COMPE-271

Ckick below to enter/change your Name and RedID

Giorgi Aptsiauri / 820968337

Final Project

VERNAM CIPHER

- I declare that all material in this assignment is my own work except where there is clear reference to the work of others.
- I have read, understood and agree to the SDSU Policy on Plagiarism and Cheating on the university website at http://go.sdsu.edu/student_affairs/srr/cheating-plagiarism.aspx , the syllabus and the student-teacher contract for the consequences of plagiarism, including both academic and punitive sanctions.

_____

Remark. By submitting this assignment report electronically, you are deemed to have signed the declaration above.*

5/6/2018

Vernam Cipher

1) There have been many cryptographic functions around with certain weaknesses except for one which has been mathematically proven as unbreakable and secure given certain prerequisites, Vernam cipher (proof). Vernam cipher is also called one-time pad because each message is encrypted with a truly random key of one-time pad of the same length as the message. The project which I present here implements Vernam Cipher with certain functions written both in C and x86 assembly language. The user interface, although a little messy, displays values computed by both C and assembly functions to make sure both work the same way. Functions ***written in assembly are Galois LFSR*** which generates pseudorandom sequence of numbers and ***Reverse Number function*** which reverses an int value passed to it.
   Here's how it works. For example, say, message to be encrypted is "Hello_World!":
   A) Any implementation of Vernam cipher would have to figure out the length because the key must be the same length as the message.
   B) So, say the function generated the key "H91"\dp&u@10" which is 12 symbols long just like the input message.
   C) When both message and key are in place, here's what happens:

```
Plain text  =      H  e  l  l  o  _  W  o  r  l  d  !
                   ^
key  =             H  9  1  "  \  d  p  &  u  !  1  0
                   -------------------------------------------------
Cipher text  =        \     N  3  ;  '  I     M  U
```

   As you can see, some characters of the cipher text are not in the displayable part of the ASCII table. For that reason, my function prints all characters in binary as well. So, key ^ message = cipher text, and, key ^ cipher text = message.
2) No special hardware was used. I developed the project solely in Code Blocks on Windows. I used debugger heavily.
3) What follows is the sequence of steps which achieves what's printed on screen:
   A) Getting a time value as a seed to Galois LFSR. Shift the value one time to the right to decrease the value otherwise it overflows when reversed and revere the number to make sure the most frequently updated bits of the time value are the ones applied to the Galois LFSR, otherwise, the same values would produce from the LFSR. Two different values of time are created: *currentTime* is reversed by C function called *currentTime*() and *currentTimeA* reversed by an assembly function called *reverseTimeValueA*().
   B) Get input message from the user with maximum of 100 characters. Message is printed in binary using "void *printStringAsBinary(char* s*)" function.
   C) C version of LFSR is called which is passed a pointer to a char array (*key*), *currentTime* and message length. Message length has significance for LFSR because it must produce sufficient amount of random byte sequences to make sure that at least one random byte corresponds to one byte (character) from the message. To make sure it happens, LFSR function has an if statement which makes sure sufficient amounts of bytes have been generated before breaking out of the loop and returning. After function returns, array **key** has all the hex values of the LFSR sequence of numbers as characters (NOTE: when XORING later, they need to be converted to real values and they will be).
   D) As in *C)*, assembly equivalent function of LFSR is called which is passed a pointer to a char array (*keyA*), *currentTimeA* and message length. After function returns, array *keyA* must have the same characters in it as the array *key*.

