



# SYMMETRIC ENCRYPTION ATTACK

*A way to exploit encrypted text*
*in order to autonomously find its access key*

Lab of Computer Security
H-FARM College
A.Y. 2020-2021
Prof. L. Maccari

## Structure of the Report

1. Introduction
   Our mission, reasonings to develop this project and how we managed it.
2. What is Cryptography?
   Brief description and basis on the touched topics at a theoretical level.
3. Project Description
   Walkthrough of all the major steps in the work development ranging from Linux bash to Python coding used to obtain our final results.
4. Considerations
   Based on coding implications and possible solutions/problems we state our position on the obtained results.
5. Conclusions
   Our final thoughts, conclusions inherent to the project itself. The results, successes and obstacles encountered.
6. Commands and References
   Section dedicated to the listing of all the supporting implemented elements and codes used in the project itself, all recalled in the description section.

**GROUP MEMBERS**
Mattia Brocco – 873058
Alessandra Cancellari – 876052
Anna Costa – 875243
Filippo Guerra – 873140
Eleonora Sartori – 872366

# Introduction

Nowadays, cryptography and encryption processes are essential for data privacy. Pursuing and testing the strength of these kinds of securities captured our attention since we deal with them daily by using our messaging applications. Our main aim was to learn how our conversations are protected, the different scenarios depending on the encryption security, the privacy strength provided based on the selected technology, and the overall effectiveness.

By following the SEED Lab instructions provided, we decided to develop our project "Secret-Key Encryption Lab" in which we structured a Symmetric Encryption Attack based on different ciphertext using various encryption keys generated by a script.

The quantity of work was divided equally among us. In particular, we worked together during the afternoons after the lectures. The material was shared through Google Drive that we used to implement the contents of our project simultaneously. The project development part in Ubuntu shell was made during a physical meeting of the whole team, as well as the frequency analysis. The following parts come mainly from brainstorming and attempts on .ipynb files that were continuously shared up-to -date among team members to discuss on attempts.

In our project, we analysed what Cryptography is and how it works, focusing on the relevance that it could have in our work.

## What is cryptography?

Cryptography is a technique used to secure information and communications, in order to avoid unauthorized access to information from malicious third parties, known as "adversaries". This technique is possible through the use of secret codes that can be understood only by those people for whom the information is intended. Despite the oxymoron, the word "cryptography", which aims to hide secrets, has a very transparent etymology. It derives from Greek, and its prefix "crypt" means "hidden", and the suffix graph means "writing".
It is divided into two branches: cryptography and cryptanalysis. The former is the one we have shortly described above, i.e. the ability to hide information through the use of codes. The latter is the set of techniques that are used to break those codes.

Cryptography has four main features:

1. Confidentiality: the information can only be accessed by the person for whom it is intended, and no other person except him can access it;
2. Integrity: it is not possible to modify information in storage or transition between sender or receiver without any changes to information being detected;
3. Non-repudiation: the creator/sender of the information cannot deny its intention to send information at a later time;
4. Authentication: the identities of sender and receiver are confirmed, accordingly also the destination/origin of the information is.

In order to guarantee the confidentiality of the data, they are usually made incomprehensible by encryption, that is a technique used to hide and secure data basically through algorithms. It consists in the process of translating a message (called a "plaintext") into an encrypted message (called a "ciphertext"). The process of converting the encrypted message in the original one is known as decryption, and it is possible by the use of a secret key.

All these features regarding cryptography let us think that this system guarantees the highest security. However, this is not true. As we have dealt with during our lessons, this security also depends on other factors. They are the channel through which the key is sent and the underlying protocol used to allow communication between the sender and the recipient. Moreover, cryptography is only one of the techniques that aim to increase security, but it is not the only one. Other essential techniques are message digests, digital signatures, digital certificates, public key infrastructure, notary and traffic padding that can strengthen the level of trust between two entities.

