# Introduction to Cyber Security
# – Secret Key Cryptography –

**Deadline: 16th November, 2023**

## Introduction

The main goal of cryptography is confidentiality, often achieved through encryption. This first practical exercise concerns itself with some historical and modern cryptographic algorithms and methods to analyze encrypted text. There is a wide variety of literature on basic cryptographic methods [2, 3, 6].

In this exercise we will first work with the open source e-learning tool *CrypTool*. The tool serves as a demonstration for many cryptographic algorithms. There are several versions of that tool. In this lab we will focus on the Java implementation of *CrypTool*, namely *JCrypTool 2.0* and the online version *CTO*. These programs are cross-platform compatible. You can either download the tool from the website [1] or use it directly in your web browser at `https://www.cryptool.org/en/cto/`.

*JCrypTool* provides many historical and modern cryptographic algorithms and adequate analysis methods to study and break weakly encrypted messages. The tool also comes with visualization capabilities, which makes *CrypTool* a handy tool to get a better understanding of the complex world of cryptography.

Once a basic understanding of the historical and modern cryptographic algorithms has been obtained, you will analyze some encrypted files on your own. To this end, we will consider different kinds of scenarios typically covered in the field of cryptography. We will consider ciphertext-only attacks as well as (partial) known plaintext attacks. In the setting of ciphertext-only attacks, we will implement a brute-force search to break an AES encryption with a weak password. Later, we will improve the naive brute-force by utilizing a method called hill climbing to reverse a mono-alphabetic substitution. On the side of (partial) known plaintext attacks, we will analyze an encrypted zip container. Here, the predictable header information of the file format specification serves as a partially known plaintext.

# Notes

Note that this task assumes *basic knowledge* to have been learned in previous studies. If you find yourself missing such knowledge[1], it is your responsibility to attain it in self study. This also holds for topics that are covered in the lecture or exercise classes of introduction to cyber security that have not yet been held, i.e., this practical task is independent of the lectures and exercises with respect to this issue.

# 1 Preparation

To get familiar with some historical and modern cryptographic algorithms, we start with a case study utilizing *CrypTool*. Try to *understand the analysis/attack for every scenario* presented below. Think about reasons why these attacks were possible in the first place. While walking through the different scenarios, the overall goal is to gain an adequate understanding of how the cryptographic algorithm works, as well as how the cryptanalysis is mounted. Simply "clicking" through the examples may not be enough to solve the tasks. To this end, it is recommended to walk through these steps before starting the actual tasks listed in section 2.

Even though this first part of the task sheet is ungraded, you may be asked questions about these scenarios during the oral consultation of this task, influencing the points you attain. Keep in mind that the practical tasks are mandatory to get approval for the final exam.

## 1.1 Caesar Cryptography

Inform yourself about the classical Caesar algorithm and possible methods of analysis. Classify the cryptanalysis according to known-plaintext and ciphertext-only attacks. Which fundamental fact is used to break the Caesar cipher?

**Hints:**

1. To de- and encrypt with the Caesar cipher using *JCrypTool*, choose the menu *Algorithms* $\rightarrow$ *Classic* $\rightarrow$ *Caesar*. In the online version of *CrypTool* you can find the Caesar cipher in the "Ciphers" menu.

2. To find additional resources you can use either the built in help function of *JCrypTool* or the tutorial provided by online tool. Keep in mind that external resources are allowed as well.

[1] Examples for this could be matrix calculus or basic statistical methods.

3. To calculate and visualize frequency use *Analysis* → *Entropy Analysis*.

## 1.2 Vigenère Cryptography

The main vulnerability in Caesar cipher is the monoalphabetic substitution. Thus, Vigenère extended the idea of the Caesar algorithm by using different shifts of the same alphabet during the encryption. Inform yourself about how the polyalphabetic substitution is being realized. How can it still be analyzed? – Inform yourself about the Kappa test[2] as well as the Kasiski test. Which one is used in the current implementation of *JCrypTool*?

**Hints:**

1. To de- and encrypt the Vigenère cipher with *JCrypTool* choose the menu *Algorithms* → *Classic* → *Vigenère*. In the online version of *CrypTool* you can find the Vigenère cipher in the "Ciphers" menu.

2. To find additional resources you can use either the built in help function of *JCrypTool* or the tutorial provided by online tool. Keep in mind that external resources are allowed as well.

3. Note, that there is a built-in Vigenère-breaker in *JCryptTool*, which may be helpful to understand the methodology behind the cryptanalysis of Vigenère ciphers.

## 1.3 Mono-alphabetic Substitution

The monoalphabetic substitution is another generalization of the Caesar algorithm. It substitutes a letter from the plaintext with an arbitrarily chosen one[3]. However, there is still the possibility for an easy cryptanalysis. Inform yourself about the monoalphabetic substitution, analysis techniques that are available and why these techniques work.

