DAT 510 Assignment 1

Abstract

The focus of this project was to solve problems related to implementing and cracking cryptographic ciphers. The project consisted of two main parts, the first of which was related to polyalphabetic ciphers. In this part the main task consisted of developing tools to analyze and crack encrypted text for which the specific method and key from which the encrypted text was produced, was unknown. This task was executed successfully, where the cipher method was identified as an autokey-cipher, with the key being found as: "DATFBA". The second part consisted of implementing and utilizing simplified DES, a lightweight model-version of the larger DES (data encryption standard), on which we were to encode and decode binary strings. Followingly, TripleSDES a threefold composition of DES ciphers, was also implemented. From these implementations a part of the task was to find the key on which two binary messages was encoded on each respective algorithm. Using a brute force approach, the key was found to be "11111010" for the SDES encoded message and "11111010--0101011111" for the TripleSDES encoded message.

Introduction

The field of cryptography is vital for the modern data and communication technology, being the primary source of means for ensuring secure communication, privacy, and data security. To get some semblance over how methods from this field is utilized in the real word, this project consists of solving simplified but relevant problems. More specifically, these problems consist of implementing and cracking different cipher algorithms.

Design and Implementation

Part 1 Cracking the polyalphabetic cipher

Of all the tasks in the project, this task proved to be the most challenging and time consuming. At first an attempt was made to decode the text as a Vigenère cipher, but this yielded no intelligible results. After this attempt, an implementation for the autokey cipher algorithm yielded the correct result. The general method for performing cryptanalysis on such a cipher is described on Wikipedia [1], and by Rodrigues-Clark at at interactive maths [2]. Hence the task was to create the necessary tools to be able to perform this analysis. The solution can be summarized by first describing the general method, and then describe the specific implementation.

An autokey substitution cipher is a polyalphabetic substitution-cipher which is like the Vigenère cipher in that it consists of a plaintext and a key of equal length, where the characters in both texts belong to a given alphabet. The characters in the alphabet can then be treated as numeric values, and the result of the cipher is the sum of the key and plaintext value. Where autokey differs from Vigenère is when how the key may be generated. In the case of Vigenère the key may be a simple keyword repeated throughout the length of the plaintext. With this type of encoding, statistical characteristics of the language is preserved for every key-length spaced letter, and cryptanalysis can be performed by bucketing the text and analyzing the statistical similarity to English text. The autokey cipher works by appending a keyword to the front of the plaintext and using this as the key. The advantage of this method is that the statistical properties of the text is not preserved, and the statistical methods one would use in the Vigenère case will not work. Rather, the strategy for cracking an autokey cipher can be inducted by first realizing that the ciphertext itself is most of the key in the cipher. If one were to guess or know a word of length n which appears in the plaintext, one would also know that this word encodes the next n letters a keyword-length further down the plaintext. Furthermore, one would also be able to find the word which together with the ciphertext n letters up, would produce the guessed/known word. From this fact, one can lead the following strategy: Choose a word/n-gram that is likely to appear in the plaintext such as "the, and, tha, …" then for key-lengths up to a set limit, decode the next and previous n-gram one key length apart. If the n-grams that results from the decoding seems likely to appear in the plaintext one can repeat the process. Should one guess correctly, then it is likely that enough of the plaintext is revealed to be able to surmise the contents.

The most important functions are summarized as follows. First of, most of the functions abvelong to the Autokey_cryptanalysis class which stores the ciphertext and initializes keystream and plaintext

In [ ]:
```
class Autokey_cryptanalysis:
    def __init__(self, ciphertext):
        self.ciphertext = ciphertext
        self.plaintext = '-'*len(ciphertext)
        self.keystream = '-'*len(ciphertext)

        self.matches = {}
```

The critical functions of the crytpanalysis became a method to find a viable word/n-gram suggestion which was likeliy to appear in the ciphertext, and a method to place and decrypt the plaintext given a keyword with correct placeement and distance to next likely n-gram. These functions were:

