Classical Cryptography Report

Eva Imbens, Chiara Michelutti, Jan Przystał, Moritz Klopstock

Tuesday 18th March, 2025

Abstract

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.

1.1 Why Haskell

Haskell offers several advantages that make it a strong candidate fior 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 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.

```
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)

-- xor first bytestring with all the others
xorCipherTexts :: [B.ByteString] -> [B.ByteString]
xorCipherTexts (c:cx) = map (xorBytes c) cx


-- 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

One Time Pad (OTP) is an encryption technique which relies on having a unique symmetric key for each encrypted message. We use a simple XOR operation on the plaintext and the key as the encyption function. In OTP the message must be of equal length or shorter than the key, or else it could not be completely encrypted or the key would have to loop. This technique is proven to be unbreakable - the cipher is completely resistant to attacks.

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.

3.1 Code

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

```
module Main where

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

-- 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, decrypt or execute the Multi-
    Time Pad attack? (generate/encrypt/decrypt/mtp)"
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
    "mtp" -> do
         hexciphertexts <- loadHexList "ciphertexts/mtp.txt"
         let ciphertexts = map hexToBytes hexciphertexts
         mapM_ (breakIO ciphertexts) ciphertexts
    _ -> 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:

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

5 Many Time Pad Attack

As previously stated, the One Time Pad is secure and resistant to attacks. However, in case where not all the keys are unique, so a key is reused, it is possible to break the encyption. The method to do it is called a Many Time Pad Attack. The cipher becomes vulnerable because if two plaintexts have been ecypted with the same key, performing the XOR operation on the cipher-texts will have the same result as doing it with the original plaintexts. What it means is that we remove the secret key from the equation completely. This is shown in the equation: [insert equation]

```
module MTP where

import System.Environment (getArgs)
import Data.Char (chr, ord, isAscii)
import Data.Word (Word8)
import Data.Bits (xor)
import Data.List (intercalate, foldl')
import qualified Data.ByteString as BS
import qualified Data.ByteString.Char8 as C8
import Text.Printf (printf)
```

```

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

-- 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

main :: IO ()
main = do
    hexciphertexts <- loadHexList "ciphertexts/mtp.txt"
    let ciphertexts = map hexToBytes hexciphertexts
    mapM_ (breakIO ciphertexts) ciphertexts

-- 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



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

-- 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"



-- 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



-- 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



-- Decrypt a ciphertext using the partial key
breakWithPartialKey :: [Word8] -> [Maybe Word8] -> String
breakWithPartialKey = zipWith decryptByte
  where
    decryptByte b 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.