# Classical Cryptography

András Zsolt Sütő, Azim Kazimli, Tobias Innleggen

Tuesday 18th March, 2025

**Abstract**

In this project, we investigate the effectiveness of using Haskell in implementing and breaking classical ciphers. Our report focuses on three historically significant ciphers: the Caesar cipher, the Vigenére cipher, and the Playfair cipher. We develop automated methods to recognize which cipher has been used on a given ciphertext by analyzing statistical features such as letter frequencies, $n$-gram patterns, entropy, and periodicity. Based on the identified cipher, we then apply appropriate cryptanalytic attacks: brute-force key search for Caesar and Playfair, and a combination of the Friedman test, Kasiski examination, and frequency analysis for Vigenére. We evaluate the success rate of these attacks and the performance of our Haskell implementations. **(Final results will be inserted here upon completion, summarizing the accuracy and efficiency achieved.)**

# Contents

# 1 Introduction

Classical ciphers like Caesar, Vigenère, and Playfair demonstrate the rich history and fundamental principles of cryptography[BR23], while also providing an opportunity to assess modern programming paradigms. In this project, we focus on applying Haskell's functional approach to implement and break these ciphers: we design tools for automatically recognizing the encryption method (via frequency analysis, entropy metrics, and periodicity checks) and then apply tailored attacks. Notably, Vigenère is attacked using the Friedman and Kasiski tests, and Playfair is targeted through guided search and heuristic improvements.

We seek to determine how effectively functional patterns—from list comprehensions to higher-order functions—can simplify cryptographic implementation and cryptanalysis. Specifically, we measure the performance of our tools and the success rates of brute-force or frequency-based attacks, highlighting where Haskell's immutable data structures and clear abstractions streamline even the more computationally heavy tasks. The upcoming sections describe each cipher, our recognition framework, and the methods used to recover plaintext.

# 2 Implementing classical cryptography ciphers

Some more theory, some more citing like [**?**].

## 2.1 Substitution cipher

- General explanation of the cipher
- Explanation of the code
  - Explain both of the key types
  - Explanation of the map construction
  - Explanation of the encryption and decryption functions

The substitution cipher is a way of encrypting where the units of the plaintext is replaced with cipher text. There are several diffrent kinds of substitution ciphers, but the one considered in this paper is the simple substitution cipher. this is a version of the substitution cipher that works on single letters. The two versions of the substitution cipher considered by this paper is a caesar cipher, taking an int as key value, and a version using a scrambled alphabet that uses a keyword as a key value. To accomodate diffrent cipher key formats in a uniform and maintable way the typeclass CipherInput was used. The CipherInput typeclass specifies that any type used as a cipher key must implement the create function. This allows for treating multiple cipher types unifomly, making use of haskell's polymorphism.

```
class CipherInput a where
    createCipherMap :: a -> M.Map Char Char
```

The caesar cipher is constructed by writing the alphabet twice, once as the normal alphabet and once with the letters shifted by a number of steps or even reversed. To create this mapping

a concrete implementation of the eralier defined CipherInput typeclass for the input type Int. This instance allows integers to serve as keys for the caesar cipher. The function generates a cipher map by spliting the regular alphabet at the rotation point, and then recombining the two segments into a shifted alphabet. The shifted alphabet is then ziped with the regular alphabet, foming the mapping between the plaintext letters and the cipher letters.

```
instance CipherInput Int where
    createCipherMap :: Int -> M.Map Char Char
    createCipherMap rotation =
        let baseAlphabet = ['A'..'Z']
            (firstPart, secondPart) = splitAt (rotation `mod` 26) baseAlphabet
            flippedAlphabet = secondPart ++ firstPart
        in M.fromList $ zip baseAlphabet flippedAlphabet
```

The keyword cipher is constructed by reordering the alphabet based on a given keyword, ensuring that each letter appears only once while preserving the remaining order of the alphabet. To create this mapping, a concrete implementation of the earlier defined CipherInput typeclass is provided for the input type String. This instance allows strings to serve as keys for the keyword cipher. The function generates a cipher map by extracting unique letters from the keyword, appending the remaining letters of the alphabet in order, and then zipping this modified sequence with the regular alphabet to form the mapping between plaintext and cipher letters.

```
instance CipherInput String where
    createCipherMap :: String -> M.Map Char Char
    createCipherMap keyword =
        let baseAlphabet = ['A'..'Z']
            uniqueKeyword = map toUpper . nub $ keyword
            remainingLetters = baseAlphabet \\ uniqueKeyword
            cipherAlphabet = uniqueKeyword ++ remainingLetters
        in M.fromList $ zip baseAlphabet cipherAlphabet
```

The `generateCipherTextFromMap` function transforms plaintext using a cipher mapping by filtering out non-alphabetic characters and replacing valid ones through `M.lookup`, which returns a `Maybe Char`. The use of `mapMaybe` ensures only successful lookups (`Just` values) are kept, while non-alphabetic or unmapped characters are discarded. The encrypted output is then formatted into five-character chunks omitting punctuation and spacesusing a recursive `chunk` function. This is done in order to disguise word boundaries from the plaintext as well as to help aviod transmission errors.

