# DAT 510: Assignment 1

Tomasz Gliniecki

17.09.2016

## Abstract

The assignment was to crack ciphertexts, encrypted with different encryption algorythms, and also implement Simplified DES, and Triple SDES with two keys. To crack the ciphertexts, neccesary software tools were written in Python and C++. Ciphertexts provided with the assignment were cracked succesfully.

# Part I

## Decrypted text

CRYPTOGRAPHY CAN BE STRONG OR WEAK CRYPTOGRAPHIC STRENGTH IS MEASURED IN THE TIME AND RESOURCES IT WOULD REQUIRE TO RECOVER THE PLAINTEXT THE RESULT OF STRONG CRYPTOGRAPHY IS CIPHERTEXT THAT IS VERY DIFFICULT TO DECIPHER WITHOUT POSSESSION OF THE APPROPRIATE DECODING TOOL HOW DIFFICULT GIVEN ALL OF TODAYS COMPUTING POWER AND AVAILABLE TIME EVEN A BILLION COMPUTERS DOING A BILLION CHECKS A SECOND IT IS NOT POSSIBLE TO DECIPHER THE RESULT OF STRONG CRYPTOGRAPHY BEFORE THE END OF THE UNIVERSE

Given the hint about the encryption algorithm that was used - Poly-alphabetic ciper, first assumtion was Vigenre cipher, this together with the knowledge about the type of text -English text without newlines and spaces, gave enough information to try the following algorythms together with different ciphertext analysys.

## First approach failure - Vigenre cipher

To try and extract some information about the text, frequency analysis was used. If Vigenre cipher was used it would be possible to discover letter frequencies in the decrypted text that would match those of standard English text[4]. Fist to do that, the ciphertext had to be split into rows of length 6,4,3, and 2, then frequency analysis could be performed on the letters in each column. The reason for that is - Vigenre cipher with the keylength of 3, would encrypt every 3rd letter with the same character, combining all those letters in one could be a good sample for finding letter frequencies. The results of that attack gave no usefoul information, the letter frequencies were too similar to eachother, and did not mach any of the letter frequencies in English[1]. After that analysis, and investigating other possible cipher algorytms next step was to try the Autokey cipher.

## Autokey cipher - succes

With the autokey cipher, it is not possible to extract any meaninful frequencies of letters the same way it would be done for Vigenre cipher, it is possible but has to be done slightly differently. The technique to crack the encrypted messages was to use quadgram statistics to determine the fitness measure of the decrypted text. This means measuring how similar to english the text is and simply trying out different keys to find the best scoring one. This technique seemed like the most robust one and turned out to be effective in cracking this ciphertext[2].
Given the ciphertect, one has to assume a keylength that was used, divide the text into chunks that are the same length as the assumed key, then run Autokey deciphering algorythm on the first chunk, the result of that decryptionwill be the plaintext which also the next part of the key, it is then appended to the initial key, as described in the Autokey description here[3].

```
plaintext:   MEET ATTHEFOUNTAIN (unknown)
key:         KILT MEETATTHEFOUN (unknown)
ciphertext:  WMPM MXXAEYHBRYOCA (known)
```

This continues, until the whole text is decrypted, the output of one decryption is used as the key for the next one.
Cracking the ciphertext:
For any assumed keylength $n$, we cycle through the possible letters in the first posision leaving the other n-1, letters unaffected, we decipher the message - this will yeald 26 possible decriptions, for every one of those decriptions a fitness score is calculated based on the quadgram statistics**??**, the best score and associated letter is being saved- it is important to mention that the fitness score is only calculated with the letters that were actually decrypted, that means if the keylength is 2, and we only choose the first letter, we take every other letter from the output of the decryption and calclate fitness score on those.
We go forward with the keypositions from 1 to n, choosing only the letters that have already been changed and passing them for fitness measure. When the best letter for each position has been choosen, we cycle through all positions again starting from the first letter, only on this cycle we use the whole decryption for calculation the fitness score since every letter has been choosen already. The advandage of this algorithm, is that we do not go through every