In [ ]:
```
def keyword_searching(self, searchword, vocab = common_words):
    """Running window decoding of searchword over ciphertext,
    if decode yields a likely n-gram, then runs again with the decoded  word,
    ïf this yields a likely n-gram then store original keyword, position and
    distance to next keyword."""
    key_len =0
    pos = 0
    for i in range(len(self.ciphertext)):
        dec = vigenere_decode(self.ciphertext[i:i+len(searchword)], searchword)
        if dec in vocab:
            for j in range(i+1, len(self.ciphertext)):
                next_dec = vigenere_decode(self.ciphertext[j:j+len(searchword)],dec)
                if next_dec in vocab:
                    key_len = j-i
                    pos = i

                    # Keeping search which minimizes key length
                    if searchword in self.matches.keys():
                        val = sorted([self.matches[searchword],
                            ((pos, pos+len(searchword)), key_len)],
                            key = lambda x: x[1])[0]

                        self.matches[searchword] = val
                    else:
                        self.matches[searchword]= ((pos, pos+len(searchword)), key_len)

def get_keystream_suggestion(self, vocab1 = common_words, vocab2= common_words):
    """Keyword searching for keywords in vocab """
    for gram in tqdm(vocab1):
        self.keyword_searching(gram, vocab2)
```

In [ ]:
```
    def unfolding(self, gram, gram_loc, offset):
        """Places ngram suggestion at specified position in the keystream,
        decodes every ngram  m offsets away in both direcitons """

        self.keystream_insert(gram, gram_loc)

        right_len = len(self.ciphertext[gram_loc[0]:])
        left_len = len(self.ciphertext[:gram_loc[0]])

        next_r_gram = next_l_gram = gram

        for i in range(0, right_len, offset):
            next_r_gram = vigenere_decode(self.ciphertext[gram_loc[0]
                        + i : gram_loc[1] + i], next_r_gram)

            self.keystream_insert(next_r_gram, (gram_loc[0]+i + offset, gram_loc[1]+ i+ offset))
            self.keystream = self.keystream[:len(self.ciphertext)]


        for i in range(0, left_len-offset, offset):
            next_l_gram = vigenere_decode(self.ciphertext[gram_loc[0]
                     -i -offset :gram_loc[1] - i-offset], next_l_gram)
```

```
            self.keystream_insert(next_l_gram, (gram_loc[0]- i -offset, gram_loc[1]-i- offset))

        self.plaintext_update()
```

The output of these functions yields:

In [ ]:
```
from polyalphabetic import*

inst = Autokey_cryptanalysis(ciphertext)
print('\nSearching for potential words in key\n')
inst.get_keystream_suggestion(common_trigrams[:20], common_trigrams[:20])
print(inst.matches)
```

```
100%|██████████| 20/20 [00:00<00:00, 589.83it/s]
Searching for potential words in key

{'AND': ((75, 78), 6), 'ENT': ((38, 41), 37), 'OUR': ((81, 84), 214), 'IST': ((293, 296), 2)}
```

In [ ]:
```
print('\n AND at index 75, with a distance of 6 to next word looks promising, we can place it ir
inst.unfolding('AND',(75,78), 6)
inst.plaintext_update()
inst.disp(width = 75)
```

```
 AND at index 75, with a distance of 6 to next word looks promising, we can place it in the keys
tream and decode


 FRRUUOIIYEAMIRNQLQVRBOKGKNSNQQIUTTYIIYEAWIJTGLVILAZWZKTZCJQHIFNYIWQZXHRWZQW
 ---FBA---PTO---PHY---BES---NGO---AKC---TOG---HIC---ENG---SME---RED---HET---
 ---PTO---PHY---BES---NGO---AKC---TOG---HIC---ENG---SME---RED---HET---AND---

 OHUTIKWNNQYDLKAEOTUVXELMTSOSIXJSKPRBUXTITBUXVBLNSXFJKNCHBLUKPDGUIIYEAMOJCXW
 AND---OUR---ITW---DRE---RET---COV---HEP---NTE---HER---LTO---RON---YPT---APH
 OUR---ITW---DRE---RET---COV---HEP---NTE---HER---LTO---RON---YPT---APH---CIP

 FMJVMMAXYTXFLOLRRLAAJZAXTYYWFYNBIVHVYQIOSLPXHZGYLHWGFSXLPSNDUKVTRXPKSSVKOWM
 ---CIP---TEX---ATI---RYD---ICU---ODE---HER---HOU---SSE---ONO---EAP---PRI---
 ---TEX---ATI---RYD---ICU---ODE---HER---HOU---SSE---ONO---EAP---PRI---DEC---

 QKVCRTUUPRWQMWYXTYLQXYYTRTJJGOOLMXVCPPSLKBSEIPMEGCRWZRIYDBGEBTMFPZXVMFMGPVO
 DEC---NGT---HOW---FIC---GIV---LLO---DAY---MPU---GPO---AND---ILA---TIM---ENA
 NGT---HOW---FIC---GIV---LLO---DAY---MPU---GPO---AND---ILA---TIM---ENA---LIO

 OKZXXIGGFESIBRXSEWTYOOOKSPKYFCZIEYFDAXKGARBIWKFWUASLGLFNMIVHVVPTYIJNSXFJKNC
 ---LIO---MPU---SDO---ABI---ONC---KSA---OND---SNO---SSI---TOD---PHE---ERE---
 ---MPU---SDO---ABI---ONC---KSA---OND---SNO---SSI---TOD---PHE---ERE---TOF---

 HBLUKPDGUIIYEAMHVFDYCULJSEHHMXLRXBNOLVMR
 TOF---ONG---PTO---PHY---ORE---END---HEU-
 ONG---PTO---PHY---ORE---END---HEU---ERS-
```

