

Vigenere Cipher Cracking and Simplified Data Encryption Standard Implementation

Abstract

In this assignment, we study and implement two different cryptographic systems and examine a way to break them, using Python. We observe the efficiency of statistical analysis techniques on the Vigenere cipher, and the increasing difficulty of brute-force techniques as keys grow in size on the DES cipher. We briefly discuss optimized ways of breaking these ciphers with modern computers.

1 Vigenere Cipher Cracking

We are given the following cipher :

BQZRMQ KLBOXE WCCEFL DKRYYL BVEHIZ NYJQEE BDYFJO PTLOEM EHOMIC
UYHHTS GKNJFG EHIMK NIHCTI HVRIHA RSMGQT RQCSXX CSWTNK PTMNSW
AMXVCY WEOGSR FFUEEB DKQLQZ WRKUCO FTPLQT GOJZRI XEPZSE ISXTCT
WZRMXI RIHALE SPRFAE FVYORI HNITRG PUHITM CFCDLA HIBKLH RCDIMT
WQWTOR DJCNDY YWMJCN HDUWOF 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 TEXTCJZ VTSGBA NUDYAP IUGTLD WLKVRI HWACZG PTRYCE
VNQCUP AOSPEU KPCSNG RIHLRI KUMGFC YTDQES DAHCKP BDUJPM KPYMBD
IWDQEF WSEVKT CDDWLI NEPZSE OPYIOW

We know that it is an English message which has been encrypted by a polyalphabetic substitution cipher, and that the encryption key is not longer than 10 characters. To break this cipher, one approach could be to try to find the key length. To do so, we are looking for repeating sequences, which have a high probability of corresponding to a repetition of the key. This method is called Kasiski examination.¹²

To perform this examination, I implemented a Python method which loops through the whole cipher, and searches for patterns which appear again in the rest of the cipher. If an occurrence of the current pattern is found, the distance between the two patterns is added to the possible key lengths list. Note that all divisors of the distance are also added to the list, as the key could have repeated multiple times between the two occurrences of the pattern. The `get_divisors()` subroutine does not return divisors which are greater than the maximum key length.

```
def kasiski_examination(cipher):  
    possible_key_lengths = []  
  
    for pattern_length in range(MIN_PATTERN_LENGTH, MAX_PATTERN_LENGTH):  
        for index in range(len(cipher)):  
            substring = cipher[index:index+pattern_length]  
            distance = cipher[index+pattern_length:].find(substring)
```

```

        if distance != -1:
            possible_key_lengths.extend(get_divisors(distance + pattern_length))

probable_key_lengths = {length:possible_key_lengths.count(length) for length in possible_key_lengths}

return sorted(probable_key_lengths.items(), key=lambda x:x[1], reverse=True)

```

When executed on the given ciphertext, we get the following output.

```

key length : 2 -> 115 occurrences
key length : 4 -> 115 occurrences
key length : 8 -> 115 occurrences
key length : 3 -> 57 occurrences
key length : 6 -> 54 occurrences
key length : 7 -> 31 occurrences
key length : 9 -> 19 occurrences
key length : 5 -> 2 occurrences
key length : 10 -> 2 occurrences

```

Which implies that the key is probably of length 2, 4 or 8. Since 2 and 4 are divisors of 8, it is safe to assume that they appear so often here because the `get_divisors()` method returns all divisors smaller than the maximum key length. Our guess is therefore that the key is of length 8, because it has the most sub-pattern repetition occurrences. Next, we perform a frequency analysis for each set of character with an equal position in the ciphertext, modulo the key length. These sets will be called columns from now on, because they each represent a column of the ciphertext when it is splitted every key length characters.

This frequency analysis is implemented in the following method.

```

def frequency_analysis(cipher, key_length):
    candidates = []

    for column in range(key_length):
        frequencies = char_frequency(message[column::key_length])
        column_candidates = []

        for most_common in frequencies[:1]:
            E_offset = (LETTERS.find(most_common[0]) - LETTERS.find('E')) % 26
            column_candidates.append(LETTERS[E_offset])
        print("key[{}] candidates : {}".format(column, column_candidates))
        candidates.append(column_candidates)

    return candidates

```

This method computes character frequency for each column, and returns the key character which would have encrypted this column if the most common character effectively corresponds to the letter E, which is the most used letter in the English language.

```

key[0] candidate : ['B']
key[1] candidate : ['D']
key[2] candidate : ['A']
key[3] candidate : ['A']
key[4] candidate : ['E']

```

```
key[5] candidate : ['T']
key[6] candidate : ['C']
key[7] candidate : ['Y']
```

This gives us the following possible key : 'BDAAETCY'. We then try to decrypt the ciphertext with this key using the following method.

```
def poly_decrypt(cipher, key):

    # make sure key and cipher are in uppercase and without whitespace
    cipher = cipher.upper().replace(' ', '')
    key = key.upper().replace(' ', '')

    # expand the key so that it matches the length of the cipher
    expanded_key = ''.join(key[i % len(key)] for i in range(len(cipher)))

    decrypted_message = ''

    for cipher_letter, key_letter in zip(cipher, expanded_key):
        decrypted_index = (LETTERS.find(cipher_letter) - LETTERS.find(key_letter)) % 26
        decrypted_message += LETTERS[decrypted_index]

    return decrypted_message
```

When using this method on the given ciphertext with the key 'BDAAETCY', we get a glimpse of the original plaintext. If we format the text by columns, we get the following output.

```
01234567
-----
ANZRIXIN
ALXESJAG
EIDKNFVN
ASEHEGLA
INEEXKWH
...
CRJPTRNA
LYDISKOG
ETSERRRE
CAWLEUCR
YPEOLFGY
```

This is not the original message, but we can clearly identify possible words in this text, like CRYPTANALYSIS or CRYPTOLOGY. However, it seems that columns 2 and 5 are offset by the wrong key. To find the right key, we try matching the last 10 characters of the ciphertext (EPZSEOPYIW) to the guessed word (CRYPTOLOGY).