**Hints:**

1. To analyze the cipher, you have to choose the menu entry *Analysis* → *Substitution Analysis*.

2. If you know the language of the text to analyze, it is recommended to utilize this knowledge in the analysis dialog.

[2] alternative name: Friedman test
[3] Without repetitions of course, i.e., the substitution is a permutation of the alphabet.

## 1.4 XOR Cryptography

While working with binary data, the XOR encryption is a common and performant algorithm[4]. In *CrypTool*, we can apply an XOR encryption, e.g., to a compressed data or hex file. Think about the knowledge required to analyze an XOR cipher, and bring at least one example of such knowledge.

## 1.5 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is a method used to date, but the theory of elliptic curves dates way back. Already at the end of the 19th century, all mathematical foundations used in contemporary ECC were known. Here we will look at the properties of elliptic curves over the field of the real numbers, as well as finite fields. To get familiar with elliptic curves, you can start using the visualization provided by *JCrypTool*,

```
Visuals -> Elliptic Curve Calculations
```

Try to understand the basic properties of elliptic curves over the real numbers and how the point addition can be realized geometrically. Why is this not possible over a finite number field? Compare the graphs of the curve in the finite (discrete) field and the continuous case.

[4] The XOR cipher can be seen as Vigenère cipher over the alphabet $\{0, 1\}$.

# 2 Tasks

## Task 1: Ciphertext-only Analysis

Cryptanalysis based on knowledge of the ciphertext only, i.e., we do not have any plaintext, is the most difficult type of analysis. A method that can always be used is exhaustive key search, where every key is tried out. For this lab you were provided a file called `enc_2.hex`, which has been encrypted twice, using monoalphabetic substitution first and then the AES algorithm in CBC mode. The AES encryption used a weak key, where from the $128$ bit key only the first $16$ bits were chosen and the rest of the key was padded with zeros. Here, the initialization vector (IV) is the first block of the ciphertext.

### Task 1.1: Brute-force Weak AES

Due to the small key space, a brute-force attack on `enc_2.hex` seems to be a reasonable technique to break the outer AES encryption layer. Write a short prototype script to automate the attack. Thereby, think about a criterion to distinguish the right key from wrong ones. The decryption has to be computed and stored in `enc_1.hex` for further processing in the next step.

**Hints:**

1. Notice that there are already many implementations (libraries) available to apply AES encryption and decryption, e.g., *PyCryptodome* to interface with Python.

2. Take a look at Shannon entropy. How can it help to find out the correct key among all possible combinations?

### Task 1.2: The Hill-Climbing Method to Break Monoalphabetic Substitution

To break the inner monoalphabetic substitution, a naive brute-force attack, would require to test up to $26! \approx 4 \cdot 10^{26}$ possible permutations[5] of the alphabet[6]. Assuming each probe takes $1\,\mu$s, approximately $10^{13}$ years would be required. A more sophisticated method is required. The monoalphabetic substitution corresponds to a permutation of the alphabet, which can be represented by a substitution defined in a lookup table like the example depicted in table 1.

[5] What key size does this correspond to?
[6] Aassuming the alphabet `A-Z` only.

| reference alphabet | A | B | C | D | E | F | G | H | I | J | K | ⋯ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| key alphabet | Z | H | Q | P | L | A | G | I | Y | X | M | ⋯ |

Table 1: Exemplary substitution key for monoalphabetic substitution

From combinatorics, it is known that starting from an arbitrary permutation of a finite alphabet, we can define any key permutation by applying a finite number of transpositions[7], i.e., assuming, we start with the initial permutation $I$ and the substitution permutation (i.e., the key) used for monoalphabetic encryption is $K$, then there exist transpositions $T_1, T_2, \ldots, T_l$, such that

$$K = T_l \circ T_{l-1} \circ \cdots \circ T_1 \circ I\,.$$

In other words, finding out for given $I$ the transpositions $T_i$ is equivalent to finding the encryption key. That way the cryptanalysis becomes an iterative approach, where we successively test substitutions $I$, $T_1 \circ I$, $T_2 \circ T_1 \circ I$, …, starting from $I$ until we find $K$. To formalize this idea into an algorithm called hill climbing, three critical questions have to be answered:

1. How to choose the *initialization key* $I$, i.e., the permutation to start with?
2. How to derive from one key, e.g., $I$, the following keys $T_1 \circ I$, $T_2 \circ T_1 \circ I$, …?
3. How to measure the quality of each permutation, i.e., did we improve from $T_i \circ \cdots \circ T_1 \circ I$ to $T_{i+1} \circ T_i \circ \cdots \circ T_1 \circ I$?