E) Finally, C *Vernam* function is called with arguments: pointer to *key, message, and cipher* arrays. *Vernam* function iterates message length times. At each iteration, it:
   a) Converts two hex symbols of *key* char array to the real value of it and normalized it. NOTE: normalization is not required and it actually decreases the reliability of this implementation, so line 109 from main.c can be freely commented out. I chose to normalize the value to ASCII displayable characters to visualize more as to what values are XORed.
   b) Each byte of the *key* value and message are XORed and two char arrays: *keyGenerated,* and *cipherGenerated* are created to store the values to print them later when loop is broken and encryption is done. *keyGenerated* will store ASCII symbols of the length of the message where each one is converted from two hex digits of *key* char array to a real value which, roughly speaking, is stored as one single value in *keyGenerated,* interpreted as an ASCII symbol.
   c) After XORing, *cipherGenerated* is also created which stores XORed ASCII symbols to display later, after all the values are encrypted.
   d) *keyGenerated* and *cipherGenerated* are printed to *stdout.*
   e) For demonstrational purposes, *Vernam* function also decrypts the cipher text and prints it to *stdout* as a final step of the *Vernam* function.

F) Assembly equivalent of *Vernam* function is called with arguments: *keyA, message, and cipher* arrays. It does everything *a)* through *e)* in *E).*

G) *Goto A)*

4) Source:

Main.c:

```
/*
   San Diego State University
   Giorgi Aptsiauri / 820968337
   Final Project for COMPE 271, Spring, 2018.

   Unbreakable Vernam Cipher using Galois LFSR.
*/

# include <stdint.h>
# include <stdio.h>
# include <string.h>
# include <time.h>
# include <stdlib.h>

extern int sprintfkeys(char* keyPtr, unsigned int LFSR_STATE);   // simplifies galoislfsrina() function a lot
extern int GaloisLFSRinC(unsigned int seedIN, unsigned int charCount, char key[]);
extern int galoislfsrina(unsigned int seedIN, unsigned int charCount, char key[]);
int Vernam(char key[], char message[], char cipher[]);
int reverseTimeValue (int num);
```

```
extern int reverseTimeValueA (int num);
void printBinary(char stringPtr[]);
void printCharAsBinary(char c);
void printStringAsBinary(char* s);

int main(void) {
   unsigned int currentTime = 0, currentTimeA = 0;
   unsigned int stringLen = 0;
   char key[202];     // key generated from C code goes here
   char keyA[202];    // key generated from Assembly code goes here
   char cipher[101];  // cipher text is written here
   char message[101]; // message to be encrypted

   while(1)
   {
      currentTime = (int)time(NULL) >> 1;  // just to make sure it fits within int
      currentTimeA = (int)time(NULL) >> 1; // just to make sure it fits within int

      /* currentTime value has to be reversed to accommodate "a more pseudorandom" sequence from
LFSR.
        The reason is that time(NULL) returns time in seconds, where the most frequently updated bit is
        the least significant bit(LSB). Since the taps into the LFSR are most significant bits (MSB), LFSR
would not
        consider currentTime's LSB into key generation and the same sequence of pseudorandom values
would pop up
        for a long time which is not the objective of my program. By reversing, this problem is fairly
solved for my
        application.*/
      currentTime = reverseTimeValue(currentTime);
      currentTimeA = reverseTimeValueA(currentTimeA);

      // get input
      do{
         printf("Enter a message to encrypt: ");
         fgets(message, 100, stdin);
         stringLen = strlen(message)-1;
         message[stringLen] = '\0';
      } while (stringLen > 100);

      printf("Message in binary: ");
      printStringAsBinary(message);

      printf("Generating the key and cipher...\n");
      if(GaloisLFSRinC(currentTime, stringLen, &key[0]) != 1)
         exit(0);
      printf("\nLFSR sequence by C LFSR function: %s\n", key);
```

```
        if(galoislfsrina(currentTimeA, stringLen, &keyA[0]) != 1)
            exit(0);
        printf("LFSR sequence by Assembly LFSR function: %s\n\n", keyA);

        printf("1) Generated by C LFSR\n\n");
        if(Vernam(&key[0], &message[0], &cipher[0]) != 1)
            exit(0);
        printf("\n");

        printf("2) Generated by Assembly LFSR\n\n");
        if(Vernam(&keyA[0], &message[0], &cipher[0]) != 1)
            exit(0);
        printf("\n");
    }

    return 0;
}

int Vernam(char key[], char message[], char cipher[])
{
    char bufferMessageByte[2]; // one byte for symbol, one for \0
    char bufferKeyByte[3]; // two bytes for symbol, one for \0
    char keyGenerated[strlen(message)+1];
    char cipherGenerated[strlen(message)+1];
    char reconstructedMessage[strlen(message)+1];
    char* messagePtr = &message[0];
    char* keyPtr = &key[0];
    int stingLen = strlen(messagePtr);
    unsigned char keySymbolAscii;
    unsigned char messageSymbolAscii;
    unsigned char cipherByte;
    int i;
    for(i = 0; i < stingLen; i++)
    {
        bufferMessageByte[0] = *messagePtr++;
        bufferMessageByte[1] = '\0';

        bufferKeyByte[0] = *keyPtr++;
        bufferKeyByte[1] = *keyPtr++;
        bufferKeyByte[2] = '\0';

        keySymbolAscii = (unsigned char)strtol(bufferKeyByte, NULL, 16);
        /* normalize key symbol to ASCII displayable symbols, i.e. decimal 33 - 126.
           the next line can be commented to allow all values, decimal 0 - 255, to be used during encryption
as a key.
```

```
        Even though each byte of key is normalized, final cipher text will not be normalized.*/

        keySymbolAscii = 33 + (keySymbolAscii%93);
        keyGenerated[i] = (unsigned char)keySymbolAscii; // save to final key array
        messageSymbolAscii = (unsigned char)bufferMessageByte[0];

        // XORING
        cipherByte = keySymbolAscii ^ messageSymbolAscii;
        cipherGenerated[i] = (unsigned char)cipherByte; // save cipher text
    }

    keyGenerated[i] = '\0';
    cipherGenerated[i] = '\0';

    printf("key in ASCII: %s\n", keyGenerated);
    printf("key in binary: ");
    printStringAsBinary(keyGenerated);
    printf("\ncipher text: %s\n", cipherGenerated);
    printf("cipher text in binary: ");
    printStringAsBinary(cipherGenerated);

    // Decryption
    printf("Now let's decrypt the cipher text...\n");
    messagePtr = &message[0];
    keyPtr = &keyGenerated[0];
    for(i = 0; i < stingLen; i++)
    {
        reconstructedMessage[i] = ((unsigned char)keyPtr[i] ^ (unsigned char)cipherGenerated[i]);
    }
    reconstructedMessage[i] = '\0';
    printf("Decrypted message: %s\n", reconstructedMessage);

    return 1;
}

int reverseTimeValue (int num)
{
    int reverseNum = 0;
    while(num > 0)
    {
        reverseNum = reverseNum*10 + num%10;
        num = num/10;
    }
    return reverseNum;
}
```

```
int sprintfkeys(char* keyPtr, unsigned int LFSR_STATE)
{
   return sprintf(keyPtr , "%x", LFSR_STATE);
}

void printBinary(char stringPtr[])
{
   char* strPtr = &stringPtr[0];
   while(*strPtr)
   {

      printf(" ");
   }
   printf("\n");
}

void printCharAsBinary(char c) {
   int i;
   for(i = 0; i < 8; i++){
      printf("%d", !!((c << i) & 0x80));
   }
}

void printStringAsBinary(char* s){
   for(; *s; s++){
      printCharAsBinary(*s);
   printf(" ");
   }
   printf("\n");
}
```

