UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

EHN 410

E-Business and Network Security

Practical 2 Code

Name and Surname	Student Number	Contribution
Willem A. Fourie	17028002	33
Michelle Hartman	17090823	33
Franco M. van Tonder	17007594	33

By submitting this assignment we confirm that we have read and am aware of the University of Pretoria's policy on academic dishonesty and plagiarism and we declare that the work submitted in this assignment is our own as delimited by the mentioned policies. We explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. We understand that we will be subjected to disciplinary actions should it be found that the work we submit here does not comply with the said policies.

Contents

I. AES Files

A. *AES.c*

```
1  //
2  // Created by armandt on 2020/04/07.
3  //
4
5  #include "AES.h"
6  #include "stdio.h"
7  #include "math.h"
8  #include "stdlib.h"
9  #include "string.h"
10 #include "CipherModes.h"
11
12 int number_of_rounds = -1;
13 int expanded_key_size = -1;
14 int key_length = -1;
15
16 void set_number_of_rounds(int r){
17     number_of_rounds = r;
18 }
19
20 void set_expanded_key_size(int s){
21     expanded_key_size = s;
22 }
23
24 void set_key_length(int l){
25     key_length = l;
26     if(key_length == 128)     {
27         number_of_rounds = 9;
28         expanded_key_size = 176;
29     }
30     else if (key_length == 192) {
```

```
31          number_of_rounds = 11;
32          expanded_key_size = 208;
33      }
34      else if (key_length == 256) {
35          number_of_rounds = 13;
36          expanded_key_size = 240;
37      }
38  }
39
40  int s_box[256] =
41      {
42          0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0
               x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, 0
               xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0
               xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0
               xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0
               x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, 0
               x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0
               x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0
               x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0
               x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0
               x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0
               x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0
               xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0
               x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0
               x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0
               xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0
               xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0
               xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, 0
               x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0
               x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0
               xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0
               xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0
               xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0
               x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, 0
               xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0
               xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0
               x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0
               x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, 0
               xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0
               x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0
               x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0
               x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};
43
44  unsigned char inverse_s_box[256] =
45      {
46          0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38,
47          0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
48          0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87,
```

```
49          0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
50          0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D,
51          0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
52          0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2,
53          0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
54          0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16,
55          0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
56          0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA,
57          0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
58          0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A,
59          0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
60          0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02,
61          0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
62          0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA,
63          0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
64          0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85,
65          0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
66          0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89,
67          0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
68          0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20,
69          0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
70          0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31,
71          0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
72          0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D,
73          0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
74          0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0,