Luckily, we can see that this first guess seemed to work out. We can see probable words in the text, let's input 'HER' right after 'CIP':

In [ ]:
```
cip_loc = inst.keystream_get_loc('CIP')
inst.unfolding('HER', (cip_loc[0] +3,cip_loc[1]+3), 6)
inst.disp(width=75)
```

```
 FRRUUOIIYEAMIRNQLQVRBOKGKNSNQQIUTTYIIYEAWIJTGLVILAZWZKTZCJQHIFNYIWQZXHRWZQWOHUTI
 ---FBACRYPTOGRAPHYCANBESTRONGORWEAKCRYPTOGRAPHICSTRENGTHISMEASUREDINTHETIMEANDRE
 ---PTOGRAPHYCANBESTRONGORWEAKCRYPTOGRAPHICSTRENGTHISMEASUREDINTHETIMEANDRESOURCE

 KWNNQYDLKAEOTUVXELMTSOSIXJSKPRBUXTITBUXVBLNSXFJKNCHBLUKPDGUIIYEAMOJCXWFMJVMMAXYT
 SOURCESITWOULDREQUIRETORECOVERTHEPLAINTEXTTHERESULTOFSTRONGCRYPTOGRAPHYISCIPHERT
 SITWOULDREQUIRETORECOVERTHEPLAINTEXTTHERESULTOFSTRONGCRYPTOGRAPHYISCIPHERTEXTTHA

 XFLOLRRLAAJZAXTYYWFYNBIVHVYQIOSLPXHZGYLHWGFSXLPSNDUKVTRXPKSSVKOWMQKVCRTUUPRWQMWY
 EXTTHATISVERYDIFFICULTTODECIPHERWITHOUTPOSSESSIONOFTHEAPPROPRIATEDECODINGTOOLHOW
 TISVERYDIFFICULTTODECIPHERWITHOUTPOSSESSIONOFTHEAPPROPRIATEDECODINGTOOLHOWDIFFIC

 XTYLQXYYTRTJJGOOLMXVCPPSLKBSEIPMEGCRWZRIYDBGEBTMFPZXVMFMGPVOOKZXXIGGFESIBRXSEWTY
```

```
DIFFICULTGIVENALLOFTODAYSCOMPUTINGPOWERANDAVAILABLETIMEEVENABILLIONCOMPUTERSDOIN
ULTGIVENALLOFTODAYSCOMPUTINGPOWERANDAVAILABLETIMEEVENABILLIONCOMPUTERSDOINGABILL

OOOKSPKYFCZIEYFDAXKGARBIWKFWUASLGLFNMIVHVVPTYIJNSXFJKNCHBLUKPDGUIIYEAMHVFDYCULJS
GABILLIONCHECKSASECONDITISNOTPOSSIBLETODECIPHERTHERESULTOFSTRONGCRYPTOGRAPHYBEFO
IONCHECKSASECONDITISNOTPOSSIBLETODECIPHERTHERESULTOFSTRONGCRYPTOGRAPHYBEFORETHEE

EHHMXLRXBNOLVMR
RETHEENDOFTHEUN
NDOFTHEUNIVERSE
```

From this we produced the plaintext which is:

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 DOINGA BILLION CHECKS A SECOND IT IS NOT POSSIBLE TO DECIPHER THE RESULT OF STRONGCRYPTOGRAPHY BEFORE THE END OF THE UNIVERSE

In [ ]:
```
print('And the keyword is: ' + vigenere_decode('FRRUUO', 'CRYPTO') + '\n')
```

```
And the keyword is: DATFBA
```

Note that a necessary component in this code is the search vocab. In this project two search vocabulary has been utilized, one a list of the 50 most common words in english text, the other a list of tri-grams which was extracted from a selection of text in the syllabus.

