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 Vigenère 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 Vigenère. 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

The motivation behind this report is twofold. First, we aim to demonstrate that a functional language like Haskell can elegantly implement classical ciphers and their cryptanalysis. Second, we address a practical scenario: given an unknown piece of encrypted text, can we automatically determine the cipher used and subsequently crack it?

First we implement three classical ciphers:

- **Caesar cipher** is a simple monoalphabetic substitution cipher that shifts letters by a fixed offset; it was famously used by Julius Caesar and exemplifies the most basic substitution technique.
- **Vigenère cipher** extends this idea using a keyword to vary the shift across the message, making it a polyalphabetic cipher that was once considered "le chiffre indéchiffrable" (the indecipherable cipher) due to its resistance to simple frequency analysis.
- **Playfair cipher** is a digraph substitution cipher that encrypts pairs of letters using a 5x5 key matrix. This cypher mixes letter coordinates and thus complicates frequency patterns. [BR23]

Caesar is trivial to break with brute force, Vigenère requires analysis to uncover its repeating key, and Playfair demands strategic searching using a list of common words due to its larger key space. Both the ciphered text and the key are assumed to be in English.

We built a **cipher recognition** mechanism that inspects an unknown ciphertext and infers which cipher produced it. We base this recognition on statistical features of the ciphertext, such asletter frequency distributions, entropy and coincidence measures and periodicity, as explained in Section 3.

We measure the success rate of our attacks by testing each cipher-breaking method on sample ciphertexts and checking if the correct plaintext (or key) is recovered. We also profile the performance (time complexity and efficiency) of the Haskell solutions. Another aspect of our evaluation is the *ease of coding* and clarity of the solutions in Haskell. Throughout the project, we found that Haskell's expressive features (like higher-order functions, list comprehensions, and immutable data structures) allowed us to write cryptographic algorithms in a concise manner.

2 Implementing classical cryptography ciphers

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

2.1 Caesar and Keyword cipher

The substitution cipher is a way of encrypting where the units of the plaintext is replaced with cipher text. There are several different kinds of substitution ciphers, but the one considered in this section is the Caesar and Keyword cipher. These ciphers are versions of the substitution cipher that works on single letters. The caesar cipher takes a int as key value, and the keyword cipher generates a scrambled alphabet based on a keyword. To accommodate different cipher key formats in a uniform and maintainable way the typeclass CipherInput is 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 uniformly, making use of Haskell's polymorphism.

```haskell
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 earlier 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 splitting the regular alphabet at the rotation point, and then recombining the two segments into a shifted alphabet. The shifted alphabet is then zipped with the regular alphabet, forming the mapping between the plaintext letters and the cipher letters.

```haskell
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.

```haskell
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 spaces using a recursive `chunk` function. This is done in order to disguise word boundaries from the plaintext as well as to help avoid transmission errors.

The `encryptCaesarAndKeyword` 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.

```haskell
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)

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

The decryption process of both the caesar and keyword 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, `decryptCaesarAndKeyword`, 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
decryptCaesarAndKeyword :: CipherInput p => p -> String -> String
decryptCaesarAndKeyword key ciphertext =
    let cipherMap = createCipherMap key
        decryptMap = invertCipherMap cipherMap
    in decryptCipherText ciphertext decryptMap
```

2.2 Vigenère Cipher

```haskell
module Vigenere where
```

Unlike a simple monoalphabetic shift, the Vigenère cipher uses multiple shifts determined by a repeating keyword. In Haskell, this design naturally translates into a few elegant higher-order functions, list manipulations, and local bindings.

We define `vigenere`, which takes a higher order function `op`, a `key` and a plaintext, and returns the ciphertext. We rely on `cycle` to stream each character of the `key` infinitely, then pair it with each plaintext character. The recursive helper `enc` uses pattern matching to either shift a character or leave them as-is, depending on whether it is an alphabetical letter - this is an elegant way of replacing manual loop constructs.

```
vigenere :: (Int -> Int -> Int) -> String -> String -> String
vigenere op key = enc (cycle key)
  where
    enc _ [] = []
    enc [] _ = []
    enc (k:ks) (t:ts)
      | isAlpha t = shift k t : enc ks ts
      | otherwise = t : enc (k:ks) ts
```

Here, `shift` calculates the offset by converting the key character `k` to uppercase and subtracting `'A'`. We conditionally decide the base code point (uppercase or lowercase) for the plaintext letter and then wrap around with modular arithmetic. Because all values are passed immutably, we avoid side effects and can directly transform each character.

