Hacettepe University

Computer Engineering Department

BBM465 Information Security Laboratory - 2024 Fall

Homework Assignment 1

October 23, 2024

Group ID: 28
Students name:
Mehmet Yiğit
Mehmet Oğuz Kocadere

Students Number:
b2210356159
b2210356021

1 Problem Definition

The goal is to understand basic cryptographic techniques through practical application. The assignment consists of two main parts:

1. **Cipher Implementation**: We are required to implement encryption and decryption functions for three classic ciphers:
   - **Caesar Cipher**: Involves shifting letters in the plaintext by a fixed number of positions.
   - **Affine Cipher**: Uses a mathematical formula for encryption, requiring two keys to encode and decode messages.
   - **Monoalphabetic Substitution Cipher**: Each letter in the plaintext is replaced by a corresponding letter from a fixed substitution alphabet.
2. **Cryptanalysis**: We must develop methods to break the ciphers implemented in Part 1 by using techniques such as brute force and frequency analysis. The decryption should be validated against a dictionary to ensure accurate results.

The assignment is designed to be completed in Python and emphasizes understanding basic cryptographic methods and developing problem-solving skills related to encryption and decryption.

2 Encryption Implementations

2.1 Caesar Cipher Encryption

```
#You can run this code from the console with this command
python ciphers.py caesar --plain_text_file_path --mode --shift_amount
```

This function encrypts a text using the Caesar cipher technique, where each letter in the text is shifted by a specified number of positions (the shift). Non-letter characters remain unchanged, and both uppercase and lowercase letters are treated accordingly. This code just can be run with "-e" (encryption) mode. This command call encrypt_caesar(plaintext, shift) function. And it returns a ciphered text that is calculated with shift amount.

2.2 Affine Cipher Encryption

```
#You can run this code from the console with this command
python ciphers.py affine --plain_text_file_path --mode -a  -b
```

The `encrypt_affine(plaintext, a, b)` function encrypts a given plain text using the Affine cipher by applying a linear transformation to each alphabetic character, based on the multiplicative key `a` and additive key `b`, while leaving non-alphabetic characters unchanged. It returns the ciphered text.

2.3 Mono-alphabetic Substitution Cipher Encryption

```
#You can run this code from the console with this command
python ciphers.py mono --plain_text_file_path --mode --key
```

The `encrypt_mono` function encrypts a given plain text using a mono alphabetic cipher by substituting each letter with a corresponding letter from a key, preserving case, and leaving non-alphabetic characters unchanged. It returns the ciphered text.

2.4 Cipher.py Main

This code defines a `main()` function that parses command-line arguments to determine which encryption technique (Caesar, Affine, or Monoalphabetic) to use and whether to encrypt or decrypt a given text file. It reads the input file, applies the specified cipher using the provided parameters, and outputs the result.

3 Decryption Implementations

3.1 Caesar Cipher Decryption

```
#You can run this code from the console with this command
python break.py caesar --ciphered_text_file_path
```

The `decrypt_caesar` function attempts to decrypt a given cipher text using the Caesar cipher by checking potential shifts (from 0 to 25) until it finds a valid word that exists in a provided dictionary. If a valid shift is found, it decrypts the entire cipher text with that shift and returns the decrypted text; otherwise, it returns `None` if no valid decryption is identified. It is a brute force approached method. It validate the final decrypted text with the help of process_text() function. It splits the decrypted text and it controls each word exists in the dictionary.txt or not exist. If it is not exist, it place "[REDUCTAD]" word in place of this word and if exist it gives the verified decrypted text. It also use the encrypt_caesar() function for decryption with negative shift amount.

3.2 Affine Cipher Decryption

```
#You can run this code from the console with this command
python break.py affine --ciphered_text_file_path
```

This code implements the decryption of text encrypted with the Affine cipher, utilizing brute force to find the correct keys. The `mod_inverse` function calculates the modular inverse of a given integer "a" with respect to a modulus "m", which is essential for reversing the encryption process. The `coprime_with_26` function checks if a number is coprime with 26, ensuring that the multiplicative key "a" has a valid inverse needed for decryption.

The `decrypt_affine_single_word` function attempts to decrypt the cipher text by iterating over potential keys "a" and "b", checking if the resulting words exist in a dictionary of

valid words. Finally, the `decrypt_affine_with_keys` function performs the actual decryption using the identified keys, returning the decrypted text if successful. Overall, this code systematically addresses the problem of decrypting Affine cipher text through mathematical properties and brute-force search.

3.3 Mono-alphabetic Substitution Cipher Decryption With Frequency Analysis and Brute Force

```
#You can run this code from the console with this command
python break.py mono --ciphered_text_file_path
```

General Overview

Purpose: The main objective of the program is to decrypt a cipher text using a mono-alphabetic substitution cipher, which replaces each letter in the plain text with a different letter.

