Classical Cryptography

Tobias Innleggen, András Zsolt Sütő, Azim Kazimli

Tuesday 18th March, 2025

Abstract

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

Contents

1 Introduction

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

First we implement three classical ciphers:

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

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

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

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

2 Implementing classical cryptography ciphers

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

2.1 Substitution cipher

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

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE InstanceSigs #-}
module Substitution where
```

2.2 Vigenère Cipher

```
module Vigenere where
```

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

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

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

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

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

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

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

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

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

2.3 Playfair cipher

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

```
module Playfair where
```

3 Cipher recognition

This section is about cypher recognition.

4 Attacks

4.1 Attack on Vigenère Cipher

4.2 Attack on Playfair Cipher

4.3 Attacks based on recognition

This section should be the main code part, where attacks are done based on recognized cipher.

```
module Main where

import Vigenere
import Substitution
import Playfair
import PlayfairBreaker

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

We can show the output with

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

5 Tests

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

```
module Tests where

import Vignere
import Substitution
import Playfair

import Test.Hspec
import Test.QuickCheck

main :: IO ()
main = hspec $ do
```

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

6 Conclusion

Here we conclude what has to be concluded.

References

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