75          0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
76          0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26,
77          0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
78
79  };
80  unsigned char multiply_2[] =
81      {
82          0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0
                x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
83          0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0
                x30, 0x32, 0x34, 0x36, 0x38, 0x3a, 0x3c, 0x3e,
84          0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0
                x50, 0x52, 0x54, 0x56, 0x58, 0x5a, 0x5c, 0x5e,
85          0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0
                x70, 0x72, 0x74, 0x76, 0x78, 0x7a, 0x7c, 0x7e,
86          0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0
                x90, 0x92, 0x94, 0x96, 0x98, 0x9a, 0x9c, 0x9e,
87          0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0
                xb0, 0xb2, 0xb4, 0xb6, 0xb8, 0xba, 0xbc, 0xbe,
88          0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0
                xd0, 0xd2, 0xd4, 0xd6, 0xd8, 0xda, 0xdc, 0xde,
89          0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0
                xf0, 0xf2, 0xf4, 0xf6, 0xf8, 0xfa, 0xfc, 0xfe,
```

```
90          0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15, 0
               x0b, 0x09, 0x0f, 0x0d, 0x03, 0x01, 0x07, 0x05,
91          0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35, 0
               x2b, 0x29, 0x2f, 0x2d, 0x23, 0x21, 0x27, 0x25,
92          0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55, 0
               x4b, 0x49, 0x4f, 0x4d, 0x43, 0x41, 0x47, 0x45,
93          0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75, 0
               x6b, 0x69, 0x6f, 0x6d, 0x63, 0x61, 0x67, 0x65,
94          0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95, 0
               x8b, 0x89, 0x8f, 0x8d, 0x83, 0x81, 0x87, 0x85,
95          0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5, 0
               xab, 0xa9, 0xaf, 0xad, 0xa3, 0xa1, 0xa7, 0xa5,
96          0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5, 0
               xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1, 0xc7, 0xc5,
97          0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5, 0
               xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5};
98
99  unsigned char multiply_3[] =
100     {
101         0x00, 0x03, 0x06, 0x05, 0x0c, 0x0f, 0x0a, 0x09, 0
               x18, 0x1b, 0x1e, 0x1d, 0x14, 0x17, 0x12, 0x11,
102         0x30, 0x33, 0x36, 0x35, 0x3c, 0x3f, 0x3a, 0x39, 0
               x28, 0x2b, 0x2e, 0x2d, 0x24, 0x27, 0x22, 0x21,
103         0x60, 0x63, 0x66, 0x65, 0x6c, 0x6f, 0x6a, 0x69, 0
               x78, 0x7b, 0x7e, 0x7d, 0x74, 0x77, 0x72, 0x71,
104         0x50, 0x53, 0x56, 0x55, 0x5c, 0x5f, 0x5a, 0x59, 0
               x48, 0x4b, 0x4e, 0x4d, 0x44, 0x47, 0x42, 0x41,
105         0xc0, 0xc3, 0xc6, 0xc5, 0xcc, 0xcf, 0xca, 0xc9, 0
               xd8, 0xdb, 0xde, 0xdd, 0xd4, 0xd7, 0xd2, 0xd1,
106         0xf0, 0xf3, 0xf6, 0xf5, 0xfc, 0xff, 0xfa, 0xf9, 0
               xe8, 0xeb, 0xee, 0xed, 0xe4, 0xe7, 0xe2, 0xe1,
107         0xa0, 0xa3, 0xa6, 0xa5, 0xac, 0xaf, 0xaa, 0xa9, 0
               xb8, 0xbb, 0xbe, 0xbd, 0xb4, 0xb7, 0xb2, 0xb1,
108         0x90, 0x93, 0x96, 0x95, 0x9c, 0x9f, 0x9a, 0x99, 0
               x88, 0x8b, 0x8e, 0x8d, 0x84, 0x87, 0x82, 0x81,
109         0x9b, 0x98, 0x9d, 0x9e, 0x97, 0x94, 0x91, 0x92, 0
               x83, 0x80, 0x85, 0x86, 0x8f, 0x8c, 0x89, 0x8a,
110         0xab, 0xa8, 0xad, 0xae, 0xa7, 0xa4, 0xa1, 0xa2, 0
               xb3, 0xb0, 0xb5, 0xb6, 0xbf, 0xbc, 0xb9, 0xba,
111         0xfb, 0xf8, 0xfd, 0xfe, 0xf7, 0xf4, 0xf1, 0xf2, 0
               xe3, 0xe0, 0xe5, 0xe6, 0xef, 0xec, 0xe9, 0xea,
112         0xcb, 0xc8, 0xcd, 0xce, 0xc7, 0xc4, 0xc1, 0xc2, 0
               xd3, 0xd0, 0xd5, 0xd6, 0xdf, 0xdc, 0xd9, 0xda,
113         0x5b, 0x58, 0x5d, 0x5e, 0x57, 0x54, 0x51, 0x52, 0
               x43, 0x40, 0x45, 0x46, 0x4f, 0x4c, 0x49, 0x4a,
114         0x6b, 0x68, 0x6d, 0x6e, 0x67, 0x64, 0x61, 0x62, 0
               x73, 0x70, 0x75, 0x76, 0x7f, 0x7c, 0x79, 0x7a,
115         0x3b, 0x38, 0x3d, 0x3e, 0x37, 0x34, 0x31, 0x32, 0
               x23, 0x20, 0x25, 0x26, 0x2f, 0x2c, 0x29, 0x2a,
```

```
116         0x0b, 0x08, 0x0d, 0x0e, 0x07, 0x04, 0x01, 0x02, 0
              x13, 0x10, 0x15, 0x16, 0x1f, 0x1c, 0x19, 0x1a};
117
118 unsigned char multiply_9[] =
119     {
120         0x00, 0x09, 0x12, 0x1b, 0x24, 0x2d, 0x36, 0x3f, 0
              x48, 0x41, 0x5a, 0x53, 0x6c, 0x65, 0x7e, 0x77,
121         0x90, 0x99, 0x82, 0x8b, 0xb4, 0xbd, 0xa6, 0xaf, 0
              xd8, 0xd1, 0xca, 0xc3, 0xfc, 0xf5, 0xee, 0xe7,
122         0x3b, 0x32, 0x29, 0x20, 0x1f, 0x16, 0x0d, 0x04, 0
              x73, 0x7a, 0x61, 0x68, 0x57, 0x5e, 0x45, 0x4c,
123         0xab, 0xa2, 0xb9, 0xb0, 0x8f, 0x86, 0x9d, 0x94, 0
              xe3, 0xea, 0xf1, 0xf8, 0xc7, 0xce, 0xd5, 0xdc,
124         0x76, 0x7f, 0x64, 0x6d, 0x52, 0x5b, 0x40, 0x49, 0
              x3e, 0x37, 0x2c, 0x25, 0x1a, 0x13, 0x08, 0x01,
125         0xe6, 0xef, 0xf4, 0xfd, 0xc2, 0xcb, 0xd0, 0xd9, 0
              xae, 0xa7, 0xbc, 0xb5, 0x8a, 0x83, 0x98, 0x91,
126         0x4d, 0x44, 0x5f, 0x56, 0x69, 0x60, 0x7b, 0x72, 0
              x05, 0x0c, 0x17, 0x1e, 0x21, 0x28, 0x33, 0x3a,
127         0xdd, 0xd4, 0xcf, 0xc6, 0xf9, 0xf0, 0xeb, 0xe2, 0
              x95, 0x9c, 0x87, 0x8e, 0xb1, 0xb8, 0xa3, 0xaa,
128         0xec, 0xe5, 0xfe, 0xf7, 0xc8, 0xc1, 0xda, 0xd3, 0
              xa4, 0xad, 0xb6, 0xbf, 0x80, 0x89, 0x92, 0x9b,
129         0x7c, 0x75, 0x6e, 0x67, 0x58, 0x51, 0x4a, 0x43, 0
              x34, 0x3d, 0x26, 0x2f, 0x10, 0x19, 0x02, 0x0b,
130         0xd7, 0xde, 0xc5, 0xcc, 0xf3, 0xfa, 0xe1, 0xe8, 0
              x9f, 0x96, 0x8d, 0x84, 0xbb, 0xb2, 0xa9, 0xa0,
131         0x47, 0x4e, 0x55, 0x5c, 0x63, 0x6a, 0x71, 0x78, 0
              x0f, 0x06, 0x1d, 0x14, 0x2b, 0x22, 0x39, 0x30,
