Ryan Shimizu

ID: 823053121

CompE-271

Project Report

Vigenère Encryption/Decryption Program

Table of Contents

Introduction:

This program takes a user-inputted “plaintext” and a user-inputted “key” and encrypts the plaintext into a “ciphertext” using the Vigenère encryption/decryption algorithm. The Vigenère cipher was originally described by Giovan Battista Bellaso in 1553 (and was subsequently cracked in 1863) but it was misattributed to Blaise de Vigenère in the 19th century, hence its name. The algorithm is a classic example of a polyalphabetic substitution cipher.

Discussion:

To encrypt the plaintext characters to a cipher text using a Vigenere cipher, the chosen keyword is first repeated or reduced to match the length of the plaintext string. Then, using polyalphabetic substitution, we take each letter of the plaintext and shift its value to the corresponding letter key. For example, if the plaintext letter is A and its corresponding key letter is C, the plaintext letter is shifted according to the tabula recta table as shown in **Appendix A.** In this case, column A, row C is used and the ciphertext character would become C. If P was the plaintext character with X as the keyword character, we would go to column P, row X and the ciphertext character becomes M. This process is then repeated for the remainder of the string.

To decrypt the cipher text, this process is applied in reverse. The keyword is broken up letter-by-letter and the column corresponding to the keyword is matched up with the corresponding letter-row that matches with the ciphertext.

To apply this method in assembly, however, we need to take a different approach. First, we need to take the user-inputted strings and convert them into upper-case. This is required for the ensuing algorithm since upper-case letters and lower-case letters have different ASCII hex values. Hence, the function convertUpper() takes the string and converts the lower-case characters by evaluating each character individually and bit-shifting the hex values to their corresponding upper-case variant (subtracting by 0x20) and then storing it back into the string. If the character is already in upper-case, the character is ignored. This procedure is repeated for the entire length of the string.

Second, we need to get the length of the plaintext and key string to generate a "keystream." This is done by repeating or reducing the key string to match the length of the plaintext. This ensures that every character in the plaintext corresponds to exactly one letter in the keystream. This means that we first need to retrieve the lengths of both the plaintext string and the key string. This procedure is handled by the function, getLength(), which takes any given string pointer, procedurally goes character-by-character and compares it to the null-byte value. If the character loaded is NOT a null-byte character, that means we haven't reached the end of the string and we increase the character counter by 1. If the character loaded IS a null-byte character, then we have reached the end of the string. Now, we need to reposition the pointer back to the beginning of the string. Since we already have the length of the string in one of our registers, this process is simple. We take advantage of the fact that a string stores its characters linearly in memory and we subtract the memory address value by the length of the string to put the pointer back to where it was when the function first started. This function is applied to both the plaintext and key string.

Next, we need to apply the values we retrieved to generate the keystream. This procedure is handled by the function, repeatKey(). The function takes the length of the plaintext and key strings and either repeats or reduces the key string to match the length of the plaintext string. The function first evaluates the lengths of the two strings and makes the decision to either reduce,

extend, or do nothing to the keystring. A flowchart of this function is shown in **Appendix B**. If we need to extend the key, we use two pointers. One pointer points to the beginning of the string and the other points to the end of the string. In assembly, this means we have two memory addresses loaded into two separate registers. The program loads the value of the pointer located in the beginning of the string into another register and stores it in the pointer at the end of the string. Then, both pointers are incremented by 1 byte and the process is repeated until the length of the keystring matches the length of the plaintext. If we need to reduce the key, we place a pointer at the end of the string (again calculated using the length of the key we found earlier) and storing a null-byte in the specified memory address. The pointer is then decremented by 1 byte and the process is repeated until the length of the keystring matches the length of the plaintext. A visual diagram of this process is shown in **Appendix C**.

Now, we have the necessary data and adjustments to start encrypting and decrypting the text. The encryption algorithm is handled by the function vigenereEncrypt(). First, we need to "normalize" the ASCII values to reflect the correct offset of each letter. We do this by subtracting the hex value 0x41 to each letter in the key string. This takes advantage of the fact that the letters A-Z are properly ordered in ASCII so that, after normalizing the values, A = 0, B = 1, C= 2…etc. We also do the same procedure on the plaintext. The encryption algorithm can be summarized algebraically, which we will use to calculate the ciphertext character value: (plaintext + keytext)mod26 = ciphertext. Here, we run into our first issue. Assembly does not perform real division (let alone modulo division) and so we need to be creative in our approach to calculate modulo 26. Since Assembly can do addition and subtraction quickly, I used a loop to repeatedly compare the sum of the plaintext and keytext characters to the number 26. If the number is greater than 26, we subtract that value by 26 and repeat until the number is less than 26. The number that remains is the modulo of 26. This procedure is incredibly fast since the maximum number of times we need to repeat the loop is once and the maximum value we would get is 50 (Z + Z = 25 + 25) and the modulo of that would be 24. Once we have the modulo value, we de-normalize the value by adding 0x41 back and we now have its corresponding upper-case ciphertext character in ASCII.

To decrypt the ciphertext, the process is similar. The only part that is different is the modulo calculation. To decrypt the text algebraically, we apply the formula (ciphertext – keytext)mod26 = plaintext. Since there is a possibility that this result is negative (i.e. keytext > ciphertext), we need to adjust our modulo algorithm to account for that fact. We do this by looping the value back around to the end of the alphabet. For example, Z corresponds with the value 25, but it can also correspond to the value of -1. The same goes for (Y = 24) V (Y = -1), (A = 0) V (A = -26), etc. We can normalize these negative values easily by adding 26 if we find that the result we get after subtracting the ciphertext and the keytext is less than zero. Then, we apply the same procedure as before to find the modulo. After the entire string is traversed, we return the pointers back to the beginning of the string using the same procedure as described before.

