DAT510 - Assignment 1

Eirik Sakariassen

September 22, 2020

Contents

1 Abstract

This assignment is divided into two parts; Poly-alphabetic Ciphers and Simplified DES. The objective of part one is to decipher a text that was enciphered using a polyalphabetic substitution cipher. A cryptanalysis tool based on Kasiski's method and frequency analysis is implemented and tested on different use cases. Part two focuses on the implementation of a simplified version of the DES block cipher algorithm. Both SDES and TripleSDES is implemented and used for various encryption and decryption tasks. A webserver with TripleSDES decryption is created in the last task of part two. Execution time will be considered in all of the tasks, and different adjustments for improved execution will be discussed along the way.

2 Part I. Poly-alphabetic Ciphers

2.1 Introduction

We are given the following enciphered text:

BQZRMQ KLBOXE WCCEFL DKRYYL BVEHIZ NYJQEE BDYFJO PTLOEM EHOMIC UYHHTS GKNJFG EHIMK NIHCTI HVRIHA RSMGQT RQCSXX CSWTNK PTMNSW AMXVCY WEOGSR FFUEEB DKQLQZ WRKUCO FTPLOT GOJZRI XEPZSE ISXTCT WZRMXI RIHALE SPRFAE FVYORI HNITRG PUHITM CFCDLA HIBKLH RCDIMT WQWTOR DJCNDY YWMJCN HDU-WOF DPUPNG BANULZ NGYPQU LEUXOV FFDCEE YHQUXO YOXQUO DDCVIR RPJCAT RAQVFS AWMJCN HTSOXQ UODDAG BANURR REZJGD VJSXOO MSDNIT RGPUHN HRSSSF VFSINH MSGPCM ZJCSLY GEWGQT DREASV FPXEAR IMLPZW EHQGMG WSEIXE GQKPRM XIBFWL IPCHYM OTNXYV FFDCEE YHASBA TEXCJZ VTSGBA NUDYAP IUGTLD WLKVRI HWACZG PTRYCE VNQCUP AOSPEU KPCSNG RIHLRI KUMGFC YTDQES DAHCKP BDUJPX KPYMBD IWDQEF WSEVKT CDDWLI NEPZSE OPYIW

Our prior knowledge is that this is originally an English text that was enciphered using a polyalphabetic substitution cipher. We can leverage these insights when developing a cryptanalysis tool that will encipher the given text. One common strategy in such a scenario is to first discover the key length n by using Kasiski's method. We can then divide the ciphertext into n columns and treat each column as ciphertext from a monoalphabetic substitution cipher. The frequency of the letters in each ciphertext will then in some degree reflect the frequency of letters in English text [1]. We can compare the frequencies and derive the most probable letters in the key. This key does not necessarily decipher the text perfectly, but we might be able to understand parts of the text. We can then simply change the remaining wrong letters of the key and complete the deciphering.

2.2 Task 1: Maximum 10 bit key

We want to find a key of maximum length 10. The first step of Kasiski's approach is to find repeating strings of characters in the ciphertext. We do this by looking at all character level n-grams in the range of 2 to 10. An example of the 3-gram based on the sequence "BQZRMQ" is shown below:

$$BQZRMQ \rightarrow [BQZ, QZR, ZRM, RMQ]$$

We can simply discard all strings with just one occurrence once all n-grams within the range of our maximum key length is produced. The next step is to calculate the distances between similar strings as these are likely to be a multiple of the keyword length. This is accomplished by the two methods $get\_ngram\_distances1$ and $get\_distances2$. The method *get_ngram_distances* iterates through every unique string in the set of n-grams and finds the positions of where it occurs in the ciphertext. The list of position indices for a given string is then fed into the other method *get_distances*, which calculates and returns all distances between the occurrences of the string. For example, if the string QZR occurs in the positions [2,6,15,25], it returns [6-2,15-2,25-2,15-6,25-6,25-15]=[4,13,23,9,19,10].

```python
def get_ngram_distances(ngram):
    res = []
    for item in set(ngram):
        pos_idx = [i for i,_ngram in enumerate(ngram) if _ngram==item]
        if len(pos_idx) > 1:
            res = res + get_distances(pos_idx)
    return res
```