Despite all these notable characteristics, we will focus on cryptography and its sub-branches, such as symmetric key encryption, asymmetric key encryption, and cryptographic protocols. We intend to describe them briefly to contextualise our report. Let us start with the difference between the two types of encryptions. Symmetric and asymmetric encryption are two techniques used in encryption and decryption. Symmetric encryption is the method of using the same cryptographic keys for both encryptions of plaintext and decryption of ciphertext.
Asymmetric encryption is the method of using a pair of keys: the public key, which is disseminated widely, and a private key, which is known only to the owner. However, they differ in many areas such as complexity, speed and algorithms. For what concerns the complexity, since only one key is used in both operations, symmetric encryption is more simple than asymmetric encryption that uses separate keys for both operations. From the speed side, Asymmetric encryption is slower compared to symmetric encryption, which has a faster execution. Finally, they use different types of algorithms. RC4, AES, DES, 3DES are some of the most used in Symmetric Encryption. Instead, Diffie-Hellman and RSA are some standard algorithms used in Asymmetric Encryption.

A protocol, on the other hand, is a document that describes how the algorithms should be used to protect information transfer. More profoundly, it is a convention between two entities that

communicate with each other on how to exchange data. The description of a protocol must include details about all representations, structures of the data, and all details about how to employ the protocol by programs. An example of a protocol is Secure Shell (SHH), is an Internet communication protocol used to allow users to log into other laptops and run commands. It gives the possibility to exchange data using a secure channel between two PCs.

We are fascinated by these advanced encryption mechanisms; for this reason, we chose this topic for our project. Our goal is to decrypt a ciphertext based on a mono alphabetical algorithm in which each letter is swapped with another. Now we can start the analysis of our project by highlighting the processes and difficulties that we have discovered to clear up with it.

## Project Description

The implementation of the project is the main and more interesting part of the report being the demonstration of our efforts. We started our task following the instructions of the SEED Lab, reproducing point 1-2.1 of Crypto Lab - Secret-Key Encryption which is called 'Symmetric Encryption Attack'.
The intent of this project is to get familiar with the concepts of secret-key encryption. In particular, we learned how to decrypt a test using frequency analysis.

We had the following materials:

- Corpus text;
- Ciphertext;
- Hash of the decrypted ciphertext;

To launch the project we downloaded the ciphertexts and the corpus text into the Virtual Machine, saving them into a shared folder in the Desktop. Then, we located the directory in the GNU\ Linux monitor. To continue our project we adopted two commands: tr and cat. The first one, 'tr' was utilized to convert the words in the file from lowercases into uppercases, in order to create a new file denominated 'step-1.txt'. [Fig. 1]

Then, the second command 'cat' helped us to remove the punctuation, paragraph, spaces and quads, creating a decipherable text. More deeply, the 'cat' was used to open the file, with the bar '|' its output was re-addressed to the first 'tr'.
Then we used the Regex '-d "[:punct:] \n\r" ' to eliminate the punctuation, spaces and '\n'. The regex (regular expression) in GNU/Linux is 'POSIX compliant' and is the character writing syntax. Then to advance the program, with another bar '|' the output of 'tr' was re-sent to 'sed -e' to remove a character similar to double quotes. [Fig. 2]

Once the ciphertext was clean enough, we moved to Python to perform the actual decryption of the ciphertext. The steps are the following:

FIRST STEP: Frequency Analysis
It is the count of how many times each letter appears in the ciphertext. To do this we used the freq_analysis function, which takes the text as input, creates a dictionary with the frequency of the individual letters, then converts it into a list of tuples with the structure (letter, frequency) so that it can be sorted by frequency values in the tuples, and finally the function returns a string of letters sorted by frequency by joining the letters in the tuples.
This function has been applied to both ciphertext and corpus. The string obtained from the analysis on the ciphertext will be called $K_0$.

