DAT-510 Assignment 1

Christian Stigen

UiS, September 18th, 2017

Abstract

This report presents procedures and results for recovering plaintexts and keys for two cipher systems — The classical *Vigenère polyalphabetic* substitution cipher, and the educational *Simplified DES* (SDES) symmetric cipher. The Vigenère plaintext was deciphered using the *Kasiski method* for finding key-lengths, while the individual Caesar cipher alphabets were found by *Kerckhoffs method*: Using alphabet shifts with the highest English letter frequency correlation. Recombining the shifted alphabets produced the plaintext directly, while the key was found by simply decrypting the ciphertext with the plaintext as key. The SDES and TripleSDES keys were found through brute-force. The 20-bit TripleSDES key was found in less than 50 milliseconds by decomposing the decryption function and creating two separate lookup tables and encoding key candidates in a memory efficient bitset. Keys that did not produce all-printable ASCII characters were discarded.

1 Part I — Polyalphabetic ciphers

Below is the raw and formatted plaintext message recovered from the Vigenère cipher [1, 2]. Spaces were added to highlight the coincidence with the length of the key `TSHTAD`.

```
CRYPTO GRAPHY CANBES TRONGO RWEAKC RYPTOG RAPHIC STRENG
THISME ASURED INTHET IMEAND RESOUR CESITW OULDRE QUIRET
ORECOV ERTHEP LAINTE XTTHER ESULTO FSTRON GCRYPT OGRAPH
YISCIP HERTEX TTHATI SVERYD IFFICU LTTODE CIPHER WITHOU
TPOSSE SSIONO FTHEAP PROPRI ATEDEC ODINGT OOLHOW DIFFIC
ULTGIV ENALLO FTODAY SCOMPU TINGPO WERAND AVAILA BLETIM
EEVENA BILLIO NCOMPU TERSDO INGABI LLIONC HECKSA SECOND
```

```
ITISNO TPOSSI BLETOD ECIPHE RTHERE SULTOF STRONG CRYPTO
GRAPHY BEFORE THEEND OFTHEU NIVERS E
```

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 today's computer 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.

1.1 Strategy

The strategy was first to find the key length using the Kasiski method [3, 4], and then use frequency analysis to find each individual monoalphabet required to recover the plaintext.

It is important to note that the goal was solely to decipher the given ciphertext, not to make a general program to solve all Vigenère ciphers.

I will now describe each step in more detail.

1.2 Finding the key length

The very first step was to find repeated substrings in the ciphertext. A repeat section of the ciphertext could mean that it was encrypted using the same part of the key. In other words, by finding enough of these, it should be possible to guess the key length.

Performing such a search was straight-forward: The problem description stated that the key length would not be more than ten letters, so I iterated through the ciphertext with a moving window of decreasing lengths from ten and down to four. Whenever I found a repeat, I recorded the distance between them. Below are excerpts from the `vigenere.py` program output.

```
Looking for repeated substrings with lengths [4, 10]:

  Found 'VJFITRZJHI' at   0, 378 distance 378
  Found 'JFITRZJHIH' at   1, 379 distance 378
  Found 'FITRZJHIHB' at   2, 380 distance 378
  ...
  Found 'UVFPXM'     at 259, 301 distance  42
  Found 'VJFIT'      at   0, 378 distance 378
  Found 'JFITR'      at   1, 379 distance 378
  Found 'FITRZ'      at   2, 380 distance 378
  ...
  Found 'MHVLS'      at 192, 342 distance 150
  Found 'UVFPX'      at 259, 301 distance  42
  Found 'VFPXM'      at 260, 302 distance  42
  ...
  Found 'UDLM'       at 282, 348 distance  66
```

Next, I looked at the recorded distances, trying to find common factors. Of course, many of the repeated substrings could have been mere coincidences. Fortunatley, for this ciphertext, all the distances contained common factors.

```
Attempting to deduce key length:

  Set of distances: 378 66 42 150
  Common factors:   2 3
  Proposed length:  2*3 = 6
```

The idea here is that since the key is repeated over the ciphertext, each duplicate substring would represent the same passage of plaintext. Because of the constant key length, each distance should then be a multiple of the key length. Therefore I factorized each distance and threw away those that were not common to all distances.

