Homework 3

Bellew, Nathan D

Problem 1

A. I notated the progam very well so hopefully a quick look will help. It has two core functions, one for encryption of the DES which follows the standard pattern using some functions that convert between hex, dec, and binary. The other function creates a list of all round keys that will be used to encrypt at each round.

B. They are found in textfiles/Bellew_output_?.txt

C. I created a program called GenerateInputs.py that does this.

D. 1.999906301498413 seconds Timed using Python's time function so its accurate.

E. Year 4,582,202 : Week 27 : Hour 10 : Min 41 : Sec 6 So i took the amount of time it took to perform 1k encryptions and because the DES decryption is the same process but reversed I assumed the time for decryptions would be the same. Then i divided 1000 by the time in seconds. That gives me decryptions/second. Then I divided 2^56 (number of possible keys) by my decrypts per second and got a huge number of seconds. From there i converted seconds into minutes into hours into weeks into years. So it would take me Year 4,582,202 : Week 27 : Hour 10 : Min 41 : Sec 6 to brute force DES. Also to improve my time i encorporated python's version of strings (multiprocesses) so i was able to use all of my cores, previously it took 12 million years to complete. So i chunked off 8 million years!

Problem 2

A.

To derive K_1 you must first change the hex key into binary, then use the Key Parity permutation to change the key into 56 bits, then separate the into halves, circular left shift both sides and then recombine the key, after recombining the key you can use the compression permutation on the key to compress the key into 48 bits. Creating K_1. First you must change the key 89ABCDEF01234567 into Binary which has been provided as `1000100110101011110011011110111100000001001000110100010101100111`. From here the key needs to be permuted following the Key Parity permutation set as described from the supplementary material as

```
[57, 49, 41, 33, 25, 17, 9,
1, 58, 50, 42, 34, 26, 18,
10, 2, 59, 51, 43, 35, 27,
19, 11, 3, 60, 52, 44, 36,
63, 55, 47, 39, 31, 23, 15,
7, 62, 54, 46, 38, 30, 22,
14, 6, 61, 53, 45, 37, 29,
21, 13, 5, 28, 20, 12, 4 ]
```

Once permuted the key becomes: K_L = `0000 1111 1100 1100 1010 1010 0000` K_R =`1010 1010 1100 1100 0000 1111 0000` These are written seperately because the key needs to be split in half creating a left and a right side. these sides are each shifted to the left using a circular left shift. They are shifted a single time because the normal DES has the shift occur once on the first run and then twice on the next few. Because we are only doing a single round we only need the first shift which is only a single shift. After shifting the values become: $K_{Lshifted}$ = `0001 1111 1001 1001 0101 0100 0000` $K_{Rshifted}$ = `1010 1010 1100 1100 0000 1111 0000` Now that the two have been shifted the key sides can be recombined and permuted. The permutation will compress the now 56bit key into a 48 bit key. K_{P48} = `0000 0100 0100 0011 0111 0110 1001 0011 1001 0101 1000 1101` Which when converted to Hex becomes K_{P48} = `84 43 76 93 95 8B`.

B.

In order to derive the initial Left and Right sides of the cipher the plaintext must first be permuted through the initial Permutation or IP, then the result of that initial Permutation can be split in half creating the left and right sides. IP =

```
[58, 50, 42, 34, 26, 18, 10, 2,
60, 52, 44, 36, 28, 20, 12, 4,
62, 54, 46, 38, 30, 22, 14, 6,
64, 56, 48, 40, 32, 24, 16, 8,
57, 49, 41, 33, 25, 17, 9, 1,
59, 51, 43, 35, 27, 19, 11, 3,
61, 53, 45, 37, 29, 21, 13, 5,
63, 55, 47, 39, 31, 23, 15, 7]
```

So taking the Initial Permutation of the original line (do not need to convert to binary since its the same as the key and this was done already) of `1000 1001 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111` becomes: L_0 = `1100 1100 0000 0000 1100 1100 1111 1111` R_0 = `0000 1111 1010 1010 0000 1111 1010 1010`

C.

This one is easier than it sounds in order to Expand R_0 to get $E[R_0]$ you must use the Expansion function which is described as:

```
[32, 1 , 2 , 3 , 4 , 5 , 4 , 5,
6 , 7 , 8 , 9 , 8 , 9 , 10, 11,
12, 13, 12, 13, 14, 15, 16, 17,
16, 17, 18, 19, 20, 21, 20, 21,
22, 23, 24, 25, 24, 25, 26, 27,
28, 29, 28, 29, 30, 31, 32, 1 ]
```

After applying this function to only R_0 the result will modify the 32bit right side to 48 bits so it can be xored. Applying the function the result becomes: $R_{0expanded}$ = `0000 0101 1111 1101 0101 0100 0000 0101 1111 1101 0101 0100`

D.