132         0x9a, 0x93, 0x88, 0x81, 0xbe, 0xb7, 0xac, 0xa5, 0
              xd2, 0xdb, 0xc0, 0xc9, 0xf6, 0xff, 0xe4, 0xed,
133         0x0a, 0x03, 0x18, 0x11, 0x2e, 0x27, 0x3c, 0x35, 0
              x42, 0x4b, 0x50, 0x59, 0x66, 0x6f, 0x74, 0x7d,
134         0xa1, 0xa8, 0xb3, 0xba, 0x85, 0x8c, 0x97, 0x9e, 0
              xe9, 0xe0, 0xfb, 0xf2, 0xcd, 0xc4, 0xdf, 0xd6,
135         0x31, 0x38, 0x23, 0x2a, 0x15, 0x1c, 0x07, 0x0e, 0
              x79, 0x70, 0x6b, 0x62, 0x5d, 0x54, 0x4f, 0x46};
136
137 unsigned char multiply_11[] =
138     {
139         0x00, 0x0b, 0x16, 0x1d, 0x2c, 0x27, 0x3a, 0x31, 0
              x58, 0x53, 0x4e, 0x45, 0x74, 0x7f, 0x62, 0x69,
140         0xb0, 0xbb, 0xa6, 0xad, 0x9c, 0x97, 0x8a, 0x81, 0
              xe8, 0xe3, 0xfe, 0xf5, 0xc4, 0xcf, 0xd2, 0xd9,
141         0x7b, 0x70, 0x6d, 0x66, 0x57, 0x5c, 0x41, 0x4a, 0
              x23, 0x28, 0x35, 0x3e, 0x0f, 0x04, 0x19, 0x12,
142         0xcb, 0xc0, 0xdd, 0xd6, 0xe7, 0xec, 0xf1, 0xfa, 0
              x93, 0x98, 0x85, 0x8e, 0xbf, 0xb4, 0xa9, 0xa2,
143         0xf6, 0xfd, 0xe0, 0xeb, 0xda, 0xd1, 0xcc, 0xc7, 0
```

```
            xae, 0xa5, 0xb8, 0xb3, 0x82, 0x89, 0x94, 0x9f,
144         0x46, 0x4d, 0x50, 0x5b, 0x6a, 0x61, 0x7c, 0x77, 0
            x1e, 0x15, 0x08, 0x03, 0x32, 0x39, 0x24, 0x2f,
145         0x8d, 0x86, 0x9b, 0x90, 0xa1, 0xaa, 0xb7, 0xbc, 0
            xd5, 0xde, 0xc3, 0xc8, 0xf9, 0xf2, 0xef, 0xe4,
146         0x3d, 0x36, 0x2b, 0x20, 0x11, 0x1a, 0x07, 0x0c, 0
            x65, 0x6e, 0x73, 0x78, 0x49, 0x42, 0x5f, 0x54,
147         0xf7, 0xfc, 0xe1, 0xea, 0xdb, 0xd0, 0xcd, 0xc6, 0
            xaf, 0xa4, 0xb9, 0xb2, 0x83, 0x88, 0x95, 0x9e,
148         0x47, 0x4c, 0x51, 0x5a, 0x6b, 0x60, 0x7d, 0x76, 0
            x1f, 0x14, 0x09, 0x02, 0x33, 0x38, 0x25, 0x2e,
149         0x8c, 0x87, 0x9a, 0x91, 0xa0, 0xab, 0xb6, 0xbd, 0
            xd4, 0xdf, 0xc2, 0xc9, 0xf8, 0xf3, 0xee, 0xe5,
150         0x3c, 0x37, 0x2a, 0x21, 0x10, 0x1b, 0x06, 0x0d, 0
            x64, 0x6f, 0x72, 0x79, 0x48, 0x43, 0x5e, 0x55,
151         0x01, 0x0a, 0x17, 0x1c, 0x2d, 0x26, 0x3b, 0x30, 0
            x59, 0x52, 0x4f, 0x44, 0x75, 0x7e, 0x63, 0x68,
152         0xb1, 0xba, 0xa7, 0xac, 0x9d, 0x96, 0x8b, 0x80, 0
            xe9, 0xe2, 0xff, 0xf4, 0xc5, 0xce, 0xd3, 0xd8,
153         0x7a, 0x71, 0x6c, 0x67, 0x56, 0x5d, 0x40, 0x4b, 0
            x22, 0x29, 0x34, 0x3f, 0x0e, 0x05, 0x18, 0x13,
154         0xca, 0xc1, 0xdc, 0xd7, 0xe6, 0xed, 0xf0, 0xfb, 0
            x92, 0x99, 0x84, 0x8f, 0xbe, 0xb5, 0xa8, 0xa3};
155
156 unsigned char multiply_13[] =
157     {
158         0x00, 0x0d, 0x1a, 0x17, 0x34, 0x39, 0x2e, 0x23, 0
            x68, 0x65, 0x72, 0x7f, 0x5c, 0x51, 0x46, 0x4b,
159         0xd0, 0xdd, 0xca, 0xc7, 0xe4, 0xe9, 0xfe, 0xf3, 0
            xb8, 0xb5, 0xa2, 0xaf, 0x8c, 0x81, 0x96, 0x9b,
160         0xbb, 0xb6, 0xa1, 0xac, 0x8f, 0x82, 0x95, 0x98, 0
            xd3, 0xde, 0xc9, 0xc4, 0xe7, 0xea, 0xfd, 0xf0,
161         0x6b, 0x66, 0x71, 0x7c, 0x5f, 0x52, 0x45, 0x48, 0
            x03, 0x0e, 0x19, 0x14, 0x37, 0x3a, 0x2d, 0x20,
162         0x6d, 0x60, 0x77, 0x7a, 0x59, 0x54, 0x43, 0x4e, 0
            x05, 0x08, 0x1f, 0x12, 0x31, 0x3c, 0x2b, 0x26,
163         0xbd, 0xb0, 0xa7, 0xaa, 0x89, 0x84, 0x93, 0x9e, 0
            xd5, 0xd8, 0xcf, 0xc2, 0xe1, 0xec, 0xfb, 0xf6,
164         0xd6, 0xdb, 0xcc, 0xc1, 0xe2, 0xef, 0xf8, 0xf5, 0
            xbe, 0xb3, 0xa4, 0xa9, 0x8a, 0x87, 0x90, 0x9d,
165         0x06, 0x0b, 0x1c, 0x11, 0x32, 0x3f, 0x28, 0x25, 0
            x6e, 0x63, 0x74, 0x79, 0x5a, 0x57, 0x40, 0x4d,
166         0xda, 0xd7, 0xc0, 0xcd, 0xee, 0xe3, 0xf4, 0xf9, 0
            xb2, 0xbf, 0xa8, 0xa5, 0x86, 0x8b, 0x9c, 0x91,
167         0x0a, 0x07, 0x10, 0x1d, 0x3e, 0x33, 0x24, 0x29, 0
            x62, 0x6f, 0x78, 0x75, 0x56, 0x5b, 0x4c, 0x41,
168         0x61, 0x6c, 0x7b, 0x76, 0x55, 0x58, 0x4f, 0x42, 0
            x09, 0x04, 0x13, 0x1e, 0x3d, 0x30, 0x27, 0x2a,
169         0xb1, 0xbc, 0xab, 0xa6, 0x85, 0x88, 0x9f, 0x92, 0
```

```
            xd9, 0xd4, 0xc3, 0xce, 0xed, 0xe0, 0xf7, 0xfa,
170         0xb7, 0xba, 0xad, 0xa0, 0x83, 0x8e, 0x99, 0x94, 0
            xdf, 0xd2, 0xc5, 0xc8, 0xeb, 0xe6, 0xf1, 0xfc,
171         0x67, 0x6a, 0x7d, 0x70, 0x53, 0x5e, 0x49, 0x44, 0
            x0f, 0x02, 0x15, 0x18, 0x3b, 0x36, 0x21, 0x2c,
172         0x0c, 0x01, 0x16, 0x1b, 0x38, 0x35, 0x22, 0x2f, 0
            x64, 0x69, 0x7e, 0x73, 0x50, 0x5d, 0x4a, 0x47,
173         0xdc, 0xd1, 0xc6, 0xcb, 0xe8, 0xe5, 0xf2, 0xff, 0
            xb4, 0xb9, 0xae, 0xa3, 0x80, 0x8d, 0x9a, 0x97};
174
175 unsigned char multiply_14[] =
176     {
177         0x00, 0x0e, 0x1c, 0x12, 0x38, 0x36, 0x24, 0x2a, 0
            x70, 0x7e, 0x6c, 0x62, 0x48, 0x46, 0x54, 0x5a,
178         0xe0, 0xee, 0xfc, 0xf2, 0xd8, 0xd6, 0xc4, 0xca, 0
            x90, 0x9e, 0x8c, 0x82, 0xa8, 0xa6, 0xb4, 0xba,
179         0xdb, 0xd5, 0xc7, 0xc9, 0xe3, 0xed, 0xff, 0xf1, 0
            xab, 0xa5, 0xb7, 0xb9, 0x93, 0x9d, 0x8f, 0x81,
180         0x3b, 0x35, 0x27, 0x29, 0x03, 0x0d, 0x1f, 0x11, 0
            x4b, 0x45, 0x57, 0x59, 0x73, 0x7d, 0x6f, 0x61,
181         0xad, 0xa3, 0xb1, 0xbf, 0x95, 0x9b, 0x89, 0x87, 0
            xdd, 0xd3, 0xc1, 0xcf, 0xe5, 0xeb, 0xf9, 0xf7,
182         0x4d, 0x43, 0x51, 0x5f, 0x75, 0x7b, 0x69, 0x67, 0
            x3d, 0x33, 0x21, 0x2f, 0x05, 0x0b, 0x19, 0x17,
183         0x76, 0x78, 0x6a, 0x64, 0x4e, 0x40, 0x52, 0x5c, 0
            x06, 0x08, 0x1a, 0x14, 0x3e, 0x30, 0x22, 0x2c,
184         0x96, 0x98, 0x8a, 0x84, 0xae, 0xa0, 0xb2, 0xbc, 0
            xe6, 0xe8, 0xfa, 0xf4, 0xde, 0xd0, 0xc2, 0xcc,
185         0x41, 0x4f, 0x5d, 0x53, 0x79, 0x77, 0x65, 0x6b, 0
            x31, 0x3f, 0x2d, 0x23, 0x09, 0x07, 0x15, 0x1b,
186         0xa1, 0xaf, 0xbd, 0xb3, 0x99, 0x97, 0x85, 0x8b, 0
            xd1, 0xdf, 0xcd, 0xc3, 0xe9, 0xe7, 0xf5, 0xfb,
187         0x9a, 0x94, 0x86, 0x88, 0xa2, 0xac, 0xbe, 0xb0, 0
            xea, 0xe4, 0xf6, 0xf8, 0xd2, 0xdc, 0xce, 0xc0,
188         0x7a, 0x74, 0x66, 0x68, 0x42, 0x4c, 0x5e, 0x50, 0
            x0a, 0x04, 0x16, 0x18, 0x32, 0x3c, 0x2e, 0x20,
