CSCI 2461 The George Washington University

Professor Leontie Timothy Lytkine

**CSCI 2461 Project 4 – Simple Cryptography Report**

**Implementation:**

An overview of the way that my implementation works can be described as follows:

**Encryption occurs by applying:**

*bitShift(VigenereCipher(CaesarCipher(W)))*

**Decryption occurs by applying:** *invertedCaesarCipher(invertedVignereCipher(invertedBitShift(encrypted\_string)))*

In order to implement the functions and inverse functions for the Caesar Cipher, Vigenere Cipher and Bit Shift operations, there were subroutines made for XOR, multiplication and division. In addition to this there were many labels to allocate the many variables and addresses needed. There were strings allocated for every prompt outputted to the user, storage locations the key with its every individual variable (z1,x1,y1,y2,y3), the encrypted / decrypted message or input and the initial memory addresses for the key and the message. Many ideas had to be considered when applying these functions to modify the input string such as when to increment or decrement a memory address before storing or when to clear the proper registers so the proper values go into their according locations. This required a lot of trial and error and patience, looking through every line of code to spot potential errors.

The pseudo code attached gives an overall explanation of how my implementation of the encryption-decryption module is structured. Some of the small details not mentioned in the pseudo code is that subroutines had to be created for XOR, multiplication and division. In order to make the algorithm as effective as possible, I made these subroutines take values from the same two registers as input: R3 and R4. The output was also the same for all the subroutines (R5) with the exception of the subroutine for division which also outputted the resulting remainder of the division operation in register R6. A challenge I ran into was using as little registers as possible for each step of the encryption and decryption functions in order to be able to track the number of memory addresses or characters stored in the MESSAGE label. This required alternating between incrementing and decrementing a variable that tracked the number of characters stored. This did make the implementation more complex but allows there to be any amount of characters encrypted from 1 to 10. To make this fully functional, I had to use the ASCII value for the enter / carriage return key to detect when the user was done typing the string. Another detail worth mentioning about my explanation was that I had to convert the ASCII values from both the key and the message in order to convert them to decimal and then I had to convert them back accordingly in order for the original message being encrypted to match the message outputted by the decryption algorithm.

**Pseudocode for T-Lytkine.asm**

0) Print "Starting Privacy Module"

Start of loop

1) a) Print prompt "ENTER E to ENCRYPT, D to DECRYPT, X to EXIT"

1) b) Take user input

2) a) If not E, D, or X, print "INVALID ENTRY, PLEASE TRY AGAIN"

2) b) If X is typed, program exits and halts

2) c) If E or D, Store user input in memory (register dedicated to this)

2 a) If E or D print "ENTER KEY (Length 5, non-zero digit less than 8 followed by non-numeric character followed by 3 digit number between 0 and 127)"

b) print "WHEN DONE PRESS ENTER"

c) Take user input (user will type key) (single digit less than 8 followed by non numeric character followed by 3 digit number between 1 and 127)

d) Store input (used to decrypt message) (must be stored in memory, label KEY)

3) if input in 1B was D:

a) Check if key is correct

b) Apply inverse Bit Shift function on message, overwriting old message

c) Apply inverse Vigenere Cipher Function, overwriting old message

d) Apply inverse Caesars Cipher Function, overwriting old message

e) Print decrypted message to screen (correct message will not print if key is incorrect)

d) returns to 1A) (Branch, loop at 1a)

4) if input in 1A was E:

a) Print "Enter input text of length 10 character. When done press <enter>"

b) User will input string of 10 characters (ASCI printable character, do not include control characters), terminates with enter key

c) String stored in memory location MESSAGE starting at x4000, writes encrypted characters into same location (plain text must not persist so that message can not be accessed without being decrypted)

d) using string at MESSAGE at x4000 and KEY entered in 2c, use encryption algorithm and store at MESSAGE (string is w)

Encryption Algorithm:

First apply, Caesars Cipher to the string entered by adding each character to the numeric value of the first three numbers of the key, then divide this sum by 128 and store the remainder as the new character at each position in the string

Then, apply the Vigenere Cipher by using the same key and applying pi XOR k, (pi being the characters in the new string resulting from Caesars Cipher in the previous step). XOR can be done simply with the AND & NOT operations in LC3 Assembly.

Finally, shift each of the characters in the new resulting string by K=z1 values to the right.

e) return to step 1A) (Branch, loop at 1a)

***Question (during your design): what do you echo when the user types in the key?***

When the user types in the key, nothing should be echoed because this would make it less secure as someone else other than the user could see the key being outputted onto the console.

***b. Your program must store this ASCII string in memory location MESSAGE starting at x4000. It must then write the encrypted characters into the same location since you do not want plain text to persist. Question: – why?***

Plain text should not persist because this compromises the integrity of the encryption-decryption module. If the characters are written into some location and not overwritten during each step, then they could potentially be accessed by someone just looking at what is stored in those memory locations.

***Question: How would you change the program if instead of at most 10 message, you want to allow strings of any length ?***

This would simply be a matter of ending input being taken from the user when they press the carriage return / enter key. This way as long as enter is not pressed, (in which case a register with the negative ASCII value of carriage return will be added to the input and equal 0, causing a branch statement to exit the loop) the loop that allows the user to input characters would continue to run allowing for a string of any length. This of course would be limited by the memory of the LC3, however, it would certainly allow for a message of a length greater than 10.

***Question: Why are we using modulo 128 ? Should we be using 128 or a different value if the characters have to be alpha numeric or special characters (i.e., they have to be visible characters from the keyboard – cannot be space, tab etc.).***

We are using modulo128 because the number entered by the user used as the key for the XOR and modulo operations is between 0 and 127. The key is 7 bits so its maximum value of 127 is at the end of this range (127). Therefore, applying modulo 128 after subtracting the key value from each character ensures that all of the encryption bits in the plain text can be properly decrypted without any loss of data. Length is a big factor in this because applying modulo or bit wise operation such as XOR can have difference outputs depending on the lengths of their inputs. The reason why the value if 127 is because this is also the max value of a non extended ASCII table. If the characters had to be alpha numeric or special characters, this would imply that it is an extended ASCII table in which case the maximum value or the value that should follow modulo should be 255.

***Question: How do you decrypt a message that has been encrypted using this scheme?***

The inverse of XOR is XOR, therefore, all that needs to be done is the encrypted string needs to have the XOR operation applied with the key that is still stored. (After the inverse bit shift operation is applied to the string). This will output the string inputted into the initial XOR function or Vigeneres Cipher.

***Question: How do you shift left in LC3? How do you right shift in LC3 ?***

In order to left shift in LC3, you can add the value of a character to itself K times. In order to do this to the whole string there would be a loop with the previously stated operation put into a loop within this one where the address of each character in the string would be adjusted and stored accordingly.

In order to right shift in LC3, you can put the value of the string in one register, then in another register you can divide the value of the string by 2 using a division subroutine and putting the result in the corresponding register. Then take the two’s complement of this register and add one to it so the value of it is negative. After this add the first register mentioned with just the value of the string with a negative half of this value so it will be (value) – (1/2value). Doing this in a loop K times will be the inverse operation of the left bit shift.