhhmh`hhhh
0x40	6868	6707	7878	636C	6969	6F6F	6B6B	6160	hhg.xxcliiookka`
0x50	6869	6C61	606F	6C65	6A69	686B	6A68	6968	hila`olejihkjhih
0x60	6868	6868	6868	6868	6D68	6060	6765	7878	hhhhhhhhmh``gexx
0x70	0378	696A	6B6C	6D6E	6F60	6168	781D	362C	.xijklmno`ahx.6,
0x80	3D2A	787D	1A6C	6969	6F6F	6B6B	6160	6869	=*x}.liiookka`hi
0x90	6C61	606F	6C06	0B19	0C10	1916	1716	770B	la`ol.........w.
0xA0	0D0A	1916	1F06	6A69	686B	6A68	6968	6868	......jihkjhihhh
0xB0	6868	6868	6868	6068	6868	6868	6D68	6068	hhhhhh`hhhhhmh`h
0xC0	6868	6868	6867	0778	7863	6C69	696F	6F6B	hhhhhg.xxcliiook
0xD0	6B61	6068	696C	6160	6F6C	656A	6968	6B6A	ka`hila`olejihkj
0xE0	6869	6868	6868	6868	6868	686D	6860	6067	hihhhhhhhhhmh``g
0xF0	6578	7803	7869	6A6B	6C6D	6E6F	6061	6878	exx.xijklmno`ahx
0x0100	1D36	2C3D	2A78	7D1A	6C6A	6E6E	606C	696C	.6,=*x}.ljnn`lil
0x0110	616F	6A68	6A6B	686F	0610	1909	0D1D	7715	aojhjkho......w.

Figure 10 - Cipher text

5.6.3.1 One may notice either through an automated frequency analysis or just by looking at the cipher text, the value 0x68 seems to be repeated a lot, possibly even used as blank space in between track data.

5.6.3.2 Following the above premise, one can attempt to find the key by XOR ciphering 0x68 (cipher text) by 0x30 (0, possible plain text).

0x6 = 0b0110 0x8 = 0b1000
0x3 = 0b0011 0x0 = 0b0000

For the left nibble:

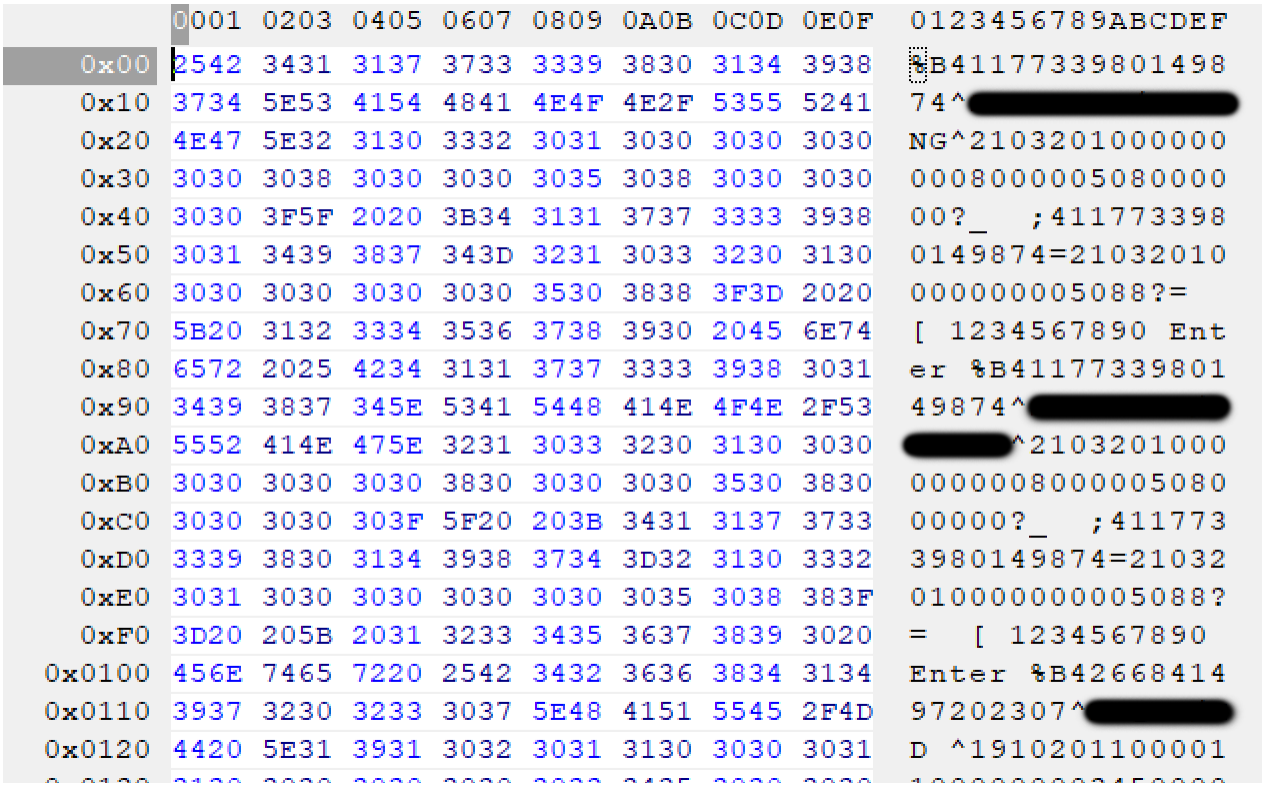
	0	1	1	0
⊕	0	0	1	1
	0	1	0	1

For the right nibble:

	1	0	0	0
⊕	0	0	0	0
	1	0	0	0

0101 1000 = 0x58
So if the above premise is true, then the XOR key = 0x58.

5.6.3.3 With 0x58 applied against the above extraction, the following data is presented:



	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x00	2542	3431	3137	3733	3339	3830	3134	3938	%B41177339801498
0x10	3734	5E53	4154	4841	4E4F	4E2F	5355	5241	74^[redacted]
0x20	4E47	5E32	3130	3332	3031	3030	3030	3030	NG^2103201000000
0x30	3030	3038	3030	3030	3035	3038	3030	3030	0008000005080000
0x40	3030	3F5F	2020	3B34	3131	3737	3333	3938	00?_ ;411773398
0x50	3031	3439	3837	343D	3231	3033	3230	3130	0149874=21032010
0x60	3030	3030	3030	3030	3530	3838	3F3D	2020	000000005088?=
0x70	5B20	3132	3334	3536	3738	3930	2045	6E74	[1234567890 Ent
0x80	6572	2025	4234	3131	3737	3333	3938	3031	er %B41177339801
0x90	3439	3837	345E	5341	5448	414E	4F4E	2F53	49874^[redacted]
0xA0	5552	414E	475E	3231	3033	3230	3130	3030	[redacted]^2103201000
0xB0	3030	3030	3030	3830	3030	3030	3530	3830	0000008000005080
0xC0	3030	3030	303F	5F20	203B	3431	3137	3733	00000?_ ;411773
0xD0	3339	3830	3134	3938	3734	3D32	3130	3332	3980149874=21032
0xE0	3031	3030	3030	3030	3030	3035	3038	383F	0100000000005088?
0xF0	3D20	205B	2031	3233	3435	3637	3839	3020	= [1234567890
0x0100	456E	7465	7220	2542	3432	3636	3834	3134	Enter %B42668414
0x0110	3937	3230	3233	3037	5E48	4151	5545	2F4D	97202307^[redacted]
0x0120	4420	5E31	3931	3032	3031	3130	3030	3031	D ^1910201100001

Figure 11 – Deciphered text

One can see that the 0x58 successfully decoded 8-bit ASCII encoded track data.

5.6.3.4 Ciphered data in unallocated space

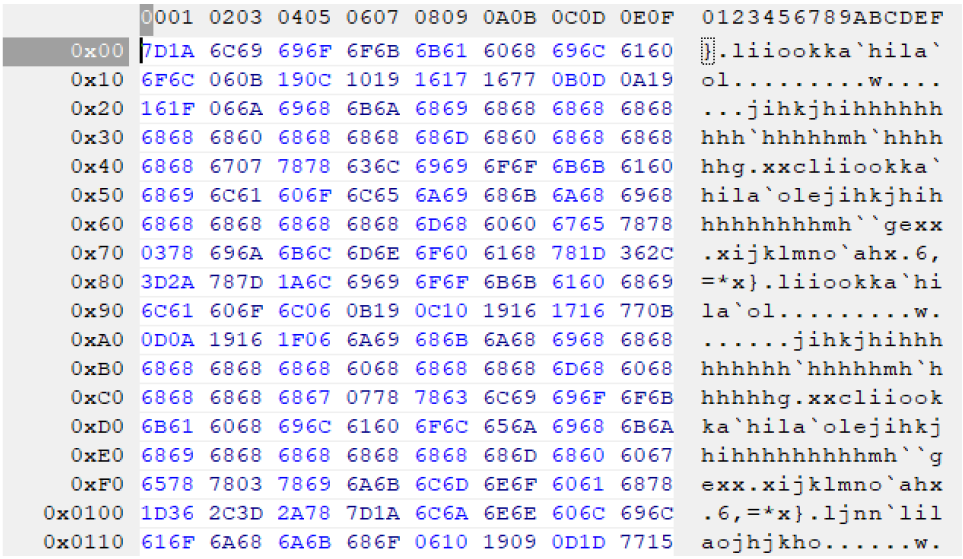
In regard to SD card skimmers, after decoding data saved in any recovered file(s), using the same XOR key, one should decode track data that may reside in unallocated space on the SD card.

5.6.4 Incorrect nibbles can still be winners

Given the same parameters mentioned above (plain text of 8 bit ASCII) if one uses an incorrect first nibble of a one byte key but a correct second nibble, due to the nature of XOR ciphering data one byte cipher text by a one byte key, the resulting plain text will properly decode the primary account numbers in a format that closely resembles binary coded decimal, but not the remaining track ASCII data, e.g. name.

For example:

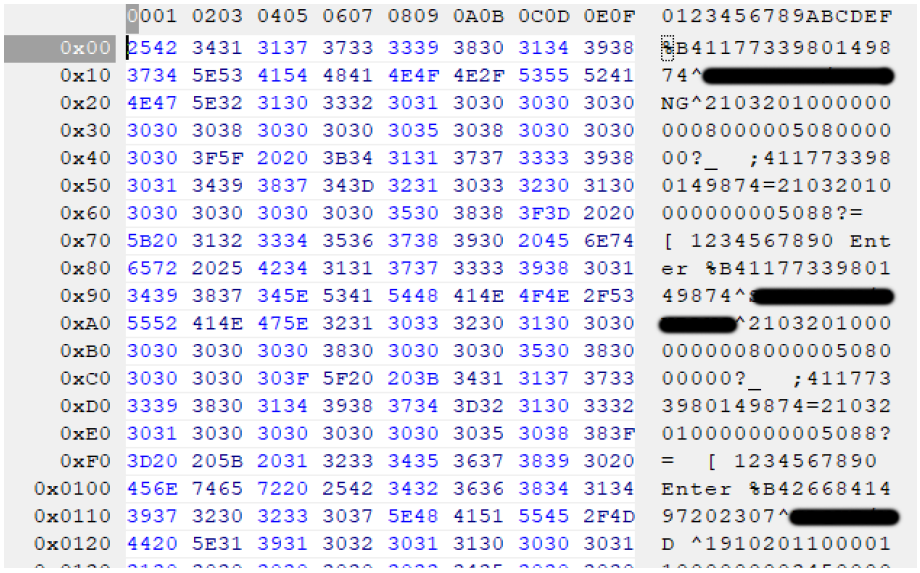
Cipher text:



	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x00	7D1A	6C69	696F	6F6B	6B61	6068	696C	6160	}.liiookka`hila`
0x10	6F6C	060B	190C	1019	1617	1677	0B0D	0A19	ol.........w....
0x20	161F	066A	6968	6B6A	6869	6868	6868	6868	...jihkjhihhhhhh
0x30	6868	6860	6868	6868	686D	6860	6868	6868	hhh`hhhhhmh`hhhh
0x40	6868	6707	7878	636C	6969	6F6F	6B6B	6160	hhg.xxcliiookka`
0x50	6869	6C61	606F	6C65	6A69	686B	6A68	6968	hila`olejihkjhih
0x60	6868	6868	6868	6868	6D68	6060	6765	7878	hhhhhhhhmh``gexx
0x70	0378	696A	6B6C	6D6E	6F60	6168	781D	362C	.xijklmno`ahx.6,
0x80	3D2A	787D	1A6C	6969	6F6F	6B6B	6160	6869	=*x}.liiookka`hi
0x90	6C61	606F	6C06	0B19	0C10	1916	1716	770B	la`ol.........w.
0xA0	0D0A	1916	1F06	6A69	686B	6A68	6968	6868	......jihkjhihhh
0xB0	6868	6868	6868	6068	6868	6868	6D68	6068	hhhhhh`hhhhhmh`h
0xC0	6868	6868	6867	0778	7863	6C69	696F	6F6B	hhhhhg.xxcliiook
0xD0	6B61	6068	696C	6160	6F6C	656A	6968	6B6A	ka`hila`olejihkj
0xE0	6869	6868	6868	6868	6868	686D	6860	6067	hihhhhhhhhhmh``g
0xF0	6578	7803	7869	6A6B	6C6D	6E6F	6061	6878	exx.xijklmno`ahx
0x0100	1D36	2C3D	2A78	7D1A	6C6A	6E6E	606C	696C	.6,=*x}.ljnn`lil
0x0110	616F	6A68	6A6B	686F	0610	1909	0D1D	7715	aojhjkho......w.