189         0xec, 0xe2, 0xf0, 0xfe, 0xd4, 0xda, 0xc8, 0xc6, 0
            x9c, 0x92, 0x80, 0x8e, 0xa4, 0xaa, 0xb8, 0xb6,
190         0x0c, 0x02, 0x10, 0x1e, 0x34, 0x3a, 0x28, 0x26, 0
            x7c, 0x72, 0x60, 0x6e, 0x44, 0x4a, 0x58, 0x56,
191         0x37, 0x39, 0x2b, 0x25, 0x0f, 0x01, 0x13, 0x1d, 0
            x47, 0x49, 0x5b, 0x55, 0x7f, 0x71, 0x63, 0x6d,
192         0xd7, 0xd9, 0xcb, 0xc5, 0xef, 0xe1, 0xf3, 0xfd, 0
            xa7, 0xa9, 0xbb, 0xb5, 0x9f, 0x91, 0x83, 0x8d};
193
194 unsigned char RCon[11] =
195         {
196                 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0
                    x40, 0x80, 0x1b, 0x36
```

```
197         };
198
199 void key_expansion_core(unsigned char *in, unsigned char i)
200 {
201     //rotate left
202     unsigned char t = in[0];
203     in[0] = in[1];
204     in[1] = in[2];
205     in[2] = in[3];
206     in[3] = t;
207
208     //S-box on all four bytes
209     in[0] = s_box[in[0]];
210     in[1] = s_box[in[1]];
211     in[2] = s_box[in[2]];
212     in[3] = s_box[in[3]];
213
214     //RCon operation
215     in[0] ^= RCon[i];
216 }
217
218 void key_expansion(unsigned char *input_key, unsigned char
      *expanded_key)
219 {
220     //The first 16 bytes of the expanded key are simply the
          encryption key that the user entered.
221     for (int i = 0; i < (key_length / 8); i++)
222         expanded_key[i] = input_key[i];
223
224     //Variables
225     int bytes_generated = key_length / 8;
226     int RCon_iteration = 1;
227     unsigned char temp[4];
228
229     while (bytes_generated < expanded_key_size)
230     {
231         // Assign previous four bytes in the expanded key
              to temp
232         for (int i = 0; i < 4; i++)
233             temp[i] = expanded_key[i + bytes_generated -
                  4];
234
235         //Send t to the core key scheduler along with the
              RCon value.
236         if (bytes_generated % 16 == 0)
237             key_expansion_core(temp, RCon_iteration++);
238
239         /*XOR the output of the core key scheduler with a
              four-byte block 16 bytes before the
```

```
240             expanded key (i.e bytes 0-3). The result becomes
                    the next 4 bytes of the expanded key.*/
241             for (unsigned char i = 0; i < 4; i++)
242             {
243                 expanded_key[bytes_generated] = expanded_key[
                        bytes_generated - 16] ^ temp[i];
244                 bytes_generated++;
245             }
246         }
247     }
248
249     void sub_bytes(unsigned char *state)
250     {
251         for (int i = 0; i < 16; i++)
252             state[i] = s_box[state[i]];
253     }
254     void inverse_sub_bytes(unsigned char *state)
255     {
256         for (int i = 0; i < 16; i++)
257             state[i] = inverse_s_box[state[i]];
258     }
259
260     void shift_rows(unsigned char *state)
261     {
262         unsigned char tmp[16];
263
264         tmp[0] = state[0];
265         tmp[1] = state[5];
266         tmp[2] = state[10];
267         tmp[3] = state[15];
268
269         tmp[4] = state[4];
270         tmp[5] = state[9];
271         tmp[6] = state[14];
272         tmp[7] = state[3];
273
274         tmp[8] = state[8];
275         tmp[9] = state[13];
276         tmp[10] = state[2];
277         tmp[11] = state[7];
278
279         tmp[12] = state[12];
280         tmp[13] = state[1];
281         tmp[14] = state[6];
282         tmp[15] = state[11];
283
284         for (int i = 0; i < 16; i++)
285             state[i] = tmp[i];
286     }
```

```
287 void inverse_shift_rows(unsigned char *state)
288 {
289     unsigned char tmp[16];
290 
291     tmp[0] = state[0];
292     tmp[5] = state[1];
293     tmp[10] = state[2];
294     tmp[15] = state[3];
295 
296     tmp[4] = state[4];
297     tmp[9] = state[5];
298     tmp[14] = state[6];
299     tmp[3] = state[7];
300 
301     tmp[8] = state[8];
302     tmp[13] = state[9];
303     tmp[2] = state[10];
304     tmp[7] = state[11];
305 
306     tmp[12] = state[12];
307     tmp[1] = state[13];
308     tmp[6] = state[14];
309     tmp[11] = state[15];
310 
311     for (int i = 0; i < 16; i++)
312         state[i] = tmp[i];
313 }
314 
315 void mix_columns(unsigned char *state)
316 {
317     unsigned char tmp[16];
318 
319     tmp[0] = (unsigned char)(multiply_2[state[0]] ^
        multiply_3[state[1]] ^ state[2] ^ state[3]);
320     tmp[1] = (unsigned char)(state[0] ^ multiply_2[state
        [1]] ^ multiply_3[state[2]] ^ state[3]);
321     tmp[2] = (unsigned char)(state[0] ^ state[1] ^
        multiply_2[state[2]] ^ multiply_3[state[3]]);
322     tmp[3] = (unsigned char)(multiply_3[state[0]] ^ state
        [1] ^ state[2] ^ multiply_2[state[3]]);
323 
324     tmp[4] = (unsigned char)(multiply_2[state[4]] ^
        multiply_3[state[5]] ^ state[6] ^ state[7]);
325     tmp[5] = (unsigned char)(state[4] ^ multiply_2[state
        [5]] ^ multiply_3[state[6]] ^ state[7]);
326     tmp[6] = (unsigned char)(state[4] ^ state[5] ^
        multiply_2[state[6]] ^ multiply_3[state[7]]);
327     tmp[7] = (unsigned char)(multiply_3[state[4]] ^ state
        [5] ^ state[6] ^ multiply_2[state[7]]);
```

```
328
329       tmp[8] = (unsigned char)(multiply_2[state[8]] ^
              multiply_3[state[9]] ^ state[10] ^ state[11]);
330       tmp[9] = (unsigned char)(state[8] ^ multiply_2[state
              [9]] ^ multiply_3[state[10]] ^ state[11]);
331       tmp[10] = (unsigned char)(state[8] ^ state[9] ^
              multiply_2[state[10]] ^ multiply_3[state[11]]);
332       tmp[11] = (unsigned char)(multiply_3[state[8]] ^ state
              [9] ^ state[10] ^ multiply_2[state[11]]);
333
334       tmp[12] = (unsigned char)(multiply_2[state[12]] ^
              multiply_3[state[13]] ^ state[14] ^ state[15]);
335       tmp[13] = (unsigned char)(state[12] ^ multiply_2[state
              [13]] ^ multiply_3[state[14]] ^ state[15]);
336       tmp[14] = (unsigned char)(state[12] ^ state[13] ^