Finally, the program returns these strings to the user by printing them into the I/O stream using the printf() function.

Assembly Code:

```
/*
 * AFunctions.s
 *
 *
 */

 .global convertUpper
  .data
  // declare any global variables here

 .text
  convertUpper:
  mov   r12,r13              // save stack pointer into register r12
  sub   sp,#32               // reserve 32 bytes of space for local variables
  push  {lr}                 // push link register onto stack -- make sure you pop it out before you return



 // Your solution here
  //R0 = pointer to first char in string, R1 = counter, R2 = increment
  MOV R2, #0    //clear registers
  MOV R1, #0

  FOR:
    LDRB R2, [R0]
    CMP R2, #0X0    //check if pointer points to end of string
    BEQ DONE

    CMP R2, #0X61   //check if char is an upper case
    BLT SKIP
```

```
  CMP R2, #0X7A
  BGT SKIP
                //if here, char must be a lower case letter
  SUB R2, R2, #0X20     //sub by 32 to get upper case variant


SKIP:
  STRB R2, [R0]       //put back value into address stored at r0
  ADD R1, R1, #16       //increment counter by 4, needed to return pointer back to original position
  ADD R0, R0, #0x1       //move pointer to next char
  B FOR            //repeat loop

DONE:
  //total distance travelled since beginning of function is now stored in R1
  ADD R0, R0, R1      //r0 = r0 - offset, r0 now points back to beginning of string


  pop {lr}                          // pop link register from stack
  mov sp,r12                 // restore the stack pointer -- Please note stack pointer should be equal to the
                            // value it had when you entered the function .
  mov pc,lr                         // return from the function by copying link register into  program counter




.global getLength
  .data
  // declare any global variables here
 .text
  getLength:
  mov   r12,r13                // save stack pointer into register r12
```

```
 sub   sp,#32                // reserve 32 bytes of space for local variables
 push  {lr}                  // push link register onto stack -- make sure you pop it out before you return



// Your solution here
 //R0 = pointer, R1 = dereferencer, R2 = for counter

 MOV R1, #0    //clear registers
 MOV R2, #0

 FOR1:
   LDRB R1, [R0]    //dereference r0
   CMP R1, #0X0     //check if it is equal to nullbyte
   BEQ DONE1       //exit loop

   ADD R2, R2, #1    //increment counter
   ADD R0, R0, #0X1  //move to next char in array
   B FOR1        //repeat loop

 DONE1:
   MOV R0, R2    //return counter



 pop {lr}                          // pop link register from stack
 mov sp,r12              // restore the stack pointer -- Please note stack pointer should be equal to the
                      // value it had when you entered the function .
 mov pc,lr                         // return from the function by copying link register into  program counter



.global repeatKey
  .data
  // declare any global variables here
```

```
.text
repeatKey:
mov   r12,r13              // save stack pointer into register r12
sub   sp,#32               // reserve 32 bytes of space for local variables
push  {lr}                 // push link register onto stack -- make sure you pop it out before you return



// Your solution here
//R0 = string pointer, R1 = userLength, R2 = keyLength, R3 = dummy, R4 = endPointer, R5 = DUMMY2, R6 = dereferencer

MOV R3, #0     //clear registers
MOV R4, #0
MOV R5, #0
MOV R6, #0
MOV R7, #0

CMP R1, R2     //analyze lengths
BEQ DONE2      //if they are the same, do nothing
BLT REDUCE     //if userLength is less than keyLength, reduce key string

EXTEND:        //if here, userlength is more than keylength
 SUB R3, R1, R2   ///R3 = userLength - keyLength
 MOV R4, R0    //copy address of beginner pointer into r4
 MOV R7, #0X1
 MUL R5, R2, R7    //calculate offset (distance between beginning of string to the end), kinda redundant oops

 ADD R4, R4, R5    // R4 now points to the end of the string
 FOR2:
   CMP R3, #0      //check if userlength is now 0
   BEQ ENDFOR2       //exit loop

   LDRB R6, [R0]
```

```
  STRB R6, [R4]
  SUB R3, R3, #1    //decrement counter by 1
  ADD R0, R0, #0X1  //increment both pointers to next char
  ADD R4, R4, #0X1
  B FOR2

ENDFOR2:
  MOV R6, #0X0      //reuse dummy register
  STRB R6, [R4]     //add nullbyte into end of string
  B DONE2

REDUCE:           //if here, userLength is less than keyLength
  SUB R3, R2, R1   //r3 = keyLength - userLength
  MOV R7, #0X1
  MUL R4, R2, R7    //calculate offset (distance between beginning of string to the end), also redundant

  ADD R0, R0, R4      //move pointer to end of string

  MOV R5, #0X0          //move nullbyte into r5
  SUB R0, R0, #0X1      //r0 points to char before nullbyte

  FOR3:
    CMP R3, #0    //check if r3 is 0
    BEQ DONE2   //exit loop

    STRB R5, [R0]   //move null byte character into address stored in r0 (end of string)
    SUB R0, #0X1      //decrement R0 to point to previous char
    SUB R3, #1      //decrement counter

    B FOR3        //repeat

  DONE2:
    MOV R0, #0    //return void
```

```
pop {lr}                        // pop link register from stack
mov sp,r12                  // restore the stack pointer -- Please note stack pointer should be equal to the
                           // value it had when you entered the function .
mov pc,lr                       // return from the function by copying link register into  program counter


.global vigenereEncrypt
 .data
 // declare any global variables here
.text
 vigenereEncrypt:
 mov   r12,r13              // save stack pointer into register r12
 sub   sp,#32               // reserve 32 bytes of space for local variables
 push  {lr}                 // push link register onto stack -- make sure you pop it out before you return

// Your solution here
 //R0 = plaintext pointer, R1 = keytext pointer, R2 = plaintext dereferencer, R3 = keytext dereferencer, R4 =
modulo, R5 = lengthcounter, R6 = 0X1
 MOV R2, #0     //clear registers
 MOV R3, #0
 MOV R4, #0
 MOV R5, #0
 MOV R6, #0

 MASTERLOOP:
  LDRB R2, [R0]   //ldrb needed because we only want one char at a time
  LDRB R3, [R1]   //do the same for keytext
  CMP R2, #0    //is it null-byte (end of string?)
  BEQ FINISH    //if so, end loop

  SUB R2, R2, #0X41       //normalize ascii values...A=0, B=1, etc...
  SUB R3, R3, #0X41
  ADD R4, R2, R3    //(plaintext + keytext)
```

```
MODULO26:        //assembly doesn't do real division, so we take this approach to calculate the modulo of 26
  CMP R4, #26       //is the number greater than 26?
  BLT ENDMODULO26

  SUB R4, R4, #26   //if not, subtract 26
  B MODULO26        //result will be modulo (even though we didn't divide or multiply anything)

//R4 = (plaintext + keytext)mod26 = encrypted letter
ENDMODULO26:
  ADD R4, R4, #0X41     //convert back to ascii
  STRB R4, [R0]      //store encrypted letter into plaintext string
  ADD R0, R0, #0X1      //move onto next char in string
  ADD R1, R1, #0X1      //same here
  ADD R5, R5, #1        //increment counter by 1
B MASTERLOOP

FINISH:
  //need to revert pointer back to beginning of string
  ADD R0, R0, #0X1    //move pointer forward
  ADD R1, R1, #0X1    //
  MOV R6, #0          //null byte
  STR R6, [R0]        //put terminating null byte
  STR R6, [R1]        //
  MOV R6, #0X1        //constant byte holder
  ADD R5, R5, #1      //since we are one byte ahead from previous steps
  MUL R5, R5, R6      //total distance travelled, redundant
  SUB R0, R0, R5
  SUB R1, R1, R5      //go back to beginning of string

pop {lr}                            // pop link register from stack
mov sp,r12                  // restore the stack pointer -- Please note stack pointer should be equal to the
                           // value it had when you entered the function .
mov pc,lr                           // return from the function by copying link register into  program counter
```

```
.global vigenereDecrypt
 .data
 // declare any global variables here
.text
 vigenereDecrypt:
 mov   r12,r13              // save stack pointer into register r12
 sub   sp,#32               // reserve 32 bytes of space for local variables
 push  {lr}                 // push link register onto stack -- make sure you pop it out before you return

// Your solution here
 //R0 = plaintext pointer, R1 = keytext pointer, R2 = plaintext dereferencer, R3 = keytext dereferencer, R4 = modulo, R5 = lengthcounter, R6 = 0X1
 MOV R2, #0      //clear registers
 MOV R3, #0
 MOV R4, #0
 MOV R5, #0
 MOV R6, #0

 MASTERLOOPD:
  LDRB R2, [R0]   //need to ldrb because we only want to work with one char at a time
  LDRB R3, [R1]   //same for keytext
  CMP R2, #0    //is it null-byte (end of string?)
  BEQ FINISH2    //if so, end loop

  SUB R2, R2, #0X41       //normalize ascii values...A=0, B=1, etc...
  SUB R3, R3, #0X41
  SUB R4, R2, R3    //(plaintext + keytext)

  MODULO26D:        //can use the same algorithm as before, but since we are subtracting, we need to account for the possibility of a negative value
    CMP R4, #0        //negative check, if it comes out to be negative we loop it back around from Z towards A
    ADDLT R4, #26
```

```
    BLT MODULO26D   //repeat as necessary


    CMP R4, #26       //same as before
    BLT ENDMODULO26D


    SUB R4, R4, #26
    B MODULO26D


  //R4 = (plaintext + keytext)mod26 = encrypted letter
  ENDMODULO26D:
    ADD R4, R4, #0X41     //normalize back to ascii
    STRB R4, [R0]      //store encrypted letter into plaintext string, need strb because we don't want to overwrite
more than one char at a time
    ADD R0, R0, #0X1      //move onto next char in string
    ADD R1, R1, #0X1      //same here
    ADD R5, R5, #1        //increment counter by 1
  B MASTERLOOPD


 FINISH2:
  //need to revert pointer back to beginning of string, also put null byte at end of string
  ADD R0, R0, #0X1    //move pointer forward
  ADD R1, R1, #0X1    //
  MOV R6, #0          //null byte
  STR R6, [R0]        //put terminating null byte
  STR R6, [R1]        //
  MOV R6, #0X1        //constant byte holder
  ADD R5, R5, #1      //since we are one byte ahead from previous steps
  MUL R5, R5, R6      //total distance travelled
  SUB R0, R0, R5
  SUB R1, R1, R5      //go back to beginning of string


 pop {lr}                              // pop link register from stack
 mov sp,r12                  // restore the stack pointer -- Please note stack pointer should be equal to the
                             // value it had when you entered the function .
```

```
mov pc,lr                    // return from the function by copying link register into  program counter
```