The frequency analysis on ciphertext actually returns a key, in fact through the "decryption" function (which is in the Decryption_fun.py class), each letter of $K_0$ is associated with the corresponding letter in order of frequency in the corpus (by creating a dictionary with a zip of the two strings). In this way, it is possible to simply replace the respective letter of $K_0$ in the ciphertext to generate the first (partial) plaintext. [Fig. 3]

SECOND STEP: Bigram Analysis & Scores
In order to evaluate the quality of a decrypted ciphertext, we used bigram analysis, so that it was possible to create an evaluation metric for candidates.
The function is called "bigram_analysis". The function looks for all bigrams (pairs of letters) in the input text (which is the ciphertext without any punctuation sign), each with its own frequency. The "collection" library has been used for this step and in particular the "Counter" module, which by definition returns a dictionary. We therefore have a dictionary {bigram: frequency}, which is transformed into {bigram:score}, with the following formula, in order to penalize lower scores (as a property of the logarithm function):

$$Score = natural\ logarithm(frequency)$$

Then, the dictionary obtained is ordered with respect to the bigram score and this function is applied to both corpus and ciphertext. [Fig. 4]

On the basis of the bigram analysis, we've written the "candidate_plain_score" function. It takes a text (string, which would be a plaintext candidate without any space or punctuation) and then calculates, based on the "bigram_analysis" function, the dictionaries with the bigram score of both the corpus and the candidate plaintext. For each bigram in the candidate, we get the score of that bigram in the corpus. This way, the bigrams in the candidate that are not in the corpus are ignored, thus they don't increase the score. Now, given the bigrams in the candidate and the scores that each of them has in the corpus, we can create a dictionary {bigram:score} by zipping these two lists. Finally, the total score of a candidate will be given (as the return of the function) by the sum of all the scores (values) of the dictionary. The score of our first candidate (the ciphertext translated with $K_0$) is 4110. [Fig. 5]

THIRD STEP: Attempts

We have tried several attempts to carry out this project, so we can say that in the end, a test-and-trial approach got us to the correct solution, and it made us learn more on how to handle strings in Python, and how to reduce computational efforts.

1st attempt

The first attempt concerns the list of all alphabet permutations. This attempt failed as it would be a list of 26! elements, a vast number that cannot be processed by Jupyter Notebook, as that list would require a huge amount of allocated memory.

2nd attempt

In the second attempt, for each character in the key, that character is swapped with a letter of the alphabet. In the wheel, initially, the first character is replaced with each of the letters of the alphabet, then all the others. Then we get 26 keys where the first letter for the first time is A and then all the keys, then B and all the keys and so on.
This creates $26 * 26$ keys, i.e. 676 keys. Obviously, to avoid duplicate letters in the key, the letter that has been replaced (e.g. I in the first iteration), is moved to the $K_0$'s index of the new letter (e.g. A in the first iteration).
Given the total number of new keys obtained, we thought this could be a suitable method, but it was not successful. Swapping letters in the first part of $K_0$, interferes heavily with the results of frequencies analysis. The score obtained from each of these keys tended to be lower than the initial Key.

3rd attempt

Once the first candidate was partially translated and given an intermediate step of ciphertext cleaning (where all spaces were still preserved), we've tried to check all the partial decrypted words against the most similar word in the English dictionary (provided by the "enchant" library). Similarity of words was computed with the Levenshtein algorithm for string similarity. This way, the output for each word would have been the most similar word in the dictionary, then all the results could have been merged to obtain a more than satisfying decrypted version of the ciphertext without using a decryption key.
While performing this solution, we got stopped by a computational limitation. This was because of the length of the ciphertext, and the continuous search with the enchant dictionary. Even if we've tried to do it completely with NumPy and vectorized functions to improve performance, this method was computationally too demanding.

FOURTH STEP: Improving $K_0$ with swaps

In this part of the project, we applied the function 'key_improver' to the key obtained from frequency analysis, which swaps each character with the next one. Then, the score of each swap is compared with that of the initial key, and only the "winning" swaps are returned. The score used is the log of the bigram frequency in the corpus.

The function 'key_improver' gets $K_0$ as an input. Subsequently, for each character in the key, it swaps the character with the following one and stores the new key. Then, for each key found, the scores calculated for that key is compared with one obtained with the first key. Finally, return the output of the function that is a list of tuples (number of iteration, keys) of only those keys that produce a better score than the first key. [Fig. 6]

FIFTH STEP: Permutations

Next, with this chunk, for each winning swap (5, 7, 12, 13, 14, 13, 24), we created all the permutations of the substring (since some winning swaps were consecutive, we switched all consecutive swaps together, as from 12 to 14). Then, with the nested for loop below, we created 575 new keys (because K0 was excluded), that are all the possible combinations of the above-mentioned permutations. Lists with permutations have length equal to $len(string)!$, and all the combinations of those lists is defined as the product of all these lengths, indeed [Fig. 7]:

$$2! * 2! * 3! * 4! = 576$$

It is necessary to underline that, at this point, we were quite sure to be heading in the right direction since we've previously computed the correct key through manual swaps. Indeed, thanks to this passage, we have been able to check the presence of the right key in the list obtained with the permutations just done (see Fig. 7).