Task 2

For this task, the execution can be considered separately for the two main functions. Searching for potential keywords in the as demonstrated, occurs almost immedeatly as reported with the tqdm progress bar package. However the execution time is dependent on a number of factors. Notice that in this notebook the vocabulary for whioch the program searches after keywords is the 20 first entries of the common_trigrams list, which is a list of the most common trigrams that appeared in a selection of text of the course book. Changing the size of the search vocabulary to the entire field of about 2000 entries greatly increases the search time. However, this vocabulary is unessecarily large and one is not necessarily more likely to get good results with a large vocab. Adding letters to the key will not have to much of an impact on the search unless one were to greatly extend the length of the keyword. If the keyword is random, the cipher will approach a vernamm cipher as the keyword approaces the length of the plaintext. Then, no statistical method would help in producing the plaintext.

Once a keyword which incidentaly is right is found, the speed of the decyrption method is trivial.

Task 3:

By decrypting this new cipherstring with the key found in task 2 we get:

In [ ]:
```
print(autokey_decode(ciphertext_task3, 'DATFBA'))
```

```
FRRUUOIIYEAMIRNQLQVRBOKGKNSNQQIUTTYIIYEAWIJTGLVILAZWZKTZCJQHIFNYIWQZXHRWZQWOHUTIKWNNQYDLKAEOTUVX
ELMTSOSIXJSKPRBUXTITBUXVBLNSXFJKNCHBLUKPDGUIIYEAMOJCXWFMJVMMAXYTXFLOLRRLAAJZAXTYYWFYNBIVHVYQIOSL
PXHZGYLHWGFSXLPSNDUKVTRXPKSSVKOWMQKVCRTUUPRWQMWYXTYLQXYYTRTJJGOOLMXVCPPSLKBSEIPMEGCRWZRIYDBGEBTM
FPZXVMFMGPVOOKZXXIGGFESIBRXSEWTYOOOKSPKYFCZIEYFDAXKGARBIWKFWUASLGLFNMIVHVVPTYIJNSXFJKNCHBLUKPDGU
IIYEAMHVFDYCULJSEHHMXLRXBNOLVMR
```

This is recognized as the ciphertext from task1, which means that the addition in the cipher process is encrypting the text twice with the same keyword. Using the implemented solution for single autokey encryption from the last task will not work. The reason being that the second layer of encoding obscures the words on which one can check if belongs to a vocabulary. However, I am positive that given that one knows that the ciphertext is produced by this type of double encoding with the same key, it should be possible to decode the cipher through a modification of the technique or with some other method; although I was not able to find such a solution. The reasoning behind this is that we still know that

a part of the plaintext encodes other parts of the plaintext, just with a step intbetween. One could for example try the strategy of guessing likely pairs of words with a likely distance between them being two lengths of the keyword.

Part 2. Simplified DES

Task 1 and 2

The main task in this part was to code an implementation of the simplified DES (SDES) algorithm in concordance with the given description [2]. This algorithm is as the name suggest a simplified version of the DES (Data Encryption Standard) block cipher algorithm. This implementation was further used to create the triple des algorithm which in essence takes two keys, and essentially chains together three encryption rounds of the SDES algorithm. With these implementations a table of keys, encrypted and decrypted binary string were filled in. The results of this are as follows:

Task 1 Table:

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

Task 2 Table

Raw Key 1	Raw Key 2	Plaintext	Ciphertext
1000101110	0110101110	11010111	10111001
1000101110	0110101110	10101010	11100100
1111111111	1111111111	00000000	11101011
0000000000	0000000000	01010010	10000000
1000101110	1000101110	11111101	11100110
1011101111	1011101111	01001111	01010000
1111111111	1111111111	10101010	00000100
0000000000	0000000000	00000000	11110000