Listing 1: Get all positional distances in a ngram sequence

```python
def get_distances(idx):
    dists = []
    for i in range(len(idx)-1):
        for j in idx[i+1:]:
            dists.append(j-idx[i])
    return dists
```

Listing 2: Get all positional distances for a certain item

The final step is to calculate the factors for each distance retrieved from *get_ngram_distances*. For example, a distance of length 10 will give the factors 2, 5 and 10 indicating that the unknown keyword may be of one of these lengths. We can then count factor frequencies and sort it in descending order. Table 1 shows the output after these steps are performed. We should look for a higher length with clearly more items. We can see that the length 8 occurs pretty often as well as 2 and 4 which are divisors of 8. Thus we continue the analysis by picking 8 as the key length. The entire algorithm is showed in code block 3

```python
def get_key_lens(max_len):
    factors = []
    for n in range(2,max_len+1):
        ngram = get_ngram(doc,n)
        dists = get_ngram_distances(ngram)
        _factors = [get_factors(i, limit=max_len) for i in dists]
        # Flatten array: [[2,4], [6,3]] => [2,4,6,3]
        factors += [item for sublist in _factors for item in sublist]

    factor_frequencies = {k:factors.count(k) for k in range(2,max_len+1)}
    factor_frequencies = {k: v for k, v in sorted(factor_frequencies.
    items(), reverse=True, key=lambda item: item[1])}
    return factor_frequencies
```

Listing 3: Get key length frequencies

Table 1: Key length frequencies sorted in decreasing order

2	4	8	3	6	7	9	5	10
358	324	280	178	138	92	71	60	37

Frequency analysis is applied once the user picks a probable key length. We divide the ciphertext into 8 columns and treats each column as a ciphertext from a monoalphabetic substitution cipher. For example, column one begins with **BBF** and column 2 begins with **QOL**:

BQZRMQ KL**B**OXE WCCE**F**L...
B**Q**ZRMQ KLB**O**XE WCCEF**L**...

We count the letter frequency in each column and compares it against the letter frequency of English text [1]. Figure 1 shows the relative frequency of letters in English text (green) compared to the letter frequency of column 1 (blue). The letter H occurs most often and hence we can guess that it translates to E. We calculate the offset from E which is equal to 3, and shift our column frequency by the offset. The red bar shows the shifted column frequency. We can see clear similarities between the shifted frequency and the original one in green. The second letter of the unknown key is then set to index 3 of the alphabet (starting from index 0) which gives us the letter D.