The `encryptSubstitution` function orchestrates this by generating the cipher map with `createCipherMap` and passing it to `generateCipherTextFromMap`. Haskell's `Maybe` type makes this implementation safe by forcing explicit handling of missing values, preventing common lookup errors found in imperative languages.

```
generateCipherTextFromMap :: String -> M.Map Char Char -> String
generateCipherTextFromMap plainTxt pairs =
    let filteredText = mapMaybe (\c -> if isAlpha c then M.lookup (toUpper c) pairs else
        Nothing) plainTxt
    in unwords (chunk 5 filteredText)

chunk :: Int -> String -> [String]
chunk _ [] = []
chunk n str = take n str : chunk n (drop n str)

encryptSubstitiution :: CipherInput p => p -> String -> String
encryptSubstitiution key plaintext =
    let cipherMap = createCipherMap key
    in generateCipherTextFromMap plaintext cipherMap
```

The decryption process of a substitution cipher is straightforward, as it involves inverting the cipher map used for encryption. This is exactly what happens in the functions below. The main decryption function, `decryptSubstitution`, first generates the key map using `createCipherMap`, then inverts it using `invertCipherMap`, which swaps the keys and values by converting the map to a list of pairs with `M.toList`, mapping each `(k, v)` pair to `(v, k)`, and reconstructing a new map with `M.fromList`. This functional approach is both concise and efficient because it avoids explicit loops and operates in a single pass over the map structure. The `decryptCipherText` function then applies the decryption map while preserving spaces and punctuation, using `map` to iterate over the text and `fromMaybe` to handle unmapped characters safely. This ensures that decryption is expressive and declarative, leveraging Haskell's built-in higher-order functions and persistent data structures to process transformations in a clear and optimized manner without unnecessary state mutations.

```haskell
-- Invert a cipher map to create a decryption map
invertCipherMap :: M.Map Char Char -> M.Map Char Char
invertCipherMap = M.fromList . map (\(k, v) -> (v, k)) . M.toList

-- Process ciphertext for decryption (preserve spaces and punctuation)
decryptCipherText :: String -> M.Map Char Char -> String
decryptCipherText cipherTxt decryptMap =
    map (\c -> if isAlpha c
               then fromMaybe c (M.lookup (toUpper c) decryptMap)
               else c) cipherTxt

-- Main decryption function that works with both String and Int keys
decryptSubstitution :: CipherInput p => p -> String -> String
decryptSubstitution key ciphertext =
    let cipherMap = createCipherMap key
        decryptMap = invertCipherMap cipherMap
    in decryptCipherText ciphertext decryptMap
```

## 2.2 Vignere cipher

Explain code of cipher implementation in codeblocks, unhide important things from below

```haskell
module Vignere where
```

## 2.3 Playfair cipher

Explain code of cipher implementation in codeblocks, unhide important things from below

```haskell
module Playfair where
```

# 3 Cipher recognition

This section is about cypher recognition.

# 4 Attacks

In this section we perform attacks on the ciphers implemented in 2.

```
module Main where

import Vignere
import Substitution
import Playfair
import PlayfairBreaker

main :: IO ()
main = do
  putStrLn "Hello!"
```

We can show the output with

```
Hello!
Output, statistics, timings, encoded decoded results whatever:
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

# 5 Tests

We now use the library QuickCheck to randomly generate input for the ciphers, and test whether they work correctly. We can also use QuickCheck to test attacks maybe?

```
module Tests where

import Vignere
import Substitution
import Playfair

import Test.Hspec
import Test.QuickCheck

-- Define the functions that are referenced in the tests
somenumbers :: [Int]
somenumbers = [1..10]

funnyfunction :: Int -> Int
funnyfunction n = n * 2

myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = myreverse xs ++ [x]

main :: IO ()
main = hspec $ do
  describe "Basics" $ do
    it "somenumbers should be the same as [1..10]" $
      somenumbers `shouldBe` [1..10]
    it "if n > - then funnyfunction n > 0" $
      property (\n -> n > 0 ==> funnyfunction n > 0)
    it "myreverse: using it twice gives back the same list" $
      property $ \str -> myreverse (myreverse str) == (str::String)
```

# 6 Conclusion

Here we concluded what has to be concluded.

# References

[BR23] R. Banoth and R. Regar. Classical and modern cryptography for beginners. *Springer*, 2023.