Your task is to answer these questions and to implement, as well as to mount, the hill climbing approach to break the monoalphabetic cipher applied on `enc_1.hex`. For a successful completion of this subtask, you have to submit your work, your prototype implementation of the hill climbing attack using a suitable key initialization and the key derivation rule. Additionally, you have to submit the plaintext obtained, as well as the substitution key found by your attack. To guide you through this cryptanalysis, the following paragraphs provide help regarding the questions mentioned above.

**Initialization key.** The actual performance of the hill climbing attack will be heavily effected by the initial key from where the algorithm starts. To this end, it makes sense to not just select a random permutation. Instead, we should select a permutation which likely is close to the actual permutation, such that less transpositions are required. For the sake of this lab your task is to

[7] Naively, a transposition can be viewed as a single swap of two characters of the alphabet.

initialize the key by frequency analysis of the ciphertext. This process of initialization has to be implemented in an automated way. You may want to implement a function `init_key(cipher)` serving this purpose.

**Key derivation algorithm.** In the context of the monoalphabetic substitution, the key derivation function is rather simple. As has already been observed, the final key can be represented as a composition of single transpositions of the alphabet. To this end, it makes sense to derive the next key to probe from the given one by applying one single transposition, i.e., we randomly swap two letters in our lookup table.[8]

**Measuring the quality of the update.** The crucial part of hill climbing is the measurement function according to which we decide whether a key derived makes an improvement or not. For the sake of this lab, this part will be given. We provide a scoring function based on so called *n-gram* analysis. Implementations will be provided in C/C++ and Python. If you prefer a different language, you have to translate the given source code on your own. While this part is already implemented for you, your task is to understand the proposed/implemented technique. Questions may be asked during a examination.

```
"""
    ngram_score module to provide a mechanism to score
    texts based on a n-gram lookup table. The lookup
    table has to be created out-of-band.
"""

from math import log10


class NGramScore:
    """
        ngram_score class to calculate the n-gram score
        of a text based on a lookup table of the most
        common n-grams in a specific language.
    """

    def __init__(self, file_name, sep=' '):
        """
            Construct a new n-gram lookup table from the
            provided file. The assumed file structure is
            <ngram> <number-of-occurrences>, separated by
            a whitespace.
        """
        # read in raw file
        self.ngrams = {}
```

[8] E.g., if `A` was initially mapped to `T` and `C` to `F`, after the transposition, `A` should be map to `F` and `C` to `T` respectively.

```
        with open(file_name, 'r') as raw_file:
            for line in raw_file:
                ngram, count = line.split(sep)
                self.ngrams[ngram] = int(count)

        # store some internal parameters
        self.order = len(ngram)
        self.total_ngrams = sum(self.ngrams.values())

        # calculate probabilities
        for ix in self.ngrams:
            p = log10(float(self.ngrams[ix]) / self.total_ngrams)
            self.ngrams[ix] = p

        # define default probability for n-grams
        # not occurring in the given lookup table
        self.default_value = log10(0.01/self.total_ngrams)

    def score(self, input_text, normalize=False):
        """
            Calculate the score of the input text based on
            the lookup table. The option 'normalize' is used
            to normalize the score based on the text input
            length. While this is required to compare texts of
            different length, it has negative effects on scoring
            texts of same length! Only enable it if required!
        """
        score = 0
        text = input_text.upper()
        for idx in range(len(text)-self.order+1):
            current_ngram = text[idx:idx+self.order]
            if current_ngram in self.ngrams:
                score += self.ngrams[current_ngram]
            else:
                score += self.default_value

        if normalize:
            score = score / (len(text)-self.order+1)

        return score
```

Listing 1: n-gram scores

Notice that the source code listed above will be provided. There is no need to copy it from here.

## Task 2: Find the Right Algorithm

Given the files `sentence_1.txt` ($p_1$), `sentence_2.txt` ($p_2$) and `task02.cryp` ($c$), find an encryption scheme $(\text{Gen}, \text{Enc}, \text{Dec})$ and two keys $k_1$ and $k_2$ which fulfill the requirements

- $\text{Dec}_{k_1}(c) = p_1$ and
- $\text{Dec}_{k_2}(c) = p_2$,

i.e., the decryption of `task02.cryp` (ciphertext) results in the plaintext of `sentence_1.txt` if $k_1$ is being used and in `sentence_2.txt` if $k_2$ is being used. Submit both keys $k_1$ and $k_2$ as well as the decryption algorithm that was used. Hereby, it does not matter if you develop your own script or use a publicly available tool. However, in both cases make sure to guarantee access to the algorithm used otherwise the task can not be assessed.