Figure 12 - Cipher text, correct nibbles

Correct key (0x58):



	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x00	2542	3431	3137	3733	3339	3830	3134	3938	%B41177339801498
0x10	3734	5E53	4154	4841	4E4F	4E2F	5355	5241	74^[redacted]
0x20	4E47	5E32	3130	3332	3031	3030	3030	3030	NG^2103201000000
0x30	3030	3038	3030	3030	3035	3038	3030	3030	0008000005080000
0x40	3030	3F5F	2020	3B34	3131	3737	3333	3938	00?_ ;411773398
0x50	3031	3439	3837	343D	3231	3033	3230	3130	0149874=21032010
0x60	3030	3030	3030	3030	3530	3838	3F3D	2020	0000000050088?=
0x70	5B20	3132	3334	3536	3738	3930	2045	6E74	[1234567890 Ent
0x80	6572	2025	4234	3131	3737	3333	3938	3031	er %B41177339801
0x90	3439	3837	345E	5341	5448	414E	4F4E	2F53	49874^[redacted]
0xA0	5552	414E	475E	3231	3033	3230	3130	3030	[redacted]^2103201000
0xB0	3030	3030	3030	3830	3030	3030	3530	3830	0000008000005080
0xC0	3030	3030	303F	5F20	203B	3431	3137	3733	00000?_ ;411773
0xD0	3339	3830	3134	3938	3734	3D32	3130	3332	3980149874=21032
0xE0	3031	3030	3030	3030	3030	3035	3038	383F	0100000000050088?
0xF0	3D20	205B	2031	3233	3435	3637	3839	3020	= [1234567890
0x0100	456E	7465	7220	2542	3432	3636	3834	3134	Enter %B42668414
0x0110	3937	3230	3233	3037	5E48	4151	5545	2F4D	97202307^[redacted]
0x0120	4420	5E31	3931	3032	3031	3130	3030	3031	D ^1910201100001

Figure 13 - De-ciphered text, correct nibbles

Incorrect key, but with correct second nibble (0x68)

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x00	1572	0401	0107	0703	0309	0800	0104	0908	.r..............
0x10	0704	6E63	7164	7871	7E7F	7E1F	6365	6271	..ncqdxq~ ~.cebq
0x20	7E77	6E02	0100	0302	0001	0000	0000	0000	~wn.............
0x30	0000	0008	0000	0000	0005	0008	0000	0000	....
0x40	0000	0F6F	1010	0B04	0101	0707	0303	0908	...o............
0x50	0001	0409	0807	040D	0201	0003	0200	0100	....
0x60	0000	0000	0000	0000	0500	0808	0F0D	1010	....
0x70	6B10	0102	0304	0506	0708	0900	1075	5E44	k............u^D
0x80	5542	1015	7204	0101	0707	0303	0908	0001	UB..r...........
0x90	0409	0807	046E	6371	6478	717E	7F7E	1F63	.....ncqdxq~ ~.c
0xA0	6562	717E	776E	0201	0003	0200	0100	0000	ebq~wn..........
0xB0	0000	0000	0000	0800	0000	0000	0500	0800	....
0xC0	0000	0000	000F	6F10	100B	0401	0107	0703	......o.........
0xD0	0309	0800	0104	0908	0704	0D02	0100	0302	....
0xE0	0001	0000	0000	0000	0000	0005	0008	080F	....
0xF0	0D10	106B	1001	0203	0405	0607	0809	0010	...k............
0x0100	755E	4455	4210	1572	0402	0606	0804	0104	u^DUB..r........
0x0110	0907	0200	0203	0007	6E78	7161	6575	1F7D	........nxqaeu.}
0x0120	7410	6E01	0901	0002	0001	0100	0000	0001	t.n.............
0x0130	0100	0000	0000	0000	0003	0405	0000	0000	....
0x0140	0000	0F1A	1010	0B04	0206	0608	0401	0409	....
0x0150	0702	0002	0300	070D	0109	0100	0200	0101	....
0x0160	0000	0000	0304	050F	0510	106B	1001	0104	...........k....
0x0170	0108	1075	5E44	5542	1015	7204	0200	0706	...u^DUB..r.....
0x0180	0700	0109	0102	0402	0401	046E	6379	7E77	...........ncy~w
0x0190	781F	7E71	6279	7E74	7562	6E02	0000	0302	x.~qby~tubn.....
0x01A0	0001	0000	0000	0000	0004	0909	0000	0000	....
0x01B0	0000	0F0F	1010	0B04	0200	0706	0700	0109	....
0x01C0	0102	0402	0401	040D	0200	0003	0200	0100	....
0x01D0	0000	0000	0409	0900	0000	0000	0F03	1010	....
0x01E0	6B10	0101	0401	0810	755E	4455	4210	1572	k.......u^DUB..r
0x01F0	0403	0309	0903	0106	0807	0604	0109	0306	....

Figure 14 - De-ciphered text, incorrect second nibble

When one views the data in hex, account numbers are readable, almost as if they are padded (in this case) with a first nibble of zero, i.e. 0x04 0x01 0x01 0x07 0x07 0x3 0x3 0x9... This happens as a result of the way ASCII characters 0 – 9 are represented in hex, the fact that in this example a one byte key is used, and the XOR function itself.

Take for instance the cipher text byte of 0x6C (occurring at address 0x2). Using the correct key:
Correct key 0x58 = 0b0101 1000
Cipher text 0x6C = 0b0110 1100

For the left nibble:

	0	1	0	1
⊕	0	1	1	0
	0	0	1	1

For the right nibble:

	1	0	0	0
⊕	1	1	0	0
	0	1	0	0

0b0011 0100 = 0x34 = ASCII 4

As expected, the correct cipher text byte matches the plain text byte 0x34, ASCII 4, at address 0x02.

Using incorrect key:
Incorrect key 0x68 = 0b0110 1000 cipher text 0x6C = 0b0110 1100

For the left nibble:

	0	1	1	0
⊕	0	1	1	0
	0	0	0	0

For the right nibble:

	1	0	0	0
⊕	1	1	0	0
	0	1	0	0

0b0000 0100 = which does equal 0x04, but not ASCII 4

When one tries that key with a non-numeric, it also fails. As an example, the cipher text at byte address 0x14 should equal plain text ASCII S; however, using the same incorrect key yields the following:

Incorrect key 0x68 = 0b0110 1000 cipher text 0x0B = 0b0000 1011

For the left nibble:

	0	1	1	0
⊕	0	0	0	0
	0	1	1	0

For the right nibble:

	1	0	0	0
⊕	1	0	1	1
	0	0	1	1

0b0110 0011 = 0x63 = c

As expected, while the wrong key with correct second nibble did seem to provide the correct plain text in reference to a numerical value in an account number, it fails at other track data such as the primary account holder name, i.e. the "S".

Why does this matter?

Earlier it was stated that there are conditions to solving a one byte cipher in this matter, e.g. a lot of plain text zeros (0) in between the track data. However; knowing that hex only consists of a 16 character set, (0-9, A-F), if one is not able to identify the ciphered byte for plain text zero but is still working within ASCII plain text and a one byte cipher, one only needs to try a total of 16 variations in the second nibble space of a cipher key to work, i.e. 0xXY, where X = any hex character and y = 0-9, A-F. Once track data is found (as with the correct second nibble cipher used above), one can run through the 16 possibilities for the first nibble to recover full track data.

5.6.5 XOR key length variations

XOR ciphers can be of any length.

Encrypting 'x' (0b0111 1000) with a 4-bit / nibble key (0b0110):

	0	1	1	1	1	0	0	0
⊕	0	1	1	0	0	1	1	0
	0	0	0	1	1	1	1	0

Encrypting 'xy' (0b0111 1000 0111 1001) with a 2 byte key (0b0110 0100 1100 0001):

	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	1
⊕	0	1	1	0	0	1	0	0	1	1	0	0	0	0	0	1
	0	0	0	1	1	1	0	0	1	0	1	1	1	0	0	0

Longer keys combined with non ASCII plain text, can make XOR ciphers more difficult to resolve.

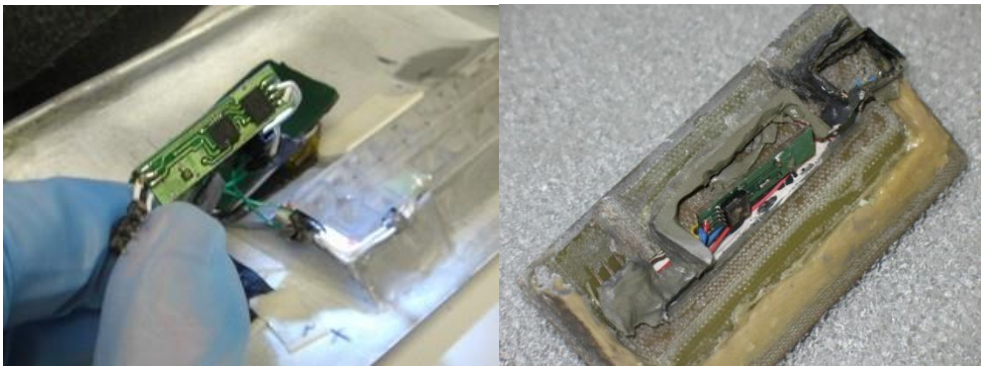


Figure 15 - Ciphered skimmer example

The above pictures are of a multi-byte XOR ciphered skimmer with microcontroller, decoder, and EEPROM chips.

In the below image extracted from the flash chip of a skimmer resembling the above, one can see possible track data isolated by hex zeros. Notice how this cipher text is dramatically different from the previous examples. Unless the plain text was padded with a non-zero plain text byte in between the track data, the only data in the file that is ciphered is the track data.

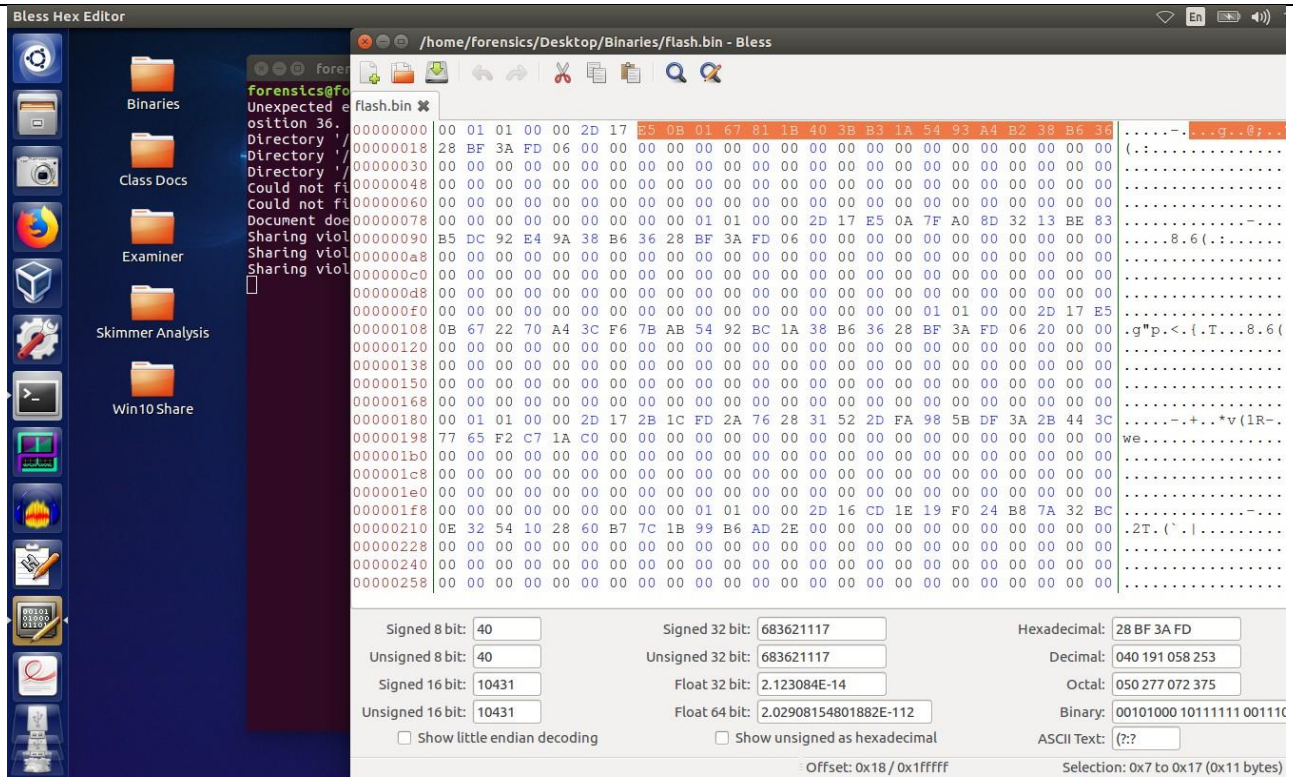


Figure 16 - Non-continuous key application

5.6.5.1 Microcontroller

When attempts to recover XOR ciphered track data as previously discussed fail, one may turn to the device's microcontroller for information that can assist in the decoding of track data. If necessary, there are decompilers that present microcontroller data in assembly language (see Appendix 2); however, many times running Strings against a microcontroller data extraction may provide a key.

5.6.5.1.1 Carve the extraction:

Strings –d filename.hex

- Note: running strings with default settings only provides hits across what one would normally see within a normal hex editor. The use of the -d switch provides hits within the data structures of the file.

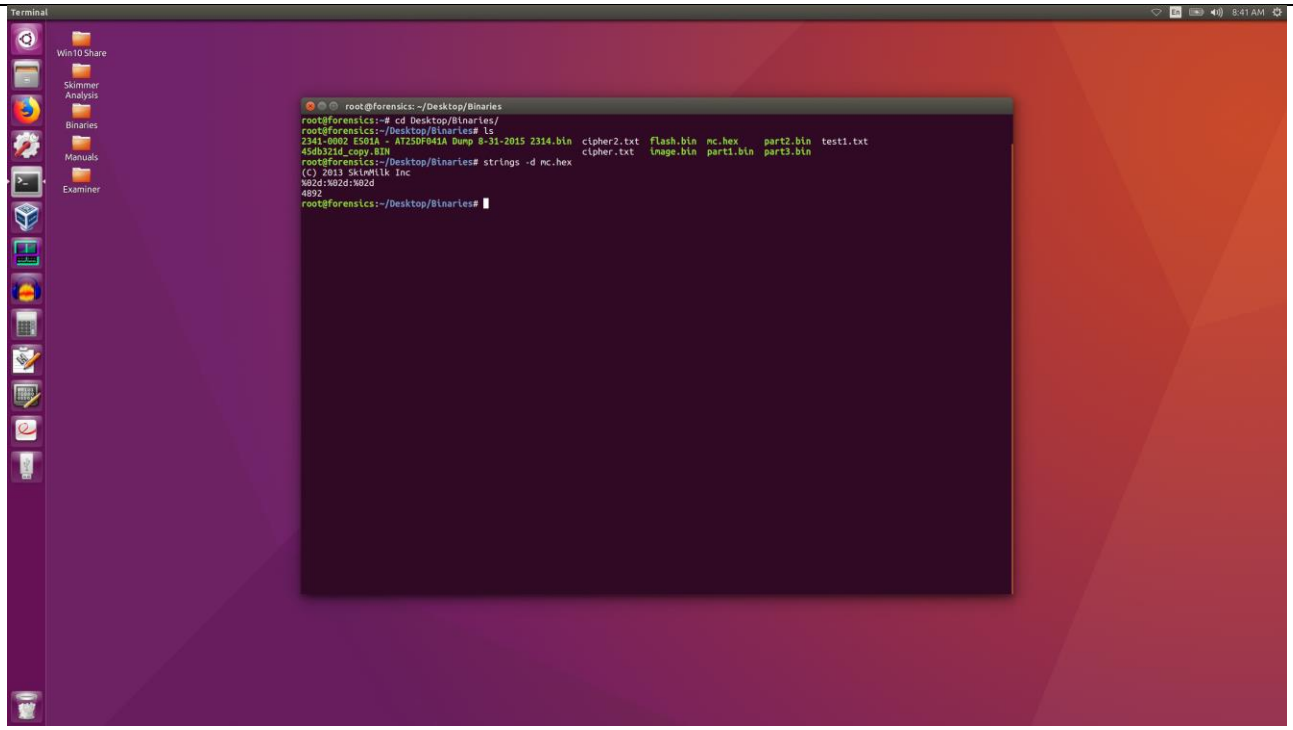


Figure 17 - Microcontroller contents

One can see the output provided a possible key of ASCII 4892.