C code:

```
#include "assemblyFunctions.h"
#include <stdio.h>
#include <stdlib.h>

/*********************************************************************
*
*       main()
*
*  Function description
*   Application entry point.
*/
void main(void) {
  char userInput[200];
  char keyInput[200];
  char *user = &userInput[0];
  char *key = &keyInput[0];
  printf("Enter the phrase to be encrypted/decrypted (max 12 characters):\n");
  scanf("%50s", user);
  printf("\nEnter the keyword to be encrypted/decrypted (max 12 characters):\n");
  scanf("%50s", key);



  convertUpper(user);     //converts user input to uppercase, necessary prep for algorithm
  convertUpper(key);



  int userLength = getLength(user);     //used for proceeding function
  int keyLength = getLength(key);

  repeatKey(key, userLength, keyLength);    //repeats the key to match the length of the user input
```

```
  printf("\nEncrypt/Decrypt?\n1. Encrypt\n2. Decrypt\n");
  int option;
  scanf("%d",&option);
  switch(option){
    case 1:
        vigenereEncrypt(user, key);     //branch to encrypt
        printf("Encrypted Text: %s", user);
        break;
    case 2:
        vigenereDecrypt(user, key);     //branch to decrypt
        printf("Decrypted Text: %s", user);
        break;
    default:
        printf("\nInvalid Option.\n");
        break;
  }
}

/*************************** End of file ****************************/
```

