# Classical Cryptography Report

Eva Imbens, Chiara Michelutti, Jan Przystał, Moritz Klopstock

Tuesday 25th March, 2025

**Abstract**

## Contents

# 1 Introduction

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

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

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 Why Haskell

Haskell offers several advantages that make it a strong candidate for implementing cryptographic attacks, such as the Many-Time Pad attack. 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 [AS13]. 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.

**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. Research confirms that memory-safe programming languages significantly reduce security risks associated with manual memory management [DS21].

**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 stringmisinterpreting 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 [Hug89].

# 2 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
```

## 2.1 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
```

## 2.2 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
```

## 2.3 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
    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/).

```
-- | 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"
```

# 3 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 process and in demonstrating the Many-Time Pad attack.

- **Module and Imports:** 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.
- **The `xorBytes` Function:** This 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.
- **The `padString` Function:** This exported function takes two strings, converts them into `ByteString`s, 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, a fundamental step in many cryptographic techniques.
- **Demonstration:** The `main` function provides an example of how `xorBytes` can be applied to two hard-coded byte arrays. This sample code illustrates the practical use of the XOR operation.

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.

```
module Pad (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.

```
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 `ByteString`s, 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)
```

# 4 Wrapping it up in an exectuable

We will now use the library form Section **??** in a program.

```haskell
module Main where

import OTP
import VigenereCipher
import CaesarCipher

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

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[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 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.

## 5.1 Many-Time Pad Implementation

```haskell
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
```

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

```haskell
-- | 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"

-- | XOR two ByteStrings together
bytesXor :: BS.ByteString -> BS.ByteString -> BS.ByteString
bytesXor a b = BS.pack $ zipWith xor (BS.unpack a) (BS.unpack b)
```

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 (bytesXor 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
```

```haskell
-- | 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.

```haskell
-- | 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)
```

# 6 Caesar Cipher

A Caesar cipher is a classical 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 frequency analysis, the Caesar cipher is a fundamental example in the study of cryptography.

## 6.1 Overview of the Implementation

The module provides functionality for:

- Encrypting text with a given shift.
- Decrypting text using the inverse of the encryption shift.
- Generating a random key (shift) for the cipher.
- Cracking the cipher using frequency analysis.
- Handling input/output operations to interact with files.

The code is structured into several functions, each addressing a specific aspect of the cipher operations.

## 6.2 Code Explanation

The code is organized into the following sections:

### 6.2.1 Module 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)
```

### 6.2.2 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)
```

### 6.2.3 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))
```

### 6.2.4 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.

```
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
```

### 6.2.5 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.

```
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 $ "Guessed shift: " ++ show bestShift
```

### 6.2.6 User Interface

The `caesar` function provides a simple command-line interface that:

- Prompts the user to choose an action: generate a key, encrypt, decrypt, or crack.
- Reads the necessary file names from the user.
- Calls the appropriate function based on the user's input.

```
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'."
```

# 7 Vigenère Cipher

The Vigenère cipher is a polyalphabetic substitution cipher that encrypts alphabetic text by using a sequence of Caesar ciphers 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.

## 7.1 Theory Behind the Vigenère Cipher

In the Vigenère cipher:

- A key (a string of letters) is used to determine a series of shifts. Each letter in the key corresponds to a shift value (e.g., A $\rightarrow$ 0, B $\rightarrow$ 1, ..., Z $\rightarrow$ 25).
- The plaintext is first cleaned (usually by removing non-alphabetic characters and converting to a common case) and then encrypted by shifting each letter according to the corresponding key letter, repeating (cycling) the key as needed.
- Decryption involves reversing the process by shifting the ciphertext in the opposite direction.

The strength of the cipher lies in the length and randomness of the key. Short or repetitive keys are vulnerable to statistical attacks such as the Kasiski examination, which looks for repeated sequences in the ciphertext, and the Friedman test, which uses the index of coincidence to estimate the key length.

## 7.2 Overview of the Implementation

This Haskell module implements the Vigenère cipher and includes:

- Functions for encrypting and decrypting text using a given key.
- Utility functions for character manipulation and shifting.
- Techniques to analyze ciphertext, such as finding repeated sequences and calculating distances, which aid in estimating the key length.
- Frequency analysis to guess the key by examining individual columns of ciphertext.
- I/O operations to handle file input and output for encryption, decryption, key generation, and cipher cracking.

## 7.3 Code Explanation

Below is the complete code with detailed comments.

### 7.3.1 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)
```

### 7.3.2 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.

```
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
```

### 7.3.3 Encryption and Decryption

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

- **Encryption:** The plaintext is first cleaned (non-alphabetic characters removed and 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.

```
vigenereEncrypt :: String -> String -> String
vigenereEncrypt key plaintext =
  let cleaned = map toUpper $ filter isAlpha plaintext
  in zipWith (\k c -> shiftChar (ord k - baseVal) c) (cycle key) cleaned

vigenereDecrypt :: String -> String -> String
vigenereDecrypt key ciphertext =
  zipWith (\k c -> shiftChar (- (ord k - baseVal)) c) (cycle key) ciphertext
```

### 7.3.4 Finding Repeated Sequences and Calculating Distances

These functions implement the Kasiski examination, a classical method for breaking polyalphabetic ciphers such as the Vigenère cipher by exploiting repeated sequences in the ciphertext.

**Theory:**

- **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.

**Implementation Details:**

- `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])
```

### 7.3.5 Frequency Analysis and Key Length Estimation

These functions employ statistical measures to refine the key length estimation and further assist in breaking the cipher.

**Theory:**

- **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 0.0667. 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:** 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), one 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 0.0667.
  3. The key length with the smallest deviation (i.e., the score closest to the expected value) is selected as the best guess.

```
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 - 0.0667)) | 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]

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)]
```

### 7.3.6 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`:
  1. Normalizes the ciphertext.
  2. Estimates the key length using the methods described above.
  3. Splits the ciphertext into columns corresponding to each key character.
  4. Determines the best shift for each column.
  5. Reconstructs the key from these shifts.

```
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
```

### 7.3.7 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.

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

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

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)
```

### 7.3.8 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.

# References


[AS13] Johan Ankner and Josef Svenningsson. An EDSL Approach to High Performance Haskell Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, pages 1–12. ACM, 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.

[DS21] Dikchha Dwivedi and Hari Om Sharan. A review article on cryptography using functional programming: Haskell. *European Journal of Molecular and Clinical Medicine*, 8(2):2098–2110, 2021.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[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.