              multiply_2[state[14]] ^ multiply_3[state[15]]);
337       tmp[15] = (unsigned char)(multiply_3[state[12]] ^ state
              [13] ^ state[14] ^ multiply_2[state[15]]);
338       tmp[16] = '\0';
339
340       for (int i = 0; i < 17; i++)
341            state[i] = tmp[i];
342  }
343
344  void inverse_mix_columns(unsigned char *state)
345  {
346       unsigned char tmp[16];
347
348       tmp[0] = (unsigned char)(multiply_14[state[0]] ^
              multiply_9[state[3]] ^ multiply_13[state[2]] ^
              multiply_11[state[1]]);
349       tmp[1] = (unsigned char)(multiply_14[state[1]] ^
              multiply_9[state[0]] ^ multiply_13[state[3]] ^
              multiply_11[state[2]]);
350       tmp[2] = (unsigned char)(multiply_14[state[2]] ^
              multiply_9[state[1]] ^ multiply_13[state[0]] ^
              multiply_11[state[3]]);
351       tmp[3] = (unsigned char)(multiply_14[state[3]] ^
              multiply_9[state[2]] ^ multiply_13[state[1]] ^
              multiply_11[state[0]]);
352
353       tmp[4] = (unsigned char)(multiply_14[state[4]] ^
              multiply_9[state[7]] ^ multiply_13[state[6]] ^
              multiply_11[state[5]]);
354       tmp[5] = (unsigned char)(multiply_14[state[5]] ^
              multiply_9[state[4]] ^ multiply_13[state[7]] ^
              multiply_11[state[6]]);
355       tmp[6] = (unsigned char)(multiply_14[state[6]] ^
              multiply_9[state[5]] ^ multiply_13[state[4]] ^
```

```
            multiply_11[state[7]]);
356         tmp[7] = (unsigned char)(multiply_14[state[7]] ^
            multiply_9[state[6]] ^ multiply_13[state[5]] ^
            multiply_11[state[4]]);
357
358         tmp[8] = (unsigned char)(multiply_14[state[8]] ^
            multiply_9[state[11]] ^ multiply_13[state[10]] ^
            multiply_11[state[9]]);
359         tmp[9] = (unsigned char)(multiply_14[state[9]] ^
            multiply_9[state[8]] ^ multiply_13[state[11]] ^
            multiply_11[state[10]]);
360         tmp[10] = (unsigned char)(multiply_14[state[10]] ^
            multiply_9[state[9]] ^ multiply_13[state[8]] ^
            multiply_11[state[11]]);
361         tmp[11] = (unsigned char)(multiply_14[state[11]] ^
            multiply_9[state[10]] ^ multiply_13[state[9]] ^
            multiply_11[state[8]]);
362
363         tmp[12] = (unsigned char)(multiply_14[state[12]] ^
            multiply_9[state[15]] ^ multiply_13[state[14]] ^
            multiply_11[state[13]]);
364         tmp[13] = (unsigned char)(multiply_14[state[13]] ^
            multiply_9[state[12]] ^ multiply_13[state[15]] ^
            multiply_11[state[14]]);
365         tmp[14] = (unsigned char)(multiply_14[state[14]] ^
            multiply_9[state[13]] ^ multiply_13[state[12]] ^
            multiply_11[state[15]]);
366         tmp[15] = (unsigned char)(multiply_14[state[15]] ^
            multiply_9[state[14]] ^ multiply_13[state[13]] ^
            multiply_11[state[12]]);
367         tmp[16] = '\0';
368
369         for (int i = 0; i < 17; i++)
370             state[i] = tmp[i];
371     }
372
373     void add_round_key(unsigned char *state, unsigned char *
        round_key)
374     {
375         for (int i = 0; i < 16; i++)
376             state[i] ^= round_key[i];
377     }
378     void AES_encrypt(unsigned char *message, unsigned char *key
        )
379     {
380         unsigned char state[16];
381         for (int i = 0; i < 16; i++)
382             state[i] = message[i];
383
```

```
384     // int number_of_rounds = 9;
385
386     unsigned char expanded_key[expanded_key_size];
387     key_expansion(key, expanded_key);
388
389     //Initial round
390     add_round_key(state, key);
391
392     //Mixing rounds
393     for (int i = 0; i < number_of_rounds; i++)
394     {
395         sub_bytes(state);
396         shift_rows(state);
397         mix_columns(state);
398         add_round_key(state, expanded_key + (16 * (i + 1)))
                ;
399     }
400
401     //Final round
402     sub_bytes(state);
403     shift_rows(state);
404     add_round_key(state, expanded_key + expanded_key_size -
            16);
405
406     for (int i = 0; i < 16; i++)
407         message[i] = state[i];
408     message[17] = '\0';
409 }
410
411 void AES_decrypt(unsigned char *message, unsigned char *key
        )
412 {
413     unsigned char state[16];
414     for (int i = 0; i < 16; i++)
415         state[i] = message[i];
416
417     //int number_of_rounds = 9;
418
419     unsigned char expanded_key[expanded_key_size];
420     key_expansion(key, expanded_key);
421
422     //Initial round
423     add_round_key(state, expanded_key + expanded_key_size -
            16);
424
425     //Mixing rounds
426     for (int i = 0; i < number_of_rounds; i++)
427     {
428         inverse_shift_rows(state);
```

```
429             inverse_sub_bytes(state);
430             add_round_key(state, expanded_key +
                    expanded_key_size - 16 - (16 * (i + 1)));
431             inverse_mix_columns(state);
432         }
433
434         //Final round
435         inverse_shift_rows(state);
436         inverse_sub_bytes(state);
437         add_round_key(state, key);
438
439         for (int i = 0; i < 16; i++)
440             message[i] = state[i];
441         message[17] = '\0';
442     }
443
444     void print_hex(const unsigned char *string, int count)
445     {
446         unsigned char *p = (unsigned char *)string;
447
448         for (int i = 0; i < count; ++i)
449         {
450             if (!(i % 16) && i)
451                 printf("\n");
452
453             printf("%02x ", p[i]);
454         }
455         printf("\n\n");
456     }
457
458     void print_hex_block(const char *string)
459     {
460         unsigned char *p = (unsigned char *)string;
461
462         for (int i = 0; i < 4; ++i)
463         {
464             int x = 0;
465             for (int j = 0; j < 4; j++)
466             {
467                 //              if (! (i % 16) && i)