Tool Set Used:

- SEGGER Embedded Studio

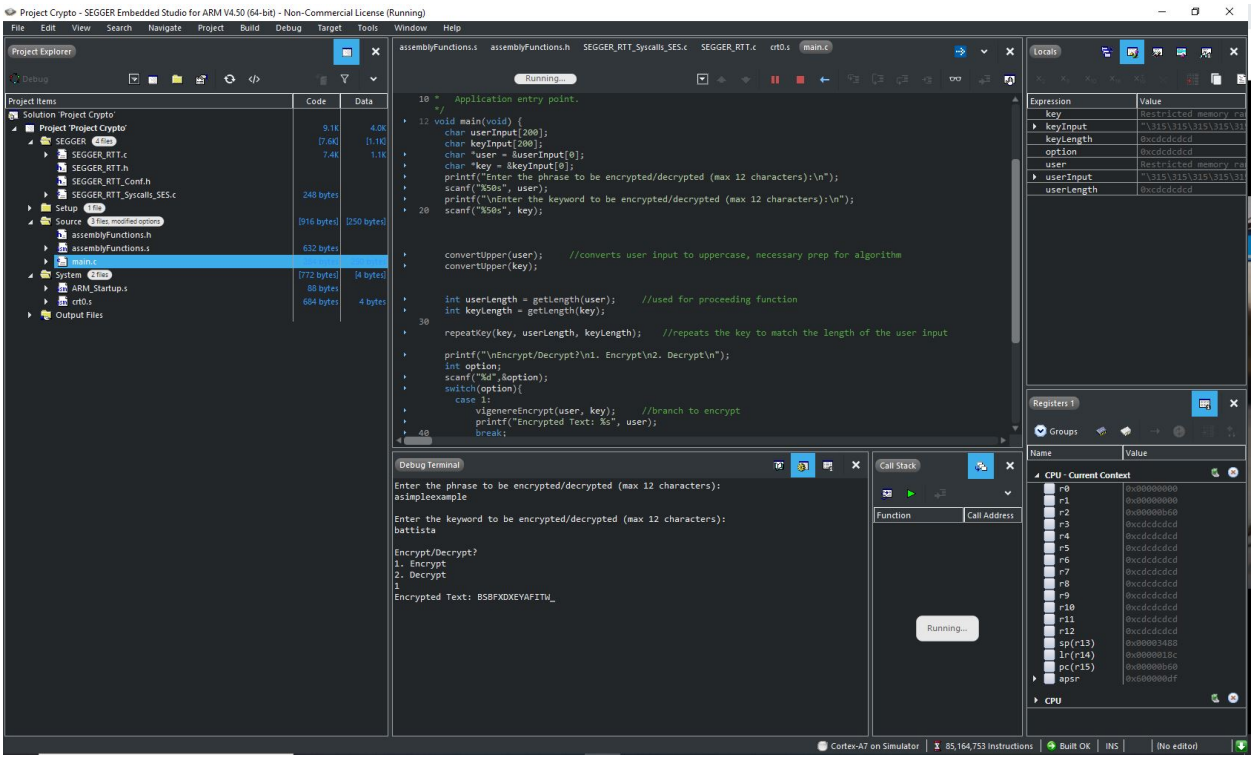
User Instructions:

- Upon running the program, the user will be prompted to type in the phrase to be encrypted/decrypted. This will be known as the "plaintext." (A known bug in this step is discussed in the project limitations section.)
- The user will then be prompted to type in a key phrase to encrypt/decrypt the plaintext/ciphertext. This can be of any size but if the user types in a key phrase that is longer than the plaintext/ciphertext, the key will be truncated to the length of the plaintext/ciphertext.
- The user will finally be prompted to type in a "1" or a "2" to encrypt or decrypt the previous phrase.
- If the user chooses to encrypt the data, the program returns the encrypted text which can be securely sent to another party along with the agreed-upon key.
- If the user chooses to decrypt the data, the program returns the decrypted text which can be used to decipher the text that was sent from another user using an agreed-upon key.

Test Results:

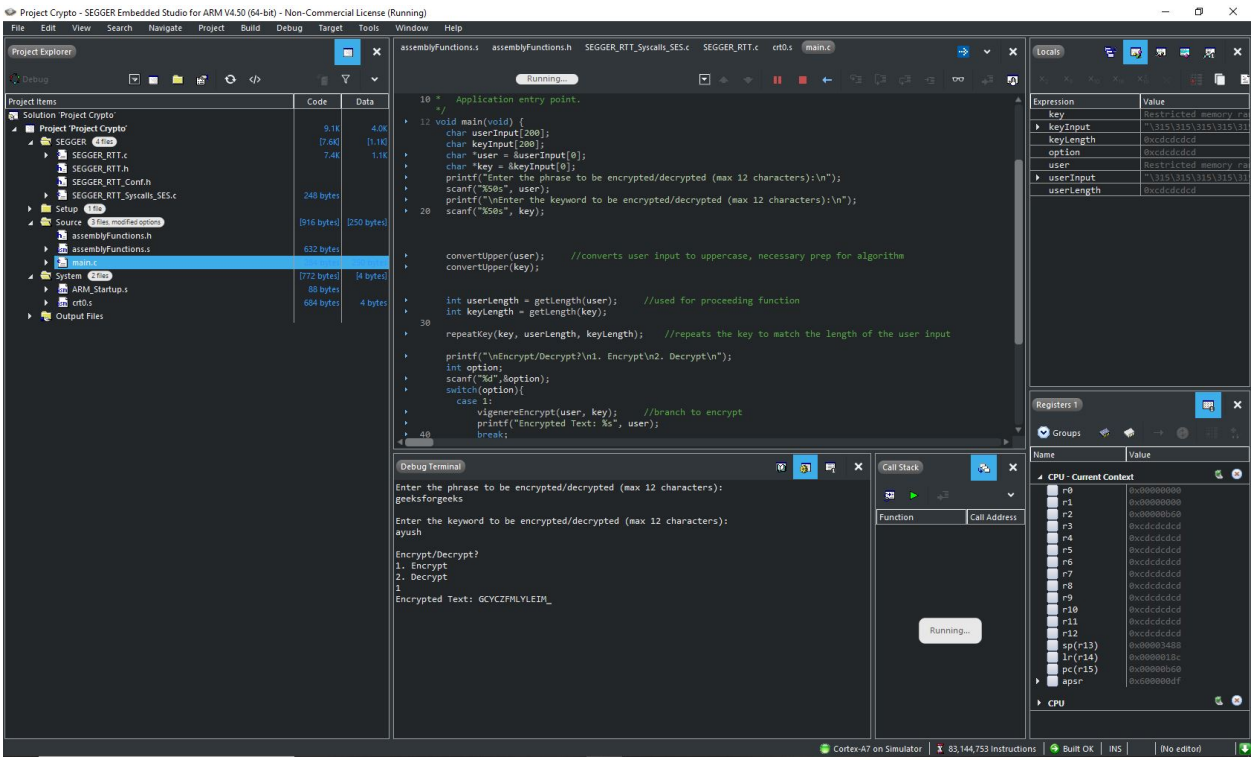
Throughout the programming process, I modeled my algorithm based on the Vigenere cipher encrypting/decrypting process on CryptoCorner, a non-profit website dedicated to educating the general public about popular cryptographic techniques. In their example, they used the phrase, "ASIMPLEEXAMPLE" paired with the keyword, "BATTISTA." Once the process was complete, the encrypted text should be "BSBFXDXEYAFITW," according to the article.[1] Below are the screenshots of the same example but performed by my program.

[1] Daniel Rodriguez Clark, "Vigenère Cipher," CryptoCorner, accessed April 3, 2020, https://crypto.interactive-maths.com/vigenegravere-cipher.html.

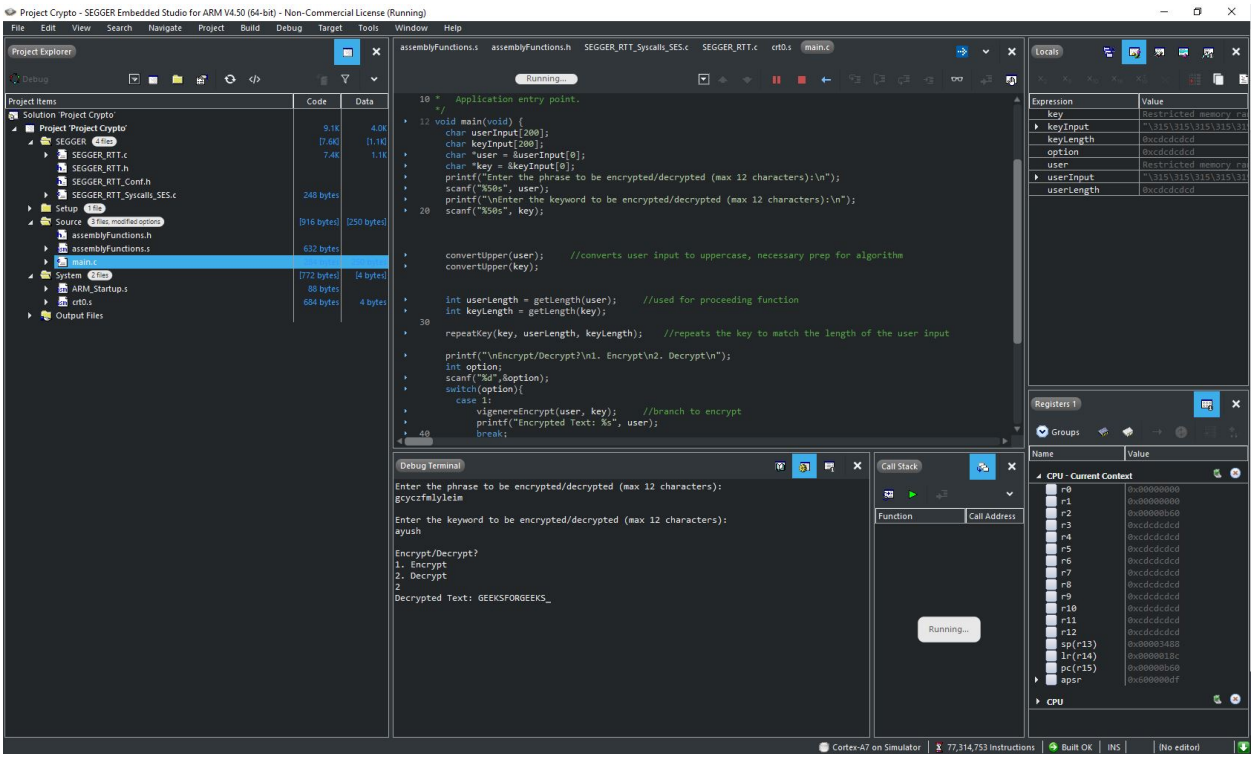


Another website, GeeksForGeeks, had a similar article explaining the process of encrypting and decrypting using a Vigenere cipher with an example. In their article, they used the phrase, "geeksforgeeks" paired with the keyword, "ayush" and obtained the ciphertext, "gcyczfmlyleim."[2] Below is a screenshot of the same example but run by my program, with no alterations from the previous test.