The following is the process we used to find the key through the manual swaps. First of all, we tried to decrypt the cipher with the key generated from the frequency analysis. While we were doing this step, we noticed that such a key was, except for eight letters, right. Another consideration that came up to our attention was that those eight letters were attributable to one bigram and two trigrams. In detail, this was the situation: "ITKSH**BE**GAOQ**YZW**JXUPCMLDN**RFV**", where the letters in bold were the misplaced ones.

At this point, we tried to swap manually the letters relying on the text we had just partially decrypted. Nevertheless, in only a few steps we were able to find the right key, which is: "**itkshebgaoqywjzxupcmldnfvr**".

SIXTH STEP: Brute Force

The function 'brute_force', starting from $K_0$, iterates for each element of the list of new keys (obtained as all the combinations of substrings' permutations in Step 5), to find an improvement of the original key in terms of score. Nonetheless, if a perfect match with the correct sha256 hash is found, the loop is interrupted, and the right key is returned. [Fig. 8]

With this function, we ended up finding a perfect match with the hash, but we additionally performed a double-check in the Ubuntu shell on our output file, to see if the text was decrypted correctly. [Fig. 9]

## Conclusions

In the end, after several attempts and afternoons spent looking for a solution, we achieved our final result. During the implementation of the project and after many tentative searches, at the end we found the key, tested it, and confirmed that it decrypts the text. This has been the main obstacle that we encountered during the execution of the project. We spent entire afternoons to reach our objective: find the right key.
In particular, after a deep analysis, we observed that the algorithm with which the cipher was encrypted was based on 'transposition' and 'substitution'. Thus, the statistical correlation between the letters of the ciphertext and plaintext has been preserved. As a consequence, there was always exact mapping of the ciphertext letters into the letters of the plaintext.
This project fascinated us from the very beginning allowing us to get involved until the first day. Cryptography is an intrinsic world full of secrets to discover, here we have only taken a first step towards its discovery. We are very satisfied with the efforts and results obtained and we hope to have been exhaustive in the explanation of the project.

Thank you,
Digital Key Exploiters

## References


"Symmetric vs Asymmetric Encryption – What Are the Differences?", *ClickSSL Blog - Information about SSL Certificates & Infosec*, 24 Sept. 2020, www.clickssl.net/blog/symmetric-encryption-vs-asymmetric-encryption

"Asymmetric Cryptography." *Asymmetric Cryptography - an Overview | ScienceDirect Topics*, www.sciencedirect.com/topics/computer-science/asymmetric-cryptography

"Secret Key Encryption." *Secret Key Encryption - an Overview | ScienceDirect Topics*, www.sciencedirect.com/topics/computer-science/secret-key-encryption

Arampatzis, Anastasios. "Encryption Protocols: Explained." *Venafi*, 8 Mar. 2019, www.venafi.com/blog/how-do-encryption-protocols-work

Khillar, Sagar. "Difference Between Encryption and Cryptography." *Difference Between Similar Terms and Objects*, 14 July 2020, www.differencebetween.net/technology/difference-between-encryption-and-cryptography/

"What Is Cryptography and How Does It Work?" *Synopsys*, www.synopsys.com/glossary/what-is-cryptography.html

"Cryptography and Its Types." *GeeksforGeeks*, 8 Jan. 2020, www.geeksforgeeks.org/cryptography-and-its-types/

## Commands

Fig. 1: Creating a new upper text file of the ciphertext called step1.txt

```
[10/25/20]seed@VM:~/.../test-1$ tr [:lower:] [:upper] <ciphertext.txt> step1.txt
```