468                 //                  printf("\n");
469                 x = j * 4;
470
471                 printf("%02x ", p[i + x]);
472             }
473             printf("\n");
474         }
475         printf("\n\n");
476     }
```

```
477
478 void test_functionality(unsigned char *input_string, int
       key_length, unsigned char* key)
479 {
480     printf("\
          n_____________________________________________\
          n");
481     unsigned char input[strlen(input_string)];
482     strncpy(input, input_string, strlen(input_string));
483     print_hex_block(input_string);
484
485     printf("\nMix␣Columns\n");
486     printf("
          _____________________________________________\n
          ");
487
488     mix_columns(input_string);
489     print_hex_block(input_string);
490     printf("
          _____________________________________________\n
          ");
491
492     printf("\nShift␣rows\n");
493     strncpy(input_string, input, strlen(input_string));
494     printf("
          _____________________________________________\n
          ");
495     shift_rows(input_string);
496     print_hex_block(input_string);
497     printf("
          _____________________________________________\n
          ");
498
499     printf("\nSub␣Bytes\n");
500     strncpy(input_string, input, strlen(input_string));
501     printf("
          _____________________________________________\n
          ");
502     sub_bytes(input_string);
503     print_hex_block(input_string);
504     printf("
          _____________________________________________\n
          ");
505
506     printf("\nExpanded␣key\n");
507     strncpy(input_string, input, strlen(input_string));
508     printf("
          _____________________________________________\n
          ");
```

```
509     unsigned char expanded_key[expanded_key_size];
510     set_key_length(key_length);
511     key_expansion(key, expanded_key);
512     print_hex(expanded_key, expanded_key_size);
513     printf("
          ------------------------------------------------\n
        ");
514 }
515
516
517 unsigned char* pad_and_encrypt(unsigned char * message,
      unsigned char * encrypted, int message_len, int key_len,
      unsigned char * k){
518     unsigned char original_message[message_len + 1];     //
          make a copy of the message (maybe this helps)
519     unsigned char key[key_len / 8];
          //make a copy of the key
520
521     for (int a = 0; a < key_len / 8; a++){
522         key[a] = k[a];
523     }
524
525     for (int a = 0; a < message_len; a++){
526         original_message[a] = message[a];
527     }
528     original_message[message_len] = '\0';
529
530     set_key_length(key_len);
531
532     //now do the padding
533     int padded_message_len = message_len;
534     if (message_len % 16 != 0){
535         padded_message_len = (padded_message_len / 16 + 1)
              * 16;
536     }
537
538     unsigned char padded_message[padded_message_len + 1];
539     for (int a = 0; a < padded_message_len; a++){
540         if (a >= padded_message_len){
541             padded_message[a] = '0';
542         } else {
543             padded_message[a] = original_message[a];
544         }
545     }
546     padded_message[padded_message_len] = '\0';
547
548     unsigned char temp[padded_message_len + 1];
549     unsigned char * encrypted_message = temp;
550     for (int a = 0; a < padded_message_len; a += 16){
```

```
551             unsigned char block_to_encrypt[17];
552             for (int b = 0; b < 16; b++){
553                 block_to_encrypt[b] = padded_message[a + b];
554             }
555 
556             block_to_encrypt[16] = '\0';
557             AES_encrypt(block_to_encrypt, key);
558 
559             for (int j = 0; j < 16; j++){
560                 encrypted_message[j + a] = block_to_encrypt[j];
561             }
562         }
563         encrypted_message[padded_message_len] = '\0';    //very
              important for decryption
564 
565         for (int a = 0; a < padded_message_len + 1; a++){
566 //             printf("%x ", encrypted_message[a]);
567             encrypted[a] = encrypted_message[a];
568         }
569 //     encrypted[padded_message_len] = '\0';
570 //     printf("\n");
571 
572         return encrypted_message;
573 }
574 
575 unsigned char * general_decrypt(unsigned char * message,
       int message_len, int key_len, unsigned char * k){
576         int padded_message_len = message_len;
577         unsigned char temp[padded_message_len + 1];         //
              the encrypted message should be the length of the
              padded original message
578         unsigned char * decrypted_message = temp;
579         unsigned char message_copy[padded_message_len + 1];
580 
581         set_key_length(key_len);
582 
583         for (int a = 0; a < padded_message_len; a++){
584             message_copy[a] = message[a];
585         }   //added this because C overwrites the contents of
              memory somewhere during the execution of this
              function
586 
587         for (int a = 0; a < padded_message_len; a += 16){
588             unsigned char block_to_decrypt[17];
589 
590             for (int b = 0; b < 16; b++){
591                 block_to_decrypt[b] = message_copy[a + b];
592             }
593             block_to_decrypt[16] = '\0';
```

```
594
595
596             AES_decrypt(block_to_decrypt, k);
597
598             for (int b = 0; b < 16; b++){
599                 decrypted_message[b + a] = block_to_decrypt[b];
600             }
601         }
602
603         decrypted_message[padded_message_len] = '\0';
604
605         for (int a = 0; a < padded_message_len; a++){
606 //              printf("%x ", decrypted_message[a]);
607             message[a] = decrypted_message[a];
608         }
609
610         message[padded_message_len] = '\0';
611 //      printf("\n");
612
613         return decrypted_message;
614 }
```

B. AES.h

```
1  //
2  // Created by armandt on 2020/04/07.
