Classical Cryptography Report

Eva Imbens, Chiara Michelutti, Jan Przystał, Moritz Klopstock

Sunday 30th March, 2025

Abstract

This project presents a comprehensive exploration of classical cryptographic techniques using Haskell. It covers the design, implementation, and analysis of three historical ciphers: the Caesar cipher, the Vigenère cipher, and the One-Time Pad (OTP). The report details the development of haskell functions for encryption and decryption, employing frequency analysis and statistical methods to crack ciphers and expose vulnerabilities, such as the Many-Time Pad attack on OTP. By leveraging Haskell's strong type system, lazy evaluation, and performance optimizations, the project illustrates both the theoretical security and practical limitations of these cryptographic systems.

Contents

1 Introduction

In this project[1], we explore fundamental principles of classical cryptography by implementing three historical ciphers: Caesar, Vigenère, and One-Time Pad (OTP). Alongside implementing these encryption techniques, we also investigate their weaknesses and known cryptographic attacks. Specifically, we focus on:

- Frequency analysis on the Caesar cipher (and monoalphabetic substitutions in general), showing how statistical patterns in natural language can reveal encrypted messages.
- Kasiski examination and Friedmann tests on Vigenère, which exploits repeating patterns in the ciphertext to determine the key length and ultimately decrypt the message.
- Many-Time Pad (MTP) attacks on OTP, demonstrating how key reuse undermines its security.

Our approach includes encrypting and decrypting natural language messages, generating secure random keys, and implementing attacks to expose vulnerabilities in these ciphers. By doing so, we aim to highlight the contrast between theoretical security (as in OTP) and practical weaknesses (as in cases of poor key management and cipher design).

The following sections outline our project's objectives, methodologies, and experimental setup, detailing our implementation and security analysis in a Haskell-based environment.

1.1 Haskell Background

Haskell offers several advantages that make it a strong candidate for implementing classical cryptographic algorithms. These benefits include performance, memory safety, and a strong type system. They all contribute to writing secure and reliable (cryptographic) code.

Compiled and Optimized Execution Despite being a high-level functional language, Haskell can perform nearly as well as C. Research has shown that cryptographic functions implemented in Haskell can perform within the same order of magnitude as C, particularly when using compiler optimizations [Tev06]. This proves that it can handle computationally intensive cryptographic tasks, with the correct optimizations.

Lazy Evaluation Haskell uses lazy evaluation, which means that it only computes the values when they are needed. This can help to improve the efficiency by avoiding unnecessary calculations. For our many-time pad attack this would help to handle large ciphertexts efficiently by processing them only when required. This also helps to reduce the memory usage and computation overhead. Lazy Evaluation may however be a security concern. It can lead to timing attacks which may leak sensitive information [Bui13].

Memory Safety Unlike languages like C, Haskell automatically manages the memory, preventing vulnerabilities such as buffer overflows and pointer-related bugs. This automatic memory management ensures that cryptographic operations do not suffer from unintended memory corruption. This is important in cryptographic applications because small memory errors can lead to security flaws.

[1] `https://github.com/moritzklp/FP-ProjectTP`

Strong Type System Haskell has a strong type system that ensures that variables hold only correct kinds of values. This prevents unintended operations, such as treating a byte array as a string, misinterpreting cryptographic data formats. Programming on a type-level allows to encode security properties at compile time, which ensures that many classes of bugs are detected early.

Immutability Haskell's immutability ensures that once a value is assigned, it cannot be altered. This prevents unintended modifications of cryptographic data during the execution, which can be a problem in other languages where variables can be overwritten accidentally. Since cryptographic attacks and defenses often rely on maintaining strict data integrity, immutability provides a significant security advantage.

Arbitrary Precision Arithmetic Haskell provides arbitrary precision integers, which means that it allows computations with arbitrary large numbers which prevents the overflow issues that are common in many other languages. This is useful in cryptographic application where calculations may involve large integers, and unexpected overflows could lead to incorrect results.

Purity Haskell's pure functions make sure that the same input always produces the same output, which makes computations easier to test and debug. The lack of hidden side effects simplifies formal reasoning about cryptographic operations, which is useful in security audits and verification processes.

2 Frequency Analysis

Frequency analysis is a powerful technique used in cryptography to break ciphers, especially monoalphabetic substitution ciphers like the Caesar cipher. The idea of frequency analysis is to exploit the fact that in English (and most other natural languages), certain letters appear with predictable frequencies. For example, the letter 'e' is the most common letter in English text, followed by 't', 'a', 'o', etc. In the code we define an expected frequency distribution for English letters [2].

```
module Frequency where

import Data.Char
import Data.List
import Data.Ord

-- | Frequency table of English letters (source: https://en.wikipedia.org/wiki/
    Letter_frequency)
englishFrequencies :: [(Char, Double)]
englishFrequencies = [('a', 8.2), ('b', 1.5), ('c', 2.8), ('d', 4.3), ('e', 12.7), ('f',
    2.2), ('g', 2.0), ('h', 6.1),
 ('i', 7.0), ('j', 0.15), ('k', 0.77), ('l', 4.0), ('m', 2.4), ('n', 6.7), ('o', 7.5), ('p
    ', 1.9),
 ('q', 0.095), ('r', 6.0), ('s', 6.3), ('t', 9.1), ('u', 2.8), ('v', 0.98), ('w', 2.4), ('x
    ', 0.15),
 ('y', 2.0), ('z', 0.074)]
```

Next, we filter the text to keep only the alphabetic characters and convert them to lowercase. This ensures that the frequency analysis is case-insensitive and not affected by non-alphabetic characters. The `normalize` function removes any non-alphabetical characters and converts the