[2] Ayush Khanduri, "Vigenère Cipher," GeeksForGeeks, accessed April 3, 2020, https://www.geeksforgeeks.org/vigenere-cipher/.

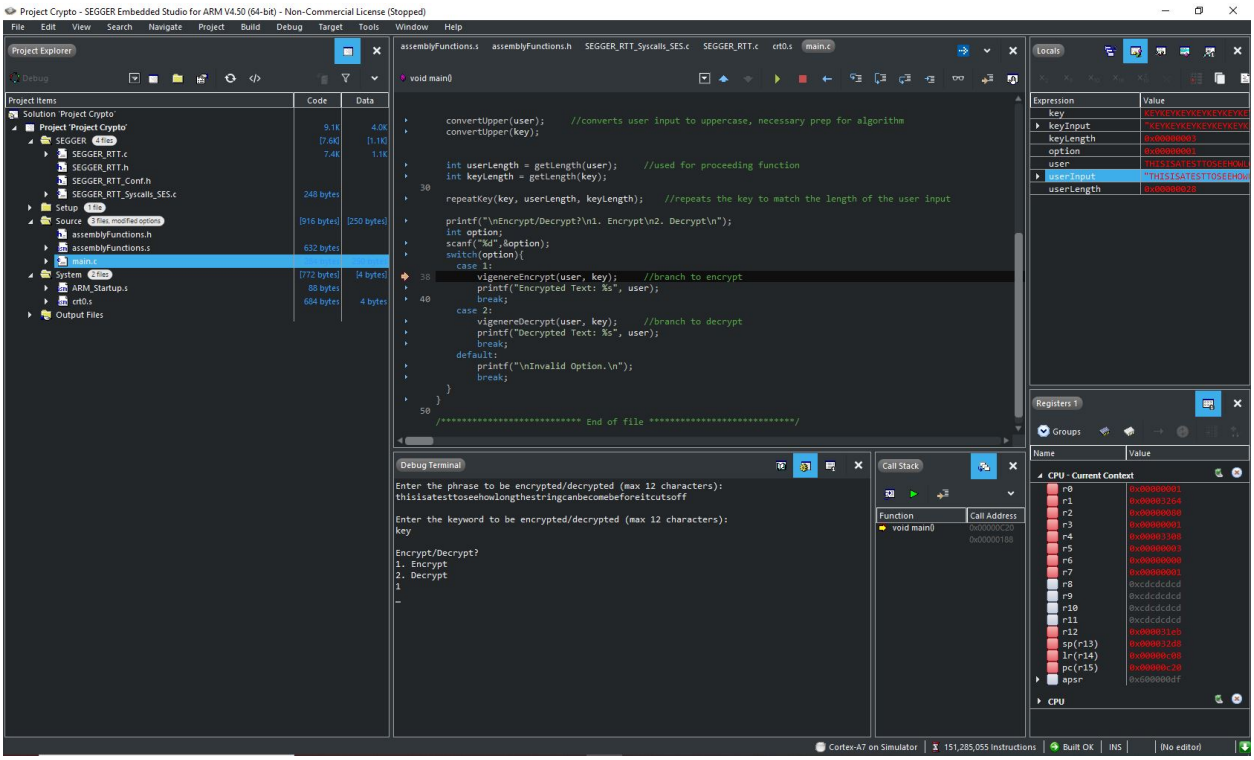


We can even apply this process in reverse. Below is the result of decrypting the text back to its original plain text form using my program.