With a possible multi-byte XOR cipher and a possible key, one needs to:

5.6.5.2 Locate the cipher text in a binary file;

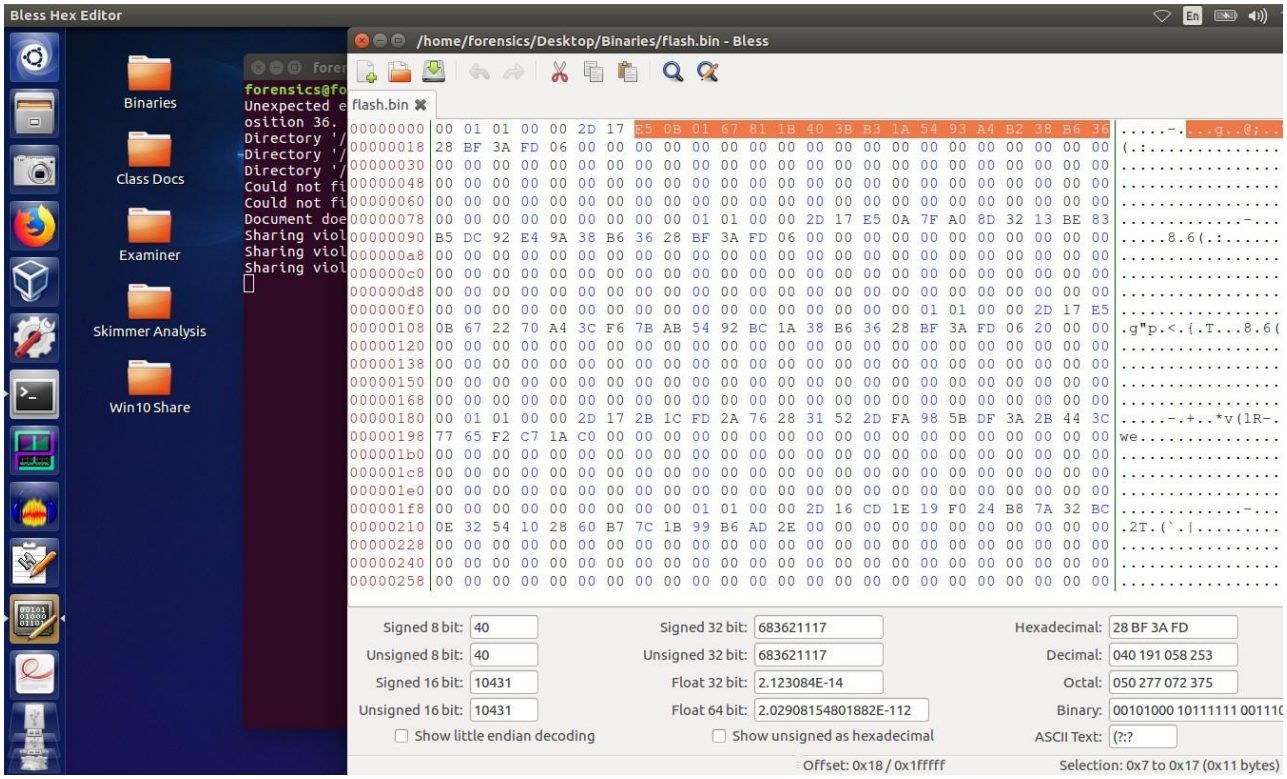


Figure 18 - Multi-byte key cipher text

- Note: There appears to be a header that precedes each chunk of encoded account data, i.e. 0x00 01 01 00 00 2D 17. It is essential to recognize these bytes in order to determine the correct starting byte to apply the four byte key of 4892, or in base 16 hexadecimal, 0x34 38 39 32. Note: 17 appears to be a record length indicator, there are instances of both 17 and 16. It is unlikely that the header and length indicator are ciphered as they are consistent throughout the extraction. For this example, 17 bytes will be deciphered.

5.6.6 XOR the hex bytes by the hex value of the key;

4 = 0x34
8 = 0x38
9 = 0x39
2 = 0x32

	E5	0B	01	67	81	1B	40	3B	B3	1A	54	93	A4	B2	38	B6	36
⊕	34	38	39	32	34	38	39	32	34	38	39	32	34	38	39	32	34
	D1	33	38	55	B5	23	79	09	87	22	6D	A1	90	8A	01	84	02

5.6.6.1 Resolve those bytes to an encoding scheme that produces Luhn validated account numbers;

- Note: For this example, the final decoding to account numbers is big endian, 5-bit binary, but that may vary from device to device.

D1 – 1101 0001
33 – 0011 0011
38 – 0011 1000
55 – 0101 0101
B5 – 1011 0101
23 – 0010 0011
79 – 0111 1001
09 – 0000 1001
87 – 1000 0111
22 – 0010 0010
6D – 0110 1101
A1 – 1010 0001
90 – 1001 0000
8A – 1000 1010
01 – 0000 0001
84 – 1000 0100
02 – 0000 0010

5.6.6.2 List the binary values as one string;

11010001001100110011100001010101101101010010001101111001000010011000011100100010011011011010000110010000100010100000000110000100000000010

5.6.6.3 List the binary values as 5-bit binary little endian chunks with their corresponding account numbers values;

11010 – Start sentinel
00100 – 4
11001 – 3
10011 – 9
10000 – 1
10101 – 5
01101 – 6
10101 – 5
00100 – 4
01101 – 6
11100 – 7
10000 – 1
10011 – 9
00001 – 0
11001 – 3
00010 – 8
01101 – 6

The account number resolved as 4391 5654 6719 0386.

5.6.6.4 Luhn validation:

8 3 9 1 1 6 1 4 3 7 2 9 0 3 7 6 = 70 = valid
8 18 10 10 12 2 0 16
4 3 9 1 5 6 5 4 6 7 1 9 0 3 8 6

5.7 Automation

If manual deciphering attempts fail, one may turn to scripting solutions. See Appendix 3 for the conceptual approach to building a model and Appendix 4 for a Python 2.7 deciphering script.

6 Appendix 1 - Decoding Scripts

6.1 Main Script

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

import argparse
import binascii
import math
import multiprocessing
import os
import signal

from tqdm import tqdm
from skimmer_analysis import track_decode


def build_parser():
    """Build the parser used for command line options."""
    parser = argparse.ArgumentParser(description="Carve credit card data from
binary files.")
    parser.add_argument('infile', type=argparse.FileType('rb'))
    parser.add_argument(
        '--blocksize', '-b', type=int, default=2**16, dest="chunksize",
        help="How many bytes to handle per process. Defaults to 2**16. The larger
this number, the more RAM required."
    )
    parser.add_argument(
        '--cores', '-c', type=int, default=None,
        help="How many cores to utilize while running. Defaults to maximum cores
available."
    )
    parser.add_argument(
        '--track', '-t', type=int, default=2, dest="tracknumber",
        help="Which track (1 or 2) to parse data from. Track 1 returns the PAN,
name, and expiration date for a card. " +
             "Track 2 only returns PANs. Use track number 0 to read from both track
1 and track 2. Defaults to track 2."
    )
    parser.add_argument(
        '--print-binary', '-p', action='store_true', dest="printbinary",
        help="Set this flag to make the script print out the binary form of all
the found verified PANs. " +
             "The binary will be the same as it is in the binary file read."
    )
    return parser


def chunk_read(file_like, chunk_size=2**16):
    """Yield chunk_size blocks from file_like object."""
    file_like.seek(0)
```

```
    while True:
        data = file_like.read(chunk_size)
        if not data:
            break
        yield data


def calculate_chunk_count(file_like, chunk_size=2**16):
    """Return the amount of chunks a given file will generate.

    Arguments:
        file_like: (file_like) File like object (open file, StringIO, etc.)
        chunk_size: (int) size (in bytes) of individual chunks

    Returns:
        int
    """
    file_size = os.fstat(file_like.fileno()).st_size
    size = int(math.ceil(file_size / float(chunk_size)))
    return size


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


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


def init_worker():
    """Ignore Interrupt signals in child processes spawned with this function."""
    signal.signal(signal.SIGINT, signal.SIG_IGN)


def auto_decode(_file, track_number, print_binary=False, chunk_size=2**16,
cores=None):
    """Manages the decoding process"""
    pool = multiprocessing.Pool(cores, init_worker)
    records = []

    chunk_count = calculate_chunk_count(_file, chunk_size)
    file_chunks = chunk_read(_file, chunk_size=chunk_size)

    if track_number == 1:
        function_pool = pool.imap_unordered(process_track1_data, file_chunks)
    else:
        function_pool = pool.imap_unordered(process_track2_data, file_chunks)

    try:
        for result in tqdm(function_pool, total=chunk_count):
            records.extend(result)
    except KeyboardInterrupt:
        print("\nKeyboard interrupt caught, dumping currently scraped data...")
        pool.close()
        pool.terminate()
    finally:
        if track_number == 1:
            track_decode.print_report_track1(records, print_binary=print_binary)
        else:
            track_decode.print_report_track2(records, print_binary=print_binary)


if __name__ == '__main__':
    parser = build_parser()
    args = parser.parse_args()
    if 1 <= args.tracknumber <= 2:
        auto_decode(args.infile, args.tracknumber, print_binary=args.printbinary,
chunk_size=args.chunksize, cores=args.cores)
    elif args.tracknumber == 0:
        print('Searching for track 1 data...')
        auto_decode(args.infile, 1, print_binary=args.printbinary,
chunk_size=args.chunksize, cores=args.cores)
        print('\n\n\n\nSearching for track 2 data...')
```

```
        auto_decode(args.infile, 2, print_binary=args.printbinary,
chunk_size=args.chunksize, cores=args.cores)
    else:
        print('Track number must be 0, 1, or 2. Track number entered: %d' %
args.tracknumber)
```

6.2 Dependency – Track Decode

```
import re
import string
import sys
import binascii

from skimmer_analysis import cc_verify


if sys.version[0] == '2':
    range = xrange


def int2bin(n, count=32):
    """Return the binary of integer n, using count number of digits"""
    # return ''.join([str((n >> y) & 1) for y in range(count - 1, -1, -1)])
    fmt_string = "{:0%db}" % count
    return fmt_string.format(n)


def get_binary_string(s):
    """Return a string of ascii 1's and 0's"""
    return ''.join([int2bin(ord(c), 8) for c in s])


def get_binary_string_reverse_endian(s):
    """Return a string of ascii 1's and 0's in reverse endian (bit order per
byte)"""
    return ''.join([int2bin(ord(c), 8)[::-1] for c in s])

# hexadecimal to binary conversion
n2b = {
    '0': '0000',
    '1': '0001',
    '2': '0010',
    '3': '0011',
    '4': '0100',
    '5': '0101',
    '6': '0110',
    '7': '0111',
    '8': '1000',
    '9': '1001',
    'a': '1010',
    'b': '1011',
```

```
    'c': '1100',
    'd': '1101',
    'e': '1110',
    'f': '1111',
}

# five bit ascii encoding scheme (4 bits + 1 parity)
f2a = {
    '00001': '0',  # (0H) Data
    '10000': '1',  # (1H)
    '01000': '2',  # (2H)
    '11001': '3',  # (3H)
    '00100': '4',  # (4H)
    '10101': '5',  # (5H)
    '01101': '6',  # (6H)
    '11100': '7',  # (7H)
    '00010': '8',  # (8H)
    '10011': '9',  # (9H)
    '01011': ':',  # (AH) Control
    '11010': ';',  # (BH) Start Sentinel
    '00111': '<',  # (CH) Control
    '10110': '=',  # (DH) Field Separator
    '01110': '>',  # (EH) Control
    '11111': '?',  # (FH) End Sentinel<>
}

# seven bit ascii encoding scheme (6 bits + 1 parity)
s2a = {
    '0000001': ' ',  # (0H)   Special
    '1000000': '!',  # (1H)      "
    '0100000': '"',  # (2H)      "
    '1100001': '#',  # (3H)      "
    '0010000': '$',  # (4H)      "
    '1010001': '%',  # (5H)   Start Sentinel
    '0110001': '&',  # (6H)   Special
    '1110000': "'",  # (7H)      "
    '0001000': '(',  # (8H)      "
    '1001001': ')',  # (9H)      "
    '0101001': '*',  # (AH)      "
    '1101000': '+',  # (BH)      "
    '0011001': ',',  # (CH)      "
    '1011000': '-',  # (DH)      "
    '0111000': '.',  # (EH)      "
    '1111001': '/',  # (FH)      "

    '0000100': '0',  # (10H)    Data (numeric)
    '1000101': '1',  # (11H)     "
    '0100101': '2',  # (12H)     "
    '1100100': '3',  # (13H)     "
    '0010101': '4',  # (14H)     "
    '1010100': '5',  # (15H)     "
    '0110100': '6',  # (16H)     "
    '1110101': '7',  # (17H)     "
```

```
    '0001101': '8',  # (18H)      "
    '1001100': '9',  # (19H)      "

    '0101100': ':',  # (1AH)    Special
    '1101101': ';',  # (1BH)       "
    '0011100': '<',  # (1CH)       "
    '1011101': '=',  # (1DH)       "
    '0111101': '>',  # (1EH)       "
    '1111100': '?',  # (1FH)    End Sentinel
    '0000010': '@',  # (20H)    Special

    '1000011': 'A',  # (21H)    Data (alpha)
    '0100011': 'B',  # (22H)      "
    '1100010': 'C',  # (23H)      "
    '0010011': 'D',  # (24H)      "
    '1010010': 'E',  # (25H)      "
    '0110010': 'F',  # (26H)      "
    '1110011': 'G',  # (27H)      "
    '0001011': 'H',  # (28H)      "
    '1001010': 'I',  # (29H)      "
    '0101010': 'J',  # (2AH)      "
    '1101011': 'K',  # (2BH)      "
    '0011010': 'L',  # (2CH)      "
    '1011011': 'M',  # (2DH)      "
    '0111011': 'N',  # (2EH)      "
    '1111010': 'O',  # (2FH)      "
    '0000111': 'P',  # (30H)      "
    '1000110': 'Q',  # (31H)      "
    '0100110': 'R',  # (32H)      "
    '1100111': 'S',  # (33H)      "
    '0010110': 'T',  # (34H)      "
    '1010111': 'U',  # (35H)      "
    '0110111': 'V',  # (36H)      "
    '1110110': 'W',  # (37H)      "
    '0001110': 'X',  # (38H)      "
    '1001111': 'Y',  # (39H)      "
    '0101111': 'Z',  # (3AH)      "

    '1101110': '[',  # (3BH)     Special
    '0011111': '\\',  # (3DH)     Special
    '1011110': ']',  # (3EH)     Special
    '0111110': '^',  # (3FH)     Field Separator
    '1111111': '_',  # (40H)     Special
}

# five and seven bit ascii without parity
f2a_np = {}
for key in f2a.keys():
    f2a_np[key[:-1]] = f2a[key]
s2a_np = {}
for key in s2a.keys():
    s2a_np[key[:-1]] = s2a[key]
```

```
# binary coded decimal
bin_dec = {
    '00': 0,
    '01': 1,
    '02': 2,
    '03': 3,
    '04': 4,
    '05': 5,
    '06': 6,
    '07': 7,
    '08': 8,
    '09': 9,
}


def trans_string(transforms, track):
    """Returns a string that explains the transformations done on the
        binary to find PANs.