Components: Reading Input: The program reads cipher text from a specified file and utilizes a dictionary for word validation. Frequency Analysis: It calculates the frequency of letter combinations (n-grams) from the dictionary to assist in decryption. Key Discovery: The program tries various key combinations to find the one that maximizes the "fitness" score based on valid words found in the decrypted text. Decryption Process: Once the best key is found, it decrypts the cipher text and outputs the resulting plain text.

Functions: Loading and Processing the Dictionary: The code loads a dictionary file to keep track of valid words. Extracting Words: It extracts 4-letter combinations (4-grams) from the dictionary to use in frequency analysis. Fitness Score Calculation: A scoring system evaluates how well the decrypted text matches the dictionary. Key Swapping: The code includes functionality to swap characters in the key to explore potential better keys.

Frequency Analysis in Detail

Frequency analysis is a method that exploits the statistical properties of the language used in the plain text to aid in breaking ciphers. Here's how it works in the context of the provided code:

Extracting 4-Grams: The program uses the extract_words function to generate a list of counts for all possible 4-grams (sequences of four letters) found in the words of the loaded dictionary. The function loops through each word in the dictionary and builds a frequency count for every 4-gram.

Calculating and Normalizing Frequencies: The function calculate_and_normalize_words normalizes the frequency counts to make them comparable. It applies logarithmic transformation to ensure that lower frequency counts are not overly favored compared to higher counts. This normalization helps in the fitness scoring of decrypted texts.

Decrypting Using Frequency Analysis: The frequency_analysis function performs the actual analysis. It tries to decrypt the given cipher text using a guessed key and computes a fitness score based on how many valid 4-grams from the decrypted text match the pre-computed frequencies. The algorithm iteratively swaps pairs of characters in the current key to see if the resulting plain text yields a higher score (i.e., better matches with known words).

Finding the Best Key: The program conducts multiple trials (up to try_number iterations) to shuffle the keys and run the frequency analysis. It keeps track of the highest fitness score and the

corresponding key. When a new key produces a higher score, it replaces the current best key. The process continues until it either finds a satisfactory key or exhausts the allowed attempts.

Final Decryption: Once the best key is found, the cipher text is decrypted using the monoalphabetic_decrypt function. The resulting plain text is checked against the dictionary to confirm word validity.

Key Functions in Frequency Analysis

extract_words(dictionary1, alphabet_map): This function generates a count of 4-grams from the provided dictionary, which is later used for scoring. calculate_and_normalize_words(words_avg): Normalizes the word frequencies to aid in comparing scores during the decryption process. frequency_analysis(key, cipher_bin, char_positions, words, alphabet_len, max_try=0): This function is the heart of the frequency analysis, where keys are tested and evaluated based on how well the decrypted text matches known words. fitness_score(plaintext, words): This function calculates a score for the decrypted plain text based on the number of valid 4-grams found in the pre-calculated word frequency data.

map_alphabet():

This function creates a mapping of characters to their respective indices in a given alphabet. It returns a dictionary where each character is a key associated with its index in the alphabet.

map_key_to_alphabet():

This function maps a substitution key to an alphabet, creating a dictionary that associates each character in the alphabet with the corresponding character in the key. It raises a `ValueError` if the lengths of the key and alphabet do not match.

extract_words:

This function extracts four-letter sequences (spells) from a dictionary, counts their occurrences, and indexes them based on their computed values. It returns a list of frequencies for these four-letter sequences.

calculate_and_normalize_words:

This function calculates the frequencies of words and normalizes them into scores. It adjusts the values based on their logarithmic frequencies, ultimately returning a list of normalized word scores.

char_to_number:

This function converts characters in a given text to their corresponding numerical values based on a specified alphabet. It returns a list of numerical values representing the characters.

decrypt_bin:

This function decrypts a binary cipher using a provided key. It returns the decrypted binary data by finding the index of each character in the binary representation of the cipher text.

swap_chars:

This function swaps two characters in a list (the key) at specified indices. It modifies the key in place.

update_plaintext_indices:

This function updates the indices of a specified character in a plain text list based on the positions of that character. It modifies the plain text in place.

compute_index:

This function computes an index based on a provided fourth word value and a character value. It combines these values to create a new computed index.

fitness_score:

This function calculates the fitness score of a given plain text based on the frequency of valid words. It returns an integer score indicating the number of valid words present.

restore_plaintext:

This function restores the indices of a character in the plain text list. It updates the plain text indices using a specified character and index.

attempt_key_swap:

This function attempts to swap two characters in the key and calculates the resulting fitness score of the plain text. It returns the temporary score and the swapped characters.

evaluate_key:

This function evaluates whether the temporary fitness score from a key swap is greater than a recorded maximum score. It returns a boolean value indicating the result.

key_swap_and_evaluation:

This function swaps two characters in the key and evaluates the new key based on the resulting fitness score. It returns the updated maximum score and a boolean indicating whether a better key was found.

frequency_analysis:

This function performs frequency analysis on the decrypted binary data to find the best key. It iteratively swaps characters in the key and evaluates the results, returning the highest fitness score found.

find_key:

This function finds the decryption key for a given cipher text by analyzing frequency patterns. It returns the final decryption key as a string, built from the best key found during analysis.