[2] `https://en.wikipedia.org/wiki/Letter_frequency`

remaining ones to lowercase. This is very important for analysing how often words appear in a text. The `charFrequency` function then calculates how often a specific character appears in the text (after normalization) in relation to the total number of letters. The `letterFrequencies` function builds a list of tuples for every letter from 'a' to 'z', each paired with its computed frequency in the text.

```
-- | Keep the lowercase (alphabetic) characters
normalize :: String -> String
normalize = map toLower . filter isAlpha

-- | Count the occurrences of a given character in a string
countOccurrences :: Char -> String -> Int
countOccurrences ch = length . filter (== ch)

-- | Calculate the frequency of a given character in the text
-- | 'fromIntegral' takes integral value (like 'Int' or 'Integer') and converts into more
    general numeric type ('Double' in our case)
charFrequency :: Char -> String -> Double
charFrequency c text =
  let norm = normalize text
      total = fromIntegral (length norm)
      count = fromIntegral (countOccurrences c norm)
  in count / total

-- | Get the frequency distribution for all the letters a-z
letterFrequencies :: String -> [(Char, Double)]
letterFrequencies text = [(c, charFrequency c text) | c <- ['a'..'z']]
```

In the next part, we compare the observed frequencies (from a decrypted candidate) with the expected letter frequencies in standard English. This comparison helps us evaluate how "English-like" the decrypted text is. The method we use is based on the *chi-squared statistic*, a technique often used in statistical hypothesis testing. The lower the chi-squared value, the closer the candidate is to what is typical in English [3].

```
-- | Calculate the chi-squared statistic between the observed and expected frequencies
-- | 'zip' pairs each observed frequency with its corresponding expected frequency - lets
    us compare the same letter's frequency from both lists
chiSquared :: [(Char, Double)] -> [(Char, Double)] -> Double
chiSquared observed expected =
  sum [((o - e) ** 2) / e | ((_, o), (_, e)) <- zip observed expected]
```

In `shiftChar` we handle the rotation of a single alphabetic character by a specified amount, wrapping around the alphabet if necessary. This function first checks whether the character is alphabetic; if it is, it calculates its position (0–25) by subtracting the base ASCII value for uppercase or lowercase letters. It then subtracts the shift amount, applies a modulo 26 to ensure the value wraps around properly, and adds the base back to convert it back to a valid ASCII character. Non-alphabetical characters are returned unchanged. The function `decryptWithShift` then applies `shiftChar` across the entire ciphertext, trying one candidate decryption by applying a given shift.

```
-- | Shift a character by a given amount
shiftChar :: Int -> Char -> Char
shiftChar shift c
  | not (isAlpha c) = c
  | otherwise = chr (base + mod (ord c - base - shift) 26)
  where
    base = if isUpper c then ord 'A' else ord 'a'


-- | Decrypt the text by applying a shift to each character
decryptWithShift :: Int -> String -> String
decryptWithShift shift = map (shiftChar shift)
```

[3] Idea adapted from http://practicalcryptography.com/cryptanalysis/text-characterisation/chi-squared-statistic/.

Now `findBestShift` tests all 26 possible shifts. For each shift, it decrypts the ciphertext, computes the frequency distribution, and calculates the chi-squared statistic based on the expected frequencies. The shift with the lowest score is assumed to be the correct one. This function is then used in `getCaesarDecryptedText` to decrypt the ciphertext using the best shift found.

```
-- | Find the best shift by comparing the frequency distributions (using chi-squared)
-- | 'minimumBy' finds the minimum element in a list based on a comparison function (in
    this case, the chi-squared value), return the shift with the lowest score
findBestShift :: String -> Int
findBestShift ciphertext = let scores = [ (shift, chiSquared (letterFrequencies (
    decryptWithShift shift ciphertext)) englishFrequencies) | shift <- [0..25] ]
  in fst (minimumBy (comparing snd) scores)

-- | Decrypt the Caesar cipher text using the frequency analysis (find the best shift and
    decrypt)
getCaesarDecryptedText :: String -> String
getCaesarDecryptedText ciphertext =
  let bestShift = findBestShift ciphertext
  in decryptWithShift bestShift ciphertext
```

3 Caesar Cipher

A Caesar cipher is a monoalphabetic substitution cipher, where each letter in the plaintext is replaced by a letter a fixed number of positions down the alphabet. For instance, with a shift of 3, the letter A is replaced by D, B by E, and so on. Despite its simplicity and vulnerability to brute force attacks, the Caesar cipher is a fundamental example in the study of cryptography.

The Caesar cipher module provides functionality for encryption, decryption, generating keys, and cracking Caesar ciphers.

Module Declaration and Imports

The module starts by declaring its name and importing necessary libraries:

- `System.IO` for input/output operations.
- `System.Random` for generating random numbers (used in key generation).
- `Data.Char` for character manipulations (e.g., converting characters to uppercase).
- Our module `Frequency` which provides a `findBestShift` function for cracking the cipher.

```
module CaesarCipher where

import System.IO
import System.Random (randomR, newStdGen)
import Data.Char (isAlpha, toUpper, toLower)
import Frequency (findBestShift)
```

Encryption and Decryption Functions

The `caesarEncrypt` function takes an integer shift and a string, applying the shift to each alphabetical character. It:

- Converts the character to uppercase.

- Computes its offset from the base character 'A'.
- Applies the shift modulo 26 to wrap around the alphabet.
- Converts the result back to a character.