Now that I had a possible key length, I organized the ciphertext in columns with the same width as the key length.

```
Finding monoalphabetic ciphers. Ciphertext arranged in
6 columns is:

  VJFITR
  ZJHIHB
  VSUUEV
  MJVGGR
  KOLTKF
  ......

  First column: VZVMKKKLMTBBKVHJHXEQXYZHRAMLBE...
```

The insight here is that if the correct key length has been found, all the letters in the same column would have been encrypted with the same key letter. As can be seen in the program output above, I arranged the text into a column, and then extracted the vertical columns of text. Each such column should then correspond to a single Caesar cipher[*].

To break each Caesar cipher, I compared their relative letter frequencies with those found in English texts. To find how closely they correlate, or match each other, I calculated the *normalized index of coincidence* [5]. I then shifted each column alphabetically and calculated its correlation with the English frequency distribution. A *shift* here means that all As become Bs, all Bs become Cs and so on.

After iterating through all 25 possible shifts, I chose the shift that produced the highest correlation. This is called *Kerckhoffs' method* [1]. The best correlations (or normalized index of coincidence) is shown in the program output below.

```
Frequency analysis:
Shifts alphabets for each column to find the best coincidence
match

  Column 0 length 70: best match 0.058826 with  7 shifts
  Column 1 length 69: best match 0.064844 with  8 shifts
  Column 2 length 69: best match 0.065840 with 19 shifts
  Column 3 length 69: best match 0.061286 with  7 shifts
  Column 4 length 69: best match 0.067704 with  0 shifts
  Column 5 length 69: best match 0.057107 with 23 shifts
```

The correlation function, taken from [5], was

$$\sum_{i=1}^{c} n_i f_i$$

where n_i is the frequency of the ith letter in the column, and f_i the frequency of the *same letter* in normal English texts. The point is to match the frequency distribution pattern. This is best shown in figure 1 on the following page.

By shifting the alphabets, which basically means moving horizontally or vertically in the Vigenère table, we can get a better correlation.

Finally, I recombined each column back to its original — the reverse of the operation mentioned earlier. This produced the plaintext directly.

After I had the plaintext, I recovered the key by simply decrypting the ciphertext with the plaintext. That produced the repeated key, which I cut off at the known length.

[*]This assumes that the Vigenère table does not use scrambled alphabets; each alphabet is written in the same order, and shifted exactly once per key letter.

