October 18, 2020

NSA Codebreaker 2020
Mr. Todd Mateer

Mr. Mateer,

Thank you again for reaching out to me about Task 6. I'm fairly new to college-level CTFs, so it means a lot that you're commending my efforts. As you suggested, I'll document here my thought process when solving the problem, and tell you about what little background I have in coding theory.

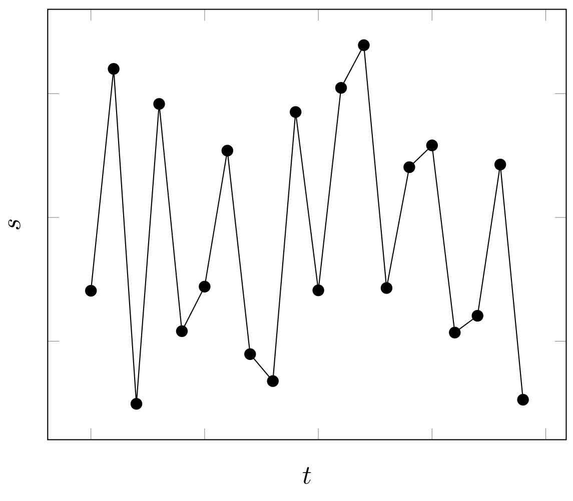
To start, I wanted to take the signal we were given and make it into something more readable. So, I wrote a simple Python program to parse each of the 16-bit floats and print them out. I was worried I'd have to write a parser myself, based off the Wikipedia article on the Half-precision Floating-point Format, but thankfully Python's `struct` library supports 16-bit floats since Python 3.6.

```python
import struct
import sys

file_name = sys.argv[1]
file_contents = open(file_name, 'rb').read()
float_iter = struct.iter_unpack('<e', file_contents)
for f in float_iter:
    print(f[0])
```

The result of this was a long list of floats, as expected. I didn't notice that the task stated the signal had already been demodulated, so I went and tried to plot the floats as a waveform. I thought the signal was still BPSK encoded and that I'd have to demodulate it, so I wanted to at least see the data before working with it.

Plot of Part of the Signal



It became clear that I wouldn't have to demodulate the signal. There weren't any smooth sine curves like I'd expect the actual transmission to have. So, I went on assuming that the transmission was already demodulated, with each float presumably corresponding to a single bit. That is, the recon team did the first step of BPSK demodulation for us, then sampled it using a bit-clock, but just didn't convert it to binary. (Note that I did the task before the clarification about one bit per float was given.)

To make the rest of the sections easier to follow, I'll diverge a bit from my process while I was solving the problem. I'll make a file that just contains a "bitstring" of the data. I use quotes since I'm just going to use the ASCII characters "0" and "1" to represent the data. Having this makes the following code much easier to follow. The actual Python code to do this is very much like the initial decoding step. The inner part of the loop is the only change.

```
for f in float_iter:
    print(1 if f[0] > 0 else 0, end='')
```

Coming back to my actual workflow, at this point it was simply a question of getting details about the Hamming code the signal used. I'd recently watched 3Blue1Brown's video on Hamming codes. It introduced the concept very well, and gave me a few takeaways useful in this task. One was that Hamming codes use blocks of size 2^r or $2^r - 1$. So, if the signal was Hamming encoded, I'd expect its length to have factors of that form:

```
sage: divisors(9572547)
[1,
 3,
 17,
 51,
 61,
 ...
```

The only factors of 9 572 547 that looked promising were $3 = 2^2 - 1$ and $17 = 2^4 + 1$. I first tried 3 since it was the only factor that fit the required form exactly. A Hamming code on three bits is just the three-bit repetition code, so I quickly implemented that in Python. The script outputs ASCII "0"s and "1"s, so I converted it to a sequence of bytes by piping the result through the Perl command I found on Stackexchange.

```
import sys

file_name = sys.argv[1]
file_handle = open(file_name, 'r')

# While the file has stuff in it
while True:
    # Check if we're done
    bit_chars = file_handle.read(3)
    if len(bit_chars) != 3:
        break

    # Check which bit is in the majority
    bit_ints = map(lambda c: int(c) - int(b'0'), bit_chars)
    sum_over = sum(bit_ints)
    print(1 if sum_over >= 2 else 0, end='')
```

```
perl -pe 'BEGIN { binmode \*STDOUT } chomp; $_ = pack "B*", $_'
```