Non-alphabetical characters are returned unchanged.
The `caesarDecrypt` function leverages `caesarEncrypt` by using the negative shift, effectively reversing the encryption process.

```
caesarEncrypt :: Int -> String -> String
caesarEncrypt shift = map (shiftChar shift)
  where
    shiftChar s c
      | isAlpha c = let upperC = toUpper c
                        base = fromEnum 'A'
                        offset = (fromEnum upperC - base + s) `mod` 26
                    in toEnum (base + offset)
      | otherwise = c

caesarDecrypt :: Int -> String -> String
caesarDecrypt shift = caesarEncrypt (-shift)
```

I/O Functions for Encryption and Decryption

The functions `encryptIO` and `decryptIO` handle file operations:

- They read the input text and key from files.
- Convert the key from a string to an integer.
- Convert the input text to uppercase before processing to ensure consistency.
- Write the encrypted or decrypted result back to an output file.

```
encryptIO :: String -> String -> String -> IO ()
encryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let shift = read keyContent :: Int
    writeFile output (caesarEncrypt shift (map toUpper inputContent))

decryptIO :: String -> String -> String -> IO ()
decryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let shift = read keyContent :: Int
    writeFile output (caesarDecrypt shift (map toUpper inputContent))
```

Key Generation Functions

The function `generateCaesarKeyIO` generates a random shift between 1 and 25 using a random number generator. The function `generateKeyFromPlaintextIO` then uses this key to create a key file for a given plaintext file, even though the plaintext is not directly used in generating the key.

```haskell
generateCaesarKeyIO :: IO String
generateCaesarKeyIO = do
    gen <- newStdGen
    let (shift, _) = randomR (1, 25 :: Int) gen
    return (show (shift :: Int))

generateKeyFromPlaintextIO :: String -> String -> IO ()
generateKeyFromPlaintextIO inputFile keyfile = do
    _ <- readFile inputFile
    key <- generateCaesarKeyIO
    writeFile keyfile key
```

Cracking the Cipher

The `crackIO` function attempts to decrypt a ciphertext without a known key by:

- Reading the ciphertext from a file.
- Using the `findBestShift` function (from the `Frequency` module) to estimate the shift based on frequency analysis.
- Decrypting the ciphertext with the guessed shift.
- Writing the result to an output file and displaying the guessed shift.

```haskell
crackIO :: String -> String -> IO ()
crackIO output inputFile = do
    ciphertext <- readFile inputFile
    let bestShift = findBestShift (map toLower ciphertext)
        decrypted = caesarDecrypt bestShift (map toUpper ciphertext)
    writeFile output decrypted
    putStrLn $ "Ciphertext: " ++ show ciphertext
    putStrLn $ "Guessed shift: " ++ show bestShift
    putStrLn $ "Guessed text: " ++ show decrypted
```

User Interface

The `caesar` function provides a simple command-line interface that promts the user to choose an action (generate, encrypt, decrypt, or crack) and then reads the necessary file names. It then calls the appropriate function based on the user's input.

```haskell
caesar :: IO ()
caesar = do
    hSetBuffering stdin LineBuffering
    putStrLn "[Caesar] Do you want to generate a key, encrypt, decrypt, or crack? (generate
        /encrypt/decrypt/crack)"
    method <- getLine
    case method of
        "generate" -> do
            putStrLn "In what file do you want to store the key? (e.g., key.txt)"
            keyFile <- getLine
            putStrLn "What plaintext do you want to generate a key for? (e.g., input.txt)"
            inputFile <- getLine
            generateKeyFromPlaintextIO inputFile keyFile
        "encrypt" -> do
            putStrLn "In what file do you want to store the ciphertext? (e.g., output.txt)"
            outputFile <- getLine
            putStrLn "What plaintext do you want to encrypt? (e.g., input.txt)"
            inputFile <- getLine
            putStrLn "What key do you want to use? (e.g., key.txt)"
            keyFile <- getLine
            encryptIO outputFile inputFile keyFile
```

```
        "decrypt" -> do
            putStrLn "In what file do you want to store the plaintext? (e.g., output.txt)"
            outputFile <- getLine
            putStrLn "What ciphertext do you want to decrypt? (e.g., input.txt)"
            inputFile <- getLine
            putStrLn "What key do you want to use? (e.g., key.txt)"
            keyFile <- getLine
            decryptIO outputFile inputFile keyFile
        "crack" -> do
            putStrLn "In what file do you want to store the decrypted text? (e.g., output.
                txt)"
            outputFile <- getLine
            putStrLn "What ciphertext do you want to crack? (e.g., input.txt)"
            inputFile <- getLine
            crackIO outputFile inputFile
        _ -> putStrLn "Invalid method. Please choose 'generate', 'encrypt', 'decrypt', or '
            crack'."
```

4 Vigenère Cipher

The Vigenère cipher is a polyalphabetic substitution cipher that encrypts alphabetic text by using a sequence of shifts based on the letters of a keyword. Each letter of the key determines a shift for the corresponding character in the plaintext. Unlike the simple Caesar cipher, which uses one fixed shift, the Vigenère cipher employs multiple shifts determined by the key, making it more resilient against basic frequency analysis. However, methods like the Kasiski examination and the Friedman test can still be used to analyze and break it.

This Haskell module implements the Vigenère cipher and includes functions for encryption, decryption, key generation, and cipher cracking.