Calculate $A = E[R_0] \oplus K_1$. Simply take the Key that was calculated earlier and xor it with the new expanded right side: K_{P48} = `0000 0100 0100 0011 0111 0110 1001 0011 1001 0101 1000 1101` $R_{0expanded}$ = `0000 0101 1111 1101 0101 0100 0000 0101 1111 1101 0101 0100` XOR Results = `0000 0001 1011 1110 0010 0010 1001 0110 0110 1000 1101 1001`

(They lined up way better on my markdown sheet than my pdf!)

E.

I ordered the XOR results above into groupings of 4 because it was easier for me to calcuate the XOR that way. But now i have to take that and rearrange it into 6s then apply the s-box. I could include every single s-box (all 8 of them) but i think you know what they look like. The way to apply the s-box (or what i did) is to take each of the 6 bit sections and apply them to a different box. Maintaining order from left to right.

Group into 6: `000000 011011 111000 100010 100101 100110 100011 011001` Apply S-box, the results: S-Box = `1110 1001 0101 0110 1100 0101 1011 0000`

F.

Take the results from above and put them into a single binary line, I will write it but it should be relatively clear how i determined the answer. `11101001010101101100010110110000` G.

This permutation is the straight permutation, it is not part of the initial nor the final. it instead will permute a 32 bit result which is necessary for the final XOR of the round.

Straight Permutation:

```
[ 16,  7, 20, 21,
29, 12, 28, 17,
1, 15, 23, 26,
5, 18, 31, 10,
2,  8, 24, 14,
32, 27,  3,  9,
19, 13, 30,  6,
22, 11,  4, 25 ]
```

The result of the permutation: `0000 0111 1100 1101 1111 0110 0000 1001`

H.

Calculate $R_1 = P(B) \oplus L_0$ Same as last XOR but with different values: `1100 1100 0000 0000 1100 1100 1111 1111` L_0 `0000 0111 1100 1101 1111 0110 0000 1001` $P(B)$ `1100 1011 1100 1101 0011 1010 1111 0110` XOR

The XOR becomes the new Right side or R_1 and the old right becomes the new left or L_1 so $L_1 =$ `0000 1111 1010 1010 0000 1111 1010 1010` $R_1 =$ `1100 1011 1100 1101 0011 1010 1111 0110`

This is important in the last step.

I.

Now to get the ciphertext I must first combine L_1 and R_1 and then with the new combination run that through the inverse of the initial permutation or IP^{-1}

$IP^{-1} =$ `1110 0100 1101 1111 0110 0110 1111 1101 0000 1010 0001 1011 1010 0010 1011 0011`

Which, after converting to Hex, becomes: `E4DF66FD0A1A2B3`

Problem 3

TABLES DID NOT LOAD PROPERLY

please see images for corresponding table sorry.

A. Addition: | + | 0 | 1 | 2 | 3 | 4 | |—|—|—|—|—|—| | **0** | **0** | 1 | 2 | 3 | 4 | | **1** | 1 | 2 | 3 | 4 | **0** | | **2** | 2 | 3 | 4 | **0** | 1 | | **3** | 3 | 4 | **0** | 1 | 2 | | **4** | 4 | **0** | 1 | 2 | 3 |

Multiplication: | x | **0** | 1 | 2 | 3 | 4 | |—|—|—|—|—|—| | **0** | **0** | **0** | **0** | **0** | **0** | | **1** | **0** | 1 | 2 | 3 | 4 | | **2** | **0** | 2 | 4 | 6 | 8 | | **3** | **0** | 3 | 6 | 9 | 12| | **4** | **0** | 4 | 8 | 12| 16|

Multiplicative and additive inverses: | w | 0 | 1 | 2 | 3 | 4 | |—|—|—|—|—|—| |$-w$ | 0 | 4 | 3 | 2 | 1 | |w^{-1}| - | 1 | 3 | 2 | 4 |

B.

i) x^3+1 is reducible by $x+1$

$$\begin{array}{r}
x^2+x+1 \\
x+1\overline{)x^3+1} \\
\underline{x^3+x^2} \\
x^2+1 \\
\underline{x^2+x} \\
x+1 \\
\underline{x+1}
\end{array}$$

ii)

x^3+x^2+1 is irreducible. If it you attempt to divide this polynomial by x or you will be unable to evenly divide the +1 part. This is also true if you attempt to divide the polynomial by $x + 1$. should you attempt this you will find the answer to be x^2 with a remainder of 1.

iii) $x^4 + 1$ is reducible by $x + 1$

$$\begin{array}{l}
\qquad\;\; x^3 + x^2 + x + 1 \\
x + 1\overline{)x^4 + 1} \\
\qquad\;\; \underline{x^4 + x^3} \\
\qquad\qquad x^3 + 1 \\
\qquad\qquad \underline{x^3 + x^2} \\
\qquad\qquad\quad\; x^2 + 1 \\
\qquad\qquad\quad\; \underline{x^2 + x} \\
\qquad\qquad\qquad\; x + 1 \\
\qquad\qquad\qquad\; \underline{x + 1}
\end{array}$$

Problem 4

TABLES DID NOT LOAD PROPERLY

please see images for corresponding table sorry.