3  //
4
5  #ifndef AES_H
6  #define AES_H
7
8  /**
9   * @brief The key expansion core is used in the key
       expansion method and contains 3 steps. 1) Rotate left.
       2) S-box on all four bytes. 3) XOR with RCons
10  * @param in A temporary 4 bytes used to generates the
       expanded key.
11  * @param i The RCon iteration index
12  */
13 void key_expansion_core(unsigned char* in, unsigned char i)
      ;
14 /**
15  * @brief This method expands the original key to the
       appropriate expanded key, to provide enough round keys
       for the AES function.
16  * @param input_key The original key.
17  * @param expanded_key The final expanded to to be used by
       the AES algorithm.
18  */
19 void key_expansion(unsigned char* input_key, unsigned char*
      expanded_key);
20 /**
21  * @brief Uses the S-box table to perform a byte-by-byte
       substitution of the current state.
22  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the encryption.
23  */
24 void sub_bytes(unsigned char* state);
25 /**
26  * @brief Uses the inverse S-box table to perform a byte-by
       -byte substitution of the current state.
27  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the decryption.
28  */
29 void inverse_sub_bytes(unsigned char* state);
30 /**
31  * @brief A simple permutation which is performed row by
       row
32  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the encryption.
33  */
```

```
34 void shift_rows(unsigned char* state);
35 /**
36  * @brief A simple permutation which is performed row by
       row
37  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the decryption.
38  */
39 void inverse_shift_rows(unsigned char* state);
40 /**
41  * @brief A substitution that alters each byte in a column
       as a function of all of the bytes in the column.
42  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the decryption.
43  */
44 void mix_columns(unsigned char* state );
45 /**
46  * @brief A substitution that alters each byte in a column
       as a function of all of the bytes in the column.
47  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the decryption.
48  */
49 void inverse_mix_columns(unsigned char* state );
50 /**
51  * @brief A simple bitwise XOR of the current block with a
       portion of the expanded key.
52  * @param state The 128-bit block is copied to a state
       which is modified at each stage of the decryption.
53  * @param round_key The portion of the expanded key used in
        a particular round.
54  */
55 void add_round_key(unsigned char* state, unsigned char*
      round_key);
56
57 /**
58  * @brief Uses AES method to encrypt a message using the
       provided key. The encrypted
59  * message is stored in the message array that it passed in
       .
60  * @param message the plaintext that will be encrypted.
61  * @param key The key used by the algorithm
62  */
63 void AES_encrypt(unsigned char* message, unsigned char* key
      );
64 /**
65  * @brief Uses AES method to decrypt a message using the
       provided key. The decrypted
66  * message is stored in the message array that it passed in
       .
67  * @param message The encrypted message that will be
```

```
     decrypted.
68   * @param key The key used by the algorithm
69   */
70  void AES_decrypt(unsigned char* message, unsigned char* key
       );
71  /**
72   * @brief Simply prints the input string in a hex format.
73   * @param string The message to be printed in hex format.
74   * @param count The number of characters that will be
       printed
75   */
76  void print_hex(const unsigned char *string, int count);
77  /**
78   * @brief Prints the input string in a hex, in a 4x4 block
       format.
79   * @param string The message to be printed in hex format.
80   */
81  void print_hex_block(const char *string);
82  /**
83   * @brief Displays each individual functions results
       independently.
84   * @param string The message to be passed into each
       individual function.
85   * @param key_length The length of the input key.
86   * @param key The input key.
87   */
88  void test_functionality(unsigned char *input_string, int
      key_length, unsigned char* key);
89
90      //Functions wat Armandt by gesit het
91      /**
92   * @brief Sets the key length, number of rounds and
       expanded key size.
93   * @param l Length of the key in bits
94   */
95      void set_key_length(int l);
96
97  /**
98   * @brief Set the number_of_rounds variable
99   * @param r Number of rounds
100  */
101 void set_number_of_rounds(int r);
102
103 /**
104  * @brief Set the expanded_key_size variable
105  * @param s Expanded key size
106  */
107 void set_expanded_key_size(int s);
108
```

```
109 /**
110  * @brief A function that combines zero padding and
        encryption of an
111  * arbitrarily-sized char array using AES_encrypt. This
        function will also
112  * call the set_key_length function to initialise those
        variables.
113  * @param message The char array that must be encrypted
114  * @param message_len Length of the message to be encrypted
        in bytes.
115  * @param encrypted The array where the encrypted message
        is stored
116  * @param key_length The length of the key in bits
117  * @param key The key used for encryption
118  * @return Returns an array containing the encrypted
        message
119  */
120 unsigned char* pad_and_encrypt(unsigned char * message,
      unsigned char * encrypted, int message_len, int
      key_length, unsigned char * key);
121
122 /**
123  * @brief Takes a longer encrypted message and decrypts it,
        returning an array containing the decrypted message
124  * @param message The encrypted message
125  * @param massage_len The length of the message in bytes
126  * @param key_length Length of the key in bits
127  * @param key The key used for encryption
128  * @return An array containing the message that has been
        decrypted (possibly padded with zeros)
129  */
130 unsigned char* general_decrypt(unsigned char * message, int
        message_len, int key_length, unsigned char * key);
131
132
133 #endif //ARMANDT_MICHELLE_H
```

II. Cipher Mode Files

A. *CipherModes.c*

```
1  //
2  // Created by fouri on 2020/03/27.
