Classical Cryptography Report

Eva Imbens, Chiara Michelutti, Jan Przystał, Moritz Klopstock

Sunday 23rd 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 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)
```

3 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 Main where

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

3.1 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
```

3.2 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
```

3.3 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
main :: IO ()
main = 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.

```haskell
-- | 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.

```haskell
-- | 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/).

```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"
```

4 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.

4.1 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
-- | 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)
```

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.