## Task 3: Partial Known Plain-Text Analysis

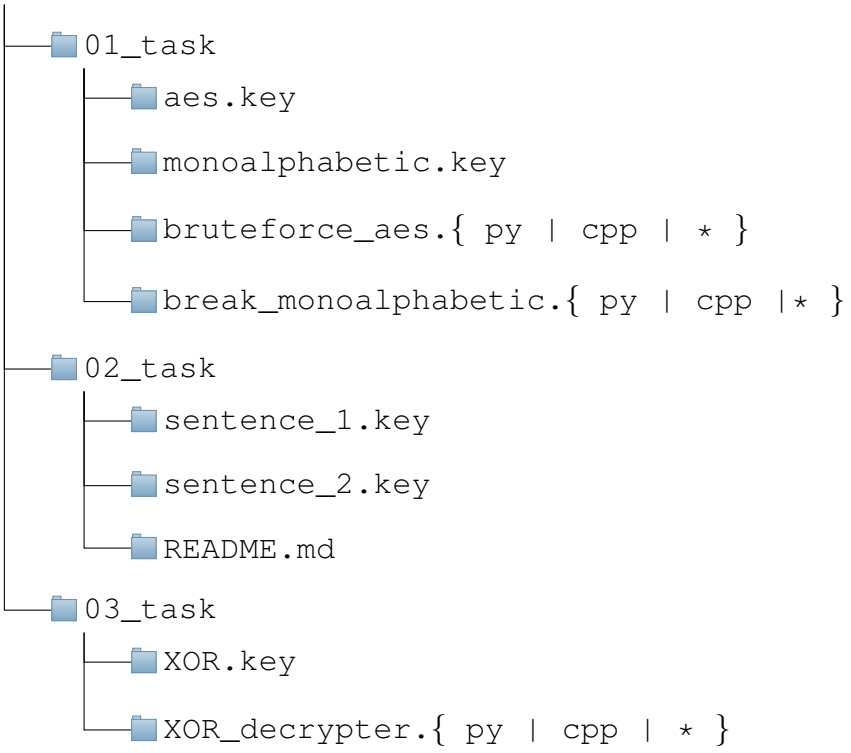
Along with this task sheet you have received the file `encrypted.zip.hex`. This zip file was encrypted with an 80-Bit-long XOR key. The compression was done with the default option using Version 3.20. Your task is to decrypt the given file. Submit the key used for the encryption, as well as a short description of how you have approached the cryptanalysis. You are allowed and encouraged to support your solution process by self-written scripts, which you must submit as well.

**Hints:**

1. Consider the encoding used in the `XOR`-algorithm if you think about using JCrypTool.
2. Sometimes it is better to reimplement a basic algorithm like `XOR` instead of converting a given input to satisfy existing programs requirements.
3. While analyzing/breaking the encryption, consider a known-plaintext attack. Which parts of the file are known for such an attack?
4. The task may not be solved in one iteration. Assuming a part of the key is known only, you can use `x00` for the missing bytes of the key. E.g., you are sure the last two bytes are `0xFF 0xFF`, then the key `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0xFF` may be used in a first iteration. Think about why this fact is important for your analysis.

## Submission

Submit **a single** compressed zip file named *firstname_lastname*.zip, where you replace *firstname* (*lastname*) with your first (last) name. The zip file contains solutions for the tasks and its contents must be named and structured as follows:




All files should contain the requested key/plaintext only. The encoding is assumed to be UTF-8 for text files. For binary files (like keys), store the data itself, not an encoding of it. The monoalphabetic key (text file) should be in the form RXLEDNOCISFQ..., corresponding to the letters ABCDEF.... The README from task 2 should contain the name of the algorithm used, e.g., *AES*, *MD5* and a short instruction on how to apply it to obtain the requested result. Additionally, source code can be submitted to support the explanations in the README.md.

# References


[1] CrypTool. *JCrypTool: Cryptography for everybody*. 2020. URL: https://www.cryptool.org/en/jct/ (visited on 10/24/2020) (cit. on p. 1).

[2] P. B. Esslinger and C. Team. *Learning and Experiencing Cryptography with CrypTool and SageMath*. 2018. URL: https://www.cryptool.org/assets/ctp/documents/CT-Book-en.pdf (visited on 10/04/2022) (cit. on p. 1).

[3] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. 2nd. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020. ISBN: 9781351133012. URL: https://books.google.tm/books?id=RsoOEAAAQBAJ (cit. on p. 1).

[4] *Letter Frequencies in the English Language*. URL: https://www3.nd.edu/~busiforc/handouts/cryptography/Letter%20Frequencies.html (visited on 11/01/2020).

[5] C. Online. *CrypTool: Cryptography for everybody*. 2020. URL: https://www.cryptool.org/en/cto/ (visited on 10/24/2020).

[6] W. Stallings. *Cryptography And Network Security : Principles And Practice*. 7th. Pearson Education, 2016. ISBN: 9788178089027 (cit. on p. 1).