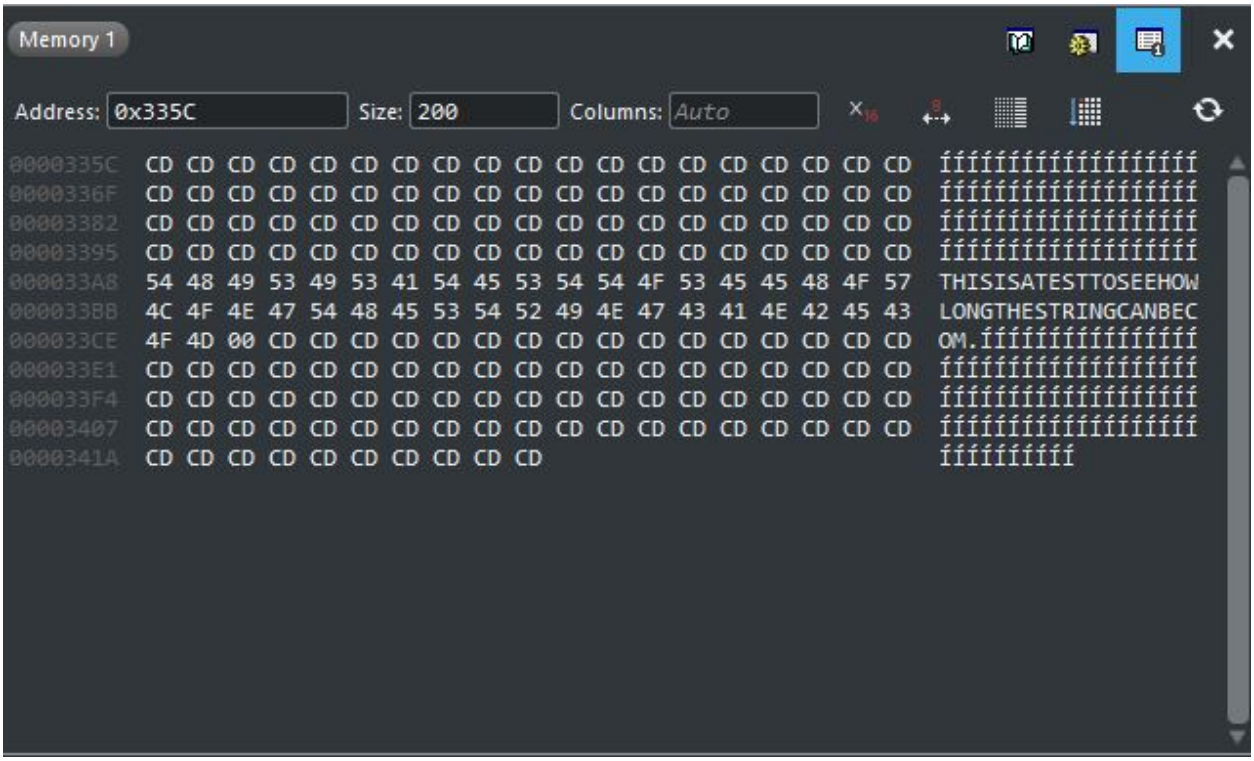


Bugs/Project Limitations:

Through my testing, I have found some found some limitations to my program. Firstly, the SEGGER Embedded Studio simulator had issues with pasting text using the Ctrl + V shortcut into the terminal. When I pasted text into the terminal instead of typing it in manually, the text would get truncated at irregular intervals. In memory, the beginning characters would be stored but the rest of the string would not show up and therefore would not be encrypted/decrypted. Below is an example of what happens when I paste the input phrase "thisisatesttoseehowlongthestringcanbecomebeforeitcutsoff" paired with the key phrase, "key."



In the terminal, the full paste can be seen. But upon closer inspection in memory, we see that the entire string was not stored.



This appears to be an issue with copy-pasting and the function that I used to capture the user input, scanf() in the Standard C library. Because of this, this issue is not an easy fix since that means we would need to change the C standard library.

Another limitation to this program is the inability to handle spaces. I have not coded my algorithm to correctly handle spaces and there seems to be an issue with the scanf() function in dealing with spaces as well. Whenever I type in a word followed by a space, the SEGGER simulator treats it as an “Enter” command and proceeds onto the next prompt before the user can finish the input phrase. This is a mild inconvenience that would require specific case-handling in all the functions that I have written for this program. I instead decided to focus my time on debugging critical errors rather than spending it fixing this issue.

Demonstration:

The link to a demonstration of my project in action is below.

https://youtu.be/6kigwjbq4ac

Improvements:

- Fix the space-bar issue as described in the Project Limitations section.
- Fix the copy-pasting issue as described in the Project Limitations section.
- Allow users to export/import mass data into a text file.
- Create a program to "guess" what the length of the key is using the Friedman test and applying frequency analysis.
- Create a web-based version of this program so that this tool can be easily accessed by anyone.

Conclusion:

This project utilized many concepts that we have learned throughout this course including how to write Assembly code, how to debug and use registers effectively, bit-manipulation, ASCII data manipulation, looping in Assembly, and memory storing/loading commands.

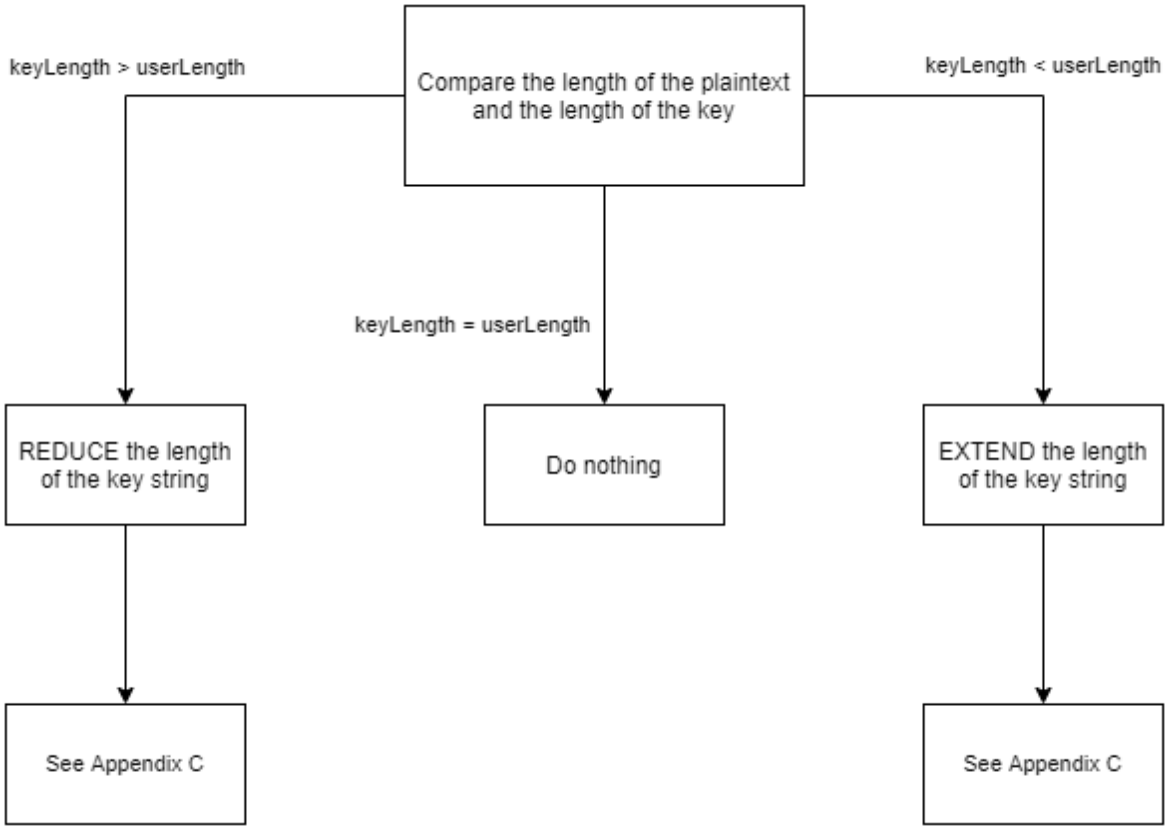
Hours spent:

- Researching ~ 2 hours
- Planning ~ 3 hours
- Coding ~ 4 hours
- Debugging ~ 6 hours
- Report ~ 5 hours

Total: ~20 hours

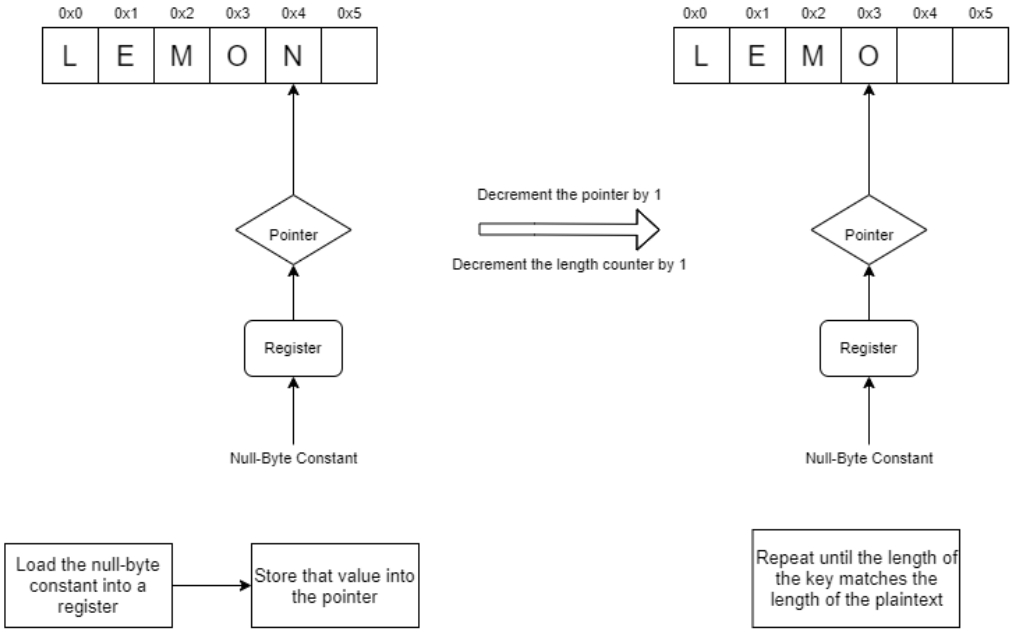
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Appendix A

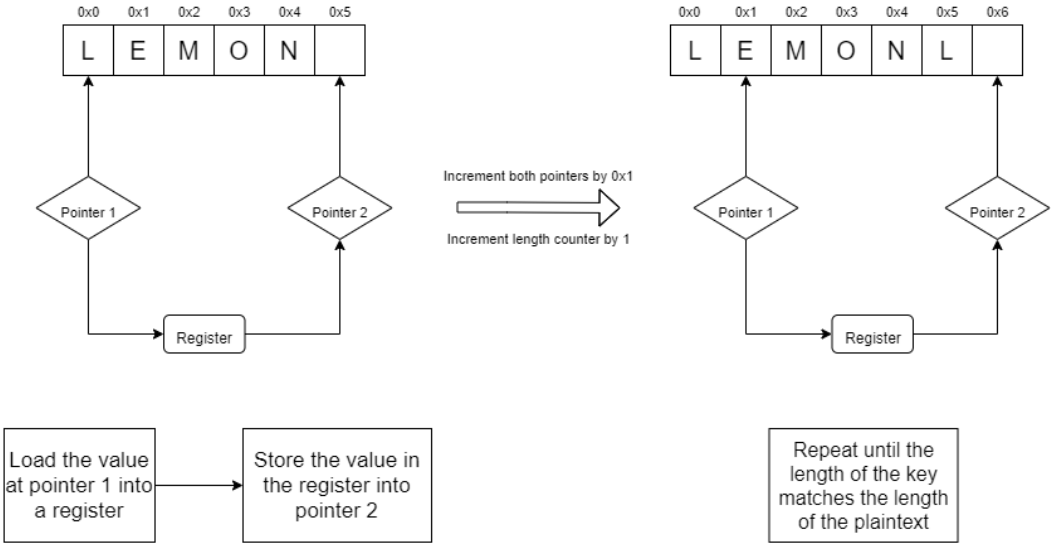

Compare the length of the plaintext and the length of the key
keyLength > userLength
keyLength = userLength
keyLength < userLength
REDUCE the length of the key string
Do nothing
EXTEND the length of the key string
See Appendix C
See Appendix C

Appendix B

Reducing the key:



Extending the Key:



Appendix C

References:

Bruden, Aiden A. & Forcinito, Mario A. “The Vigenère Cipher.” In *Cryptography, Information Theory, and Error-Correction: A Handbook for the 21st Century.* 21-22. Hoboken: John Wiley & Sons, 2011.

Clark, Daniel Rodriguez. “Vigenère Cipher.” CryptoCorner. Accessed April 3, 2020. https://crypto.interactive-maths.com/vigenegravere-cipher.html.

Khanduri, Ayush. “Vigenère Cipher.” GeeksForGeeks. Accessed April 3, 2020. https://www.geeksforgeeks.org/vigenere-cipher/.