বরেন্দ্র বিশ্ববিদ্যালয়

VARENDRA UNIVERSITY

Dept. of CSE
Varendra University

VARENDRA UNIVERSITY

Department of Computer Science and Engineering

Assignment

Cryptography and network security
Course Code: CSE 431

Submitted by	Submitted to
Name: Tasnimul Hasan ID: 191311111 Semester: 12th Batch: 20th	Name: Omar Faruqe Assistant professor Dept.of CSE Rajshahi university

Date: 05/05/2023

Signature

//Deciphering vigenere like chiper

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define KEY_LENGTH 4 // Can be anything from 1 to 13
#define MIN_KEY_LENGTH 1
#define MAX_KEY_LENGTH 13
#define KEY_SPACE 256
#define ENGLISH_LETTER_FREQUENCY .065

int vigenereEncrypt();
int crackVigenere();

int main(int argc, const char * argv[]) {
    //vigenereEncrypt();
    crackVigenere();
}

/* This function was given by the assignment, not mine. */
int vigenereEncrypt() {
    unsigned char ch;
    FILE *fpIn, *fpOut;
    int i;
    unsigned char key[KEY_LENGTH] = {0x6C, 0x75, 0x6B, 0x65}; // luke

    fpIn = fopen("ptext.txt", "r");
    if (fpIn == NULL) {
        return 1;
    }
    fpOut = fopen("ctext.txt", "w");
    i=0;
    while (fscanf(fpIn, "%c", &ch) != EOF) {
        /* avoid encrypting newline characters */
        /* In a "real-world" implementation of the Vigenere cipher,
         every ASCII character in the plaintext would be encrypted.
         However, I want to avoid encrypting newlines here because
         it makes recovering the plaintext slightly more difficult... */
        /* ...and my goal is not to create "production-quality" code =) */
        if (ch!='\n') {
            fprintf(fpOut, "%02X", ch ^ key[i % KEY_LENGTH]); // ^ is logical XOR
            i++;
        }
    }

    fclose(fpIn);
    fclose(fpOut);
    return 0;
}

size_t findKeyLength(const size_t size, const unsigned char *cipherStream) {

    unsigned char *stream = malloc(sizeof(unsigned char) * size);
    size_t *frequency = malloc(sizeof(size_t) * KEY_SPACE);
    double summations[MAX_KEY_LENGTH];
    memset(summations, 0, sizeof(double) * MAX_KEY_LENGTH);

    // Make sure we are operating on valid key lengths
    assert(MIN_KEY_LENGTH > 0);
    assert(MIN_KEY_LENGTH < MAX_KEY_LENGTH);
    // So we have to try every possible key length
    for (size_t n = MIN_KEY_LENGTH; n <= MAX_KEY_LENGTH; ++n) {

        double averageSummation = 0.;

        // For each position in the key (ki...kn)
        for (size_t k = 0; k < n; ++k) {
            memset(stream, 0x00, size);
            memset(frequency, 0, sizeof(size_t) * KEY_SPACE);

            size_t j = 0; // This will be the length of the stream
            for (size_t i = k; i < size; i += n) { // Start from the ki position
                stream[j++] = cipherStream[i];
            }

            // For each position in stream, record the frequency of each byte
            for (size_t i = 0; i < j; ++i) {
                frequency[stream[i]] += 1;
            }
            size_t summation = 0;
            for (size_t i = 0; i < KEY_SPACE; ++i) { // Could improve by slightly limiting this loop
(65-127?)
                summation += frequency[i] * (frequency[i] - 1);
            }

            averageSummation += (double)summation / (j * (j - 1));
        }
        summations[n - 1] = averageSummation / n;
        // There may be a certain threshold where we can break out of here with
        // a high degree of confidence that we have obtained the key length.
        // Probably higher than something like 0.06?
    }

    double max = 0.;
    size_t keyLength = 0;
    for (size_t i = MIN_KEY_LENGTH - 1; i < MAX_KEY_LENGTH; ++i) {
        if (max < summations[i]) {
            max = summations[i];
            keyLength = i + 1; // Add 1 to adjust for 0 based indexing.
        }
    }

    free(stream);
    free(frequency);

    return keyLength;
}

/* Frequency analysis of lower case English letters appearing in the given
 * stream.
 */
double frequencyAnalysis(const size_t size, const unsigned char *stream) {
    size_t *frequency = malloc(sizeof(size_t) * KEY_SPACE);

    for (size_t i = 0; i < size; ++i) {
        frequency[stream[i]] += 1;
    }

    size_t summation = 0;
    for (size_t i = 97; i < 122; ++i) {
        summation += frequency[i] * (frequency[i] - 1);
    }

    free(frequency);

    const size_t lowerCaseSize = KEY_SPACE;
    const double result = (double)summation / (lowerCaseSize * (lowerCaseSize - 1));

    return result;
}

/* This function takes all the data encoded by the same character in the key. It
 * will return the most suitable guess for the character in the key based on a
 * frequency analysis.
 */
unsigned char calculateKey(const size_t size, const unsigned char *stream) {

    unsigned char *shiftedStream = malloc(sizeof(unsigned char) * size);
    double guesses[KEY_SPACE];
    memset(guesses, 0x00, sizeof(size_t) * KEY_SPACE);

    for (unsigned char b = 0x00; b < 0xFF; ++b) { // Can improve if key is English (32 - 127
only)
        size_t foundGuess = 1;
        for (size_t j = 0; j < size; ++j) {
            shiftedStream[j] = stream[j] ^ b;