Module Declaration and Imports

The module is named `VigenereCipher` and imports several libraries:

- `System.IO` for file I/O.
- List and character manipulation libraries for processing the text.
- Random number generation for key creation.
- Our `Frequency` module that provides a function for frequency analysis.

```
module VigenereCipher where

import System.IO
import Data.List (nub, sort, sortBy, groupBy, maximumBy, tails, minimumBy)
import Data.Char (ord, chr, isAlpha, toUpper, toLower)
import Data.Ord (comparing)
import Data.Function (on)
import System.Random (randomRs, newStdGen)
import qualified Data.Map as M
import Frequency (findBestShift)
```

Utility Functions and Constants

Constants such as `baseChar` and `endChar` define the range of uppercase letters. The function `shiftChar` applies a shift to a character, wrapping around if necessary.

```haskell
baseChar, endChar :: Char
baseChar = 'A'
endChar = 'Z'
baseVal, range :: Int
baseVal = ord baseChar
range = 26

shiftChar :: Int -> Char -> Char
shiftChar s c
  | isAlpha c = let upperC = toUpper c
                in chr $ baseVal + (ord upperC - baseVal + s) `mod` range
  | otherwise = c
```

Encryption and Decryption

The functions `vigenereEncrypt` and `vigenereDecrypt` perform the core operations:

- **Encryption:** The plaintext is first cleaned (non-alphabetic characters removed and all letters converted to uppercase). The encryption is achieved by cycling through the key and shifting each character by the value corresponding to the key letter.
- **Decryption:** This function reverses the encryption by applying the negative of the shift.

```haskell
vigenereEncrypt :: String -> String -> String
vigenereEncrypt key plaintext
  | null key  = error "Empty key not allowed"
  | otherwise = encrypt plaintext (cycle key)
  where
    encrypt [] _ = []
    encrypt (_:_) [] = error "Unexpected empty key during encryption"
    encrypt (c:cs) ks@(k:ks')
      | isAlpha c = shiftChar (ord (toUpper k) - baseVal) (toUpper c) : encrypt cs ks'
      | otherwise = c : encrypt cs ks

vigenereDecrypt :: String -> String -> String
vigenereDecrypt key ciphertext
  | null key  = error "Empty key not allowed"
  | otherwise = decrypt ciphertext (cycle key)
  where
    decrypt [] _ = []
    decrypt (_:_) [] = error "Unexpected empty key during decryption"
    decrypt (c:cs) ks@(k:ks')
      | isAlpha c = shiftChar (- (ord (toUpper k) - baseVal)) c : decrypt cs ks'
      | otherwise = c : decrypt cs ks
```

Finding Repeated Sequences and Calculating Distances

These functions implement the Kasiski examination [Kas63], a classical method for breaking polyalphabetic ciphers such as the Vigenère cipher by exploiting repeated sequences in the ciphertext. Kasiski's method has two main components:

- **Repeated Sequences:** When a specific sequence of letters appears more than once in the ciphertext, it is likely that the same portion of the key was used to encrypt different parts of the plaintext. Consequently, the distance (i.e., the number of characters) between these repeated sequences is often a multiple of the key length.
- **Distance Analysis:** By calculating the distances between repeated sequences and then analyzing the common factors among these distances, one can infer the possible length of

the key. Typically, the greatest common divisor (or the most common divisor) of these distances is a strong candidate for the key length.

The components of the Kasiski examination are implemented as follows:

- `findRepeatedSequences`: This function takes an integer `seqLen` representing the length of the sequence to search for. It:
  1. Iterates over the ciphertext to extract all substrings of length `seqLen` along with their starting positions.
  2. Sorts these pairs so that identical sequences are grouped together.
  3. Groups the sorted list by the sequence content using `groupBy`. Only groups with more than one occurrence (i.e., repeated sequences) are retained, and their starting indices are recorded.
- `calculateDistances`: Once the repeated sequences and their positions are known, this function computes all pairwise distances between the positions of each repeated sequence. These distances are used to detect common divisors, which in turn suggest likely key lengths.

```
findRepeatedSequences :: Int -> String -> [(String, [Int])]
findRepeatedSequences seqLen ctext =
  [ (seqText, positions)
  | group <- groupedSequences
  , let seqText = fst (head group)
  , let positions = map snd group
  , length positions > 1
  ]
  where
    allSequences = [(take seqLen $ drop i ctext, i) | i <- [0..length ctext - seqLen]]
    sorted = sortBy (compare `on` fst) allSequences
    groupedSequences = groupBy ((==) `on` fst) sorted

calculateDistances :: [(String, [Int])] -> [Int]
calculateDistances = concatMap (\(_, positions) ->
  [pos2 - pos1 | (pos1:rest) <- tails positions, pos2 <- rest])
```

Frequency Analysis and Key Length Estimation

The Kasiski examination provides a set of candidate key lengths based on the distances between repeated sequences. However, it is not always definitive. To enhance the accuracy of key length estimation, and to help with further decryption, we can apply statistical methods such as the Index of Coincidence (IC) and the Friedman test.

- **Index of Coincidence (IC):** The IC is a measure of the probability that two randomly selected letters from a text are the same. For a language like English, the IC is typically around 0.067. A lower IC indicates a more uniform distribution of letters, as seen in well-encrypted text, while a higher IC suggests a distribution similar to natural language.
- **Friedman Test [Fri22]:** This test calculates the IC for columns of ciphertext. When the ciphertext is divided based on a guessed key length, each column ideally represents text encrypted with the same Caesar shift. The average IC of these columns is then compared to the expected value for the language. A key length that yields an average IC close to the expected value is more likely to be correct.

- **Combining Methods:** By merging the insights from the Kasiski examination (which provides concrete candidate key lengths from repeated patterns) and the Friedman test (which statistically evaluates each candidate's plausibility), we can robustly guess the key length.

Implementation Details:

- `indexOfCoincidence`: This function calculates the IC for a given text. It works by:
  1. Counting the frequency of each character in the text.
  2. Using these counts to compute the probability that two letters picked at random are the same. This involves summing up $n(n-1)$ for each character count n, and dividing by the total number of pairs, $N(N-1)$, where N is the length of the text.
- `friedmanTest`: The Friedman test function divides the ciphertext into several columns, each corresponding to a fixed position in the repeating key cycle. It computes the IC for each column and returns the average IC across all columns. This average is used to gauge how well a particular guessed key length fits the statistical profile of natural language text.
- `guessKeyLength`: This function synthesizes the candidate key lengths from both the Kasiski examination and a range of possible lengths evaluated via the Friedman test:
  1. It first obtains candidates from the Kasiski method by analyzing repeated sequences and calculating their most common divisor.
  2. It then tests a range of key lengths (e.g., 1 through 20) using the Friedman test, computing a score based on the deviation of the average IC from the expected 0.067.
  3. The key length with the smallest deviation (i.e., the score closest to the expected value) is selected as the best guess.