possible key that would be *26^*n possible combinations, each score calculation is *independant* from the previous one, this lives us with exactly 2*26*n key combinations. This is 26 possible letters times key size $n$ times 2 - second pass throug whole key. In out case, when the keylength is unknown but between 2-6, it will be *2*26*(6+5+4+3+2)* key combinations total = *1040*.
The program written for this purpose has three major functions, it determines the fitness score for any given plaintext, it decrypts ciphertext into plaintext and function that chooses the best score and keeps track of the keys. The most interesting part in this example is the fitness score calculation. To be able to tell the probability of the text being english we need some sort of training sample. This sample could be created by taking an english text, preferably long one, like a book. extract every quadgram - that is set of four consecutive caracters, excluding spaces and punctation. An example would be: *CRYPTOG* has those quadgrams : *CRYP RYPT YPTO PTOG*. Repeat that for the whole text and then count the repetitions of each and store the results somewhere in a file. The site practicalcryptography.com[2] had some open source resources availble, one of them being a traingin sample with quadgrams from an english book. Using that and determining the quadgrams of the decrypted plaintext it was possible to determine the score based on relative frequency of the quadgrams in the sample text and how many times they appear in the plaintext.

```
def quadgramScore(text,key,offset,quadscore,totaln):

relevantText = relevantChunks(text, offset,key)
myquads = collections.Counter(splitToQuad(relevantText))
textProb = 0;
  for quad in myquads:
     if quad in quadscore:
        textProb += quadscore[quad]
        pass
     else:
  textProb += math.log10(0.1/totaln)
pass

return textProb;
def spaceout(plain):
```

```
## Python results
 Sample calculations time | 0.8148860279998189
 Evaluation of keys  time | 0.8416476689999399
 Total   elapsed   time     | 1.6565348680005627
###### DATFBA = -1231.8030416659658
CRYPTOGRAPHYCANBESTRONGORWEAKCRYPTOGRAPHICSTRENGTHISMEA
SUREDINTHETIMEANDRESOURCESITWOULDREQUIRETORECOVERTHEPLA
INTEXTTHERESULTOFSTRONGCRYPTOGRAPHYISCIPHERTEXTTHATISVE
RYDIFFICULTTODECIPHERWITHOUTPOSSESSIONOFTHEAPPROPRIATED
ECODINGTOOLHOWDIFFICULTGIVENALLOFTODAYSCOMPUTINGPOWERAN
DAVAILABLETIMEEVENABILLIONCOMPUTERSDOINGABILLIONCHECKSA
SECONDITISNOTPOSSIBLETODECIPHERTHERESULTOFSTRONGCRYPTOG
RAPHYBEFORETHEENDOFTHEUNIVERSE
###### RYYAA = -1784.4145978659712
OTTUUAPPEKAXTNDQOXIOLANYWCSASUGCTBECGFDWUCEQKRTEVQIDVP
DRZOBERGZXEFKAADMMZQTCVVDPIBSKBQCTAZOMAUWJSLSXJWHQAAWD
ZRBYUURSDADEJINPTWBXYJLAOWBEDSYHEVMCFKOQVRVYTAVRCEYYGD
HQNLOEKNYLWNGANASSNOIDPIKIFZKBHSIPFRTZHBOZYEOEOFQWRFMH
TTNGORVJGCTATLRBUWGFPSASSEGLYFUSITOPRYVAWVZACTUTLIIYLX
HEGVFKSTWDOLIKBNIEVOMPRRRARYEXOTBTAUNFMEYVWFTUJAOFUFOW
NVFKJPEDUPQZXDVQBUYNGJLYHUJAIELEMARWRJPCCRAYQVOJMXHTSZ
XDWLHXFMNXDHVIIARHMDJBXVEDWOWTYRAPCOA
###### EOHO = -1777.7773822699783
BDKGTLYCFTCKDYLGISKLTWAVRRSSZZQCUUIGOEWUIENZYHIJNTRNMR
CMQSOVSNZDQJRWHYAAS [ . . . ]
###### ZET = -1844.8703249864418
GNYOHQUBIKZEYSJSTHDYULMMZBGOPKUFJZTZJFFRRDSCDTTFSHUESQ
PHMUJVOWSKMEGNTBEDY [ . . . ]
###### WD = -1882.4381616368698
JOIGMIWACEYIKJDHIJNIOGWAONEAMQWEXPBTHFXVZNKGWFZDMXNZML
HOVVVMNTAFIRIIPZCXX [ . . . ]
```