    Arguments:
        transforms: (string) the string of transformation codes
            generated by the PAN parsing methods
        track: (int) which track the associated PAN was carved from

    Returns:
        string
    """
    ret = 'Printing all '
    if transforms[0] == '0':
        ret += 'binary encoded, '
    elif transforms[0] == '1':
        if transforms[1] =='1':
            return ret + 'ascii, reverse endian (bit-order in byte), track %d
PANs' % track
        return ret + 'ascii track %d PANs' % track
    else:
        ret += 'binary coded decimal track %d PANs' % track
        return ret
    if transforms[1] =='1':
        ret += 'reverse endian (bit-order in byte), '
    if transforms[2] == '1':
        ret += 'reverse swipe, '
    else:
        ret += 'forward swipe, '
    if transforms[3] == '1':
        ret += 'reverse endian (bit-order in character), '
    if '1' not in transforms[1:3]:
        ret = ret[:len(ret)-2] + ' '
    if transforms[4] == '1':
        if track == 1:
            ret += '7-bit PANs'
        else:
            ret += '5-bit PANs'
```

```
    else:
        if track == 1:
           ret += '6-bit PANs'
        else:
            ret += '4-bit PANs'

    return ret


def sort_ccs(unsorted_ccs, track):
    """Sort PANs into groups based on the transformations done on the
        binary in order to find the PAN.

    Arguments:
        unsorted_ccs: (list of tuples) the list of PANs found by the
            PAN parsing methods
        track: (int) which track the associated PANs were carved from

    """
    trans_data = 4  # location of transformation sequence in tuple
    if track == 2:
        trans_data = 2
    sorted_ccs = {}
    for cc in unsorted_ccs:
        if cc[trans_data] not in sorted_ccs.keys():
            sorted_ccs[cc[trans_data]] = [trans_string(cc[trans_data], track), cc]
        else:
            sorted_ccs[cc[trans_data]].append(cc)
    return sorted_ccs


def 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


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))
    # 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


def ascii_parse_track1(data, little_endian=False):
    """Returns all track 1 PANs found in ascii format

    Arguments:
        data: (string) the string to search for track 1 PANs
        little_endian: (boolean) whether or not to reverse the
            endianness (bit order in each byte) of the data

    Returns:
        list of tuples
```

```
    """
    # if little_endian, reverse the endianness
    if little_endian:
        bin_data = get_binary_string_reverse_endian(data)
        out = ''
        for idx in range(0, len(data)):
            out += str(chr(int(bin_data[idx*8:idx*8+7],2)))
        data = out

    # regex for full credit card data
    reg1 = '%.([0-9]{13,19})\^([A-Z /]{2,26})\^([0-9]{4}).{0,50}\?.'
    pat1 = re.compile(reg1)
    # regex for missing expiration date
    reg2 = '%.([0-9]{13,19})\^([A-Z /]{2,26})\^'
    pat2 = re.compile(reg2)
    # regex for missing name
    reg3 = '%.([0-9]{13,19})\^'
    pat3 = re.compile(reg3)

    # Use a dictionary at first to store PANs found so the same PAN
    # isn't read as 3 different PANs by each regex. Dictionary keys
    # are the positions at which the regexes find the match.
    card_dict = {}

    # search first for PANs while ignoring name and expiration date
    res = pat3.search(data, pos=0)
    while res is not None:
        bin = get_binary_string(res.group(1))
        transforms = transformations(type=1,reverse_endian=little_endian)
        card_dict[res.pos] = (res.group(1),'N/A','N/A',bin,transforms)
        res = pat3.search(data, pos=res.start() + 1)

    # search for names with PANs previously found
    res = pat2.search(data, pos=0)
    while res is not None:
        bin = get_binary_string(res.group(1))
        transforms = transformations(type=1,reverse_endian=little_endian)
        card_dict[res.pos] = (res.group(1),res.group(2),'N/A',bin,transforms)
        res = pat2.search(data, pos=res.start() + 1)

    # search for expiration dates for PANs previously found
    res = pat1.search(data, pos=0)
    while res is not None:
        bin = get_binary_string(res.group(1))
        transforms = transformations(type=1,reverse_endian=little_endian)
        card_dict[res.pos] =
(res.group(1),res.group(2),res.group(3),bin,transforms)
        res = pat1.search(data, pos=res.start() + 1)

    # now convert the dictionary of PANs to a list to return
    valid_cards = []
    for key in card_dict.keys():
        valid_cards.append(card_dict[key])
```

```
    #return []  #for testing only the bin_data decode

    return valid_cards


def ascii_parse_track2(data, little_endian=False):
    """Returns all track 1 PANs found in ascii format

    Arguments:
        data: (string) the string to search for track 1 PANs
        little_endian: (boolean) whether or not to reverse the
            endianness (bit order in each byte) of the data

    Returns:
        list of tuples
    """
    # if little_endian, reverse the endianness
    if little_endian:
        bin_data = get_binary_string_reverse_endian(data)
        out = ''
        for idx in range(0, len(data)):
            out += str(chr(int(bin_data[idx*8:idx*8+7],2)))
        data = out

    # regex for full track 2 format
    t2_re = ';([0-9]{13,19})=.{1,30}\?.'
    pat_t2 = re.compile(t2_re)

    valid_cards = []

    # search for all track 2 PANs
    res = pat_t2.search(data, pos=0)
    while res is not None:
        bin = get_binary_string(res.group(1))
        transforms = transformations(type=1,reverse_endian=little_endian)
        valid_cards.append((res.group(1), bin,transforms))
        res = pat_t2.search(data, pos=res.start() + 1)

    #return []  # for testing only the bin_data decode

    return valid_cards


def track1_exists(data, parity, reverse_endian):
    """Searches for any track 1 tart sentinels, then prints out the
        next few characters using the seven-bit to ascii dictionary.
        If the binary is an invalid character, '~' gets printed in
        its place. Use this function to do a semi-manual search for
        track 1 encoded data in a file.
    Arguments:

    """
```

```
    # maximum length of chars to read/print
    max_len = 50
    # minimum number of valid chars in a result for it to be printed
    min_len = 5
    # minimum number of digits in a result for it to be printed
    min_digs = 5
    # whether or not the result must match the regex to be printed
    use_reg = True
    # regex for the start of a valid PAN
    # (without a minimum length requirement)
    reg = re.compile('%.([0-9]{0,19})\^')

    # Mostly the same as search_bin_string() methods, but upon finding
    # a possible PAN, check if it meets the requirements to be printed.
    # If it meets all requirements, then print out the found string.
    if parity:
        char_size = 7
    else:
        char_size = 6
    sentinel = '1010001'[:char_size]
    if reverse_endian:
        sentinel = sentinel[::-1]

    while data.find(sentinel) != -1:
        index = data.find(sentinel) + char_size
        startIdx = index - char_size + 1
        toPrint = ''
        digits = 0
        while len(toPrint) < max_len:
            a_char = lookup(data[index:index + char_size], parity, reverse_endian)
            if a_char in s2a.keys():
                toPrint += s2a[a_char]
                if s2a[a_char] in string.digits:
                    digits += 1
            else:
                toPrint += '~'
            index += char_size
        if use_reg:
            res = reg.search(toPrint)
            if res is not None:
                print toPrint
        else:
            if len(toPrint) > min_len and digits > min_digs and toPrint[0] in
string.uppercase:
                print toPrint
        data = data[startIdx:]
    return []


def transformations(type=0, parity=True, reverse_endian=False,
reverse_swipe=False, data_reverse_endian=False):
    """Returns a string of an encoded list of all transformations done
        on some data in order to find some PAN.
```

```
    Arguments:
        type: (int) What encoding the PAN was in. 0 for track-specific
            ascii, 1 for plain ascii, 2 for binary-coded decimal.
            Defaults to 0.
        parity: (boolean) Whether or not the parity bit is included in
            in the binary. Only applies to type = 0. Defaults to True.
        reverse_endian: (boolean) Whether or not the characters had a
            bit-wise endian reversal. Only applies to type != 2.
            Defaults to False.
        reverse_swipe: (boolean) Whether or not the PAN was read as a
            reverse swipe. (Currently) only applies to type = 0.
            Defaults to False.
        data_reverser_endian: (boolean) Whether or not the data that
            the PAN was parsed from had a bit-order per byte
            endianness reversal. Defaults to False.

    Returns:
        string
    """
    return str(type) + str(int(data_reverse_endian)) + str(int(reverse_swipe)) +
str(int(reverse_endian)) + str(int(parity))


def search_track1_bin_string(data, parity=True, reverse_endian=False,
reverse_swipe=False, data_reverse_endian=False):
    """Returns all seven-bit encoded track 1 PANs in a chunk of data.

    Arguments:
        data: (string) the binary string to parse for PANs
        parity: (boolean) whether or not the characters have the parity
            bit included. Defaults to True.
        reverse_endian: (boolean) whether or not the characters are in
            reverse bit order. Defaults to False.
        reverse_swipe: (boolean) whether or not the data passed in is
            reversed from the original. Defaults to False.
        data_reverse_endian: (boolean) whether or not the data passed
            in is reverse endian (bit-order in each byte) from the
            original. Defaults to False.
    Returns:
        list of tuples
    """
    # uncomment to check if any track 1 data exists
    #return track1_exists(data, parity, reverse_endian)
    # set the char_size and sentinel based on parity and reverse_endian
    if parity:
        char_size = 7
    else:
        char_size = 6
    sentinel = '1010001'[:char_size]
    if reverse_endian:
        sentinel = sentinel[::-1]
```

```
    records = []
    index = 0
    # Start at every start sentinel location. Since data is cut at
    # byte boundaries, use mod 8 to resume search from 'index'.
    while data[index % 8:].find(sentinel) != -1 and len(data) > 8:
        # Whether or not we hit a field separator to break from the
        # previous loop. We'll search for the next field only if the
        # previous fields are in a valid format.
        valid_break = False
        # whether or not we found a potential PAN
        valid_number = False
        index = data[index % 8:].find(sentinel) + (index % 8) + char_size
        startIdx = index  # placeholder for the start index of the PAN
        cc = ''  # ascii PAN
        ccbin = ''  # binary representation of the PAN
        name = 'N/A'
        exp = 'N/A'  # expiration date
        # in track 1, ignore the first char after the sentinel
        index += char_size
        # loop to serach for the PAN
        while index + char_size <= len(data):
            a_char = lookup(data[index:index + char_size], parity, reverse_endian)
            # if a_char is a digit in the seven-bit ascii encoding
            if a_char in s2a.keys() and s2a[a_char] in string.digits:
                    cc += s2a[a_char]
            else:
                # if a_char is a field separator and the PAN is a valid length
                if a_char in s2a.keys() and s2a[a_char] == '^' and 13 <= len(cc)
<= 19:
                    index += char_size  # make sure we skip over the field
separator
                    valid_number = True  # breaking out of loop due to a field
separator
                    valid_break = True  # "
"
                # else: # not in s2a.keys(), not a digit, nor a field separator
                    # no need to change valid_number or valid_break since we
started by setting them to false
                break
            index += char_size
        # now try getting the name if the credit card number is valid
        while index + char_size <= len(data) and valid_break:
            a_char = lookup(data[index:index + char_size], parity, reverse_endian)
            # a_char is a valid char (uppercase letter, space, or slash)
            if a_char in s2a.keys() and s2a[a_char] in (string.uppercase + ' /'):
                name += s2a[a_char]
            else:
                if a_char in s2a.keys() and s2a[a_char] == '^':  # if a_char is a
field separator
                    index += char_size  # make sure we skip over the field
separator
                else:  # we hit a completely invalid char
                    name = 'N/A'
```

```
                    valid_break = False
                break
            if len(name) > 26:  # valid char appended, but now the name is over
the size limit
                name = 'N/A'
                valid_break = False
                break
            index += char_size
        # now try to get the expiration date if we got the name correctly,
otherwise we probably can't get it
        while index + char_size <= len(data) and valid_break:
            a_char = lookup(data[index:index + char_size], parity, reverse_endian)
            if a_char in s2a.keys() and s2a[a_char] in string.digits:  # if a_char
is a digit
                exp += s2a[a_char]
            else:
                exp = 'N/A'
                break
            if len(exp) == 4:  # we have the full expiration date, so break
                break
            index += char_size

        # if the PAN found is a valid format, undo the reverse_swipe
        # and data_reverse_endian transformations on the binary PAN
        # and add it to the results list
        if valid_number:
            if not data_reverse_endian:  # if data did NOT have endianness swapped
before function call
                ccbin = data[startIdx:index]
                if reverse_swipe:  # all we need to worry about is the reverse
swipe
                    ccbin = ccbin[::-1]
            if data_reverse_endian:  # if data DID have endianness swapped before
function call
                startIdx -= startIdx % 8  # move the start index back to the
beginning of the nearest byte
                endIdx = index + (8 - (index % 8))  # move the end index forward
to the end of the nearest byte
                ccbin = data[startIdx:endIdx]  # now we grab out binary PAN
                if reverse_swipe:  # since reverse swipe happens after the
endianness swap, undo it before the endianness swap
                    ccbin = ccbin[::-1]
                newccbin = ''  # temporary variable to hold the binary PAN as we
do an endianness swap
                for i in range(0, len(ccbin)/8):  # reverse the bit order in each
byte
                    newccbin += ccbin[i*8:(i*8)+8][::-1]
                ccbin = newccbin  # set ccbin to the corrected binary PAN

            transforms = transformations(type=0,
parity=parity,reverse_endian=reverse_endian,reverse_swipe=reverse_swipe,data_rever
se_endian=data_reverse_endian)
            records.append((cc, name, exp, ccbin, transforms))
```

```
        data = data[index - (index % 8):]  # cut off data at the first byte before
the index specified by 'index'
    return records


def search_track2_bin_string(data, parity=True, reverse_endian=False,
reverse_swipe=False, data_reverse_endian=False):
    """Returns all seven-bit encoded track 1 PANs in a chunk of data.