The aformentioned functions are implemented as follows:

```
indexOfCoincidence :: String -> Double
indexOfCoincidence text =
  let counts = M.fromListWith (+) [(c, 1) | c <- text]
      total = fromIntegral (length text)
      combinations n = n * (n - 1)
  in if total < 2 then 0
     else (sum [combinations cnt | cnt <- M.elems counts]) / (total * (total - 1))

friedmanTest :: String -> Int -> Double
friedmanTest ctext klength =
  let columns = [everyNth klength i ctext | i <- [0..klength-1]]
      columnICs = map indexOfCoincidence columns
  in sum columnICs / fromIntegral klength
  where
    everyNth n k = map head . takeWhile (not.null) . iterate (drop n) . drop k

guessKeyLength :: String -> Int
guessKeyLength ctext =
  let kasiskiCandidates = take 3 $ kasiskiMethod ctext
      friedmanCandidates = [1..20]
      allCandidates = nub (kasiskiCandidates ++ friedmanCandidates)
      scores = [(kl, abs (friedmanTest ctext kl - 0.067)) | kl <- allCandidates]
  in fst $ minimumBy (comparing snd) scores
  where
    kasiskiMethod ct =
      let sequences = findRepeatedSequences 3 ct
          distances = calculateDistances sequences
      in if null distances then [3] else [mostCommonDivisor distances]
```

```haskell
mostCommonDivisor :: [Int] -> Int
mostCommonDivisor distances =
  fst $ maximumBy (comparing snd) divisorCounts
  where
    divisors n = [d | d <- [1..n], n `mod` d == 0]
    allDivisors = concatMap divisors distances
    sortedDivisors = sort allDivisors
    divisorCounts = [(d, count) | (d:ds) <- groupBy (==) sortedDivisors,
                     let count = length (d:ds)]
```

Key Guessing and Cracking the Cipher

To crack the Vigenère cipher:

- `guessShift` determines the most likely Caesar shift for a given column of ciphertext using frequency analysis.
- `crackVigenere` normalizes the ciphertext, estimates the key length (using the methods described above), and then reconstructs the key by determining the best shift for each column.

```haskell
guessShift :: String -> Int
guessShift column =
  findBestShift (map toLower column)  -- Frequency module expects lowercase

shiftToKey :: Int -> Char
shiftToKey shift = chr (ord 'A' + shift)

crackVigenere :: String -> String
crackVigenere ciphertext =
  let
      cleanText = map toUpper (filter isAlpha ciphertext)
      keyLen = guessKeyLength cleanText
      columns = [ everyNth keyLen i cleanText | i <- [0..keyLen-1] ]
      shifts = map (findBestShift . map toLower) columns
      key = map shiftToKey shifts
  in key
  where
    everyNth n k xs = map head $ takeWhile (not . null) $ iterate (drop n) (drop k xs)

normalize :: String -> String
normalize = map toLower . filter isAlpha
```

I/O Operations

The module includes functions for file input and output:

- `generateVigenereKeyIO` creates a random key of a specified length.
- `encryptIO` and `decryptIO` perform encryption and decryption on files using a key stored in another file.
- `crackIO` attempts to crack a ciphertext file by guessing the key and then decrypting the file.

```haskell
generateVigenereKeyIO :: Int -> IO String
generateVigenereKeyIO keyLength =
  take keyLength . randomRs (baseChar, endChar) <$> newStdGen

crackIO :: String -> String -> IO ()
crackIO output inputFile = do
    ciphertext <- readFile inputFile
    --putStrLn $ "Repeated sequences (length 3): " ++ show (findRepeatedSequences 3
        ciphertext)
    let guessedKey = crackVigenere ciphertext
        guessedKeyLen = length guessedKey
        plaintext = vigenereDecrypt guessedKey ciphertext
    writeFile output plaintext
    putStrLn $ "Ciphertext: " ++ ciphertext
    putStrLn $ "Guessed key: " ++ guessedKey
    putStrLn $ "Guessed key length: " ++ show guessedKeyLen
    putStrLn $ "Guessed plaintext: " ++ plaintext

encryptIO :: String -> String -> String -> IO ()
encryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    writeFile output (vigenereEncrypt keyContent inputContent)

decryptIO :: String -> String -> String -> IO ()
decryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    writeFile output (vigenereDecrypt keyContent inputContent)
```

Main Function

The `vig` function, works as entry point, just like the `caesar` and `otp` functions in the Caesar and OTP modules. It provides a simple command-line interface to generate keys, encrypt, decrypt, or crack Vigenère ciphers. Due to repetitive code, we have omitted the full implementation here.

5 Helper Functions

In this section we define functions that are essential for our implementation of the OTP cipher. The code below shows our Haskell implementation, which includes a function to perform the bitwise XOR operation on two byte strings. This functionality is a key component in both the encryption and decryption process and in demonstrating the Many-Time Pad attack.