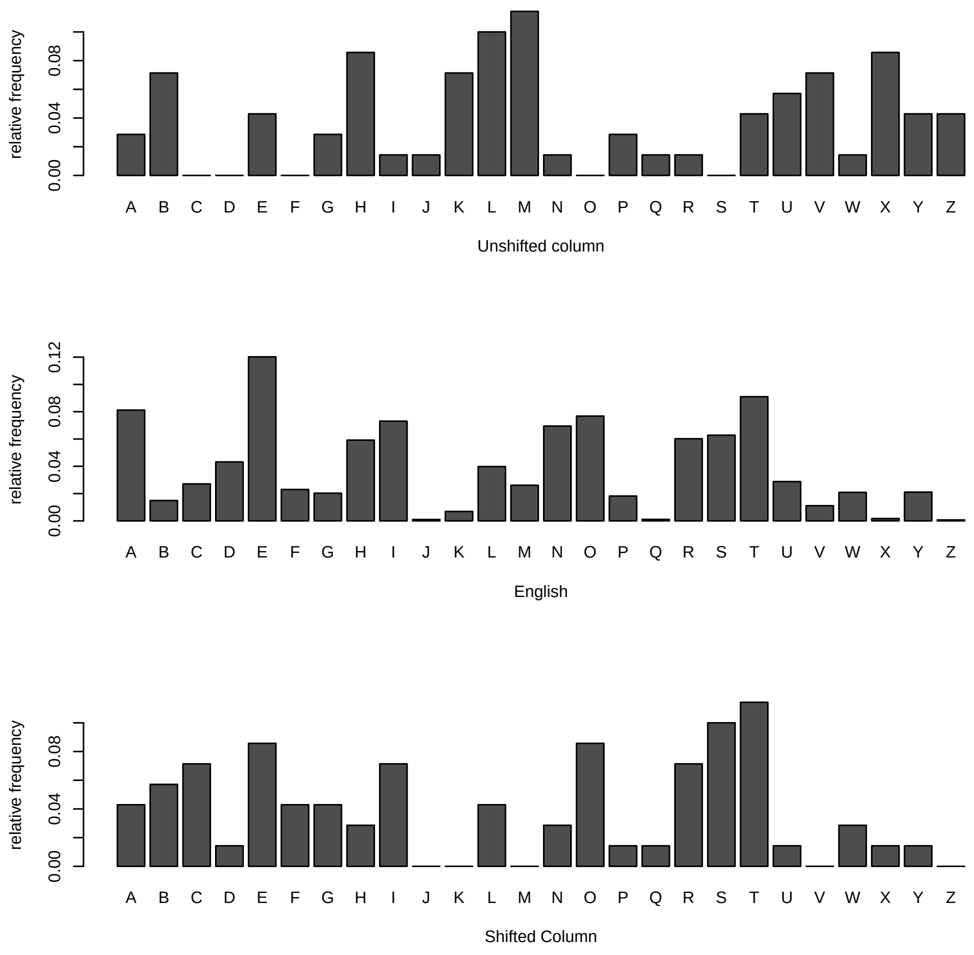

Figure 1: The lower plot shows how the column letter frequency is better correlated with the English after shifting it seven places to the right. Notice how the tall bars at KLM moves to RST.

	Task 1	
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	
Key 1	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 1: Results of SDES and TripleSDES encryption and decryption.

2 Part II — Simplified DES

2.1 Tasks 1 and 2

The two first tasks were to implement Simplified DES (SDES) [6] and TripleSDES in code, and use it to complete a table of plain- and ciphertexts. Both results are shown in table 1.

2.2 Task 3: Recovering SDES and TripleSDES keys

The SDES and TripleSDES keys are given in table 2 on the following page.

Both keys were found through brute-force: Iterating through all possible keys, I decrypted the ciphertext, discarding keys that did not produce all ASCII-printable bytes in the range 32–126.

SDES	Binary	Hexadecimal	Decimal
Key	`1111101010`	0x3ea	1002
TripleSDES	Binary	Hexadecimal	Decimal
Key 1	`1111101010`	0x3ea	1002
Key 2	`0101011111`	0x15f	351

Table 2: Recovered SDES and TripleSDES keys

The plaintext for both ciphertexts were exactly the same:

```
simplifieddesisnotsecureenoughtoprovideyousufficientsecurity
```

Simplified DES is not secure enough to provide you sufficient security.

The TripleSDES keys were found in *less than 50 milliseconds* with a highly optimized C++ library, using Python as a front-end. I opted not to look at any existing literature on how to crack it, and how to speed it up.

What I did was to split the three-part decryption function up in two lookup tables and keep key candidates were kept in a memory efficient bitset. One lookup table was completely pre-generated, while the other was generated in the outer-loop for each 10-bit key. This trades CPU time with increased memory usage, although the tables took less than a megabyte of memory.

Furthermore, the ciphertext was reduced to unique bytes, to spend as little time in the innerloop as needed. The innerloop contained the separated TripleSDES decryption calls:

```
uint8_t byte = unique[n];
byte = decrypted[byte];      // decrypt(k1, byte)
byte = encrypted[k2][byte]; // encrypt(k2, byte);
byte = decrypted[byte];      // decrypt(k1, byte)
```

After finishing the implementation, I read about *meet-in-the-middle* attacks (MITM) [7, 8]. I believe that with the approach I took, I could implement that as well, possibly further reducing the running time.

3 Conclusion

The task was to break ciphers and recover keys for the classical Vigenère polyalphabetic substitution cipher and the educational SDES and TripleSDES symmetric ciphers. I have shown that both ciphers are easily broken in less than 50 ms on modern hardware. Future improvements would be to implement a meet-in-the-middle-attack for TripleSDES, but may not be needed for keys that are only 20 bits.

References

[1] Wikipedia, "Vigenère cipher — wikipedia, the free encyclopedia," 2017. [Online; accessed 14-September-2017].

[2] Wikipedia, "Polyalphabetic cipher — wikipedia, the free encyclopedia," 2017. [Online; accessed 14-September-2017].

[3] M. Dalkilic and C. Gungor, "An interactive cryptanalysis algorithm for the Vigenere Cipher," *Advances in Information Systems*, pp. 341–351, 2000.

[4] Wikipedia, "Kasiski examination — wikipedia, the free encyclopedia," 2016. [Online; accessed 14-September-2017].

[5] Wikipedia, "Index of coincidence — wikipedia, the free encyclopedia," 2017. [Online; accessed 15-September-2017].

[6] W. Stallings, *Cryptography and network security: principles and practices*. Pearson Education India, 2006.

[7] Wikibooks, "Cryptography/meet in the middle attack — wikibooks, the free textbook project," 2017. [Online; accessed 15-September-2017].

[8] Wikipedia, "Meet-in-the-middle attack — wikipedia, the free encyclopedia," 2017. [Online; accessed 15-September-2017].