Jonathan Child

Kevin Caton-Largent

CSE 178-02L

Mukesh Singhal

November 19, 2012

**AES Encryption Implementation**

AESencrypt.m  
  
For the implementation for our main function we named it AESencrypt.m. This function takes in two parameters, an Hexadecimal plain text message, and a 16 byte Hexadecimal key. The first thing we did is we took our key, and converted it into decimal format, and make it into a four by four matrix. After converting it we put it into a key expansion function that will take the 16 byte key and expand it into a 176 byte key.  
  
Once the key expansion function returned the newly expanded key, we took the first 4 words of the key and did a bitxor on the plain text. Doing a bitxor on the plain text is how we implemented our add round key part of AES implementation.  
  
After adding the first round key, we went into our nine rounds of aes encryption where we did nine rounds of byte substitution, shiftrows, mixcolumns, and then an add round key at the end. All these functions were done in another part of our code, and were just called upon when we needed to do the functions.  
  
At the tenth round we did our last byte substitution, shiftrows, and add round key. After all that we converted the now encrypted text back into hexadecimal and sent it to the output where the user could retrieve it.  
  
**key\_exp.m**:  
        In our key\_exp function we first created a cell array of 256 bytes containing the rcon matrix values. We then made sure the values of the rcon were read as decimal. Using the first 16 elements of the key we stored those values into w which contained our words.  The following words would be determined by xoring the previous word by the next word until 44 words were created. If there were no previous word to xor,  we did a series of operations to form  the word value used for the bitwise xor operation. The first operation was a rotation on the word using a left shift by one.  The next operation was a byte substitution on that word rotation. Then using the corresponding round key, determined from the rcon table, we bitwise xored the  byte substitution  by the round key. After expanding the key we stored the result in output.  
**rotWord.m:**  
        The rotWord function is a simple function that  does a circular left shift by one on the word being passed in.  
  
  
**subByte.m:**  
        In subByte function we stored the values of the 16x16 s-box in a cell array called table. We checked for 2 cases one for when subBytes was used for key expansion and the other was when subBytes was used in the normal round processing. The first case in our code was implemented for key expansion and this dealt with reshaping the text to be a 2x2 matrix and making sure that the table values were represented in decimal format. Iterating through the 2x2 matrix we would do a table lookup using the left 4-bits as the row index and the right 4-bits as the column index. In the second case of the byte substitution we didn't reshape the text. We made sure the table was in decimal format and applied the same operation as in the first case. The result of the substitution would be then stored in the output variable.   
  
  
shiftRows.m  
  
Our shift rows function is very simple, it takes in the text message that is currently being encrypted. How we implemented the function is we first made a temp variable where we would store our values for the shift row would do. So for the first row, we just inputted the row as is, because in the AES encryption, thats what it calls for. The next row was a circular shift to the left by one, so we just inputted the values one by one into the temporary variable. The next row was a circular shift left by two. And the last row was a circular shift right by one. After putting all the variable into the temporary variable the function return the text that had gone through shift rows.  
  
mixCols.m  
  
For the mix columns function it took in a value of the text file that was in the process of being encrypted. what this function does is it takes the state and put it into a four loop. Each time it runs through the four loop it would to a matrix multiply with a xor in it. So it takes the first element of the first column and multiplied it by the shift matrix’s first element in the first row. it would then xor it with the next element in the texts matrix’s column by the next element in the shift matrix’s row, and so on. We did not use the shift matrix that we had created, but left it in if it was desired to be put in at a later time. For now we just statically put the numbers in as we multiplied and xor-ed the elements in the matrices.

Test Cases:

TEST 1

key :'0f' '15' '71' 'c9' '47' 'd9' 'e8' '59' '0c' 'b7' 'ad' 'd6' 'af' '7f' '67' '98'

input: 6b c1 be e2 2e 40 9f 96 ad 2b 41 7b e6 6c 37 10

output: F4 28 51 8F E7 97 DB 33 F2 44 CC BC D6 97 79 96

TEST 2:

key f6 cc 34 cd c5 55 c5 41 82 54 26 02 03 ad 3e cd

input ed b5 19 ba b4 cb 7b ac bd 3a 33 ac 03 4f a1 06

output: 6D 0B 0B 08 63 A9 B0 48 91 C5 06 0B 78 E6 22 B6