The module `Pad` is defined and exports the `padString` function. It imports libraries from `Data.ByteString` and `Data.ByteString.Char8` for efficient handling of binary and character data, and `Data.Bits` for bitwise operations.

```haskell
module Pad (xorBytes, padString) where

import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as C
import Data.Bits (xor)
```

Th `xorBytes` function takes two `ByteString` arguments and applies a pair-wise XOR operation using `B.zipWith xor`. The result is packed back into a `ByteString` using `B.pack`. This operation is key in combining the plaintext with the key in an OTP cipher.

```haskell
xorBytes :: B.ByteString -> B.ByteString -> B.ByteString
xorBytes bs1 bs2 = B.pack (B.zipWith xor bs1 bs2)
```

The padString function takes two strings, converts them into ByteStrings, and then applies the xorBytes function. Finally, it converts the result back into a string. This process effectively "pads" one string with another using the XOR operation.

```
padString :: String -> String -> String
padString s1 s2 = C.unpack $ xorBytes (C.pack s1) (C.pack s2)
```

6 One-Time Pad

One-Time Pad (OTP) is a symmetric encryption based on the bitwise XOR (exclusive OR) operation. Given a plaintext message m and a secret key k, the ciphertext c is computed as:

$$c = m \oplus k$$

where $\oplus$ denotes the bitwise XOR operation. Decryption is achieved by performing the bitwise XOR on the ciphertext and the key, which results in the original plaintext.

$$m = c \oplus k$$

To perform the bitwise XOR operations, the secret key must have the same length as the plaintext or ciphertext. The security of OTP relies on the key being truly random, never reused, and kept secret from any adversary.

In this section we implement the functionality of the One-Time Pad encryption and decryption. This implementation provides a command-line interface that allows users to generate keys, encrypt plaintext messages, and decrypt ciphertext back to the original text. The design emphasizes clarity and modularity, leveraging the helper functions from the Pad module.

```
module OTP where

import System.IO
import System.Random (randomRs, newStdGen)
import Pad
import MTP
```

Encryption and Decryption

Encryption and decryption is handled by the encryptIO and decryptIO functions, respectively. These functions read the input and key files, perform the encryption or decryption, and write the result to the output file. The actual bitwise XOR operations are performed by the padString function from the Pad module.

```
-- | Encrypts a plaintext file using a key file and writes the result to an output file
encryptIO :: String -> String -> String -> IO ()
encryptIO outputFile inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let ciphertext = padString inputContent keyContent
    writeFile outputFile ciphertext

-- | Decrypts a ciphertext file using a key file and writes the result to an output file
decryptIO :: String -> String -> String -> IO ()
decryptIO outputFile inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let plaintext = padString inputContent keyContent
    writeFile outputFile plaintext
```

Key Generation

The key used to encrypt the plaintext must be as long as the plaintext itself. Therefore the key generation functions ensure that the key length matches the length of the input plaintext. The key is generated using random characters from the ASCII range `'!'` to `' '`.

```
-- | Generates a random key of a given length
generateRandomKeyIO :: Int -> IO String
generateRandomKeyIO n = take n . randomRs ('!', '~') <$> newStdGen

-- | Generates a key of the same length as the plaintext and writes it to a file
generateKeyFromPlaintextIO :: String -> String -> IO ()
generateKeyFromPlaintextIO inputFile keyFile = do
    inputContent <- readFile inputFile
    let n = length inputContent
    key <- generateRandomKeyIO n
    writeFile keyFile key
```

User Interaction

The `main` function provides a command-line interface for the user to select the desired operation: key generation, encryption, decryption, or a demonstration of the Multi-Time Pad (MTP) attack.

```
-- | Handles user interactions and operations selection
otp :: IO ()
otp = do
    hSetBuffering stdin LineBuffering
    putStrLn "Hello, do you want to generate a key, encrypt, decrypt or execute the Multi-
        Time Pad attack? (generate/encrypt/decrypt/mtp)"

    method <- getLine
    case method of
        "generate" -> handleGenerateKey
        "encrypt" -> handleEncrypt
        "decrypt" -> handleDecrypt
        "mtp" -> handleMTP
        _ -> putStrLn "Invalid method. Please choose 'generate', 'encrypt', 'decrypt', or '
            mtp'."
```

If the user selects the key generation option, the program prompts for the filenames of the key and plaintext files. The key is then generated and stored in the specified key file. The plaintext file is used to determine the length of the key.

```
-- | Handles key generation interaction
handleGenerateKey :: IO ()
handleGenerateKey = do
    putStrLn "In what file do you want to store the key? (e.g., key.txt)"
    keyFile <- getLine
    putStrLn "What plaintext do you want to generate a key for? (e.g., input.txt)"
    inputFile <- getLine
    generateKeyFromPlaintextIO inputFile keyFile
    putStrLn $ "Key generated and stored in " ++ keyFile
```

If the user selects the encryption or decryption option, the program prompts for the filenames of the input and output files, as well as the key file. The encryption or decryption operation is then performed using the specified files.

```
-- | Handles encryption interaction
handleEncrypt :: IO ()
handleEncrypt = do
    putStrLn "In what file do you want to store the ciphertext? (e.g., output.txt)"
    outputFile <- getLine
    putStrLn "What plaintext do you want to encrypt? (e.g., input.txt)"
    inputFile <- getLine
```

```haskell
    putStrLn "What key do you want to use? (e.g., key.txt)"
    keyFile <- getLine
    encryptIO outputFile inputFile keyFile
    putStrLn $ "Plaintext encrypted and stored in " ++ outputFile

-- | Handles decryption interaction
handleDecrypt :: IO ()
handleDecrypt = do
    putStrLn "In what file do you want to store the plaintext? (e.g., output.txt)"
    outputFile <- getLine
    putStrLn "What ciphertext do you want to decrypt? (e.g., input.txt)"
    inputFile <- getLine
    putStrLn "What key do you want to use? (e.g., key.txt)"
    keyFile <- getLine
    decryptIO outputFile inputFile keyFile
    putStrLn $ "Ciphertext decrypted and stored in " ++ outputFile
```