```
    shift k t =
      let base   = if isUpper t then ord 'A' else ord 'a'
          offset = ord (toUpper k) - ord 'A'
      in chr $ base + mod ((ord t - base) `op` offset) 26
```

By passing an operator (`+` for encryption or `-` for decryption) we generalized our cipher logic into one function, therefore encoding and decoding are defined as specializations of `vigenere` with the appropriate operator:

```
vigenereEncode :: String -> String -> String
vigenereEncode key text = vigenere (+) key text

vigenereDecode :: String -> String -> String
vigenereDecode key text = vigenere (-) key text
```

Since each transformation is defined as a pure function, data flows in one direction without side effects. The pattern matching in `enc` clarifies when to apply shifts (only on alphabetic characters) and when to keep a character intact.

2.3 Playfair cipher

In our Haskell implementation of the Playfair cipher, we express the algorithm in a functional style, highlighting Haskells strengths in handling complex string manipulations and modular code design.

The core components of our implementation include:

- **Table Construction:** We generate a 5 x 5 table from a user-supplied keyword by removing duplicate letters and appending the remaining letters of the alphabet (with 'J' treated as 'I'). This process is performed by the `createTable` function.
- **Text Preparation:** The plaintext is filtered to remove non-letter characters, converted to uppercase, and normalized (replacing 'J' with 'I') using the `prepareText` function.
- **Digram Generation:** The normalized text is split into pairs (digrams) with the

makeDigrams function. When duplicate letters occur within a pair, a filler (typically 'X') is inserted to ensure proper encryption.

- **Encryption and Decryption:** Depending on whether a pair of letters is in the same row, same column, or forms the corners of a rectangle in the table, we perform the appropriate substitution. This logic is implemented in the encryptDigram and decryptDigram functions, while the overall encryption and decryption are handled by the encrypt and decrypt functions.

We begin with the module declaration:

```
module Playfair where
```

Below are the key functions used in our implementation:

```
import Data.Char (toUpper, isAlpha)
import Data.List (nub)

-- The alphabet for the cipher (note that 'J' is omitted)
alphabet :: [Char]
alphabet = filter (/= 'J') ['A'..'Z']
```

Table Construction: The createTable function builds the 5 x 5 table by first cleaning the keyword and then appending any missing letters.

```
createTable :: String -> [Char]
createTable keyword =
  let cleaned   = map (\c -> if c == 'J' then 'I' else c)
                  $ filter isAlpha $ map toUpper keyword
      keyUnique = nub cleaned
  in keyUnique ++ filter (`notElem` keyUnique) alphabet
```

Text Preparation: The prepareText function filters and normalizes the input text.

```
prepareText :: String -> String
prepareText = map (\c -> if c == 'J' then 'I' else c)
            . filter isAlpha . map toUpper
```

Digram Generation: The makeDigrams function splits the text into pairs (digrams), inserting an 'X' when two consecutive letters are identical or when padding is required.

```
makeDigrams :: String -> [String]
makeDigrams []       = []
makeDigrams [x]      = [[x, 'X']]
makeDigrams (x:y:xs)
  | x == y    = [x, 'X'] : makeDigrams (y:xs)
  | otherwise = [x, y]   : makeDigrams xs
```

Finding Character Positions: The helper function chunksOf is used to break lists into chunks and findPosition locates a character within the 5×5 table.

```
chunksOf :: Int -> [a] -> [[a]]
chunksOf _ [] = []
chunksOf n xs = take n xs : chunksOf n (drop n xs)
```

```haskell
findPosition :: [[Char]] -> Char -> (Int, Int)
findPosition table c =
  let positions = [ (r, col)
                  | (r, row) <- zip [0..] table
                  , (col, ch) <- zip [0..] row
                  , ch == c ]
  in case positions of
       []    -> error ("Character not found: " ++ [c])
       (p:_) -> p
```

Encryption and Decryption: The functions `encryptDigram` and `decryptDigram` perform the letter substitutions based on the rules of the Playfair cipher.