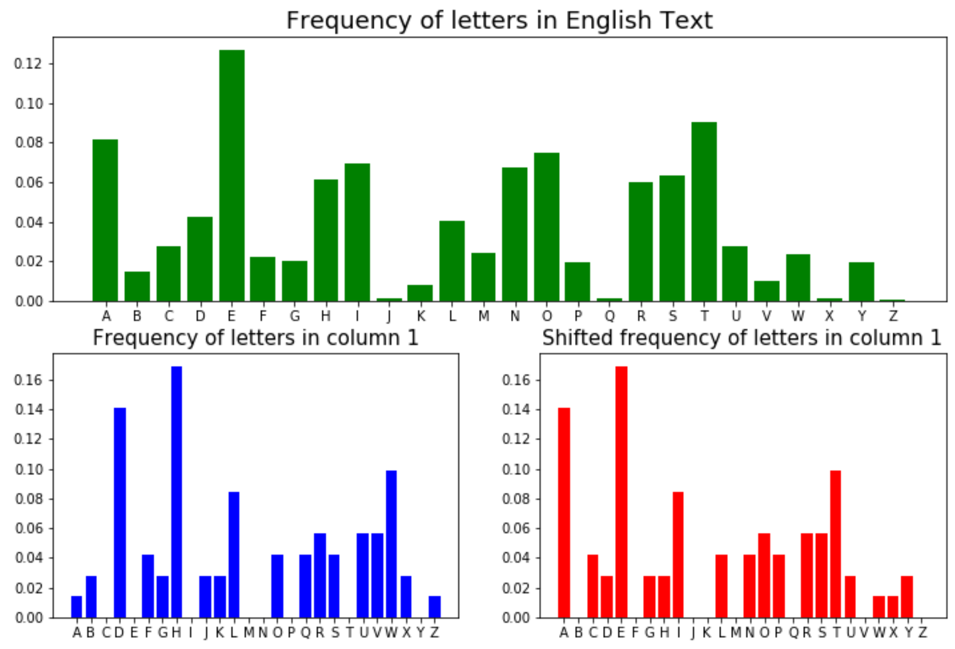

Figure 1: Column letter frequency compared to English text letter frequency

There are often cases where the most frequent letter of a column in reality doesn't map to E. This tool enables the user to pick k candidate keys which means the software calculates the offset between the k most frequent letters in English text. For k=3, it will

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

calculate the offset between H and E,T and A. This gives us 3 different offsets per letter of the key. There are $3^8 = 6561$ combinations that forms a valid key in this case. The number grows exponentially with the key length, so one has to be careful when defining this step. Code block 4 shows how all key alternatives are generated.

```python
def candidate_keys(frequencies):
    candidates = e_offset(frequencies, k=3)
    key_comboes = list(itertools.product(*candidates))
    return [''.join(combo) for combo in key_comboes] # All key
   alternatives
```

Listing 4: Candidate keys method

The user are not able to look through thousands of deciphered texts based on all key candidates, hence a ranking method is included in the software. Pythons Natural Language Toolkit *NLTK* provides a list of stopwords [2], that is high-frequency words like "its", "has" and "the". After removing words with one or two characters only, we have a list of 145 common words. The ranking algorithm counts the number of stopwords in each deciphered text. When the user is asked how many top results it will see (numres), the software simply picks the numres deciphered texts with most stopword occurences. Picking top 3 in our case gives these three alternatives:

1. Key: BDLAETCY, Stopword count: 40, Deciphered text: ANORIXINALMES...
2. Key: BDLAEECY, Stopword count: 36, Deciphered text: ANORIMINALMES...
3. Key: BDLAPTCY, Stopword count: 34, Deciphered text: ANORXXINALMEH...

We can see that the highest ranked key alternative almost deciphers the text perfectly. The first 10 letters ANORIXINAL should clearly be ANORIGINAL, meaning letter 6 of the key is wrong. Letter 6 of the original ciphertext (Q) should map to G. The offset is 10, and thus we can index the alphabet by 10 and get the letter K. Our final key is: **BDLAEKCY** and the deciphered text is:

AN ORIGINAL MESSAGE IS KNOWN AS THE PLAINTEXT WHILE THE CODED MESSAGE IS CALLED THE CIPHERTEXT. THE PROCESS OF CONVERTING FROM PLAINTEXT TO CIPHERTEXT IS KNOWN AS ENCIPHERING OR ENCRYPTION. RESTORING THE PLAINTEXT FROM THE CIPHERTEXT IS DECIPHERING OR DECRYPTION. THE MANY SCHEMES USED FOR ENCRYPTION CONSTITUTE THE AREA OF STUDY KNOWN AS CRYPTOGRAPHY. SUCH A SCHEME IS KNOWN AS A CRYPTOGRAPHIC SYSTEM OR A CIPHER. TECHNIQUES USED FOR DECIPHERING A MESSAGE WITHOUT ANY KNOWLEDGE OF THE ENCIPHERING DETAILS FALL INTO THE AREA OF CRYPTANALYSIS. CRYPTANALYSIS IS WHAT THE LAY PERSON CALLS BREAKING THE CODE. THE AREAS OF CRYPTOGRAPHY AND CRYPTANALYSIS TOGETHER ARE CALLED CRYPTOLOGY.

2.3 Task 2: Variation in key lengths

The cryptanalysis software described in Task 1 depends on the key length in several ways. Figure 2 shows execution time (in seconds) for Kasiski's method with key lengths

[2]https://www.nltk.org/book/ch02.html

ranging from 5 to 1000. We were given max key length 10 in Task 1 so the key length frequencies was created in 0.14 seconds. The attacker would have had to check for max key length 568 (length of ciphertext) without this knowledge. It would take around 3.8 seconds to generate a key length frequency table in such a case, which isn't a lot. The attacker is still required to pick the most probable key length based on this output, which will be more difficult for longer keys.