Figure 1: Sample output from Python code, Autokey decryption

The python code returns the key that is associated with the best scoring plaintext, this is done for 5 keys ranging in length from 2-6. By having only 5 output ciphertexts it is very easily recognizable which one is the correct plaintext. The output would be abviously larger if the possible keylengths were unknown, but it would still be possible to pick up the best scoring one of all of those and narrrow donw the output to only a few keys. One should also concider the possibility that if the ciphertext is too short this technique might not work as well as it did in this case. The ouput is soely dependant on the probability, if the text is short and uses uncommont english phrases or words it might be very hard to determine the correct key. In this solution, quadgram statistics were used, but also other n-grams could be calculated, like trigrams on bigrams and even single letters.

# Part II

## Task 1

Table 1: Task 1

| Raw Key | Plaintext | Ciphertext |
|---|---|---|
| 0000000000 | 00000000 | *11110000* |
| 1111111111 | 11111111 | *00001111* |
| 0000011111 | 00000000 | *01000011* |
| 0000011111 | 11111111 | *11100001* |
| 1000101110 | *00111000* | 00011100 |
| 1000101110 | *00001100* | 11000010 |
| 0010011111 | *11111100* | 10011101 |
| 0010011111 | *10100101* | 10010000 |

## Tast 2

Table 2: Task 2

| Raw Key 1 | Raw Key 2 | Plaintext | Ciphertext |
|---|---|---|---|
| 0000000000 | 0000000000 | 00000000 | *11110000* |
| 1000101110 | 0110101110 | 11010111 | *10111001* |
| 1000101110 | 0110101110 | 10101010 | *11100100* |
| 1111111111 | 1111111111 | 10101010 | *00000100* |
| 1000101110 | 0110101110 | *11111101* | 11100110 |
| 1011101111 | 0110101110 | *01001111* | 01010000 |
| 0000000000 | 0000000000 | *01010010* | 10000000 |

### Task 3

```
# Task 3
K1:1111101010
simplifieddesisnotsecureenoughtoprovideyou
sufficientsecurity
Time   SDES: 0.055122 sek

K1:1111101010
K2:0101011111
simplifieddesisnotsecureenoughtoprovideyou
sufficientsecurity
Time T2SDES: 1.96109 sek

TOTAL  TIME: 2.01638 sek
```

Since the plaintext is assumed to be english, and the encrypted strings are clear of eany spaces and punktuation it is very easy to check wether the output - decrypted byte is in the range of ascii characters. To figure out which key is correct all of the decrypted bytes must be in range. There was in total 480 bits per message (cxt1.txt and cxt2.txt), that means that every one of the 60 bytes has to be an ascii character in the range of the english alphabet, if all the bytes pass this condition it means that this is the correct key.

## Conclusion

The ciphertexts were cracked sucessfully. In case of the Part 1 of the assignment-cryptanalysis of Autokey encrypted message, as explained previously, it was not neccesary to execute bruteforce attack on every possible key. With the use of Quadgram statistics of sample english text, it was possible to make independant tries on each letter of the key, thus reducing the time dramatically. In this case, there is no clear improvement as to the algorytm for cracking, but the implementation of it could be improvement when in comes to processing the data, especially creating quadgrams for the decrypted plaintext. Part 2 is about SDES and T2SDES implementation of algorythm and cracing a relatively short message encrypted with both methods. The implementation passed all the tests in Task 1 and Task 2, also implemented bruteforce attack on both SDES and T2SDES worked. Improvements when it comes to bruteforce attack on both ciphers could be made by decrypting multiple keys parallelly with the use of threads. Wors case scenario is we have to run through all 1024*1024 keys, by running the function in 4 threads and assigning each thread 1/4 of the keys we would reduce the time to find the correct key.

# Works Cited

[1] English Letter frequencies.
`http://crypto.interactive-maths.com/autokey-cipher.html`

[2] Quadgram analysis, Practical Cryptograpy.
`http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/`

[3] Eutokey Cipher
`http://www.cryptool-online.org/index.php?option=com_content&view=article&id=104&Itemid=127&lang=en`

[4] Vigenere letter frequency analysis
`http://www.cs.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-Frequency-Analysis.html`