```haskell
encryptDigram :: [[Char]] -> String -> String
encryptDigram table [a,b] =
  let (rowA, colA) = findPosition table a
      (rowB, colB) = findPosition table b
  in if rowA == rowB then
       -- Same row: shift right
       [ table !! rowA !! ((colA + 1) `mod` 5)
       , table !! rowB !! ((colB + 1) `mod` 5) ]
     else if colA == colB then
       -- Same column: shift down
       [ table !! ((rowA + 1) `mod` 5) !! colA
       , table !! ((rowB + 1) `mod` 5) !! colB ]
     else
       -- Rectangle: swap columns
       [ table !! rowA !! colB
       , table !! rowB !! colA ]
encryptDigram _ _ = error "Invalid digram length"

decryptDigram :: [[Char]] -> String -> String
decryptDigram table [a,b] =
  let (rowA, colA) = findPosition table a
      (rowB, colB) = findPosition table b
  in if rowA == rowB then
       -- Same row: shift left
       [ table !! rowA !! ((colA - 1) `mod` 5)
       , table !! rowB !! ((colB - 1) `mod` 5) ]
     else if colA == colB then
       -- Same column: shift up
       [ table !! ((rowA - 1) `mod` 5) !! colA
       , table !! ((rowB - 1) `mod` 5) !! colB ]
     else
       -- Rectangle: swap columns
       [ table !! rowA !! colB
       , table !! rowB !! colA ]
decryptDigram _ _ = error "Invalid digram length"
```

Finally, the overall `encrypt` and `decrypt` functions tie these components together:

```haskell
encrypt :: String -> String -> String
encrypt keyword text =
  let tableFlat = createTable keyword
      table     = chunksOf 5 tableFlat
      prepared  = prepareText text
      digrams   = makeDigrams prepared
  in concatMap (encryptDigram table) digrams

decrypt :: String -> String -> String
decrypt keyword text =
  let tableFlat = createTable keyword
      table     = chunksOf 5 tableFlat
      digrams   = chunksOf 2 (prepareText text)
  in concatMap (decryptDigram table) digrams
```

In summary, our Playfair cipher implementation in Haskell clearly demonstrates how breaking down a problem into small, composable functions can result in clean, modular, and maintainable code. By leveraging Haskell's strong type system and functional abstractions, we can model classical encryption algorithms in an elegant and concise manner.

3 Cipher recognition

This section is about cypher recognition.

4 Attacks

This section will have the implementation for the attacks on the ciphers.

4.1 Attack on Caesar and Keyword Cipher

As the key size for the Caesar is only one, we can use a brute force attack to break the cipher. We iterate over all possible keys and decrypt the ciphertext with the given key. We then return the key and the plaintext. This funciton returns a list of tuples, where the first element is the key and the second element is the plaintext.

```haskell
bruteforceSubstitution :: Int -> String -> [(Int, String)]
bruteforceSubstitution iteration ciphertext
    |iteration > 26 = []
    |otherwise =
        let plaintext = decryptCaesarAndKeyword iteration ciphertext
        in (iteration, plaintext) : bruteforceSubstitution (iteration + 1) ciphertext

bruteforceCaesar :: String -> [(Int, String)]
bruteforceCaesar = bruteforceSubstitution 1
```

4.2 Attack on Vigenère Cipher

4.3 Attack on Playfair Cipher

5 Running the program

This section (our main file) provides an interface for encoding and decoding messages using different cipher techniques, including Vigenère, Substitution, and Playfair ciphers. The program will prompt the user to enter a key and a message, then display the encoded and decoded results for each cipher method.

```haskell
module Main where

import Vigenere
import CaesarAndKeywordCipher
import Playfair

import VigenereBreaker
import PlayfairBreaker
import CaesarAndKeywordCipherBreaker


main :: IO ()
main = do
  putStrLn "\nEnter the key for the Vigenere cipher:"
  key <- getLine
  putStrLn "\nEnter the message to encode:"
  message <- getLine
  let substitutionEncoded = encryptCaesarAndKeyword key message
  let substitutionDecoded = decryptCaesarAndKeyword key substitutionEncoded

  let vignereEncoded = vigenereEncode key message
  let vignereDecoded = vigenereDecode key vignereEncoded

  let playfairEncoded = encryptPlayfair key message
  let playfairDecoded = decryptPlayfair key playfairEncoded

  putStrLn $ "\nSubstitution encoded : " ++ substitutionEncoded
  putStrLn $ "Substitution decoded : " ++ substitutionDecoded ++ "\n"

  putStrLn $ "Vignere encoded : " ++ vignereEncoded
  putStrLn $ "Vignere decoded : " ++ vignereDecoded ++ "\n"

  putStrLn $ "Playfair encoded : " ++ playfairEncoded
  putStrLn $ "Playfair decoded : " ++ playfairDecoded ++ "\n"
```

6 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)
```

7 Conclusion

Here we conclude what has to be concluded.

References

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