Unsuprisingly, this didn't work. I just got garbage data out the other end. So, I reasoned that the data probably came in packets of seventeen, with some extra padding in each group. To actually see how this might be being done, I took my "bitstring" and `fold`ed it to seventeen characters.

```
$ cat solution/to_bitstring/result.txt | fold -w 17 | head
01010010010110110
01001010001011110
10010001100110110
11000101101111010
00000000101110110
10000000001101000
00000101010101010
11001001001100110
00100000010011000
01100010010100010
```

I quickly noticed that the last bit in each group of seventeen was almost always zero, and I assumed that it was just a padding bit. Using this, I was able to approximate the error rate in this data. There were 689 lines ending with a padding bit of one and 563 090 lines total, giving an error probability of about 0.12% per bit. More importantly, I now had groups of sixteen, a common size for Hamming codes. I assumed the data was using a $(15, 11)$ Hamming code with an extra parity bit, backing this by the fact many lines had even parity, as expected.

Now, I wanted to work out which bits were parity and which were data. I was given that the code was systematic, and looking up the definition on Wikipedia gives that the "plaintext" data appears inside the encoded data somewhere. So, I made the assumption that the first few groups had no errors, found an online Hamming code calculator, and started plugging in consecutive bits of the data.

I had no luck with this method. Counting the expected number of parity ones and zeros seldom gave consistent matches. Slowly it dawned on me that the data probably didn't use the "standard" Hamming code, and that I'd have to figure out what it was using. Granted, this makes sense since the task asks for the parity-check matrix, which wouldn't be very useful unless it was non-standard.

But before diving head-first into error correction, I wanted to make sure I was at least on the right track. The Wikipedia article on Hamming codes gives systematic code-generation and parity-check matricies for the $(7, 4)$ case. It seems that systematic Hamming codes have the left-most minor of $\mathbf{G}$ be the identity matrix, meaning the first 11 bits (in our case) would be the original data, assuming no errors. To test this, I took the first 11 bits in each group of 17 and wrote the data into a file using the Perl command from earlier.

```
$ cat solution/to_bitstring/result.txt                              \
    | fold -w 17                                                    \
    | sed -E -e 's/[0-1]{6}$//g'                                    \
    | tr -d '\n'                                                    \
    | perl -pe 'BEGIN { binmode \*STDOUT } chomp; $_ = pack "B*", $_'   \
    > solution/sixteen_bit_no_error_correction/result.avi
```

Miraculously, this worked, kind of. It produced a file recognized as an AVI by `file`. However, VLC complained that the file's index was missing, and trying to play the video anyway resulted in garbage. Nonetheless, the fact that the magic bytes were correct gave me the confidence to move forward with this form of error correction.

To proceed, I first tried to find the code-generation matrix. I read a bit on them, and most of the material was familar to me. 3Blue1Brown's aforementioned video mentioned XOR, priming me to think back to my experience working with $\mathbb{F}_2$. Most of the Linear Algebra we did in Georgia Tech's MATH 1564 was over $\mathbb{R}$, but we discussed how the theory can be extended to an arbitrary field, so working over $\mathbb{F}_2$ wasn't that much of a stretch.

Before going forward however, I'll introduce some notation for vectors. For some row or column vector $\mathbf{v}$, I denote its k-th component as v_k. In order to denote a sequence of vectors, I'll write $\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \cdots$. This way, I can still reference the components of each vector. For instance, $v_i^{(j)}$ denotes the i-th component of the j-th vector in the sequence $\mathbf{v}$.

From my previous experiment, it became clear that the code-generation matrix $\mathbf{G} \in M_{11\times 16}(\mathbb{F}_2)$ had form

$$\mathbf{G} = \begin{bmatrix}\mathbf{I}_{11} & \mathbf{A}\end{bmatrix}.$$

To solve for $\mathbf{A} \in M_{11\times 5}(\mathbb{F}_2)$, I considered its column vectors $\mathbf{a}^{(i)}$ as well as some messages, each consisting of eleven data bits $\mathbf{d}^{(j)}$ and five parity bits $\mathbf{p}^{(j)}$. I assumed the messages to be uncorrupted, hoping I could recognize and replace ones that were. Under that assumption