Calculating top n candidate keys will be done in milliseconds for all key lengths ranging from 5 to 1000 as shown in figure 3. This is given that k=1 in the candidate keys method. Since the key length in Task 1 was short, we could pick k=3 producing $3^8 = 6561$ key combinations, which took 2.5424 seconds to complete. However for keys in the range we search here, this is not possible. Even just a key length of 50 would produce $3^{50} = 717897987691852588770249$ combinations. This means our top results from the candidate keys are more likely to have more wrong offsets to E in the key. The deciphered text will probably be harder to interpret as the key length grows. For instance, the top key candidate with k=1 in Task 1 will produce a deciphered text with only 23 stop words, compared to 40 with k=3.

We can conclude that the technique used in this software depends a little bit on key lengths, but not much. The human has to perform the most critical parts of the decryption, and the helping functions from the software runs in just some seconds even with key lengths up to thousand characters.

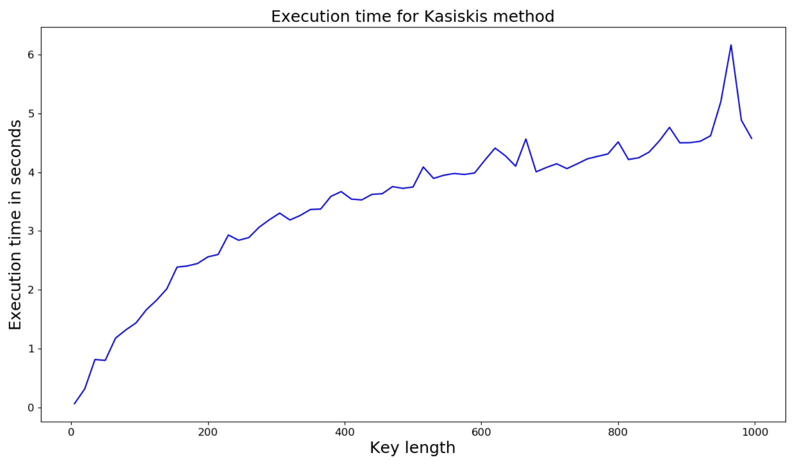

Figure 2: Execution time for Kasiski's method with different key lengths

2.4 Task 3: Adding addition in the encryption process

The software will produce a key length frequency table shown in table 2 after Kasiski's method on the new ciphertext. It is harder to clearly find a good key length based

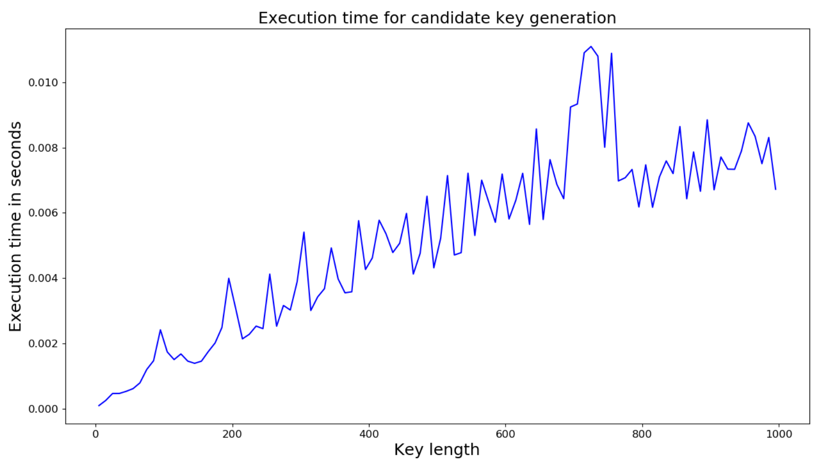

Figure 3: Execution time for candidate key generation

on it. After testing all key lengths with k=3 candidate keys, the software still fails to produce any readable deciphered text. The top candidate key in key lengths 6, 7 and 8 produced 9 stopwords, which is pure random. We can conclude that the approach in Task 1 doesn't work when additional steps are added to the decryption task.