key_mapping:

This function maps a decryption key to an alphabet, creating a dictionary that associates each character in the alphabet with the corresponding character in the key. It raises a `ValueError` if the lengths of the key and alphabet do not match.

monoalphabetic_decrypt:

This function decrypts the given ciphered text using the Mono-alphabetic cipher technique, returning the decrypted text as a string. It preserves the case of alphabetic characters and leaves non-alphabetic characters unchanged.

extract_potential_words:

This function extracts potential words from the decoded text using a regular expression. It returns a list of these words.

load_dictionary:

This function loads a dictionary from a file and returns a set of words contained in that dictionary, converted to lowercase.

break_mono:

This function decrypts a given cipher text using the Mono-alphabetic cipher technique. It utilizes the `monoalphabetic_decrypt` function and returns the decrypted text.

Conclusion

This program efficiently utilizes frequency analysis to tackle the challenge of decrypting mono-alphabetic substitution ciphers. By leveraging the statistical properties of the English language, it systematically refines its guesses for the cipher key, ultimately leading to successful decryption.

3.4 Helper Functions

write_output_file(mode, text):

This function takes a decryption mode and the decrypted text as inputs, then writes the text to a file named according to the decryption mode (e.g., break_caesar.txt. The file is created or overwritten in write mode.

validate_text(input_text, dictionary):

This function checks the words in the input text against a provided dictionary and replaces any non-matching words with "[REDACTED]," while preserving the original punctuation. It uses a nested helper function, replace_match, to process each matched word and determine whether it should be redacted based on its presence in the dictionary.

3.5 Break.py Main

The main() function serves as the entry point for a program that encrypts or decrypts text using Caesar, Affine, or Mono-alphabetic ciphers based on command-line arguments. It begins by parsing the cipher type and input file from the command line, reads the cipher text from the specified file, and processes it accordingly applying the appropriate decryption method for the selected cipher, validating the output against a dictionary, and saving the decrypted text to an output file named after the cipher used. The overall structure ensures that different cipher techniques are handled correctly, leveraging functions for decryption, validation, and output writing.

Example how to include and reference a figure: Fig. 1.

References

- BBM465 Lecture 2 Slide Basic Ciphers
- Handbook of Applied Cryptography. A. Menezes, P. van Oorschot and S. Vanstone. CRC Press page 18, 248
- https://books.google.com.tr/books?hl=tr&lr=&id=j-NeLkWNpMoC&oi=fnd&pg=PA269&dq=quad+gram+frequency+analysis&ots=YkC6GqBrIL&sig=FoF53Uvde3kxlF4_QyKaaWN2HLQ&redir_esc=y#v=onepage&q=quad%20gram%20frequency%20analysis&f=false

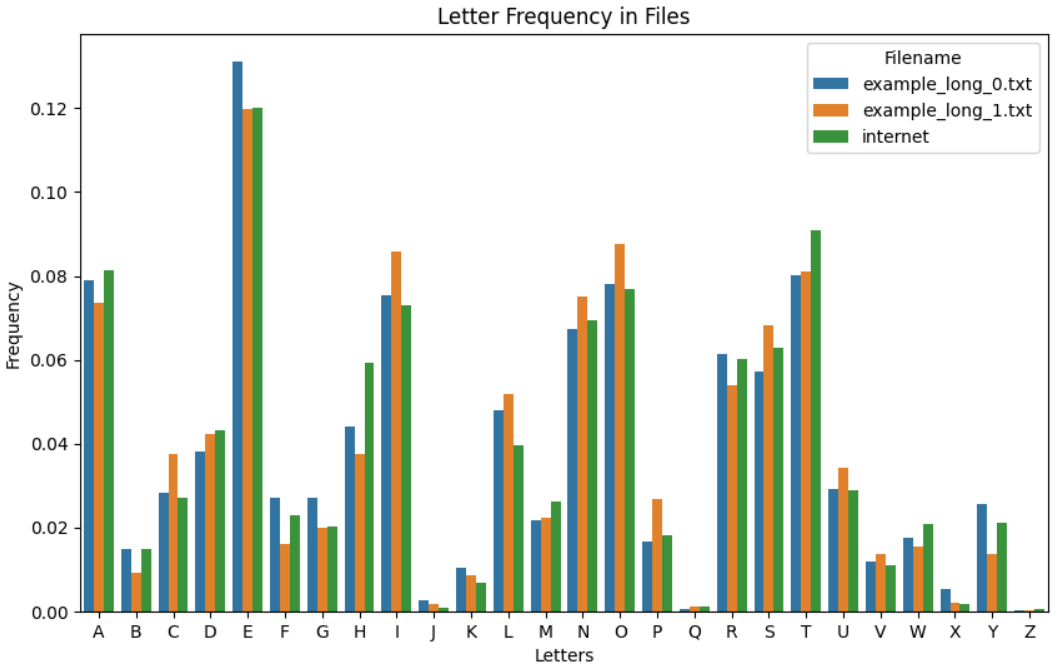

Figure 1: Plot of the functions.