Classical Cryptography Report

Eva Imbens, Chiara Michelutti, Jan Przystał, Moritz Klopstock

Tuesday 18th March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 Introduction

Cryptography has long been a cornerstone of secure communication, and among its many techniques, the One-Time Pad (OTP) cipher holds a special place. When implemented correctly, the OTP offers perfect secrecy by using a truly random key that is as long as the message itself. However, this ideal is contingent on using each key only once. When a key is reused—a situation known as the Many-Time Pad—the cipher's security can be compromised, revealing vulnerabilities that attackers may exploit.

In this project, we look into the classical principles of cryptography by implementing the OTP cipher and investigating its limitations. Our approach not only involves the encryption and decryption of natural language messages but also includes the generation of secure random keys and a demonstration of the Many-Time Pad attack. By exposing the weaknesses that arise from improper key management, we aim to provide a comprehensive understanding of the balance between theoretical security and practical implementation challenges.

Haskell has been chosen as the programming language for this project due to the course dependencies but also its pure functional nature, strong static type system, and emphasis on code safety. These features are particularly advantageous in the realm of cryptographic operations, where the avoidance of unintended side effects is critical. Through this implementation, we explore the viability and effectiveness of Haskell in developing secure cryptographic solutions, leveraging its capabilities to create a reliable and efficient cipher system.

The following sections outline our project's objectives, methodologies, and the experimental setup used to analyze the performance and security of our Haskell-based OTP implementation.

2 Helper Functions

In this section we define helper functions that are essential for our implementation of the OTP cipher. The code below shows our Haskell implementation, which includes a function to perform an 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.

```haskell
module Pad (padString) where

-- :set -package bytestring
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as C
--import Data.Word (Word8)
import Data.Bits (xor)

-- xor :: Bits a => a -> a -> a
xorBytes :: B.ByteString -> B.ByteString -> B.ByteString
xorBytes bs1 bs2 = B.pack (B.zipWith xor bs1 bs2)

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

-- main :: IO ()
-- main = do
--     let bytes1 = B.pack [0x01, 0x02, 0x03, 0x04]  -- First byte array
--     let bytes2 = B.pack [0xFF, 0x00, 0xFF, 0x00]  -- Second byte array
--     let result = xorBytes bytes1 bytes2           -- Perform XOR operation
--     print result  -- Output: "\254\STX\252\DLE"
```

3 One Time Pad

In this section we implement the main functionality of the One-Time Pad cipher. 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.

The **One-Time Pad (OTP)** is a theoretically perfect encryption scheme when used properly. However, its security guarantees completely break down when the same key is reused multiple times, creating what's known as a **Multi-Time Pad** or **Two-Time Pad** vulnerability [Lug23].

The OTP operates using:

- Plaintext message m
- Secret key k (random bits, length $= |m|$)
- Ciphertext $c = m \oplus k$ (where $\oplus$ denotes XOR)

Decryption is performed as:

$$m = c \oplus k$$

Key aspects of the implementation include:

- **Key Generation:** Two approaches are provided. One function generates a random key of a specified length, while another automatically generates a key that exactly matches the length of the input plaintext.
- **Encryption and Decryption:** Both operations use the same `padString` function to perform a bitwise XOR between the message and the key. This ensures that encryption and decryption are symmetric, as applying the XOR operation twice with the same key returns the original message.

- **Command-Line Interface:** The `main` function parses command-line arguments to determine whether to generate a key, encrypt a message, or decrypt a message. Clear usage instructions are provided for cases when the arguments do not match any of the expected patterns.
- **Random Key Generation:** By using Haskell's random number generator, the program creates a key consisting of uppercase letters, ensuring that each key is unpredictable and secure when used only once.