$$\mathbf{d}^{(j)} \cdot \mathbf{a}^{(i)} = p_i^{(j)}$$

for $i = 1, \cdots, 5$ and any j, where I use $\cdot$ to mean a dot-product. To write this in matrix form, we can take N messages in total and define (using $\mathbf{d}^{(j)}$ and $\mathbf{p}^{(j)}$ as row vectors)

$$\mathbf{D} = \begin{bmatrix}\mathbf{d}^{(1)} \\ \mathbf{d}^{(2)} \\ \vdots \\ \mathbf{d}^{(N)}\end{bmatrix}$$
$$\mathbf{P} = \begin{bmatrix}\mathbf{p}^{(1)} \\ \mathbf{p}^{(2)} \\ \vdots \\ \mathbf{p}^{(N)}\end{bmatrix}$$

to get

$$\mathbf{DA} = \mathbf{P}.$$

I arbitrarily read in the first $N = 20$ groups, however any group of 11 or more uncorrupted messages would've worked. I wrote some SageMath code to do the calculations (in `solve_a`), and fed it `to_bitstring`'s result. The output was

$$\mathbf{A} = \begin{bmatrix}1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0\end{bmatrix}.$$

From there, I found the parity-check matrix using the formula on the Parity-check Matrix's Wikipedia article:

$$\begin{aligned}\mathbf{H} &= \begin{bmatrix}-\mathbf{A}^\top & \mathbf{I}_5\end{bmatrix} \\ &= \begin{bmatrix}\mathbf{A}^\top & \mathbf{I}_5\end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.\end{aligned}$$

This solves the first half of the task.

As for the second half, we start by finding all the possible syndromes as $\mathbf{s}^{(i)} = \mathbf{H} \cdot \mathbf{e}^{(i)}$, where $\mathbf{e}^{(i)}$ is the i-th basis vector in $\mathbb{F}_2^{16}$. I used these syndromes for Syndrome Decoding, again heavily referencing the Wikipedia article. It appears the basic idea is to observe that $\mathbf{H} \cdot \mathbf{m}^\top = \mathbf{0}$ for any "valid" message $\mathbf{m}$. If it experiences a one bit error — it's added to $(\mathbf{e}^{(i)})^\top$ — then the result of computing the parity check will simply be $\mathbf{s}^{(i)}$, due to the linearity of transposition and of matrix multiplication. We then look-up this syndrome and see what error could cause it.

During the task, I computed all the syndromes as $\mathbf{H} \cdot \mathbf{I}_{16}$, however it occurs to me now that the result is just $\mathbf{H}$. So here, I just used that as our syndrome look-up table.

I went through each of the 16-bit groups, computed its syndrome, and if it wasn't $\mathbf{0}$, I looked up the column in $\mathbf{H}$ and subtracted out the error. If I couldn't find the syndrome, I just gave up. There might be a way to correct two- or more-bit errors with the information we have, but we'll see later it's not needed. Again, I wrote some SageMath code to do the calculations for me, and piped the result through the Perl script to get a binary file.

```
$ sage solution/sixteen_bit_one_bit_error_correction/code.sage solution/to_bitstring/result.txt \
    | perl -pe 'BEGIN { binmode \*STDOUT } chomp; $_ = pack "B*", $_'                          \
    > solution/sixteen_bit_one_bit_error_correction/result.avi
```

The result produced by `sixteen_bit_one_bit_error_correction` is still very corrupted, but it's nonetheless playable by VLC. The video starts by showing an empty room, with the timestamp in the top left. The screen then fades to black, then fades back in with the hostage being dragged to the chair in the center of the room, all amidst significant data corruption. The timestamp was sufficiently legible while the hostage was being shown, so I read it and submitted it, solving the second half of the task.

That's more or less how I solved Task 6. I have no idea how closely I followed the intended solution, and I would like you to send it to me if you feel comfortable doing so. I also wrote down some of the information I came across on Wikipedia when researching how to do this challenge. Please do correct me if any of that is wrong. Finally, please ask me any questions you have about this writeup. I noticed this task was much easier than last year's Task 7, but I guess most of the difficulty will be in Part 2. Other than that, it was an interesting challenge. As a person recreationally interested in math, I liked getting to apply some of the more "advanced" stuff I've learned. I look forward to seeing more challenges from you.

Thank you,
Ammar Ratnani