A. The Field GF(4) with $m(x) = x^2 + x + 1$

+	0	1	$ x $	$ x+1 $
0	0	1	$ x $	$ x+1 $
1	1	0	$ x+1 $	$ x $
x	x	x	0	1
x+1	$ x+1 $	x	1	0

x	0	1	$ x $	$ x+1 $
0	0	0	0	0
1	0	1	$ x $	$ x+1 $
x	0	x	$ x+1 $	1
$x + 1$	0	$ x+1 $	1	x

B. Generator for $GF(2^4)$ using $m(x) = x^4 + x + 1$ | Power Representation | Polynomial Representation | Binary Representation | Decimal (Hex) Representation | | ————————— | ——————————- | ————————— | ——————————- | | 0 | 0 | 0000 | 00 | | g^0 | 1 | 0001 | 01 | | g^1 | g | 0010 | 02 | | g^2 | g^2 | 0100 | 04 | | g^3 | g^3 | 1000 | 08 | | g^4 | $g + 1$ | 0011 | 03 | | g^5 | $g^2 + g$ | 0110 | 06 | | g^6 | $g^3 + g^2$ | 1100 | 12 | | g^7 | $g^3 + g + 1$ | 1011 | 11 | | g^8 | $g^2 + 1$ | 0101 | 05 | | g^9 | $g^3 + g$ | 1010 | 10 | | g^{10} | $g^2 + g + 1$ | 0111 | 07 | | g^{11} | $g^3 + g^2 + g$ | 1110 | 14 | | g^{12} | $g^3 + g^2 + g + 1$ | 1111 | 15 | | g^{13} | $g^3 + g^2 + 1$ | 1101 | 13 | | g^{14} | $g^3 + 1$ | 1001 | 09 |

Problem 5

A. The program is relatively simple. `__init__` To make it work as i imagined it i created a Calculator object class that holds the polynomial in a function variable called f. The Polynomial must look like they normally do ie $x^2 + x + 1$ they can be any length that could be in $GF(2^8)$. The polynomials are parsed into arrays of exponents, i did this because there are no coefficients when doing $GF(2^m)$. In the `__init__` function I also have a ConvDict variable that is used to map the different exponents back into the normal Polynomial form (so that way you don't have to figure out what [7,3,1,0] means). `__add__` and `__sub__` Addition and subtraction work the same, they loop through one of the polynomials and append all of the exponents to the main list. That list then goes into PolyMod which runs a modulo on the list and reduces any copies so that it conforms to $GF(2^8)$ with $modx +^8 x^4 + x^3 + x + 1$. `__mul__` Multiply loops through the left polynomials and multiplies each number to each number in the right polynomial. This is accomlished with two for loops. Because exponential multiplicaiton really means adding the exponent numbers together, for each number in the variables F and G, they are added instead of multiplied. The output is then sent to PolyMod because more than likely more than one of the numbers is greater than 8. `__truediv__` Divide would be able to find the inverse of the right polynomial and multiply that to the left polynomial. To find the inverse i was going to try the Extended Euclid, but I couldn't figure it out without converting everything to Binary, but then that gave me a headache so i decided i would try to brute force it. but that took just way to long so I decided to see if Simeon would go easy on me for trying and failing to do the division portion. So theoutput for all division will be 1 because i couldn't figure it out. `parsePolynomial` This converts the normal polynomial form into a list of exponents for me to work with. Just some string manipulation, didnt use Regex which probably would have looked cleaner. `PolyMod` takes the poly array loops through each number. If the number is greater than eight it does a floor division (it rounds down) and takes that number. If that number is even then we know the answer is 0 (any even number mod 2 is 0) if its odd and is not already in the function list then it is appended. The the mod is taken and if it is not in the function list then it is added. If there is a number that is less than 8 but is in the list then it is not added and the number in the list is also removed. (this is because if at any point there are two of the same number like $2x$ that becomes 0 because $2xmod2 = 0$) `PrepOutput` puts the polynomials back to the normal way they look.

B. Should be self explanatory, i included an image of the main function reading. I did only one test because i finished this 2 hours before it was due and it was hard. Problem5.png