    Arguments:
        data: (string) the binary string to parse for PANs
        parity: (boolean) whether or not the characters have the parity
            bit included. Defaults to True.
        reverse_endian: (boolean) whether or not the characters are in
            reverse bit order. Defaults to False.
        reverse_swipe: (boolean) whether or not the data passed in is
            reversed from the original. Defaults to False.
        data_reverse_endian: (boolean) whether or not the data passed
            in is reverse endian (bit-order in each byte) from the
            original. Defaults to False.
    Returns:
        list of tuples
    """
    # Strict search was to use 2 additional search requirements
    # (length and field separator). It was added to resolve the
    # incosistency between the ascii regex and binary search, but
    # is currently disabled until some more testing is done.
    strict = False

    # set the char_size and sentinel based on parity and reverse_endian
    if parity:
        char_size = 5
    else:
        char_size = 4
    sentinel = '11010'[:char_size]
    if reverse_endian:
        sentinel = sentinel[::-1]

    records = []
    index = 0
    # Start at every start sentinel location. Since data is cut at
    # byte boundaries, use mod 8 to resume search from 'index'.
    while data[index%8:].find(sentinel) != -1 and len(data) > 8:
        index = data[index%8:].find(sentinel) + (index % 8) + char_size
        startIdx = index  # placeholder for the start index of the PAN
        cc = ''
        ccbin = ''  # binary of PAN
        valid_break = not strict  # strict - assume invalid break;  not strict -
any break is valid
        # check the next 5 bits, if a valid number keep going else stop
        while index + char_size <= len(data):
```

```
            a_char = lookup(data[index:index + char_size], parity, reverse_endian)
# get the char for lookup
            if strict and a_char in f2a.keys() and f2a[a_char] in string.digits
and len(cc) < 19:  # strict search
                cc += f2a[a_char]
            elif a_char in f2a.keys() and f2a[a_char] in string.digits:  # relaxed
search
                cc += f2a[a_char]
            else:
                if strict and a_char in f2a.keys() and f2a[a_char] == '=':  # if
doing a strict search, check for field separator
                    valid_break = True  # valid break if we break because of a
field separator
                break
            index += char_size

        # if the PAN found is a valid format, undo the reverse_swipe
        # and data_reverse_endian transformations on the binary PAN
        # and add it to the results list
        if valid_break and 13 <= len(cc) <= 19:
            if not data_reverse_endian:  # if data did NOT have endianness swapped
before function call
                ccbin = data[startIdx:index]
                if reverse_swipe:  # all we need to worry about is the reverse
swipe
                    ccbin = ccbin[::-1]
            if data_reverse_endian:  # if data DID have endianness swapped before
function call
                startIdx -= startIdx % 8  # move the start index back to the
beginning of the nearest byte
                endIdx = index + (8 - (index % 8))  # move the end index forward
to the end of the nearest byte
                ccbin = data[startIdx:endIdx]  # now we grab out binary PAN
                if reverse_swipe:  # since reverse swipe happens after the
endianness swap, undo it before the endianness swap
                    ccbin = ccbin[::-1]
                newccbin = ''  # temporary variable to hold the binary PAN as we
do an endianness swap
                for i in range(0, len(ccbin)/8):  # reverse the bit order in each
byte
                    newccbin += ccbin[i*8:(i*8)+8][::-1]
                ccbin = newccbin  # set ccbin to the corrected binary PAN

            transforms = transformations(type=0, parity=parity,
reverse_endian=reverse_endian, reverse_swipe=reverse_swipe,
data_reverse_endian=data_reverse_endian)
            records.append((cc, ccbin, transforms))

        data = data[index - (index % 8):]  # cut off data at the first byte before
the index specified by 'index'
    return records
```

```
def unique_swipes(ccs):
    return list(set(ccs))


def binary_coded_decimal_track1(hex_data):
    """Returns all track 1 PANs found encoded in binary-coded decimal.
       Currently the only difference between track 1 and track 2 is
       the format of the returned tuples.

    Arguments:
        hex_data: (string) the data to search in hexadecimal format
    Returns:
        list of tuples
    """
    hex_bytes = [hex_data[i:i + 2] for i in range(0, len(hex_data), 2)]
    records = []
    cc = ''
    ccbin = ''
    for i in range(len(hex_bytes)):
        a_char = hex_bytes[i]

        if a_char in bin_dec.keys():
            cc += str(bin_dec[a_char])
            ccbin += a_char
        else:
            if 13 <= len(cc) <= 19:
                transforms = transformations(type=2)
                records.append((cc,'N/A','N/A',ccbin,transforms))
            cc = ''
            ccbin = ''
    if 13 <= len(cc) <= 19:
        transforms = transformations(type=2)
        records.append((cc,'N/A','N/A',ccbin,transforms))
    return records


def binary_coded_decimal_track2(hex_data):
    """Returns all track 2 PANs found encoded in binary-coded decimal.
       Currently the only difference between track 1 and track 2 is
       the format of the returned tuples.

    Arguments:
        hex_data: (string) the data to search in hexadecimal format
    Returns:
        list of tuples
    """
    hex_bytes = [hex_data[i:i + 2] for i in range(0, len(hex_data), 2)]
    records = []
    cc = ''
    ccbin = ''
    for i in range(len(hex_bytes)):
        a_char = hex_bytes[i]
```

```
        if a_char in bin_dec.keys():
            cc += str(bin_dec[a_char])
            ccbin += get_binary_string(binascii.unhexlify(a_char))
        else:
            if 13 <= len(cc) <= 19:
                transforms = transformations(type=2)
                records.append((cc, ccbin, transforms))
            cc = ''
            ccbin = ''
    if 13 <= len(cc) <= 19:
        transforms = transformations(type=2)
        records.append((cc, ccbin, transforms))

    return records


def get_aba(binary_data, in_place=False, no_parity=False, reverse_endian=False):
    """ Return a tuple (error?, ascii_data) of from 5-bit ABA track 2

    Arguments:
        binary_data: a string of ascii 0's and 1's (like "01110101"...)
        in_place: must the start sentinel be at the beginning? or can we search
for it?
        no_parity: are parity bits included?
        reverse_endian: is the bit endianness reversed?
    """
    if no_parity:
        width = 4
    else:
        width = 5
    aba_data = ''
    aba_error = False
    if width == 4 and reverse_endian:
        binary_data = ''.join([binary_data[4 * i + 3] + binary_data[4 * i + 2] +
binary_data[4 * i + 1] + binary_data[4 * i] for i in range(len(binary_data) / 4)])
    if width == 5 and reverse_endian:
        binary_data = ''.join([binary_data[5 * i + 3] + binary_data[5 * i +
3]+binary_data[5 * i + 2]+binary_data[5 * i + 1] + binary_data[5 * i] for i in
range(len(binary_data) / 5)])
    if not in_place:
        if no_parity:
            binary_data = binary_data[binary_data.find('1101'):]
        else:
            binary_data = binary_data[binary_data.find('11010'):]
    for j in range(len(binary_data) / width):
        try:
            if no_parity:
                aba_data += f2a_np[binary_data[width * j:width * j + width]]
            else:
                aba_data += f2a[binary_data[width * j:width * j + width]]
        except KeyError:
            aba_error = True
            aba_data += ' '
```

```
    return aba_error, aba_data


def get_iso(binary_data, in_place=False, no_parity=False):
    """Return a tuple (error?, ascii_data) of from 7-bit ISO track 1

    Arguments:
        binary_data: a string of ascii 0's and 1's (like "01110101"...)
        in_place: must the start sentinel be at the beginning? or can we search
for it?
        no_parity: are parity bits included?
    """
    if no_parity:
        width = 6
    else:
        width = 7
    iso_data = ''
    iso_error = False
    if not in_place:
        if no_parity:
            binary_data = binary_data[binary_data.find('101000'):]
        else:
            binary_data = binary_data[binary_data.find('1010001'):]
    for j in range(len(binary_data) / width):
        try:
            if no_parity:
                iso_data += s2a_np[binary_data[width * j:width * j + width]]
            else:
                iso_data += s2a[binary_data[width * j:width * j + width]]
        except KeyError:
            iso_error = True
            iso_data += ' '
    return iso_error, iso_data


def get_ascii7(binary_data):
    """Convert binary data from 7-bit ascii to 8-bit ascii"""
    width = 7
    ascii7 = ''
    for j in range(len(binary_data) / width):
        ascii7 += chr(int(binary_data[width * j:width * j + width], 2))
    return ascii7


def lookup(bin_char, parity, reverse_endian):
    """Normalizes a binary string for the character lookup."""
    if reverse_endian:
        bin_char = bin_char[::-1]
    if not parity:
        # add the parity bit for the lookup
        if bin_char.count('1') % 2:
            bin_char += '0'
        else:
```

```
            bin_char += '1'
    return bin_char
```

6.3 Dependency - Credit Card Verify

```
#!/usr/bin/python

import sys
import os

__doc__          = "\n\
  Usage:\n\
    %s [actions] [options] ccnumber\n\
    \n\
  Actions:\n\
    -h, --help   Display this help message\n\
    -V           Show version information\n\
    \n\
  Options:\n\
    --no-luhn    Do not use Luhn verification\n" % (sys.argv[0])

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

def verify_cc(ccnumber, file_data=None, use_luhn=True):
    """Workaround for overloading the verify_cc method. Used since both methods
        use different algorithms for verifying PANs
    Arguments:
        ccnumber: (string) the credit card number / PAN to verify
        file_data: (array of string) the lines of binlist.txt in array
            format. Defaults to None.
        use_luhn: (boolean) whether or not to use the Luhn algorithm
            to check the ccnumber. Defaults to True.
    Returns:
        if ccnumber is valid, a dictionary of data about ccnumber
        otherwise None
    """
    if file_data:  # is not None:
        return verify_from_array(ccnumber, file_data=file_data, use_luhn=use_luhn)
    else:
        return verify_from_file(ccnumber, use_luhn=use_luhn)


def verify_from_array(ccnumber, file_data, use_luhn=True):
    """Verifies the ccnumber against the binlist scrape using an array
        rather than reading through the file. Much faster to use than
        verify_from_file().
    """
```

```
    if use_luhn and not check_luhn(ccnumber):
        return None
    # use first 6 digits of ccnumber as the index to look at
    line = file_data[int(ccnumber[:6])]
    card_data = {}
    if line.find("Invalid Card") != -1:
       return None
    if line.find("scheme") != -1:
        card_data["Scheme"] = find_Between(line, "scheme\":\"", "\"")
    if line.find("type") != -1:
        card_data["Type"]= find_Between(line, "type\":\"", "\"")
    if line.find("brand") != -1:
        card_data["Brand"]= find_Between(line, "brand\":\"", "\"")
    if line.find("prepaid") != -1:
        card_data["Prepaid"]= find_Between(line, "\"prepaid\":", ",\"")
    if line.find("country") != -1:
        card_data["Country"]= find_Between(line, "\",\"name\":\"", "\"")
    if line.find("currency") != -1:
        card_data["Currency"]= find_Between(line, "currency\":\"", "\"")
    if line.find("\"bank\":{\"name\":\"") != -1:
        card_data["Bank Name"]= find_Between(line, "\"bank\":{\"name\":\"", "\"")
    if line.find("\"url\":\"") != -1:
        card_data["Bank URL"]= find_Between(line, "\"url\":\"", "\"")
    if line.find("phone") != -1:
        card_data["Bank Phone"]= find_Between(line, "phone\":\"", "\"")
    return card_data

def verify_from_file(ccnumber, use_luhn=True):
    """The original verify_cc written for using the binlist scrape.
        Incudes minor edits to some return statements and file path.
    """
    file_path = os.path.join(os.path.dirname(os.path.abspath(__file__)),
'ccs.txt')
    #print 'Reading data from %s' % file_path
    file_obj = open(file_path,"r")
    if use_luhn and not check_luhn(ccnumber):
        return None
    cc_num = ccnumber[:6]
    card_data = {}
    for line in file_obj:
        if cc_num + ": " in line:
            if line.find("Invalid Card") != -1:
               return None
            if line.find("scheme") != -1:
                card_data["Scheme"] = find_Between(line, "scheme\":\"", "\"")
            if line.find("type") != -1:
                card_data["Type"]= find_Between(line, "type\":\"", "\"")
            if line.find("brand") != -1:
                card_data["Brand"]= find_Between(line, "brand\":\"", "\"")
            if line.find("prepaid") != -1:
                card_data["Prepaid"]= find_Between(line, "\"prepaid\":", ",\"")
            if line.find("country") != -1:
                card_data["Country"]= find_Between(line, "\",\"name\":\"", "\"")
```

```
            if line.find("currency") != -1:
                card_data["Currency"]= find_Between(line, "currency\":\"", "\"")
            if line.find("\"bank\":{\"name\":\"") != -1:
                card_data["Bank Name"]= find_Between(line,
"\"bank\":{\"name\":\"", "\"")
            if line.find("\"url\":\"") != -1:
                card_data["Bank URL"]= find_Between(line, "\"url\":\"", "\"")
            if line.find("phone") != -1:
                card_data["Bank Phone"]= find_Between(line, "phone\":\"", "\"")
    return card_data


def find_Between(s, prefix, suffix):
    """Sets search parameters to parse data"""
    try:
        start = s.index(prefix) + len(prefix)
        end = s.index(suffix, start)
        return s[start:end]
    except ValueError:
        return ""


def read_file():
    """Returns the binlist scrape results as an array. Index 0
        corresponds to the first six digits being 000000, index
        1 is 000001, etc.
    """
    file_path = os.path.join(os.path.dirname(os.path.abspath(__file__)),
'ccs.txt')
    # print 'Reading data from %s' % file_path
    file_obj = open(file_path, "r")
    lines = []
    for line in file_obj:
        lines.append(line)
    return lines


def main(args):
    import getopt
    try:
        ccnumber = args[-1]
        # note: args[1:] is not typical because first arg is always a file_name
        opts, args = getopt.getopt(args[0:], 'hV', ['help','no-luhn'])
    except getopt.GetoptError as ex:
        print >> sys.stderr, 'Invalid argument: ' + str(ex)
        print >> sys.stderr, __doc__
        sys.exit(1)
    except IndexError:
        print >> sys.stderr, 'Please specify a credit card number...'
        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)

    use_luhn = True

    # 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 == '--no-luhn'):
                use_luhn = False

    try:
        assert ccnumber.isdigit()
    except:
        print >> sys.stderr, 'CC number must be all digits (got %s)...' % ccnumber
        print >> sys.stderr, __doc__
        sys.exit(1)

    # let's do it
    cc_dictionary = verify_cc(ccnumber, use_luhn = use_luhn)
    if cc_dictionary:  # is not None:
        for x in cc_dictionary:
            print x,':',cc_dictionary[x]
    else:
        print ccnumber,'is not a valid credit card number.'