The following code block contains the complete implementation of the OTP functionality:

```haskell
module Main where

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

-- Should read plaintext from input file and key from key file
-- Should write ciphertext to output file
encryptIO :: String -> String -> String -> IO ()
encryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let ciphertext = padString inputContent keyContent
    writeFile output ciphertext

-- Should read ciphertext from input file and key from key file
-- Should write plaintext to output file
decryptIO :: String -> String -> String -> IO ()
decryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let plaintext = padString inputContent keyContent
    writeFile output plaintext

-- Generate a random key of a given length (inside IO) - to not provide a seed manually
generateRandomKeyIO :: Int -> IO String
generateRandomKeyIO n = take n . randomRs ('!', '~') <$> newStdGen

-- Given a plaintext, should generate a key of the same length
generateKeyFromPlaintextIO :: String -> String -> IO ()
generateKeyFromPlaintextIO inputFile keyfile = do
    inputContent <- readFile inputFile
    let n = length inputContent
    key <- generateRandomKeyIO n
    writeFile keyfile key

main :: IO ()
main = do
    hSetBuffering stdin LineBuffering -- So we can use backspace while running this using
        ghci
    putStrLn "Hello, do you want to generate a key, encrypt, or decrypt? (generate/encrypt/
        decrypt)"
    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
        _ -> putStrLn "Invalid method. Please choose 'generate', 'encrypt', or 'decrypt'."
```

4 Multi Time Pad

When the same key k is reused for multiple messages m_1, m_2:

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

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 $m_1 \oplus m_2$ isn't immediately readable, attackers can use frequency analysis and known plaintext patterns to recover both messages [Den83].

For example:

Consider two messages encrypted with the same key:

$$m_1 = \texttt{"HelloWorld"}$$
$$m_2 = \texttt{"SecureData"}$$
$$k = \texttt{0x5f1d3a...} \quad \text{(random bytes)}$$

The attacker observes:

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

By computing $c_1 \oplus c_2$, the attacker gets $m_1 \oplus m_2$. If they guess part of m_1 (e.g., common phrase "Hello"), they can recover the corresponding part of m_2:

$$\text{Guessed } m_1 \oplus (m_1 \oplus m_2) = m_2$$

```
module MTP where

import Data.Bits (xor)
import Data.Char (chr, ord)
import Data.List (transpose, maximumBy, isSuffixOf)
import Data.Ord (comparing)
import System.IO
import System.Directory (listDirectory)

mtp :: IO ()
```

```haskell
mtp = do
    hSetBuffering stdin LineBuffering -- So we can use backspace while running this using
        ghci
    putStrLn "Please enter the folder name containing ciphertext files:"
    folder <- getLine
    files <- listDirectory folder
    let txtFiles = [folder ++ "/" ++ f | f <- files, ".txt" `isSuffixOf` f]
    if null txtFiles
        then putStrLn "No .txt files found in the specified folder."
        else do
            ciphertexts <- mapM readFile txtFiles
            let minLen = minimum (map length ciphertexts)
                truncated = map (take minLen) ciphertexts
                cipherAscii = map (map ord) truncated
                columns = transpose cipherAscii
                key = map guessKeyByte columns
                decrypted = map (\ct -> zipWith xor ct key) cipherAscii
            putStrLn "Recovered plaintexts:"
            mapM_ (putStrLn . map chr) decrypted

guessKeyByte :: [Int] -> Int
guessKeyByte column =
    let possibleCs = filter (\c -> c >= 97 && c <= 122) column  -- a-z
        candidates :: [(Int, Int)]  -- Explicit type annotation
        candidates = [ (keyCandidate, score)
                     | c <- possibleCs
                     , let keyCandidate = c `xor` 32  -- 32 is ASCII for space
                     , keyCandidate >= 65 && keyCandidate <= 90  -- Key must be A-Z
                     , let decrypted = map (`xor` keyCandidate) column
                     , let score = sum (map scoreChar decrypted) :: Int  -- Explicit type
                         annotation
                     ]
        scoreChar :: Int -> Int  -- Explicit type for clarity
        scoreChar x
            | x == 32                     = 10  -- Space
            | x >= 97 && x <= 122         = 5   -- Lowercase
            | x >= 65 && x <= 90          = 5   -- Uppercase
            | x >= 48 && x <= 57          = 3   -- Digits
            | x `elem` punctuation        = 2   -- Punctuation
            | otherwise                   = 0
        punctuation = concat [ [33..47], [58..64], [91..96], [123..126] ]
    in if null candidates
        then 0  -- Default to 0 if no valid candidates
        else fst $ maximumBy (comparing snd) candidates
```

References

[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.

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