GaloisLFSRinC.c :

```
# include <stdint.h>
# include <stdio.h>

extern int GaloisLFSRinC(unsigned int seedIN, unsigned int charCount, char key[])
{
   unsigned int startingState = seedIN;  /* this is a starting state (seed) of the LFSR */
   unsigned int LFSR_STATE = seedIN;  /* LFSR_STATE will be updated per Galois LFSR technique */
   unsigned period = 0;  /* measure period */
   char* keyPtr = key;

   do {
      unsigned lsb = LFSR_STATE & 1;   /* generating least significant bit (LSB) (i.e. output). */
```

```
        LFSR_STATE >>= 1;               /* apply shift. */
        if (lsb)                    /* taps are applied iff LSB is 1. */
           LFSR_STATE ^= 0XA3000000u;   /* taps are: 32, 30, 26, 25. binary equivalent: 1010 0011 0000
0000 0000 0000 0000 0000*/
        ++period;
        keyPtr += sprintfkeys(keyPtr, LFSR_STATE);
        if(4*period >= charCount)
           break;
    } while (LFSR_STATE != startingState);
    return 1;
}
```

Galoislfsrina.s:

```
.global _galoislfsrina
.extern sprintfkeys

_galoislfsrina:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ebx   # GET seedIN
    movl %ebx, %esi      # make a copy of seedIN
    movl 12(%ebp), %ecx  # GET charCount
    movl 16(%ebp), %edx  # GET &key[0]
    movl $0, %edi
    loop:
        andl $1, %ebx  # generate LSB (output)
        shrl $1, %esi  # apply shift (LFSR_STATE)
        cmpl $0, %ebx  # compare LSB : 0
        jle noMask     # jump if LSB is 1
        xorl $0xA3000000, %esi # apply taps
        noMask:
        incl %edi          # period++
        pushl %ecx
        pushl %esi         # pass LFSR_STATE as arg2
        pushl %edx         # pass keyPtr as arg1
        call _sprintfkeys   # sprintfkeys(keyPtr, LFSR_STATE)
        popl %edx          # clean up the stack
        popl %esi          # ...
        popl %ecx
        addl %eax, %edx # keyPtr += sprintfKeys(keyPtr, LFSR_STATE)
        movl %edi, %ebx # ...
        shll $2, %ebx   # 4*period
        cmpl %ecx, %ebx # compare 4*period : charCount
        jge end         # break if 4*period >= charCount
```

```
        movl %esi, %ebx # restore LFSR_STATE
        cmpl %esi, 16(%ebp) # compare startingState : LFSR_STATE
        jne loop
    end:
    movl $1, %eax # return 1
    movl %ebp, %esp
    popl %ebp
    ret
```