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:])
```

7 Appendix 2 - Microcontroller Code Analysis

In the event when the data retrieved from the flash chip on a skimmer proves difficult to analyze, the device's microcontroller may provide additional insight. Note: code protected microcontrollers can prevent analysis.

An example of a Bluetooth pairing name, "mariana", is shown below:

```
ROM:05AA ; ---------------------------------------------------
ROM:05AA                movff   INTCON, byte_RAM_E
ROM:05AE                bcf     INTCON, GIE_GIEH, ACCESS
ROM:05B0                clrf    TBLPTRH, ACCESS
ROM:05B2                addlw   0C4
ROM:05B4                movwf   TBLPTRL, ACCESS
ROM:05B6                movlw   5
ROM:05B8                addwfc  TBLPTRH, f, ACCESS
ROM:05BA                tblrd*+
ROM:05BC                movf    TABLAT, w, ACCESS
ROM:05BE                btfsc   byte_RAM_E, 7, ACCESS
ROM:05C0                 bsf     INTCON, GIE_GIEH, ACCESS
ROM:05C2                return  0
ROM:05C2 ; ---------------------------------------------------
ROM:05C4 aMariana       data "mariana"<0>
ROM:05CC ; ---------------------------------------------------
```

Figure 25 - Bluetooth code example

7.1 Common code examples

Typically microcontroller code is analyzed in a decompiler. What follows are some code example screen shots from such a program and an explanation of what is depicted:

's'	ROM:00EA	0000000B	C	AT+BTINFO?
's'	ROM:0110	0000000C	C	AT+BTMODE,3
's'	ROM:0136	00000013	C	AT+BTNAME=\"nokia0\"
's'	ROM:0164	0000000D	C	AT+BTSEC,1,0
's'	ROM:018C	00000012	C	AT+BTKEY=19641964
's'	ROM:01B8	0000000B	C	AT+USEDIP?
's'	ROM:01DE	00000019	C	AT+UARTCONFIG,9600,N,1,1

Figure 26 - Code decompilied

The above is a decompiler view of the configuration for a device's Bluetooth module. Notably, the "AT+BTNAME" parameter is what the device will broadcast as its name and AT+BTKEY is the pairing key needed to connect to the device. This information could prove useful in examinations if the subject uses the same password and name for a skimmer. Also, this information could be used to link a person

to a skimmer if the device the person used to connect via Bluetooth, e.g. cell phone, held corresponding information.
The below is another Bluetooth configuration parameter from the microcontroller code showing the Bluetooth pairing PIN. This can be used to possibly image the device over Bluetooth if the situation required such a process.

```
                DCB     0
aAuthS1111      DCB "AUTH %s 1111",0    ; DATA XREF: sub_370+1214↑o
                DCB     0
```

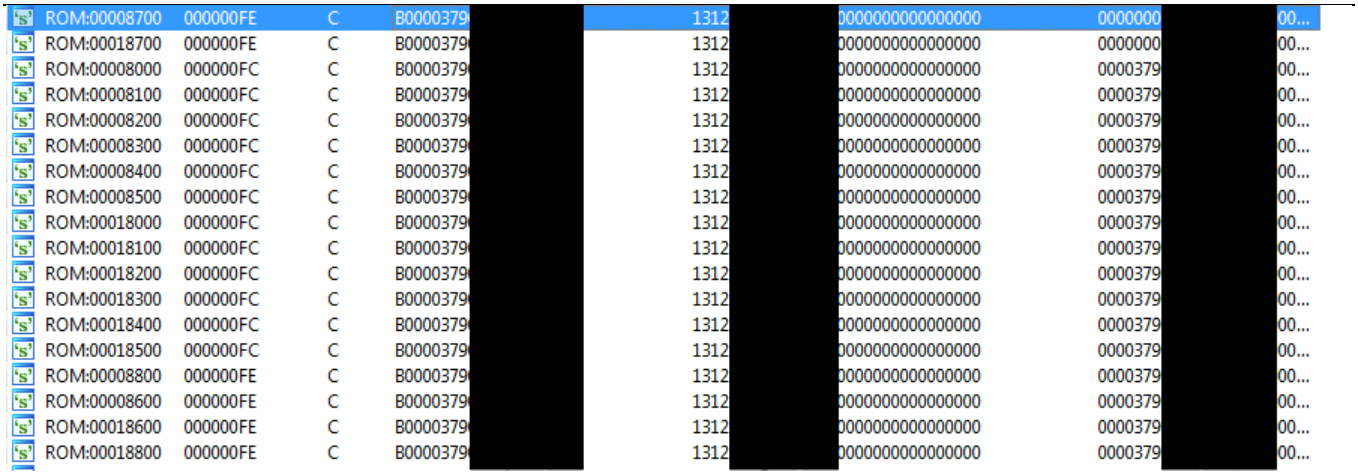
Figure 27 - Pairing PIN

This below is a snippet from a microcontroller showing the C-style format string for the card data it produces. The %XXX sequences tell the microcontroller how to format the data. In this particular case, the formatting is not doing anything tricky and pretty closely resembles a track 2 record minus the field separators and sentinels.

```
ADD     R0, SP, #0x140+var_B4
BL      sub_3334
ADD     R2, SP, #0x140+var_124
ADD     R1, SP, #0x140+var_B4
ADD     R0, SP, #0x140+var_A4
ADD     R3, SP, #0x140+var_134
STMIA   R3, {R0-R2,R7}
ADD     R3, SP, #0x140+var_84
ADD     R2, SP, #0x140+var_70
ADD     R1, SP, #0x140+var_50
STMEA   SP, {R1-R3}
ADD     R3, SP, #0x140+var_34
MOV     R2, R6
ADR     R1, aC019s26s032s01 ; "%c%019s%26s%032s%019s%032s%14s%109s%d"
LDR     R0, =unk_40000C9C
BL      vprintf
ADD     SP, SP, #0x120
LDMFD   SP!, {R4-R10,LR}
BX      LR
; End of function sub_22A4
```

Figure 28 - Format string

This formatting corresponds to the card data below.



ROM:00008700	000000FE	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000000[redacted]	00...
ROM:00018700	000000FE	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000000[redacted]	00...
ROM:00008000	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008100	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008200	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008300	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008400	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008500	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018000	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018100	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018200	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018300	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018400	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018500	000000FC	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008800	000000FE	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00008600	000000FE	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018600	000000FE	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...
ROM:00018800	000000FE	C	B0000379[redacted]	1312[redacted]	0000000000000000	0000379[redacted]	00...

Figure 29 - Formatted card data

This card data was stored on the microcontroller along with the code (1 chip for both code and data). This data could have been recovered using a strings dump of the file (or a hex editor). The format is defined by the code mentioned above.

8 Appendix 3 - Modeling Skimmers with Boolean Polynomials

The following describes the process for creating scripts for use with deciphering data extracted from skimmers.

8.1 Ring

Define a ring (a field of our variables) over Z2 = {0, 1}:

$$R = \mathrm{Z2}[k0 \ldots km{-}1, p0 \ldots pn{-}1, c0 \ldots cn{-}1]$$

Where *m* is the key length in bits and *n* is out cipher text/plain text length in bits.

8.2 Polynomials

These variables are used in sets of Boolean polynomials that describe some aspect of the skimmer.

- First polynomial
  It was stated that $pi \oplus kj = ci$. As every Boolean polynomial defined must be equal to zero, XOR both sides of the equation by *ci* so now r$pi + kj + ci = 0$.
  $$F\mathrm{cipher} = \{pi + kj + ci : j = i \text{ modulo } m \text{ and } 0 \leq i < n\}$$
- Second polynomial
  Note that every track 2 character has an odd parity bit as its fifth bit.
  So, $pi + pi1 + pi2 + pi3 + pi4 + 1 = 0$ where *i* is the first bit of the character is always true.
  $$F\mathrm{parity} = 1 + \sum_{j=i}^{i+4} pj : 0 \equiv i \pmod 5 \text{ and } 0 \leq i < n$$

Other equations:

- Key restriction to printable ASCII
- Decimal and other value-restricted fields
- Absence of specific control characters
- Predictability of control character locations
- Luhn verification of the primary account number
- Longitudinal redundancy check (LRC)

8.3 Cipher bits

Create known cipher text equations that define what the cipher text bits are. For every cipher text bit, define the polynomial:

- *ci* if the *i*th cipher text bit is zero
- *ci* + 1 if the *i*th cipher text bit is one

8.4 Gröbner basis

Together, all of the sets of defined Boolean polynomials create an *ideal*. One then attempts to compute a reduced Gröbner basis of this ideal and possibly solve for most or all of the key bits (which reveals most or all of the plain text bits).

- A Boolean polynomial/Gröbner basis python module called polybori is used for defining the equations and to compute the reduced Gröbner basis.

9 Appendix 4 - Deciphering Scripts

9.1 Single Byte – 8bit ASCII

```
# XOR Decoding Script (Works with Python >3.2 and 2.7)


import sys, os, string

# Print header
print("")
print("")
print("-------------------------------------------------------------------------")
print("$               Keyed XOR Skimmer Binary Decoder v1.5                $")
print("-------------------------------------------------------------------------")
print("")
print("Detecting python version: " + str(sys.version_info[0]) + "." +
str(sys.version_info[1]) + "." + str(sys.version_info[2]))
print("")

# Version compatibility
if (sys.version_info >= (3,2,0)): deprecated = False
else: deprecated = True

# Get filename from command line argument
if (len(sys.argv) != 3):
    print("Usage: python " + os.path.basename(sys.argv[0]) + " <input binary
filename> <output filename>")
    sys.exit()

filename = str(sys.argv[1])
output = str(sys.argv[2])

# Prompt for number of bytes
prompt = '-'
while prompt not in
['','1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16',

'17','18','19','20','21','22','23','24','25','26','27','28','29','30','31','32']:
    if (deprecated): prompt = raw_input("Key length in bytes (default: 1)? ")
    else: prompt = input("Key length in bytes (default: 1)? ")
if (prompt == '1' or prompt == ''): chunkSize = 1
else: chunkSize = int(prompt)
#chunkSize = 32

print("Loading file '" + filename + "'.")

# Prompt to skip frequency analysis
prompt = '-'
while (prompt != 'Y' and prompt != 'y' and prompt != 'N' and prompt != 'n' and
prompt != ''):
```

```
    if (deprecated): prompt = raw_input("Perform frequency analysis? (Y|n)")
    else: prompt = input("Perform frequency analysis? (Y|n)")
if (prompt == 'Y' or prompt == 'y' or prompt == ''):
    # Build blank statistical table
    statsInd = []
    statsVal = []

    # Read binary file one byte at a time for statistical analysis
    print("Frequency analysis running.")

    # Test for mathematician header
    statsMath = []
    mathOffsets = []
    offset = -6

    # Load file, looping by byte
    with open(filename, "rb") as file:
        chunk = file.read(chunkSize)
        while chunk:
            index = 0

            # Scan for mathematician and perform frequency analysis
            for x in range(0,len(chunk)):
                # Fill statsMath
                if (len(statsMath) == 6): statsMath.pop(0)

                if (deprecated):
                    statsMath.append(ord(chunk[x]))
                    index += 256**(chunkSize-x-1)*ord(chunk[x])
                else:
                    statsMath.append(chunk[x])
                    index += 256**(chunkSize-x-1)*chunk[x]

                offset += 1
                if (statsMath == [0, 1, 1, 0, 0, 45]): mathOffsets.append(offset)

            if index in statsInd:
                statsVal[statsInd.index(index)] += 1
            else:
                statsInd.append(index)
                statsVal.append(1)
            chunk = file.read(chunkSize)
    file.close()

    # Print warning for mathematician
    prompt = '-'
    if (len(mathOffsets) > 0):
        print("")
        print("WARNING: " + str(len(mathOffsets)) + " Mathematician header(s)
detected at offset(s): " + str(mathOffsets)[1:-1])
        print("")
        print("If you run the Mathematician script, press ctrl + c to exit when
you see a solved key.")
```

```
        print("")
        while (prompt != 'Y' and prompt != 'y' and prompt != 'N' and prompt != 'n'
and prompt != ''):
            if (deprecated): prompt = raw_input("Do you want to run the
Mathematician script? (Y|n)")
            else: prompt = input("Do you want to run the Mathematician script now?
(Y|n)")

    if (prompt == 'Y' or prompt == 'y' or prompt == ''):
        if (deprecated):
            os.system("python '" + os.path.dirname(os.path.realpath(__file__)) +
"/mathematician_track2.py' -f " + filename)
            print("Running Mathematician script and exiting.")
            sys.exit(0)
        else: print("Mathematician requires Python version 2, continuing with
script")

    # Get top results
    maxKey = [-1, -1, -1, -1, -1]
    freq = [-1, -1, -1, -1, -1]

    # Set max recursively
    for x in range(0,5):
        index = statsVal.index(max(statsVal))
        maxKey[x] = statsInd[index]
        freq[x] = statsVal[index]
        statsVal.pop(index)
        statsInd.pop(index)

    # Inverse encoded key XOR with ASCII zero
    pad0 = "0b00000000" # 0x00
    pad30 = "0b00110000" # 0x30 or '0' in ASCII
    padF = "0b11111111" # 0xFF
    pad = ""

    # Build key sizes based on chunkSize
    for x in range(1,chunkSize):
        pad += str("  ")
        pad0 += "00000000"
        pad30 += "00110000"
        padF += "11111111"
    pad0 = int(pad0, 2)
    pad30 = int(pad30, 2)
    padF = int(padF, 2)

    # Print frequency table
    print("")
    print("Table lists top 5 most frequent byte(s) based on the key length
selected.")
    print("Choose decode key based on which byte(s) is/are expected to be most
common.")
    print("For XOR encrypted skimmers, the default is ASCII '0' or 0x30 for most
track data.")
```

```
    print("")
    print("Byte " + pad + "| Key (0x" + hex(pad30)[2:].zfill(2*chunkSize) + ") |
Key (0x" + hex(pad0)[2:].zfill(2*chunkSize) + ") | Key (0x" +
hex(padF)[2:].zfill(2*chunkSize) + ") | Frequency ")

    for x in range(0,5):
        print("0x" + hex(maxKey[x])[2:].zfill(2*chunkSize) + " |    0x" +
hex(maxKey[x]^pad30)[2:].zfill(2*chunkSize) + "    |    0x" +
hex(maxKey[x]^pad0)[2:].zfill(2*chunkSize) + "    |    0x" +
hex(maxKey[x]^padF)[2:].zfill(2*chunkSize) + "    |    " + str(freq[x]))
    print("")

    # Choose encoded byte based on frequency analysis
    print("Most frequent byte: 0x" + hex(maxKey[0])[2:].zfill(2*chunkSize))

    print("1) Decoded key (0x" + hex(pad30)[2:].zfill(2*chunkSize) + "): 0x" +
hex(maxKey[0]^pad30)[2:].zfill(2*chunkSize) + ". (default)")
    print("2) Decoded key (0x" + hex(pad0)[2:].zfill(2*chunkSize) + "): 0x" +
hex(maxKey[0]^pad0)[2:].zfill(2*chunkSize) + ".")
    print("3) Decoded key (0x" + hex(padF)[2:].zfill(2*chunkSize) + "): 0x" +
hex(maxKey[0]^padF)[2:].zfill(2*chunkSize) + ".")
    print("4) Manual key entry.")

