Codes and Cryptography Assignment

Dom Newman

1 Cracking a DES encrypted W3W address

Before performing any cracking, I checked whether a weak key was being used for DES, that is, a key for which E(E(t)) = t. This was not the case, so I used non-trivial cracking techniques.

1.1 Training a Neural Network to learn the DES encryption

My first approach was to try and approximate the encryption using a fully connected neural network similar to the approach in [1]. My first step was to generate several hundred thousand plaintext ciphertext combinations for training. I then let the neural network learn the relationship from the ciphertext to the plaintext. As this wouldn't be able to predict perfectly after training, I would give it the supplied ciphertext for the w3w address and inspect the output by eye, or use the what3words api which can suggest corrections to input to try and find the right answer. This approach however failed. The neural network was not able to learn the relationship well and predicted the same output for every input. I tried with different network widths, depths and activation functions as well as different representations of the data such as one hot, but still very little learning took place. I suspect either better representation of the data is needed, or another model. [1] used a cascading neural network which may be more effective than a simple fully connected network.

1.2 Using a dictionary attack

My next approach was to use a dictionary attack on the ciphertext. Although using chosen plaintext attacks for DES to find the key, such as linear cryptography was an option, it would require 2^{43} or more plaintext cipher text combinations, which was not feasible. It was much faster to try and guess the ciphertext directly. I decided to use a method in which I would guess a plaintext p, encrypt it as c' = E(p), then check if c' = c, where c is 90 34 08 ec 4d 95 1a cf ae b4 7c a8 83 90 c4 75.

The biggest issue was the speed of the encrypt program. On a single core I could run the encrypt program 3-5 times a second from python. Using my machine, which has a 6 core I7 processor, I was able to run an instance of encrypt.exe on each core, giving 30 encryptions a second. If I encrypted plaintexts for 24hours, this would give me 2,592,000 encryptions. Given the input size of $16^{32} = 3.4 \times 10^{38}$, it would take more than 10^{31} days to try halve of them which is a little out of scope for the coursework.
I had to significantly reduce the search space to have a chance of cracking the code, so I used several assumptions:

1. The item had no padding, what was encrypted was the original full plain text in ascii and encoded as hex.
2. The plaintext only contains 16 ascii characters (2 nibbles per character, 32 hex values/nibbles in ciphertext)
3. The plaintext is composed of 3 English words with 2 dots between them, reduces search space to less than 170000^3
4. The plaintext is made of 3 words that are 3-8 letters in length and are somewhat common. Around 23,000 words fit this description, so we can get an upper bound of $23000^3 = 10^{13}$, although the real value is much lower than this as we do not consider how combinations of words further reduce the search space.
5. There are only 14 available characters in the plaintext, which means the average word length is a short 4.7 characters. Looking at what3words, it is evident they use more common and shorter words

for populated places, and longer words for the ocean or remote areas. Therefore, The location is somewhere populated and on land.

6. The plaintext can be broken into two halves e.g (word.wor—d2.word3), which were encrypted independently with the same key. This way, we can crack one half the plaintext and use that to make a very good guess on the next half. This provides the biggest reduction to the search space, purely considering bits it reduces a search space of 2^{128} to 2 * 2^{64}.

Building on the idea introduced above of splitting, I downloading a word list of 50,000 words, and used python to split them into two sets of text files:

- A set of files containing words of length n, for $3 \leq n \leq 7$.
- A set of files containing the starts of all words grouped by length, it does not consider words longer than length 9 in order to further reduce search space. e.g start3.txt contains all starts of words that are 3 letters long, like hun for hungry (but not sup from superstitious as too long).

Using these I was able to reduce the search space to the following for different combinations, for example:

- (words length 5) . (starts length 2) = 4226 * 270 = 1,141,020 combinations.
  $1141020 \div (30 * 60 * 60) = 10.6, \div 2 = 5.3$ hours of cracking on average
- (words length 4) . (starts length 3) = 2294 * 2251 = 5,163,794 combinations.
  $5163794 \div (30 * 60 * 60) = 47.8, \div 2 = 23.9$ hours of cracking on average

This means I could expect to find the answer after a few days of cracking. The pseudo code for this algorithm is shown below:

Algorithm 1 Cracking the first half

```
Require: firstLength, cipherText
  f = file with words of length 'firstLength'
  s = file of words starts of length 7 - firstLength
  for each firstWord in f do
    for each secondWord in s do
      plaintextGuess = firstWord + '.' + secondWordStart
      c = encrypt(plaintextGuess)
      if c == cipherText then
        print(plaintextGuess)
        exit()
      end if
    end for
  end for
```

After I have found the first half, I can then get a list of words that start with the end of the last word in the first search and then combine those with the words in my dictionary. For example, if I found 'bil' as the last word when searching the first 8 characters / 16 hexes, I might guess bo.short as the other half. I'll then try all 5 letters words with bo.`{5letword}`, if that doesn't work ill try another word that starts with bil, like bil : ly and repeat the process.

I implemented this system in python, encoding the ascii plaintext guesses to hex then calling encrypt.exe with that as an argument. After implementing this system in python, I let my cracker run on the 5 . 3 word combos then 4 . 3 combos which finally found the first half as tile.bil. I then found words of reasonable (3-8) length that started with 'bil', for which there were only 40 possible words, each of these needed to be combined with between 500-7000 other words (depending on the length of the middle word) to make predictions. This allowed me to then make a very quick search until I found the full location:

tile.bills.print

This was found after around 40 hours of searching and the location is the 4 seasons but the meme one.

References

[1] X. Hu and Y. Zhao. "research on plaintext restoration of aes based on neural network". *Security and Communication Networks*, 2018, 2018.