Fig. 2: Removing all the punctuation and spaces by defining ranges creating a final file that will be used for our project development on the encryption attack called step2.txt

```
[10/25/20]seed@VM:~/.../test-1$ cat step1.txt | tr -d "[:punct:] \n\r" | sed -e
's/[^A-Z]//g' >step2.txt
```

Fig. 3: Python function used to count the most frequent letters in a given and prepared text

```
# Frequency analysis function

def freq_analysis(text):

    # Recap the frequencies in the cipher (K letters, V frequencies)
    letterToFreq = dec.letter_freqs(text)

    # The dictionary is then converted to a list of tuples
    # (letter, frequency) that are sorted according to frequency
    freqPairs = list(letterToFreq.items())
    freqPairs.sort(key = lambda x:x[1], reverse = True)

    # The final string is created in such a way that
    # letter sare ordered by frequency
    freqOrder = [pair[0] for pair in freqPairs]

    return ''.join(freqOrder)
```

Fig. 4: Python function used to count and evaluate bigrams present in the texts

```
def bigram_analysis(self, t):
    self.t = t

    BigrtoFreq = Counter(t[idx : idx + 2] for idx in range(len(t) - 1))

    # Give a score to each bigram (score = np.log(bigram's frequency) )
    BigrtoFreq = dict(zip(BigrtoFreq.keys(), [round(np.log(v), 10)
                                              for k,v in BigrtoFreq.items()]))

    # Bigrams are finally ordered by decreasing frequency
    BigrtoFreq = {k: v for k, v in sorted(BigrtoFreq.items(),
                                          key = lambda item: item[1], reverse = True)}

    return BigrtoFreq
```

Fig. 5: Python function to compute the score of a candidate text

```
def candidate_plain_score(t):

    # Recall the previous function to generate a
    # dict and get only the list of the keys
    cand_BI_freq = dec.bigram_analysis(t)
    Corpus_BI_freq = dec.bigram_analysis(corpus)

    bigrams_in_cand = list(cand_BI_freq.keys())

    # Get the score (log(freq)) of the bigrams in the corpus
    # for every bigram found in the candidate plaintext.
    # Added a check (if) for those bigrams that are not in the corpus
    scores_corpus_bigrams_in_cand = [Corpus_BI_freq[bi] for bi in bigrams_in_cand
                                     if bi in Corpus_BI_freq.keys()]

    # Assign to every bigram of the candidate plaintext
    # the score that that biagram has in the corpus
    dict_scores_corpus_bigrams_in_cand = dict(zip(bigrams_in_cand,
                                                  scores_corpus_bigrams_in_cand))

    # The overall score of the candidate plain text is
    # computed as the sum of all the values in the above dict
    return sum(dict_scores_corpus_bigrams_in_cand.values())
```

Fig. 6: Python function used to get the best key (with a score) based on the input

```
def key_improver(k):

    F_Key_L = list(k)
    List_swap = []
    for n in range(len(F_Key_L)-1):
        F_Key_L2 = F_Key_L.copy()
        # swapping (item assigment) is not possible for string, so we used a list
        F_Key_L2[n], F_Key_L2[n+1] = F_Key_L2[n+1], F_Key_L2[n]
        List_swap += [ "".join(F_Key_L2) ]

    First_Score = candidate_plain_score(first_candidate)
    best_from_swap = []
    for e, nk in enumerate(List_swap): # keep the iteration number and the key
        new_try = dec.decrypter(nk, s2)
        new_score = candidate_plain_score(new_try)
        if new_score > First_Score: # winning scores
            best_from_swap += [ (e, nk) ] # return list of tuples
        else:
            continue

    return best_from_swap
```