    prompt = '-'
    while (prompt != '1' and prompt != '2' and prompt != '3' and prompt != '4' and
prompt != ''):
        if (deprecated): prompt = raw_input("Enter number: ")
        else: prompt = input("Enter number: ")
    if (prompt == '1' or prompt == ''): key = maxKey[0]^pad30
    elif (prompt == '2'): key = maxKey[0]^pad0
    elif (prompt == '3'): key = maxKey[0]^padF
    elif (prompt == '4'): prompt = 'n'

# Pick the most frequent key and prompt user to use it or select their own
if (prompt == 'N' or prompt == 'n'):
    prompt = '-'
    while (not (all(c in string.hexdigits for c in prompt) and len(prompt) ==
2*chunkSize)):
        if prompt != '-': print("Input must be a hex value between '0x" +
hex(pad0)[2:].zfill(2*chunkSize) + "' and '0x" + hex(padF)[2:].zfill(2*chunkSize)
+ "'.")
        if (deprecated): prompt = raw_input("Enter a decoding key (hex): 0x")
        else: prompt = input("Enter a decoding key (hex): 0x")
    key = int(prompt, 16)
    print("Using decoding key: 0x" + hex(key)[2:].zfill(2*chunkSize))
else:
    print("Using decoding key: 0x" + hex(key)[2:].zfill(2*chunkSize))

# Decode binary with key and output to file
print("Decoding file.\nSaving binary to '" + output + "'.")

# Open output and input files and parse input decoding each byte into the output
file
```

```
fileout = open(output, "wb")
with open(filename, "rb") as file:
    chunk = file.read(chunkSize)

    if (deprecated):
        while chunk:
            # Proccess each byte per chunk
            for x in range(0,len(chunk)):
                # Decrypt each byte
                decoded_byte =
ord(chunk[x])^int(hex(key)[x*2+2:x*2+4].zfill(2),16)

                # Write byte to file
                fileout.write(chr(decoded_byte))
            chunk = file.read(chunkSize)
    else:
        while chunk:
            decoded_chunk = int.from_bytes(chunk, byteorder=sys.byteorder)^key
            decoded_chunk = decoded_chunk.to_bytes(chunkSize,
byteorder=sys.byteorder)
            fileout.write(decoded_chunk)
            chunk = file.read(chunkSize)
file.close()
fileout.close()

print("Process complete.")
```

9.2 Non-8bit ASCII / Non-single byte cipher

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

# cryptanalysis

import os
from optparse import OptionParser

from skimmer_analysis.skimmer_xor_track2 import *

from skimmer_analysis.track_decode import *


def simplify_repeated_string(s):
    """ Simplifies repeated strings
            'aaa' -> 'a'
            '1212' -> '12'
    """
    if s == '':
        return s
    for i in range(1, len(s)/2 + 1):
        if s[:i]*(len(s)/i) == s:
            return s[:i]
```

```
    return s

def read_tracks(filename):
    # read in each record
    in_file = open(filename, 'rb')

    # a list containing our encrypted track datas
    tracks = []

    # read in the encrypted tracks
    record = 'cruft'
    while True:
        record = in_file.read(0x80)  # read 128 bytes
        if record == '' or record == '\xff'*0x80:  # if null string or all
unallocated memory
            break
        size = ord(record[6])
        track = record[7:7+size]
        if track != '':
            tracks.append(track)
    return tracks


def cryptanalyze(options):
    keys = []

    tracks = read_tracks(options.file)

    if options.track:
        tracks = [tracks[options.track],]
        track_number = options.track
    else:
        track_number = 0

    # now we try to cryptanalyze each one
    # we might not be able to properly align our parity bits
    # for reverse swipes....  but hopefully we'll hit a
    # nice forward swipe.
    for track_number, track in enumerate(tracks):
        if not options.max_ctxt_length >= len(track) >= options.min_ctxt_length:
            if options.max_ctxt_length < len(track):
                print 'Track %d (%s) too long to cryptanalyze' % (track_number,
repr(track))
            else:
                print 'Track %d (%s) too short to cryptanalyze' % (track_number,
repr(track))
        else:
            print 'Cryptanalyzing track %d (%s)' % (track_number, repr(track))
            # convert each track to a string of "1"'s and "0"'s
            binary_ctxt = ''.join([int2bin(ord(b), 8) for b in track])

            binary_ctxt_final = binary_ctxt
```

```
            for keylength in range(8*options.min_key_length,
8*(options.max_key_length+1), 8):
                for leading_zeros in range(options.leading_zeros):
                    for direction in ['forward', 'reverse']:
                        try:
                            if direction == 'forward':
                                print '    Forward swipe, removing %d leading
bits' % leading_zeros
                                binary_ctxt = binary_ctxt_final[leading_zeros:]
                            elif direction == 'reverse':
                                print '    Reverse swipe, removing %d leading
bits' % leading_zeros
                                binary_ctxt = binary_ctxt_final[::-
1][leading_zeros:]
                            else:
                                assert(False)

                            if options.trim:
                                if len(binary_ctxt) > options.trim:
                                    binary_ctxt = binary_ctxt[:options.trim]
                            else:
                                # we don't know if the last byte contains portions
of the last 1, 2, or 3 characters,
                                # so we shouldn't define equations for it.
additionally, in actual skimmers, the last
                                # few bytes might contain junk
                                binary_ctxt = binary_ctxt[:(8*-3)]

                            k_b = Block('k', keylength)
                            p_b = Block('p', len(binary_ctxt))
                            c_b = Block('c', len(binary_ctxt))

                            ring = declare_ring([k_b, p_b, c_b])

                            # make it simple and just set up the parity equations
                            ideal_list = []

                            # odd parity bit for every four data bits
                            for i in range(0, len(binary_ctxt) - 5 + 1, 5):
                                ideal_list.append(p(i) + p(i+1) + p(i+2) + p(i+3)
+ p(i+4) + 1)

                            # xor cipher equations that define the crypto system
used
                            if direction == 'forward':
                                for i in range(0, len(binary_ctxt)):
                                    ideal_list.append(p(i) + c(i) + k(i %
keylength))
                            elif direction == 'reverse':
                                clen = len(binary_ctxt) - 1
                                offset = (len(binary_ctxt_final) -
len(binary_ctxt) - leading_zeros) % keylength
                                for i in range(len(binary_ctxt)):
```

```
                                    ideal_list.append(p(clen - i) + c(clen - i) +
k((i + offset) % keylength))

                            # substitute known values
                            # we know our ciphertext
                            for i, each in enumerate(chosen_bits(c, 0,
binary_ctxt)):
                                ideal_list.append(each)

                            # evaluate given a ciphertext with known and chosen
                            if options.start_sentinel_loc is not None:
                                # we think we know the start sep, 5,
options.pan_sentinel
                                ideal_list.append(p(options.start_sentinel_loc +
0) + 1)
                                ideal_list.append(p(options.start_sentinel_loc +
1) + 1)
                                ideal_list.append(p(options.start_sentinel_loc +
2))
                                ideal_list.append(p(options.start_sentinel_loc +
3) + 1)
                            if options.pan_length is not None:
                                # we think we know the field separator
                                field_sep_loc = 5 * (1 + options.pan_length)
                                ideal_list.append(p(field_sep_loc + 0) + 1)
                                ideal_list.append(p(field_sep_loc + 1))
                                ideal_list.append(p(field_sep_loc + 2) + 1)
                                ideal_list.append(p(field_sep_loc + 3) + 1)
                            if options.end_sentinel_loc is not None:
                                # we think we know the end sentinel
                                ideal_list.append(p(options.end_sentinel_loc + 0)
+ 1)
                                ideal_list.append(p(options.end_sentinel_loc + 1)
+ 1)
                                ideal_list.append(p(options.end_sentinel_loc + 2)
+ 1)
                                ideal_list.append(p(options.end_sentinel_loc + 3)
+ 1)


                            if options.key_ascii:
                                # we think we know the key is printable ascii
                                for i in range(0, keylength - 8 + 1, 8):
                                    for each in ascii(k, i):
                                        ideal_list.append(each)
                            if options.key_decimal:
                                # we think we know the key is printable ascii
decimal characters
                                for i in range(0, keylength - 8 + 1, 8):
                                    for each in ascii_decimal(k, i):
                                        ideal_list.append(each)
                            if options.impossible_control:
                                # we know these control characters aren't found
here
```

```
                                for i in range(0,len(binary_ctxt)-4,5):
                                    for each in invalid_control_chars(p, i):
                                        ideal_list.append(each)
                            if options.pan_length:
                                if options.pan_prefix:
                                    # we are guessing the first few digits of the
PAN
                                    for each in chosen_track2_chars(p, 5,
options.pan_prefix):
                                        ideal_list.append(each)
                                if options.service_code and options.pan_length:
                                    # we are guessing the service code
                                    for each in chosen_track2_chars(p, 5*(1 +
options.pan_length + 1 + 4), options.service_code):
                                        ideal_list.append(each)
                                # pan is decimal
                                if len(binary_ctxt) >= 5*(1 + options.pan_length):
                                    for i in range(options.pan_length):
                                        for each in decimal(p, 5*(i + 1)):
                                            ideal_list.append(each)
                                # expiration date is decimal
                                if len(binary_ctxt) >= 5*(1 + options.pan_length +
1 + 4):
                                    for each in expiration_date(p, 5*(1 +
options.pan_length + 1)):
                                        ideal_list.append(each)
                                # service code is decimal
                                if len(binary_ctxt) >= 5*(1 + options.pan_length +
1 + 4 + 3):
                                    for each in service_code(p, 5*(1 +
options.pan_length + 1 + 4)):
                                        ideal_list.append(each)
                                # we think we know the length of the PAN and can
apply even luhn
                                if len(binary_ctxt) >= 5*(1 + options.pan_length):
                                    for each in luhn_even(p, 5, pan_length =
options.pan_length):
                                        ideal_list.append(each)

                            # solve
                            """
                            import pprint
                            pprint.pprint(ideal_list)
                            """
                            G = groebner_basis(ideal_list, heuristic = False)
                        except IndexError:
                            print '        Variable indexing was out of bounds...
Needed more ciphertext'
                            continue


                        try:
                            num_solved_key = 0
```

```
                            num_solved_ptxt = 0
                            for f in G:
                                if f.n_variables():
                                    if f.vars_as_monomial() in [k(i) for i in
range(keylength)]:
                                        num_solved_key += 1
                                if f.n_variables():
                                    if f.vars_as_monomial() in [p(i) for i in
range(len(binary_ctxt))]:
                                        num_solved_ptxt += 1
                                #         print 'solved:',
                                #print f

                            if G == [1,]:
                                print '        Contradiction for key length of %d
bits' % keylength
                            else:
                                print '        Key bits solved: %d / %d bits' %
(num_solved_key, keylength)
                            #print '        Plaintext bits solved: %d / %d' %
(num_solved_ptxt, len(binary_ctxt))

                            # if we cracked the key
                            if num_solved_key == keylength:
                                binary_key = ''
                                for i in range(keylength):
                                    if k(i) in G:
                                        binary_key += '0'
                                    elif k(i) + 1 in G:
                                        binary_key += '1'
                                    else:
                                        # this should never happen
                                        assert(False)
                                key = ''
                                for i in range(len(binary_key)/8):
                                    key += chr(int(binary_key[8*i:8*(i+1)], 2))
                                simplified_key = simplify_repeated_string(key)
                                if key == simplified_key:
                                    print '            Key: ' + repr(key)
                                else:
                                    print '            Key: %s (same as %s)' %
(repr(key), repr(simplified_key))
                                keys.append(key)
                            elif num_solved_key / float(keylength) > 0.8 or
options.verbose:
                                # if we cracked more than 80% of the key, print
what we got
                                binary_key = ''
                                for i in range(keylength):
                                    if k(i) in G:
                                        binary_key += '0'
                                    elif k(i) + 1 in G:
                                        binary_key += '1'
```

```
                                    else:
                                        binary_key += '?'
                                print '            Binary partial key: ' +
binary_key
                                key = ''
                                for i in range(len(binary_key)/8):
                                    if '?' in binary_key[8*i:8*(i+1)]:
                                        key += '?'
                                    else:
                                        key += chr(int(binary_key[8*i:8*(i+1)],
2))
                                print '            Partial key: ' + repr(key)

                        except Exception, e:
                            print e

    if keys:
        simplified_keys = [simplify_repeated_string(key) for key in keys]
        unique_keys = list(set(simplified_keys))
        unique_keys.sort()
        print ''
        print 'Potential keys found:'
        for key in unique_keys:
            print '    %s (x%d)' % (repr(key), simplified_keys.count(key))
        for key in unique_keys:
            print 'Decrypting with key %s:' % repr(key)
            options.key = key
            decrypt(options)
            print ''
    else:
        print ''
        print 'No keys found'

def search_aba(binary_ptxt):
    tracks = []
    data = binary_ptxt
    data = data[data.find('11010'):]
    while len(data) > 4:
        track = get_aba(data)[1]
        tracks.append(track)
        data = data[1:]
        data = data[data.find('11010'):]
    return tracks

def decrypt(options):
    tracks = read_tracks(options.file)

    if options.track:
        tracks = [tracks[options.track],]
        track_number = options.track
    else:
        track_number = 0
```

```
    ccs = []

    # now we decrypt each one
    for track in tracks:
        # convert each track to a string of "1"'s and "0"'s
        binary_ctxt = ''.join([int2bin(ord(c), 8) for c in track])
        binary_key  = ''.join([int2bin(ord(c), 8) for c in options.key])
        binary_ptxt = xor(binary_ctxt, binary_key)

        if options.verbose:
            print 'Analyzing forward plaintext bits:',
            print binary_ptxt
            abas = search_aba(binary_ptxt)
            for aba in abas:
                print 'Decoded track %d:' % track_number,
                print aba
            print 'Analyzing reverse plaintext bits:',
            print binary_ptxt[::-1]
            abas = search_aba(binary_ptxt[::-1])
            for aba in abas:
                print 'Decoded track %d:' % track_number,
                print aba

        ccs += search_track2_bin_string(binary_ptxt) +
search_track2_bin_string(binary_ptxt[::-1])
        track_number += 1
    print_report_track2(ccs)


if __name__ == '__main__':
    usage = 'usage: %prog [options]'
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="file",
                      help="specify the skimmer's eeprom memory dump to read and
cryptanalyze", metavar="FILE")
    parser.add_option("-k", "--key", dest="key", default='',
                      help="specify an ascii key for decryption", type="string")
    parser.add_option("", "--service-code", default='',
                      help="specify a three digit service code", type="string")
    parser.add_option("", "--pan-prefix", default='',
                      help="specify a PAN prefix", type="string")
    parser.add_option("", "--track",
                      type="int", default=None,
                      help="operate only on track a specific track number")
    parser.add_option("-z", "--leading-zeros",
                      type="int", default=10,
                      help="specify the maximum number of possible leading zeros
[default=%default]")
    parser.add_option("-t", "--trim",
                      type="int", default=None,
                      help="specify a size (in bits) to trim the ciphertext down
to before performing cryptanalysis [default=%default]")
    parser.add_option("-M", "--min-ctxt-length",
```

```
                      type="int", default=15,
                      help="specify the minimum ciphertext length in bytes to
attempt cryptanalysis [default=%default]")
    parser.add_option("-X", "--max-ctxt-length",
                      type="int", default=25,
                      help="specify the maximum ciphertext length in bytes to
attempt cryptanalysis [default=%default]")
    parser.add_option("-m", "--min-key-length",
                      type="int", default=1,
                      help="specify the minimum key length in bytes
[default=%default]")
    parser.add_option("-x", "--max-key-length",
                      type="int", default=10,
                      help="specify the maximum key length in bytes
[default=%default]")
    parser.add_option("-p", "--pan-length",
                      type="int", default=None,
                      help="specify the pan length in digits (forward only)")
    parser.add_option("-s", "--start-sentinel-loc",
                      type="int", default=None,
                      help="specify the start sentinel location in bits")
    parser.add_option("-e", "--end-sentinel-loc",
                      type="int", default=None,
                      help="specify the end sentinel location in bits")
    parser.add_option("-A", "--key-ascii", action="store_true", default=False,
                      help="specify that the key must be ascii
[default=%default]")
    parser.add_option("-D", "--key-decimal", action="store_true", default=False,
                      help="specify that the key must be ascii decimal characters
[default=%default]")
    parser.add_option("-i", "--impossible-control", action="store_true",
default=False,
                      help="specify that impossible control characters should not
appear [default=%default]")
    parser.add_option("", "--restrictive", action="store_true", default=False,
                      help="try cryptanalyzing with very restrictive defaults
[default=%default]")
    parser.add_option("", "--lax", action="store_true", default=False,
                      help="try cryptanalyzing with very lax defaults
[default=%default]")
    parser.add_option("-v", "--verbose", action="store_true", default=False,
                      help="specify verbose output [default=%default]")

    (options, args) = parser.parse_args()
    if len(args) != 0:
        parser.error('incorrect number of arguments')

    if options.min_key_length > options.max_key_length:
        parser.error('minimum key length is greater than maximum key length')

    if not os.path.isfile(options.file):
        parser.error('input file does not exist')
```

```
    if options.service_code:
        if len(options.service_code) != 3:
            parser.error('service code must be three digits')

    if options.key:
        decrypt(options)
    else:
        if options.lax:
            options.key_ascii = False
            options.impossible_control = False
            options.pan_length = None
            options.start_sentinel_loc = None
            options.end_sentinel_loc = None
        elif options.restrictive:
            options.trim = 5 * (1 + 16 + 1 + 4 + 3)
            options.key_ascii = True
            options.key_decimal = True
            options.impossible_control = True
            options.pan_length = 16
            options.start_sentinel_loc = 0
            #options.pan_prefix = '44'
            #options.pan_prefix = '43'
            #options.service_code = '101'
            #options.service_code = '521'
        cryptanalyze(options)
```