For the first character mismatch, we search the key such that T encrypts to E, and for the second, the key such that O encrypts to Y. By reading a Vigenere table, we can see that these unknown key parts we are looking for are L and K. By decrypting the ciphertext with the new key 'BDLAEKCY', we get the original plaintext.

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 Simplified DES Implementation

2.1 Test cases

The following table sums up the results of the test cases in task 1, using the SDES algorithm implementation.

Raw Key	Plaintext	Ciphertext
000000000	00000000	11110000
000001111	11111111	11100001
001001111	11111100	10011101
001001111	10100101	10010000
111111111	11111111	00001111
000001111	00000000	01000011
100010110	00111000	00011100
000001111	00000000	01000011

The following table sums up the results of the test cases in task 2, using the Triple SDES algorithm implementation.

Raw Key 1	Raw Key 2	Plaintext	Ciphertext
100010110	011010110	11010111	10111001
100010110	011010110	10101010	11100100
111111111	111111111	00000000	11101011
000000000	000000000	01010010	10000000
100010110	011010110	11111101	11100110
101110111	011010110	01001111	01010000
111111111	111111111	10101010	00000100
000000000	000000000	00000000	11110000

2.2 SDES cracking

The Simple DES encryption algorithm is very weak, as it allows only $2^{10} = 1024$ different keys. We can find the key used for encryption by bruteforce, decrypting the ciphertext with every possible key and identifying possible plaintext. The naive implementation of bruteforce cracking is to decrypt each key linearly, with the following method.

```
def des_bruteforce(cipher, probable_word):
    probable_keys = []

    for key in range(1024):
        key = format(key, '010b')
        key = create_bitfield(key)
        message = decrypt_message(cipher, key)
```

```

message = bitfield_to_string(message)
if message.find(probable_word) != -1:
    probable_keys.append(key)
    print('key : {}'.format(bitfield_to_string(key)))
    print('message : {}'.format(bin2ascii(message)))

return probable_keys

```

The ciphertext decrypted in section 1 contained almost only obvious words for a cryptography assignment, so the first probable word I used was 'des'. When running this bruteforce attack on CTX1.txt, we get the following output.

```

key : 1111101010
message : simplifieddesisnotsecureenoughtoprovideyousufficientsecurity
Elapsed time : 1.356s

```

Which gives us the key used to encrypt the message : 1111101010. We could do the same for the Triple DES algorithm, but let's first do some estimations about the time it would take to run. Bruteforcing Triple DES requires decrypting $2^{20} = 1048576$ different keys. If we approximate the time it takes for a single Triple SDES decryption to be about 3 times longer than a simple SDES decryption, we can estimate that testing all keys linearly would be $3 \times \frac{2^{20}}{2^{10}} = 3072$ times longer than the previous bruteforce. Depending on the hardware, that could take well over an hour. What we can do is parallelize the decryptions across all CPU cores of the machine. To do so, I implemented the following method.³

```

def decrypt_triple_sdes_cipher_parallel():
    numkeys = 1024 ** 2
    chunksize = int(numkeys / multiprocessing.cpu_count())
    keys = [(format(x, '010b'), format(y, '010b')) for x in range(1024) for y in range(1024)]

    probable_word = ascii2bin('security')

    with multiprocessing.Pool() as p:
        for message, keys in p.imap_unordered(func=parallel_triple_bruteforce, iterable=keys, chunksize=
                                             chunksize):
            if message.find(probable_word) != -1:
                print('keys : ({},{})'.format(bitfield_to_string(keys[0]), bitfield_to_string(keys[1])))
                print('message : {}'.format(bin2ascii(message)))

```

This method shares the decryption process across all cores, allowing to greatly speed up the computation time. On an 8 core CPU clocked at 3.6GHz, this method took 475 seconds to complete, which is way faster than what was expected by linear decryption. This also matches our estimates, because $\frac{3072}{8} = 384$, and this method took approximately 384 times longer to complete than the simple DES bruteforce ($384 \times 1.356s = 520s$). By guessing the messages in CTX1 and CTX2 might be the same, I used 'security' as the probable word, which turned out to be true. The method produced the following output.

```

keys : (1111101010, 0101011111)
message : simplifieddesisnotsecureenoughtoprovideyousufficientsecurity
Elapsed time : 474.742s

```

The keys used to encrypt CTX2.txt are '1111101010, 0101011111'. This bruteforce process could be drastically accelerated by using a compiled language or even hardware acceleration to compute the decryptions, but in the scope of this assignment this has no practical application.

3 Conclusion

In this assignment, we implemented two different ciphers using Python with no external packages. We saw that even a cipher which remained unbroken for three centuries can be attacked by a smart modification allowing statistical analysis, when this is not possible on the whole cipher. We also could see the vulnerabilities of DES, even though a simplified version was used here. We discussed optimization ideas for speeding up the attack process in the case of longer keys. It would be interesting to try the same attacks with a compiled language such as C to benchmark the time difference between interpreted and compiled programs when it comes to intense CPU loads.

References

- [1] Wikipedia contributors. Vigenre cipher — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Vigen%C3%A8re_cipher, 2019. [Online].
- [2] Daniel Rodriguez-Clark. Kasiski analysis: Breaking the code. <https://crypto.interactive-maths.com/kasiski-analysis-breaking-the-code.html>.
- [3] Rosetta Code contributors. Parallel brute force. http://rosettacode.org/wiki/Parallel_Brute_Force#Python.