Fig. 7: Python function used to try all the possible permutations of our keys

```
# In this chunk, given the output from the previous one,
# we now generate all the permutations of the improving swaps
# (5,7,12,13,14,23,24). So every substring, will generate len(substring)! permutations.
# In practice, we've created for lists of length 2!, 2!, 4!, 3!

from itertools import permutations

perms_5_6 = [''.join(p) for p in permutations(First_Key[5]+First_Key[6])]
perms_7_8 = [''.join(p) for p in permutations(First_Key[7]+First_Key[8])]
perms_12_13_14_15 = [''.join(p) for p in permutations(First_Key[12:16])]
perms_23_24_25 = [''.join(p) for p in permutations(First_Key[23:26])]

print(len(perms_5_6),len(perms_7_8),len(perms_12_13_14_15),len(perms_23_24_25),sep = " // ")

# ------------------------------------------------------------------------
# From these sublists, we've replace that part of the initial key
# in order to create a list that has all the combinations of substrings
# that we've just generated.

# The result is a list, whose length is (2*2*24*6) and to which
# the elements that are equal to First_Key are removed (actually only one
# according to combination definition)

next_step_keys = []
for s in perms_5_6:
    for t in perms_7_8:
        for u in perms_12_13_14_15:
            for v in perms_23_24_25:
                next_step_keys += [ First_Key[:5] + s + t + First_Key[9:12] + u + First_Key[16:23] + v]

next_step_keys = [x for x in next_step_keys if x != First_Key]
# converted to lowercase, since the ciphertext is in lowercase as well!
```

executed in 89ms, finished 00:53:05 2020-10-26

```
2 // 2 // 24 // 6
```

Fig. 8: Python function used to get the final key and check the final hash of the plain text

```
# Hash to compare our result with
true_digest = open("sha256sum.txt").read()

def brute_force(key0, list_of_keys, correct_hash):

    best_key = key0
    best_cand = dec.decrypter( key0, s2 )
    max_score = candidate_plain_score(best_cand)
    for key in list_of_keys:
        candidate = dec.decrypter( key, s2 )
        score = candidate_plain_score( candidate )
        candidate_for_hash = dec.ciphertext_decrypter(key, ciphertext) # to get the hask check right
        # we are sure that encoding is "utf-8" after professor Maccari's tutorial
        utf8_dig = hashlib.sha256( bytes(candidate_for_hash, "utf-8") ).hexdigest()
        if utf8_dig == correct_hash:
            return key
        elif score > max_score:
            max_score = score
            best_key = key
            best_cand = candidate
    return best_key.lower()
```

Fig. 9: Decrypting the text with our verified key and the letter frequency of the given corpus

```
[10/25/20]seed@VM:~/.../test-1$ tr 'ITKSHEBGAOQYWJZXUPCMLDNFVR' 'etaonihsrdluwcm
fgypbvkxjqz'\
> <step1.txt> textdecr.txt
```

```
swarmed with well-provisioned sutlers and austrian jews offering
all sorts of tempting wares. the pvlograds held feast after feast,
celebrating awards they had received for the campaign, and made
expeditions to olmtz to visit a certain caroline the hungarian,
who had recently opened a restaurant there with girls as waitresses.
rostv, who had just celebrated his promotion to a cornetcy and bought
densovs horse, bedouin, was in debt all round, to his comrades and
the sutlers. on receiving bors letter he rode with a fellow officer
to olmtz, dined there, drank a bottle of wine, and then set off alone
to the guards camp to find his old playmate. rostv had not yet had
time to get his uniform. he had on a shabby cadet jacket, decorated with
a soldiers cross, equally shabby cadets riding breeches lined with
worn leather, and an officers saber with a sword knot. the don horse
he was riding was one he had bought from a cossack during the campaign,
and he wore a crumpled hussar cap stuck jauntily back on one side of his
head. as he rode up to the camp he thought how he would impress bors
and all his comrades of the guards by his appearancethat of a
fighting hussar who had been under fire.

the guards had made their whole march as if on a pleasure trip, parading
their cleanliness and discipline. they had come by easy stages, their
knapsacks conveyed on carts, and the austrian authorities had provided
excellent dinners for the officers at every halting place. the regiments
had entered and left the town with their bands playing, and by the grand
dukes orders the men had marched all the way in step (a practice on
which the guards prided themselves), the officers on foot and at their
proper posts. bors had been quartered, and had marched all the
way, with berg who was already in command of a company. berg, who had
obtained his captaincy during the campaign, had gained the confidence of
his superiors by his promptitude and accuracy and had arranged his money
matters very satisfactorily. bors, during the campaign, had made the
acquaintance of many persons who might prove useful to him, and by
```