9.2.1 Dependencies – XOR Track 2

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

from polybori import *

def subst(f,x,c):
    # from polybori docs
    i=x.index()
    c=Polynomial(c)#if c was int is now converted mod 2,
    #so comparison to int(0) makes sense
    s=f.set()
    if c==0:
        #terms with x evaluate to zero
        return Polynomial(s.subset0(i))
    else:
        #c==1
        return Polynomial(s.subset1(i))+Polynomial(s.subset0(i))


f2a = {'00001': '0', # (0H) Data
       '10000': '1', # (1H)
       '01000': '2', # (2H)
       '11001': '3', # (3H)
```

```
        '00100': '4', # (4H)
        '10101': '5', # (5H)
        '01101': '6', # (6H)
        '11100': '7', # (7H)
        '00010': '8', # (8H)
        '10011': '9', # (9H)
        '01011': ':', # (AH) Control
        '11010': ';', # (BH) Start Sentinel
        '00111': '<', # (CH) Control
        '10110': '=', # (DH) Field Separator
        '01110': '>', # (EH) Control
        '11111': '?', # (FH) End Sentinel<>
}
a2f = {}
for key, value in f2a.iteritems():
    a2f[value] = key

def int2bin(n, count=32):
    """returns the binary of integer n, using count number of digits"""
    return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)])

def xor(binary_ptxt, binary_key):
    return ''.join([str(int(x) ^ int(binary_key[i % len(binary_key)])) for i,x in
enumerate(binary_ptxt)])

def eq_binary_string(var, i, b):
    polys = []
    for j, bit in enumerate(b):
        assert bit in ['0','1']
        polys.append(var(j+i) + int(bit, 2))
    return polys

def neq_binary_string(var, i, b):
    #poly = Polynomial(1)
    #for j, bit in enumerate(b):
    #    assert bit in ['0','1']
    #    poly = poly * (var(j+i)+((int(bit)+1) % 2))
    #return poly
    loops = 0
    poly = 0
    for j, bit in enumerate(b):
        assert bit in ['0','1']
        if(loops == 0):
            poly = Polynomial(var(j+i)+((int(bit)+1) % 2))
        else:
            poly = poly * (var(j+i)+((int(bit)+1) % 2))
        loops += 1
    return poly

def decimal(var, i):
    """ we think that this character is a number
    """
    return [
```

```
                neq_binary_string(var, i, '0101'),
                neq_binary_string(var, i, '1101'),
                neq_binary_string(var, i, '0011'),
                neq_binary_string(var, i, '1011'),
                neq_binary_string(var, i, '0111'),
                neq_binary_string(var, i, '1111'),
            ]

def invalid_control_chars(var, i):
    """ we think that there are no control symbols ('01011', '00111', '01110')
through the entire plaintext
        we could do a subrange of this if we are unsure
        does not appear to affect the grobner basis
    """
    return [
                neq_binary_string(var, i, '0101'),
                neq_binary_string(var, i, '0011'),
                neq_binary_string(var, i, '0111'),
            ]

def ascii(var, i):
    """ we think that this is a printable ascii character (0b00100000 -
0b01111110)
        this should only be used on the key
    """
    return [
        var(i),
        (var(i+1)+1)*(var(i+2)+1),
        var(i+1)*var(i+2)*var(i+3)*var(i+4)*var(i+5)*var(i+6)*var(i+7),
        ]

def ascii_decimal(var, i):
    """ we think that this is a printable ascii decimal character (0b00110000 -
0b00111001)
        this should only be used on the key
    """
    return [
        var(i),
        var(i+1),
        var(i+2)+1,
        var(i+3)+1,
        neq_binary_string(var, i+4, '101'),
        neq_binary_string(var, i+4, '110'),
        neq_binary_string(var, i+4, '111'),
        ]

def service_code(var, i):
    """ we think we know where the service code is and that it is left to right
    """
    return decimal(var, i) + decimal(var, i + 5) + decimal(var, i + 10) + \
        [
            neq_binary_string(var, i, '0000'), # 0
            neq_binary_string(var, i, '1100'), # 3
```

```
            neq_binary_string(var, i, '0010'), # 4
            neq_binary_string(var, i, '0001'), # 8

            neq_binary_string(var, i+5, '1000'), # 1
            neq_binary_string(var, i+5, '1100'), # 3
            neq_binary_string(var, i+5, '1010'), # 5
            neq_binary_string(var, i+5, '0110'), # 6
            neq_binary_string(var, i+5, '1110'), # 7
            neq_binary_string(var, i+5, '0001'), # 8
            neq_binary_string(var, i+5, '1001'), # 9

            neq_binary_string(var, i+10, '0001'), # 8
            neq_binary_string(var, i+10, '1001'), # 9
        ]

def expiration_date(var, i, possible_years = range(2004, 2029), possible_months =
range(1,13)):
    """ we think we know where the expiration date is and that it is left to right
    """
    dec = decimal(var, i) + decimal(var, i + 5) + decimal(var, i + 10) +
decimal(var, i + 15)
    decimals = '0123456789'
    possible_y1 = set([('%0.2d' % year)[-2] for year in possible_years])
    possible_y2 = set([('%0.2d' % year)[-1] for year in possible_years])
    possible_m1 = set([('%0.2d' % month)[-2] for month in possible_months])
    possible_m2 = set([('%0.2d' % month)[-1] for month in possible_months])
    y1 = [neq_binary_string(var, i, int2bin(int(digit),4)[::-1]) for digit in
decimals if digit not in possible_y1]
    y2 = [neq_binary_string(var, i+5, int2bin(int(digit),4)[::-1]) for digit in
decimals if digit not in possible_y2]
    m1 = [neq_binary_string(var, i+10, int2bin(int(digit),4)[::-1]) for digit in
decimals if digit not in possible_m1]
    m2 = [neq_binary_string(var, i+15, int2bin(int(digit),4)[::-1]) for digit in
decimals if digit not in possible_m2]
    return dec + y1 + y2 + m1 + m2

def luhn_even(var, i, pan_length = 16):
    """ we think we know where the pan is and how long it is and that it passes
luhn verification
            var: plaintext variable
            i: index of the first bit of pan
            pan_length: length of the primary account number
    """
    def b(var, i):
        return neq_binary_string(var, i, '1010') + \
               neq_binary_string(var, i, '0110') + \
               neq_binary_string(var, i, '1110') + \
               neq_binary_string(var, i, '0001') + \
               neq_binary_string(var, i, '1001')
    singles = [var(5*j + i) for j in range(pan_length-1, -1, -2)]
    doubles = [b(var, 5*j + i) for j in range(pan_length-2, -1, -2)]
    even = sum(singles + doubles)
    return [even,]
```

```
def chosen_bits(var, i, b):
    """ we know the bits characters at this location
            var: variable
            i: index of the first chosen bit
            b: string of chosen bits
    """
    return eq_binary_string(var, i, b)


def chosen_track2_chars(var, i, s):
    """ we know the bits characters at this location
            var: variable
            i: index of the first chosen bit
            s: string of chosen track 2 characters
    """
    chosen = []
    for lcv in range(len(s)):
        chosen += chosen_bits(var, i + 5*lcv, a2f[s[lcv]])
    return chosen


# if this script was run directly rather than imported,
# run an example script
if __name__ == '__main__':
    # randomly generated, passes luhn
    ptxt = ';0373382470817953=14129478901234567890?'
    # a 48-bit ascii key
    # you can change this to whatever you want to observe how the effectiveness
    # of cryptanalysis changes with the key size
    key = '123456'
    # knowing or properly guessing the primary account number (pan) can produce
    # some very useful equations
    pan_length = 16
    # strings of ascii 1's and 0's will represent our binary
    binary_ptxt = ''.join([a2f[x] for x in ptxt])
    binary_key = ''.join([int2bin(ord(x),count=8) for x in key])
    binary_ctxt = xor(binary_ptxt, binary_key)
    # the keylength is something that we have to guess correctly.
    # if we do not guess it correctly but the rest of our equations are
    # correct, we will be presented with a boolean polynomial of 1 as an
    # answer.  this corresponds to the equation 1 = 0 which is never true.
    # this is called a contradiction, and it let's us know that something
    # we assumed cannot be true.  this lets us identify incorrectly guessed
    # keylengths.
    keylength = len(binary_key)

    k = Block('k', keylength)
    p = Block('p', len(binary_ptxt))
    c = Block('c', len(binary_ctxt))

    ring = declare_ring([k, p, c])
```

```
    for ideals in range(4):
        ideal_list = []
        # odd parity bit for every four data bits
        for i in range(0, len(binary_ptxt), 5):
            ideal_list.append(p(i)+p(i+1)+p(i+2)+p(i+3)+p(i+4)+1)

        # xor cipher
        for i in range(0, len(binary_ptxt)):
            ideal_list.append(p(i)+c(i)+k(i % keylength))

        G = groebner_basis(ideal_list, heuristic = False)

        # evaluate given a ciphertext with known values
        #print 'Grobner Basis evaluated with known values:'
        # substitute known values
        # we know our ciphertext
        for i in range(len(binary_ctxt)):
            ideal_list.append(c(i)+int(binary_ctxt[i]))

        G = groebner_basis(ideal_list, heuristic = False)
        #for f in G:
        #    print f

        # evaluate given a ciphertext with known and chosen
        # print 'Grobner Basis evaluated with known and chosen values:'
        # substitute known values
        if ideals >= 1:
            # we think we know the start sentinel
            ideal_list.append(p(0)+1)
            ideal_list.append(p(1)+1)
            ideal_list.append(p(2))
            ideal_list.append(p(3)+1)
            # we think we know the field separator
            ideal_list.append(p(85)+1)
            ideal_list.append(p(86))
            ideal_list.append(p(87)+1)
            ideal_list.append(p(88)+1)
            # we think we know the end sentinel
            ideal_list.append(p(190)+1)
            ideal_list.append(p(191)+1)
            ideal_list.append(p(192)+1)
            ideal_list.append(p(193)+1)

        if ideals >= 2:
            # we think we know the key is printable ascii
            for i in range(0, keylength, 8):
                for each in ascii(k, i):
                    ideal_list.append(each)
            # we know these control characters aren't found here
            for i in range(0,len(binary_ctxt)-4,5):
                for each in invalid_control_chars(p, i):
                    ideal_list.append(each)
            # pan is decimal
```

```
            for i in range(pan_length):
                for each in decimal(p, 5*i+5):
                    ideal_list.append(each)
            # expiration date is decimal
            for each in expiration_date(p, 5*(1 + pan_length + 1)):
                ideal_list.append(each)
            # service code is decimal
            for each in service_code(p, 5*(1 + pan_length + 1 + 4)):
                ideal_list.append(each)

        if ideals >= 3:
            # we think we know the length of the PAN and can apply even luhn
            for each in luhn_even(p, 5, pan_length = pan_length):
                ideal_list.append(each)

        try:
            if ideals >= 1:
                G = groebner_basis(ideal_list, heuristic = False)
            num_solved_key = 0
            num_solved_ptxt = 0
            for f in G:
                if f.n_variables():
                    if f.vars_as_monomial() in [k(i) for i in range(keylength)]:
                        num_solved_key += 1
                if f.n_variables():
                    if f.vars_as_monomial() in [p(i) for i in
range(len(binary_ptxt))]:
                        num_solved_ptxt += 1
                #        print 'solved:',
                #print f

            ideals_str = 'parity'
            if ideals >= 1:
                ideals_str += ', known sentinels'
            if ideals >= 2:
                ideals_str += ', value restriction'
            if ideals >= 3:
                ideals_str += ', even luhn'

            print '  Ideals %s:' % ideals_str
            print '    Key bits solved: %d / %d' % (num_solved_key, keylength)
            print '    Plaintext bits solved: %d / %d' % (num_solved_ptxt,
len(binary_ptxt))
        except Exception, e:
            print e
        #raw_input()
```

9.2.2 Dependencies – Track Decode

As included above

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