If the user selects the MTP attack option, the program loads the ciphertexts from a file and performs the Many-Time Pad attack. By default, the mtp.txt file contains the ciphertexts from the MTP challenge from the "Introduction to Modern Cryptography" course (https://homepages.cwi.nl/ schaffne/courses/crypto/2012/).

```haskell
-- | Handles Multi-Time Pad attack
handleMTP :: IO ()
handleMTP = do
    hexCiphertexts <- loadHexList "ciphertexts/mtp.txt"
    let ciphertexts = map hexToBytes hexCiphertexts
    mapM_ (breakIO ciphertexts) ciphertexts
    putStrLn "MTP attack completed"
```

7 Many-Time Pad

OTP is considered unbreakable when used correctly. The key must be completely random, must be as long as the message, and should be used only once. When these conditions are met, OTP provides perfect secrecy: even if an attacker intercepts the encrypted message, they cannot determine the original plaintext, no matter how much computing power they have. This is because the randomness of the key ensures that the encrypted message appears as complete gibberish. The probability of any plaintext message given the ciphertext is the same as the probability of any other plaintext message [Sha49].

However, OTP's security depends entirely on strict key management: if a key is reused, the ciphertexts are vulnerable the Many-Time Pad (MTP) attack, which allows attackers to break the encryption and recover parts of the secret key and the original messages [Lug23].

When the same key k is reused to encrypt two messages m_1, m_2, an attacker can exploit the XOR operation's properties to recover the XOR of the plaintexts.

Given two ciphertexts c_1, c_2 encrypted with the same key k:

$$c_1 = m_1 \oplus k$$
$$c_2 = m_2 \oplus k$$

an attacker can compute:

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2$$

.

This eliminates the key and reveals the XOR of plaintexts. While $m_1 \oplus m_2$ is not immediately readable, attackers can use frequency analysis and known plaintext patterns to recover both messages [Den83].

In our implemenation, we assume the encrypted texts are mostly English text, and we use the space character as a reference point to recover the key. We use an important property of ASCII characters: when a letter is XOR-ed with a space, it toggles the case. Therefore, if a space is XOR-ed with a letter, the result is another letter. If two ciphertexts are XOR-ed, and a letter is found in the result, it means that one of the plaintexts had a space in that position.

Many-Time Pad Implementation

```
module MTP where

import Data.Char (chr, ord)
import Data.Word (Word8)
import Data.Bits (xor)
import qualified Data.ByteString as BS
import qualified Data.ByteString.Char8 as C8
import Pad (xorBytes)
```

The ciphertexts are loaded from a file, and then the program iterates over each ciphertext, and performs the Many-Time Pad attack.

```
-- | Load a list of hex strings from a file, and perform the Many-Time Pad attack on each
    of them
main :: IO ()
main = do
    hexciphertexts <- loadHexList "ciphertexts/mtp.txt"
    let ciphertexts = map hexToBytes hexciphertexts
    mapM_ (breakIO ciphertexts) ciphertexts

-- | Function to read hex strings from a file and split them
loadHexList :: FilePath -> IO [String]
loadHexList filePath = do
    contents <- C8.readFile filePath
    return $ splitHexStrings (C8.filter (/= '\n') contents) -- Remove newlines if present

-- | Function to split a ByteString containing comma-separated hex values
splitHexStrings :: BS.ByteString -> [String]
splitHexStrings = map C8.unpack . C8.split ','

-- | Process and decrypt a single ciphertext using information from all ciphertexts
breakIO :: [BS.ByteString] -> BS.ByteString -> IO ()
breakIO allCiphertexts targetCiphertext = do
    -- | Make all ciphertexts the same length as the target ciphertext
    let normalizedCiphertexts = map (BS.take (BS.length targetCiphertext)) allCiphertexts

    -- | Find space positions for all ciphertexts
    let ciphertextsWithSpaceInfo = analyzeAllCiphertexts normalizedCiphertexts

    -- | Initialize empty key and update it with space information
    let emptyKey = replicate (fromIntegral $ BS.length targetCiphertext) Nothing
    let partialKey = createPartialKey emptyKey ciphertextsWithSpaceInfo

    -- | Decrypt the target ciphertext
    putStrLn $ breakWithPartialKey (BS.unpack targetCiphertext) partialKey
```

To perform the attack, the hex strings are converted to ByteStrings, so that the bitwise XOR operation can be used on them.

```
-- | Convert a hexadecimal string to a ByteString
hexToBytes :: String -> BS.ByteString
hexToBytes [] = BS.empty
hexToBytes (a:b:rest) = BS.cons (fromIntegral $ hexValue a * 16 + hexValue b) (hexToBytes
    rest)
    where
```

```
        hexValue :: Char -> Int
        hexValue c
            | c >= '0' && c <= '9' = ord c - ord '0'
            | c >= 'a' && c <= 'f' = ord c - ord 'a' + 10
            | c >= 'A' && c <= 'F' = ord c - ord 'A' + 10
            | otherwise = error $ "Invalid hex character: " ++ [c]
hexToBytes _ = error "Invalid hex string: ciphertext must have even number of characters
    hex characters"
```