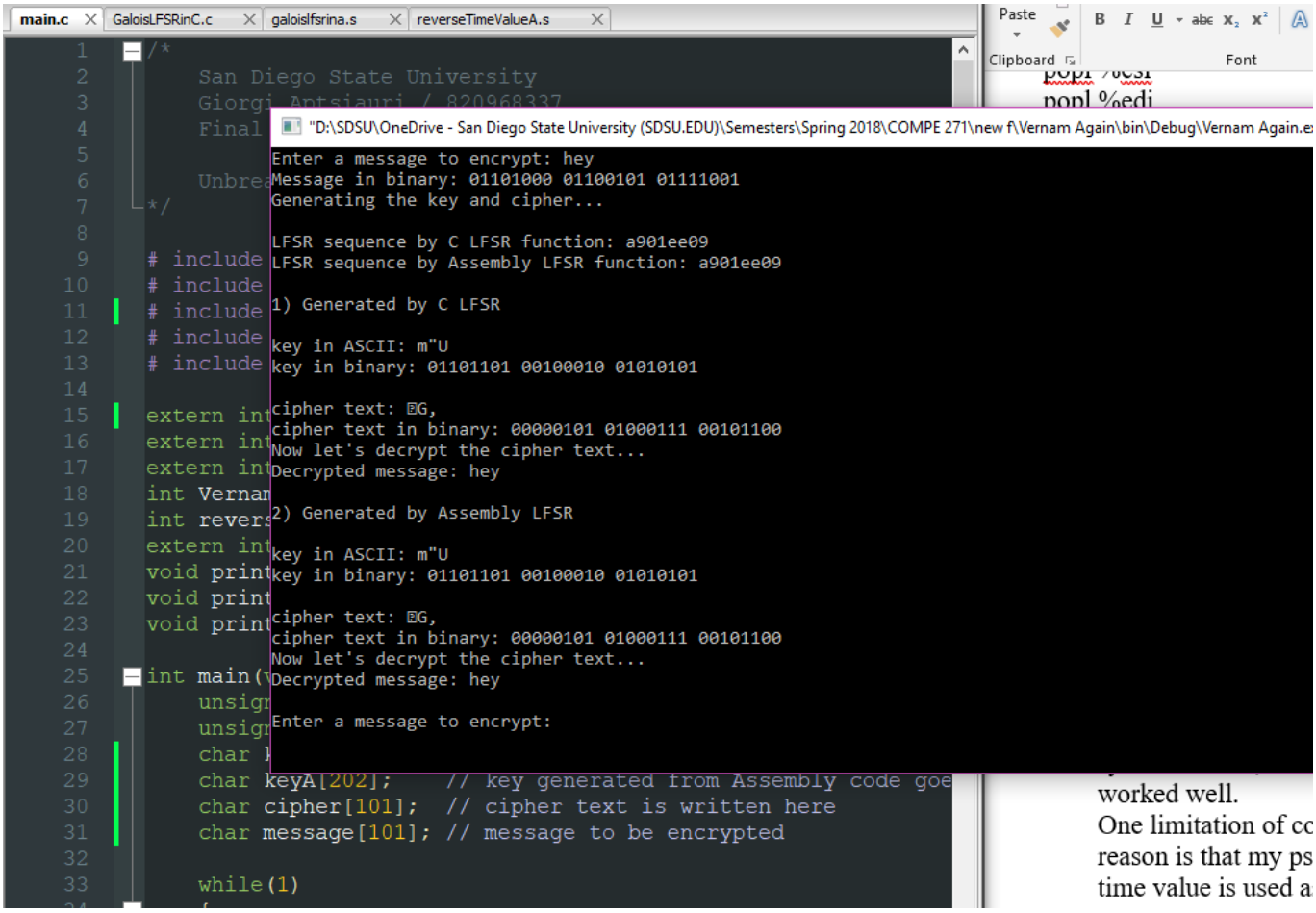
reverseTimeValueA.s :