Task 3

Task 3 gives two strings of binary digits, which are the results of converting ascii characters to binary and encoding both strings with their respective of the two algorithms, where the keys are unknown. The task was then to attempt to crack the ciphers and retrieve the decrypted plaintext and the encryption key. The strategy used to solve this task was a simple brute force attack. In the SDES algorithm, the key can be a value between 0 and 1024 in binary. One can then check for every value in the range if it is a potential key. One can check this by choosing a word that is likely to appear in the plaintext, in this case the word “des” was chosen. Since the output of the SDES algorithm vary strongly dependent on the key and plaintext, and since the output will be encoded into ascii which is a relatively larger set of characters, it is unlikely for the search word to appear in an incorrect decryption from a false key.

The brute force function:

In [ ]:
```
def bruteforce_des(ctx_bitlist, searchword):
    for key in tqdm(range(1024)):
        plaintext =''
        key =[int(bit) for bit in bin(key)[2:].zfill(10)]
        for bitslist in ctx_bitlist:
            char = bitlist_to_ascii(des_decrypt(bitslist, key))
            plaintext += char
        if plaintext.find(searchword) != -1:
            return plaintext, key
    return 'Did not work'
```

By using this strategy, and by incidentally choosing a correct keyword, the plaintext was found to be:

“Simplified des is not secure enough to provide you sufficient security”

With key: [1111101010]

This takes aproximately 2 seconds.

The case for TripleDES is similar, however in this case the range for potential key is much greater, in fact it is the range of SDES squared which accounts for over a million keys to check. Account the fact that it takes some time to perform the encryption chain of the algorithm, brute forcing the triple des algorithm takes a significantly greater amount of time. However, with patience and a relatively powerful computer, the same strategy works without augmentations to the method.

Method:

In [ ]:
```
def bruteforce_tripledes(ctx_bitlist, searchword):
    for key in tqdm(range(1024**2-1,0, -1)):
        key = [int(bit) for bit in bin(key)[2:].zfill(20)]
        key1 = key[:10]
        key2 = key[10:]
        plaintext =''
        for bitlist in ctx_bitlist:
            char = bitlist_to_ascii(tripledes_decode(bitlist, key1, key2))
            plaintext += char
        if plaintext.find(searchword) != -1:
            return plaintext, key1, key2
    return 'Did not work'
```

Note that this method loops from the top, this is because in this particular case, the key is a large number and the method will converge quicker. Looping from the bottom will yield the right answer after about two hours. The message was found to be the same as previous with the keys:

[key1,key2] = [[1111101010, 0101011111]

Task 4:

In this task a small we server was set up in in which takes binary strings as input into the TripeDes algorithm and yields the decrypted text. This reflects a communication protocol which utilizes TripleSDES as the cipher, which has been demonstrated in this project to be not secure as it is susceptible to brute force attacks

Discussion:

A few critiques can be raised for some of the implementations in the project. The first being that I was not able find a method to produce plaintext from the cipher in part 1 task 3. One could also critique the implemented function as not being a function which takes in ciphertext and produces plaintext, since it requires some human input in-between processes. However, the task was after all to produce tools to help produce the plaintext.
The TripleDes brute force also took quite a while, and it would be better suited to the theme of the project to find a faster solution, (to demonstrate that in fact this cipher also can be easily cracked). Since the slow solution gave the correct answer, I chose not to investigate it further although I will mention that one can use strategies such as multithreading /processing to speed up this process. In this specific case when the plaintext is pure letters, one can also

choose to break the loop earlier if the key suggestions produce non-letter ascii symbols, this might speed up the process.

Conclusion:

In this project we were able to see how certain encryption schemes can be cracked. Statistical techniques can be used to crack polyalphabetic ciphers, and the SDES algorithm which models the larger DES is susceptible to brute-force attacks.

Sources

[1] https://en.wikipedia.org/wiki/Autokey_cipher

[2] 2013 - 2019 Daniel Rodriguez-Clark https://crypto.interactive-maths.com/autokey-cipher.html

[3] William Stallings Cryptography and Network Security, Fifth Edition Prentice Hall 2010 ISBN-10:0136097049 http://williamstallings.com/Crypto/Crypto5e.html