            // When the guess 'b' is correct, all bytes in the plaintext
            // stream will between 32 and 127 (ASCII values of the English
            // alphabet, including punctuation)
            if ((shiftedStream[j] < 0x20) || (shiftedStream[j] > 0x7A)) {
                foundGuess = 0;
                break;
            }

            // Modify this list when examining the output to narrow down guesses
            // These are characters that shouldn't appear in the plaintext
            if ((shiftedStream[j] == 0x2A ||
                 shiftedStream[j] == 0x5F ||
                 shiftedStream[j] == 0x5E ||
                 shiftedStream[j] == 0x60 ||
                 shiftedStream[j] == 0x24 ||
                 shiftedStream[j] == 0x26 ||
                 shiftedStream[j] == 0x23 ||
                 shiftedStream[j] == 0x2B )) {
                foundGuess = 0;
                break;
            }
        }
        if (foundGuess) {
            guesses[b] = frequencyAnalysis(size, shiftedStream);
        }
    }

    double max = 0.;
    unsigned char key = 0x00;
    for (unsigned char b = 0x00; b < 0xFF; ++b) {
        if (max < guesses[b]) {
            max = guesses[b];
            key = b;
        }
    }

    free(shiftedStream);
    return key;
}

size_t readFile(unsigned char **out) {

    // Open the ciphertext to decrypt
    FILE *cipherFile = fopen("ciphertext.txt", "r");
    //FILE *cipherFile = fopen("ctext.txt", "r");
    if (cipherFile == NULL) {
        return 0;
    }

    // Get the length of the file and set it back to the beginning
    fseek(cipherFile, 0L, SEEK_END);
    const size_t fileSize = ftell(cipherFile);
    rewind(cipherFile);
    unsigned char *rawData = malloc(sizeof(unsigned char) * fileSize);

    // Read the entire file into memory
    if (fread(rawData, sizeof(char), fileSize, cipherFile) != fileSize) {
        free(rawData);
        fclose(cipherFile);
        return 0;
    }
    fclose(cipherFile);

    // The following code will convert the data from hex to ASCII.
    const size_t size = fileSize / 2; // Divide by 2 since 1 hex value occupies 2 bytes
    unsigned char *cipherStream = malloc(sizeof(unsigned char) * size);

    size_t count = 0;
    for (size_t i = 0; i < fileSize; i += 2) {
        if (sscanf((char *)rawData + i, "%2hhX", &cipherStream[count++]) != 1) {
            break;
        }
    }

    if (count == size) {
        // Conversion to hex worked, so return the data
        free(rawData);
        *out = cipherStream;
        return size;
    } else {
        // File was not in hex, so return the raw data
        free(cipherStream);
        *out = rawData;
        return fileSize;
    }
}

/* Attacking the (variant) Vigenere cipher requires two steps:
 * - Determine the key length (bytes in the key)
 * - Determine each byte of the key
 */
int crackVigenere() {

    unsigned char *cipherStream = NULL;
    const size_t size = readFile(&cipherStream);

    const size_t keyLength = findKeyLength(size, cipherStream);

    // Below needs to refactored out into a function (or more)
    // Load the cipher text grouped by ci where ki was used as the shift
    // keyMemorySize must account for size not evenly divided by keyLength
    const size_t keyMemorySize = (size / keyLength) + (size % keyLength);
    const size_t streamSize = keyMemorySize * keyLength;
    unsigned char *stream = malloc(sizeof(unsigned char) * streamSize);

    // Length of each group of characters encoded by the same shift.
    size_t *groupLengths = malloc(sizeof(size_t) * keyLength);

    for (size_t n = 0; n < keyLength; ++n) {
        size_t j = keyMemorySize * n;
        size_t k = 0;
        for (size_t i = 0 + n; i < size; i += keyLength) {
            stream[j++] = cipherStream[i];
            k++;
        }
        groupLengths[n] = k;
    }

    unsigned char *key = calloc(keyLength, sizeof(unsigned char));
    for (size_t n = 0; n < keyLength; ++n) {
        key[n] = calculateKey(groupLengths[n], stream + (n * keyMemorySize));
    }

    for (size_t count = 0; count < size; ++count) {
        fprintf(stderr, "%c", cipherStream[count] ^ key[count % keyLength]);
    }

    free(cipherStream);
    free(groupLengths);
    free(stream);
    free(key);

    return 0;
}
```