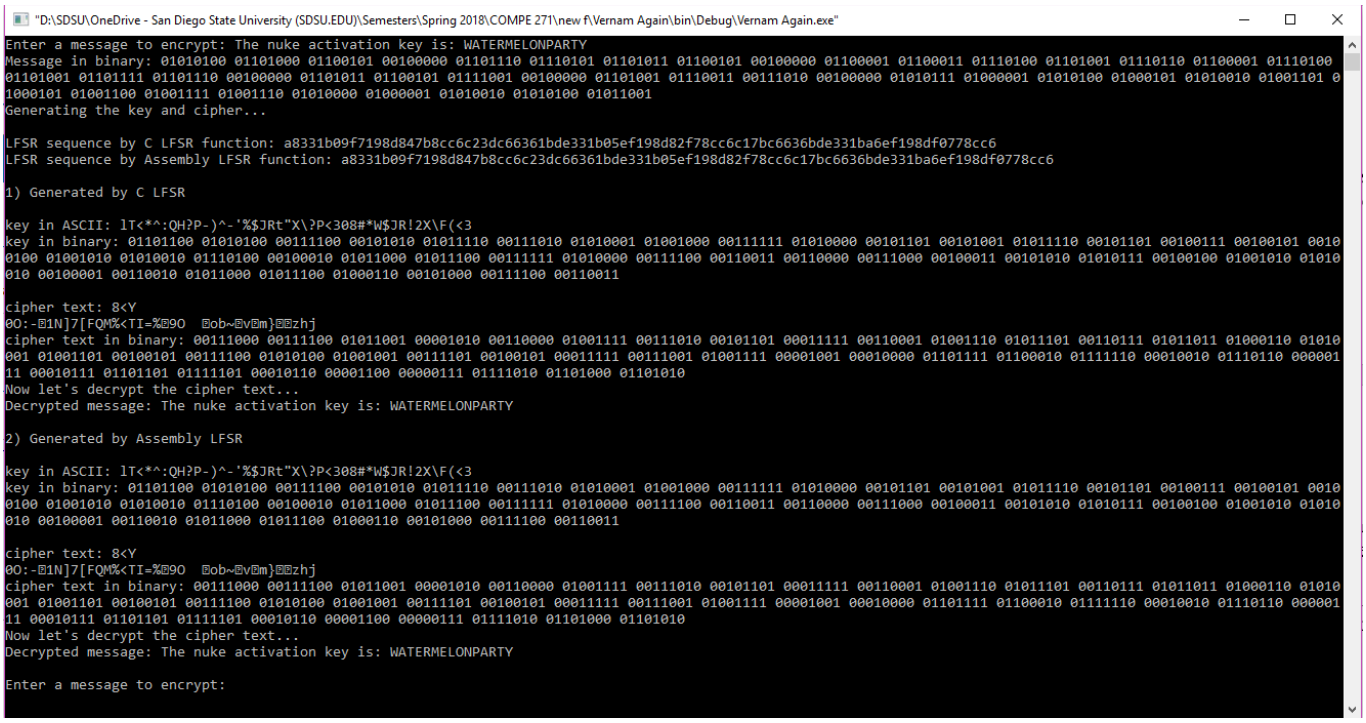
```
.global _reverseTimeValueA

_reverseTimeValueA:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx    # save registers
    pushl %esi
    pushl %edi
    movl 8(%ebp), %eax # GET num
    xorl %edi, %edi    # reverseNum = 0
    cmpl $0, %eax      # compare num : 0
    jle end            # exit if num <= 0
    loop:
        movl $10, %esi    # 10
        imull %edi, %esi  # reverseNum*10
        movl $10, %ebx    # 10
        sarl $31, %edx
        idivl %ebx        # num%10
        addl %esi, %edx   # reverseNum*10 + num%10
        movl %edx, %edi   # reverseNum = reverseNum*10 + num%10
        # num = num/10 is implied (in %eax)
        cmpl $0, %eax     # compare num : 0
        jg loop           # jump if num > 0
    end:
    movl %edi, %eax
    popl %ebx    # restore registers
    popl %esi
    popl %edi
    movl %ebp, %esp
    popl %ebp
    ret
```

5) No special items were used.

6) Tests, results, limitations. The first tests I ran involved entering long messages. First, LFSR function in assembly used to crash, I fixed it by doubling the capacity of keyA char array getting it 200 instead of 100, because each character corresponds to two hex digits written in ASCII symbols. Now, it works quite well for relatively large messages. Apart from large functions, it worked well.
   One limitation of course, is that my implementation is not as unbreakable as the ideal one. The reason is that my pseudo random sequence of numbers can be predicted if a person finds out that time value is used as the seed into the LFSR.
   For design validation, I heavily used debugger to keep track of all the values.

7) Screen captures:





as message size increases, the output becomes a little more messy.

8) ***If I had more time***, I would rewrite the entire *Vernam* function to assembly as well.
9) ***Time spent:*** well over 40 hours including research.
10) ***Conclusion:***

The project seemed simpler than it was. I did not expect I would be calling other procedures from assembly function, but I had to do it and I did it, although a few tricks simplified the assembly code.

References

https://stackoverflow.com/questions/111928/is-there-a-printf-converter-to-print-in-binary-format

https://stackoverflow.com/questions/9794416/x86-assembly-multiply-and-divide-instruction-operands-16-bit-and-higher