The Many-Time Pad attack uses the fact that a letter XOR-ed with a space returns a letter. If two plaintexts are XOR-ed, and there is a letter in the result, one of the plaintext had a space in that position. The following functions are used to analyze the ciphertexts and find likely space positions in each of them.

```
-- | Check if a byte is likely to be a space in plaintext (1 for space locations, 0
    otherwise)
markAsSpace :: Word8 -> Int
markAsSpace byte | isLikelySpace byte = 1
                 | otherwise = 0
    where isLikelySpace b = (b >= 65 && b <= 90) || (b >= 97 && b <= 122) || b == 0

-- | Find likely space positions for two ciphertexts
detectSpacePositions :: BS.ByteString -> BS.ByteString -> [Int]
detectSpacePositions ciphertext1 ciphertext2 =
    map (markAsSpace . BS.index (xorBytes ciphertext1 ciphertext2)) [0 .. BS.length
        ciphertext1 - 1]

-- | Find likely space positions for all ciphertexts
-- | A position is likely a space if it produces a letter when XORed with most other
    ciphertexts
findLikelySpaces :: BS.ByteString -> [BS.ByteString] -> [Int]
findLikelySpaces target otherCiphertexts =
    let initialCounts = replicate (BS.length target) 0
        spaceIndicators = map (detectSpacePositions target) otherCiphertexts
        voteCounts = foldr (zipWith (+)) initialCounts spaceIndicators
        threshold = length otherCiphertexts - 2
    in map (\count -> if count > threshold then 1 else 0) voteCounts

-- | Perform the findLikelySpaces function on each ciphertext to find likely space
    positions in each of them
analyzeAllCiphertexts :: [BS.ByteString] -> [(BS.ByteString, [Int])]
analyzeAllCiphertexts ciphertexts =
    map (\cipher -> (cipher, findLikelySpaces cipher (filter (/= cipher) ciphertexts)))
        ciphertexts
```

Using the information about the locations of the spaces in each of the ciphertext, a partial key is created. Only bytes of the key in positions where one of the original plaintext had a space are recovered.

```
-- | Create the partial key from the space information
-- | Apply the updatePartialKey to the key for each ciphertext
createPartialKey :: [Maybe Word8] -> [(BS.ByteString, [Int])] -> [Maybe Word8]
createPartialKey emptyKey ciphertextsWithSpaces = xorWithSpace (foldl updatePartialKey
    emptyKey ciphertextsWithSpaces)
    where
        xorWithSpace = map (fmap (\b -> b `xor` fromIntegral (ord ' ')))

-- | Update the partial key with the space information from a single ciphertext
updatePartialKey ::  [Maybe Word8] -> (BS.ByteString, [Int]) -> [Maybe Word8]
updatePartialKey oldKey (cipherText, spaceIndicators) = zipWith updateKeyByte oldKey (zip (
    BS.unpack cipherText) spaceIndicators)
    where
        updateKeyByte currentByte (ciphertextByte, isSpace) = case (currentByte, isSpace)
            of
            (Nothing, 1) -> Just ciphertextByte
            (Just existing, 1) | existing == ciphertextByte -> Just existing
            _ -> currentByte
```

Using the partial key obtained using the space information, a ciphertext can be partially decrypted.

```
-- | Decrypt a ciphertext using the partial key
breakWithPartialKey :: [Word8] -> [Maybe Word8] -> String
breakWithPartialKey = zipWith decryptByte
  where
    decryptByte _ Nothing = '.'
    decryptByte b (Just keyByte) =
        if b == keyByte
        then ' '
        else chr $ fromIntegral (b `xor` keyByte)
```

8 Wrapping it up in an exectuable

The libraries with implementations for the OTP, Vigenere and Caesar ciphers are all imported into the main module. The main module is the entry point of the program. It provides a simple command-line interface for the user to choose between the three ciphers. The user is prompted to select a cipher method (OTP, Vigenere, or Caesar) and the corresponding function is called based on the user's input.

```
module Main where

import OTP
import VigenereCipher
import CaesarCipher

main :: IO ()
main = do
  putStrLn "Do you want to do One Time Pad (OTP), Vigenere Cipher or Caesar Cipher? (o/v/c)
    "
  method <- getLine
  case method of
    "o" -> do
      otp
    "v" -> do
      vig
    "c" -> do
      caesar
    _ -> putStrLn "Invalid method"
```

9 Conclusion

Our project successfully implemented encryption, decryption, and key generation capabilities for three classical ciphers: Caesar, Vigenère, and One-Time Pad. The Haskell implementation includes automated attacks for each cipher, enabling users to recover plaintext from ciphertexts through methods such as brute force attacks, frequency analysis, Kasiski examination, Friedman tests, and the Many-Time Pad attack.

The comprehensive user interface provides access to all implemented cryptographic operations and attack methods. Throughout this project, we observed a good alignment between Haskell's functional programming paradigm and the implementation of classical ciphers. The fact that classical ciphers are simply operations that transform inputs to outputs without maintaining state, pairs elegantly with Haskell's functional approach. Each cipher could be modeled as a series of pure functions, making the code both easy to understand and closely resembling the mathematical descriptions of the cryptographic algorithms.

9.1 Future Work

Our program has several opportunities for improvement. A priority would be implementing more sophisticated word prediction algorithms to complete partially recovered plaintexts, particularly when using the Many-Time Pad attack. This would increase the effectiveness of our cryptanalysis tools.

It would also be interesting to compare the performance of the Haskell implementations of the ciphers to other programming languages like C or Java. Cryptographic operations often need to be performed quickly in resource constrained systems so it would be interesting to see how well Haskell performs in such scenarios.

References

[Bui13] A. Buiras, P. ; Russo. Lazy programs leak secrets. NordSec, 2013.

[Den83] Dorothy E. Denning. The many-time pad: Theme and variations. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 25-27, 1983*, pages 23–32. IEEE Computer Society, 1983.

[Fri22] William Frederick Friedman. *The index of coincidence and its applications in cryptography.* Aegean Park Press, 1922.

[Kas63] Friedrich Wilhelm Kasiski. *Die Geheimschriften und die Dechiffrir-Kunst. Mit besonderer Berücksichtigung der deutschen und der französischen Sprache*. ES Mittler und sohn, 1863.

[Lug23] Thomas Lugrin. *One-Time Pad*, pages 3–6. Springer Nature Switzerland, Cham, 2023.

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[Tev06] JE Tevis. Secure programming using a functional paradigm. In *Proceedings of the Illinois State Academy of Science (ISAS) Conference. Citeseer*. Citeseer, 2006.