Table 2: Key length frequencies sorted in decreasing order for task 3

2	3	7	4	5	9	6	8	39
174	138	77	76	68	64	56	50	39

3 Part II. Simplified DES

3.1 Task 1: SDES implementation

The first step of both encryption and decryption is to generate two keys from the given Raw key; *K1* and *K2*. A method named *generate_key* combines a set of given permutation and left shift operations which in turn produces *K1* and *K2*. Code block 5 shows an example of a permutation method where 10 bits is permutated into new 8 bits. The SDES implementation contains several similar permutation methods.

```python
def P8(bits):
    return [bits[i-1] for i in [6,3,7,4,8,5,10,9]]
```

Listing 5: Select and permutate method

The ciphertext/plaintext goes through 2 rounds with different permutation and switch methods before and after. An SDES round is shown in code block 6 where *inp* either is a ciphertext or plaintext that has gone through its initial permutation. In encryption, first round accepts *K1* while second round accepts *K2*. Decryption is simply the opposite of encryption. The SDES round performs permutation, XOR and S-Box methods before it concatenates two 4-bit lists and returns.

```
def sdes_round(inp, K):
    inp_0 = inp[:4]
    inp_1 = inp[4:]
    out = EP(inp_1)
    out = XOR(out, K)
    S0,S1 = S_box(out[:4], out[4:])
    out = P4(S0+S1)
    out = XOR(inp_0,out)
    return out + inp_1
```

Listing 6: SDES round

The results from running this implementation of SDES is shown in table 3. The bold text is the output from the encryption/decryption of the given raw key and text.

Raw key	Plaintext	Ciphertext
0000000000	00000000	**11110000**
0000011111	11111111	**11100001**
0010011111	11111100	**10011101**
001001111	10100101	**10010000**
1111111111	**11111111**	00001111
0000011111	**00000000**	01000011
1000101110	**00111000**	00011100
1000101110	**00001100**	11000010

Table 3: SDES results

3.2 Task 2: TripleSDES implementation

While SDES only uses a 56-bit key, TripleSDES uses two 56-bit keys. The implementation is done by composing DES with itself three times:

$$Enc_{3DES}(p) = Enc_{DES}(k_1, Dec_{DES}(k_2, Enc_{DES}(k_1, p)))$$

$$Dec_{3DES}(c) = Dec_{DES}(k_1, Enc_{DES}(k_2, Dec_{DES}(k_1, c)))$$

where p is the plaintext to encrypt and c is the ciphertext to decrypt. Implementing this in python is straightforward and shown in code blocks 7 and 8. The results from this implementation is shown in table 4

```python
def encrypt_3DES(K1, K2, plaintext):
    return encrypt(K1,decrypt(K2,encrypt(K1,plaintext)))
```

Listing 7: TripleSDES encryption

```python
def decrypt_3DES(K, ciphertext):
    return decrypt(K[:10], encrypt(K[10:], decrypt(K[:10], ciphertext)))
```

Listing 8: TripleSDES decryption

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	0110101110	**11111101**	11100110
1011101111	0110101110	**01001111**	01010000
1111111111	1111111111	**10101010**	00000100
0000000000	0000000000	**00000000**	11110000

Table 4: TripleSDES results

3.3 Task 3: Cracking SDES and TripleSDES

Simple DES enryption is easy to crack as it is only contains $2^{10} = 1024$ possible keys. Even the most naive bruteforce strategies will succeed within a second on modern computers. The length of the ciphertexts CTX1 and CTX2 is 480. Each character is made up of 8 bits, meaning we have $480/8 = 60$ characters in the decrypted text. The most naive implementation looked at each possible key. Then for each key it decrypted 60 8-bit chunks with the key. It would then go through all 60 characters and check its ASCII value. Assuming that the key are based on English text, the filtering method validates a-z (97-122) and A-Z (65-90) values only. This approach gave us 1 key with all 60 characters within the filter ranges:

Execution time: 1.1368 seconds
Key: 1111101010
Decrypted text: simplifieddesisnotsecureenoughtoprovideyousufficientsecurity

A more clever approach was then developed in order to speed up the execution time. Each key goes through a *key_check* method, shown in code block 9. This method evaluates one chunk at the time and exits whenever it fails the filtering criterion. In other words, it will not spend time decrypting chunk 3 if chunk 2 isn't within the a-z/A-Z range. We stop the key iteration once the *key_check* returns a string. This new approach returns the same key and decrypted text as above in 0.0315 seconds.

```python
def key_check(key, chunks=chunks):
    res = []
    for _chunk in chunks:
        dec_str = ''.join(str(i) for i in decrypt(key, _chunk))
        dec_str = binary_to_ascii(dec_str)
        c = ord(dec_str)
        if (c > 64 and c < 91) or (c > 96 and c < 123):
            res.append(dec_str)
        else:
            return False
    return ''.join(res)

keys = [val_to_bits(i, return_len=10) for i in range(2**10)]
for key in keys:
    pos_key = key_check(key, chunks)
    if pos_key: #done
        break
```

Listing 9: Key ckeck

TripleSDES requires two 10-bit keys, that is $2^{20} = 1048576$ possible keys. Average decryption time for a character was $14.6\mu s$ for SDES and $44.1\mu s$ for TripleSDES, which is approxemately 3 times longer. The bruteforce approach would therefore take $3 \times 1024 = 3072$ times longer than SDES. The most naive approach would then spend $(1.1368 \times 3072)/60 \approx 58.2$ minutes. We can speed things up by using a *key_check_3des* that works similarly as *key_check*. This approach produced the following results:

Execution time: 67.4261 seconds
Key1: 1111101010
key2: 0101011111
Decrypted text: simplifieddesisnotsecureenoughtoprovideyousufficientsecurity

We can speed things up further by allowing multiple CPU cores to work on the task in parallell. Code block 10 shows the implementation.

```python
chunksize = int(len(keys) / multiprocessing.cpu_count())
with multiprocessing.Pool() as pool:
    for key, decoded_text in pool.imap_unordered(func=key_check_3des,
    iterable=keys_3des, chunksize = chunksize):
        if decoded_text: #done
            break
```

Listing 10: Parellalizing decryption

This implemention was executed on 4 CPU cores and decrypted the same keys and text as before in 22.7213 seconds.

3.4 Task 4: Webserver with TripleSDES decryption

The webserver was implemented by Pythons Flask micro framework. The user could paste a ciphertext into a basic HTML form and get the decrypted message back. The keys are stored locally on the server side, but that does not mean that this end to end encryption is safe. As proven in task 3, TripleSDES is fairly simple to crack, and thus an attacker listening to traffic on this network will be able to decrypt it. The decryption algorithm could be implemented in other programming languages (such as C++ or Go) and run on more CPU cores. It will be no problem to discover the secret keys within a second in that case. Any ciphertext over the network can then directly be decrypted once these keys are known.

4 Conclusion

The cryptanalysis tool for Poly-alphabetic Ciphers seemed to work fine given the test case in this assignment. However, it was not fully automated as it relies on human intelligence both in picking key length based on result from Kasiski's method and replacing letters in the final top key candidates. The software could be more intelligent by scanning through the most probable key lengths itself. It should also use a more sophisticated statistical comparison in the frequency analysis section. Calculating offset by E tends to fail quite often and might produce unreadable deciphered text, especially with long keys. One could instead compare the entire distribution between columns and English text frequencies to determine most probable shift.

SDES and TripleSDES was both proven to be "too simple" for serious cryptographic applications. Even a bruteforce strategy broke TripleSDES in 22.7 seconds. Small adjustments in different subparts of the decryption algorithms improved execution time by a significant amount. The initial code would spend hours cracking TripleSDES. Python is not considered as a fast language for tasks like this, so implementing and further improving subparts in another programming language like C++ or Go would speed things up even more.

References

[1] Wikipedia contributors. Kasiski examination — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Kasiski_examination&oldid=976443999`, 2020. [Online; accessed 22-September-2020].