Alec Mendiola

11/24/19

Lab Assignment #1 Write-Up

For this assignment we were to implement the 128-bit version of the Advanced Encryption System (AES) algorithm. The algorithm, in very simplistic terms, takes in a message and a key then outputs an encrypted message. This algorithm has 4 steps which are key expansion, initial round key addition, 9 rounds of sub-steps, and a final round comprising of most of the previous step's sub-steps. The sub-steps for the third step are sub bytes, shift rows, mix columns, and add round key. The last step is similar in that it is sub bytes, shift rows, but it lacks the mix columns sub-step, instead it ends with an add round key. I will structure this write up into explaining my implementation at each step and sub-step but taking a brief aside explaining my main function then concluding by displaying the two test cases, their results, and how I concluded that the tests were correct.

The main function simply serves as a conduit for the user to select from one the two test cases, it is a Boolean statement that sets the character array to either "Mukesh is great!" or "Tralfamadorian ." either way the message is 16-bytes or 128-bits as is stated in the specification sheet. Independent of which choice the user selects, the program then calls aesPreKeySet(msg) which is an additional function, made in an effort to reuse code, that takes the message selected by the user and sends it to the aesEncrypt(msg,key) function but only after the aesEncrypt function generates the 16-byte key for the encryption method. The key was selected by a random number generator that selects numbers between 0 and 100, underneath this section in the code I have provided the key in hexadecimal so you may easier check the results. This function also prints out the encrypted message at the program's end.

With the message selected, the key generated, and aesEncrypt() invoked we now need to create a state character array that gets the original message copied to it so we can change the state without changing the original message. We also create a variable that will dictate how many rounds AES goes for, we set it to 9 but only because the final round is different, and then create another character array that we will use to expand our key. For our key expansion function we put our original key into the expanded key array and use a while loop that will run until our key is at our desired amount, 176 bytes. In this function are working in 4-byte increments which we then send to our key scheduler. This scheduler has 3 steps, rotating left, substitution with S-Box, and Rcon substitution. The rotating left is simple enough in that we move our second value of the 4 to the first position and so on until we have moved what was the value in our first position to the fourth position. For the substitution box we use a look up table, sourced from Wikipedia, and replace the values of our array with that of the those where their values would lie in the substitution box table. In the last step of the key scheduler we take every fourth bit and XOR add it from the place in the rcon lookup table, also sourced from Wikipedia, where our iterator, which is changed in the keyExpand function, is currently pointing.

The next step in our encryption is the initial round key addition which is invoked by addRdKey. This function's input is the current state our message and the round key. In this step we take every byte of the current state and XOR add it to the block of the round key. We accomplish this by utilizing a for loop that will ensure that all 16 bytes are modified and with a ^= operator to rdKey[iterator].

Following the initial round key step, we move to the 9 rounds of sub-steps. To implement the 9 steps we use a for loop iterating from 0 to 8 running the functions in the order of subByte, shiftRow, mixColumn, and addRdKey. The sub byte function is relatively simple with the use of our S-Box lookup table. We use a for loop to set all 16 bytes of our current state to that of where in the S-Box that value links to.

The next sub-step is that of Shift Row where the last three rows of the current state are shifted by 1, 2, and 3 spaces respectively. To accomplish this task we create a temporary character array that will hold our values in their updated positions until the end of the shifts where we will then set the current state to that of the shifted rows. As stated before, the first row is not changed, the second row is shifted by 1 to the left, the third row is shifted 2 to the left, and the final row is shifted 3 to the left. After all of our values are correctly set we use a for loop to set our state array equal to that of our temporary array at all 16 indices.

Continuing through the sub-steps, the mix columns sub-step is where we mix the columns of the current state to combine the four bytes in each column. In an effort to cut down on math and to not have to in-line implement Galois multiplication in the code we use 2 look up tables, sourced from Wikipedia, that are pre-calculated to perform byte multiplication by 2 and 3. Again, we create a temporary character array that we will hold our updated values in until ready to update the current state. The code follows the equation for mix columns of $b_0 = 2a_0 + 3a_1 + 1a_2 + 1a_3$, $b_1 = 1a_0 + 2a_1 + 3a_2 + 1a_3$, $b_2 = 1a_0 + 1a_1 + 2a_2 + 3a_3$, $b_3 = 3a_0 + 1a_1 + 1a_2 + 2a_3$ but uses the look up tables of that state value at that position, either multiplied by 2 or 3, to perform the multiplication and then for addition in the equation uses XOR.

The last sub-step is to add the round key which we do in the same way as before by calling the function, which follows the same steps, but instead of just inputting the expanded key into the function we now need to iterate it for the current position which we do by multiplying 16 then by the iterator + 1.

For our 10th and final round we do the same as above expect for the exclusion of the mix columns sub-step so our function calls are subByte, shiftRow, mixColumn, and addRdKey with the only notable difference in this round being that we again need to update what we input into addRdKey by taking our expandedKey and adding 160 to it to put it at the 176-bytes we need it to be at.

Finally, we output the encrypted message, printing it in hexadecimal, and can verify whether it is correct. As previously stated, for the two test cases I used the phrases “Mukesh is the best!” and “Tralfamadorian .” I chose these two phrases as they are 16-bytes long, as stated is the goal in the prompt, and feature a mixture of uppercase, lowercase, and special characters. After getting the output I put them into a reputable website, http://aes.online-domain-tools.com/, in their encrypted forms to see if they would output correctly decrypted, they did, and again in their unencrypted forms to see if they would they be the same when encrypted on their website as they were in my code, and again they were. I believe this is valid reasoning for the code’s correctness. In the folder I have included screenshots of my code’s output next to the website’s verification and if you would like to test the cases yourself select 0 at the start of the code for “Mukesh is the best!” and 1 for “Tralfamadorian .”