3  //
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include "CipherModes.h"
9  #include "AES.h"
10
11
12 void iterativeEncryptCBC(struct CBC *c){
13     int blockSize = (*c).blockSize;
14     int pSize = (*c).pSize;
15
16     int round = 0;
17
18     while
19         (!((((round + 1) * blockSize - pSize) >= blockSize)
               || (round * blockSize - pSize == 0))){
20             unsigned char temp[blockSize];
21
22             int i = round * blockSize; //i is the index
                  from where we will begin to copy chars
23             int j = 0;                 //j is the number of
                   items that have been copied over so far
24
25             while ((i < pSize) && (j < blockSize))
26             {
27                 temp[j] = (*c).plaintext[i];
28                 j++;
29                 i++;
30             } //first, copy chars from p to temp without
                  going beyond the scope of p
31             while (j < blockSize)
32             {
33                 temp[j] = 0;
34                 j++;
35             } //pad with zeros if necessary
36
37             //now we have temp, the block of input data we
                  want to work with
38             //         printf("XOR Output block: \t");
39             if (round == 0)
40             {
41                 for (int a = 0; a < blockSize; a++)
```

```
42                  {
43                      temp[a] = temp[a] ^ (*c).iv[a];
44                      //                  printf("%x ", temp[a
                            ]);
45                  } //xor the plaintext with the IV
46              }
47              else
48              {
49                  for (int a = 0; a < blockSize; a++)
50                  {
51                      temp[a] = temp[a] ^ (*c).ciphertext[a +
                            (round - 1) * blockSize]; //replace
                            this with the previous block's
                            ciphertext values
52                  } //xor the plaintext with the previous
                        block of ciphertext
53              }
54
55
56              //call AES function on the temp array
57              set_key_length((*c).keySize); //pass in the
                    size of the key in bits
58              unsigned char encrypted[blockSize + 1];
59              pad_and_encrypt(temp, encrypted, blockSize, (*c
                    ).keySize, (*c).key);
60
61
62              for (int a = 0; a < blockSize; a++)
63              {
64                  (*c).ciphertext[a + blockSize * round] =
                        encrypted[a];
65              }
66              round++;
67          }
68  }
69
70
71  void iterativeDecryptCBC(struct CBC *c){
72      int blockSize = (*c).blockSize;
73      int cSize = (*c).cSize;
74
75      int round = 0;
76
77      while ((round * blockSize) - cSize != 0) {
78          unsigned char temp[blockSize + 1];
79          unsigned char *decrypted;
80
81          //call aes decryption
82          for (int a = 0 + round * blockSize; a < blockSize +
```

```
                round * blockSize; a++)
83          {
84              temp[a - round * blockSize] = (*c).ciphertext[a
                    ];
85          }                           //copy ciphertext into temp
86          temp[blockSize] = '\0'; //append an endline char
87
88          general_decrypt(temp, blockSize, (*c).keySize, (*c)
                .key);    //decrypt the block
89
90          if (round == 0)
91          {
92              for (int a = 0; a < blockSize; a++)
93              {
94                  temp[a] = temp[a] ^ (*c).iv[a];
95                  (*c).plaintext[a + blockSize * round] =
                        temp[a];
96              } //perform the XOR step using the IV and store
                    the output in the plaintext array
97          }
98          else
99          {
100             for (int a = 0; a < blockSize; a++)
101             {
102                 temp[a] = temp[a] ^ (*c).ciphertext[a + (
                        round - 1) * blockSize];
103                 (*c).plaintext[a + blockSize * round] =
                        temp[a];
104             }//perform the xor step using the ciphertext
                    block and store the output in the plaintext
                    array
105         }
106         round++;
107     }
108 }
109
110 void shiftBytesIn(unsigned char* shiftReg, int regSize,
        unsigned char* newData, int newDataSize){
111     for (int a = 0; a < regSize - newDataSize; a++){
112         shiftReg[a] = shiftReg[a + newDataSize];
113     }
114
115     int b = 0;
116     for (int a = regSize - newDataSize; a < regSize; a++){
117         shiftReg[a] = newData[b++];
118     }
119 }
120
121
```

```
122 void iterativeEncryptCFB(struct CFB *c){
123     int blockSize = (*c).blockSize;
124     int pSize = (*c).pSize;
125     int shiftRegSize = (*c).shiftRegSize;
126 
127     int round = 0;
128 
129     while (round * blockSize <= pSize){
130         //step 1: encrypt the IV/Shift Register using the
              provided Key, K
131         unsigned char temp[shiftRegSize];
132         for (int a = 0; a < shiftRegSize; a++)
133         {
134             temp[a] = (*c).shiftRegister[a];
135         } //copy the shift register into a temp array
136 
137         unsigned char storage[shiftRegSize + 1];
138         unsigned char keyCopy[c->keySize / 8];
139         for (int a = 0; a < c->keySize / 8; a++)
140         {
141             keyCopy[a] = (*c).key[a];
142         }
143         pad_and_encrypt(temp, storage, shiftRegSize, (*c).
              keySize, keyCopy);
144 
145         //Step 2: XOR the LSB of the temp array with the
              plaintext
146         int b = 0;
147         for (int a = 0 + round * blockSize; a < round *
              blockSize + blockSize; a++)
148         {
149             //            (*c).ciphertext[a] = temp[b] =
                  temp[b] ^ (*c).plaintext[a];
150             (*c).ciphertext[a] = storage[b] = storage[b] ^
                  (*c).plaintext[a];
151             b++;
152         } //the first [blockSize] bytes of temp now
              contains the new block of ciphertext. And
              ciphertext has the new
153         //data in it as well
154 
155         //Step 3: Shift the ciphertext into the shift
              register before starting the next round.
156         shiftBytesIn((*c).shiftRegister, shiftRegSize,
              storage, blockSize);
157         round++;
158     }
159 }
160 
```

```
161
162 void iterativeDecryptCFB(struct CFB *c){
163     int blockSize = (*c).blockSize;
164     int pSize = (*c).pSize;
165     int shiftRegSize = (*c).shiftRegSize;
166
167     int round = 0;
168
169     while ((round)*blockSize <= pSize){
170         if (round == 0)
171         {
172             for (int a = 0; a < shiftRegSize; a++)
173             {
174                 (*c).shiftRegister[a] = (*c).iv[a];
175             } //the shift register should begin the same as
                    the IV
176         }
177         else
178         {
179             //              printf("Shift Register: \t");
180             //              printArr((*c).shiftRegister,
                    shiftRegSize, 'x');
181         }
182
183         //step 1: encrypt the IV/Shift Register using the
                provided Key, K
184         unsigned char temp[shiftRegSize];
185         if (round == 0)
186         {
187             for (int a = 0; a < shiftRegSize; a++)
188             {
189                 temp[a] = (*c).iv[a];
190             } //copy the IV into a temp array
191         }
192         else
193         {
194             for (int a = 0; a < shiftRegSize; a++)
195             {
196                 temp[a] = (*c).shiftRegister[a];
197             } //copy the shift register into a temp array
198         }
199
200         unsigned char storage[shiftRegSize + 1];
201         pad_and_encrypt(temp, storage, shiftRegSize, (*c).
                keySize, (*c).key);
202
203         //step 2: XOR the first s bits of the encrypted
                output with the first s bits of ciphertext to get
                 the plaintext
```

```
204             int b = blockSize * round;
205             for (int a = 0; a < blockSize; a++)
206             {
207                 //              (*c).plaintext[a + b] = temp[a] ^
                        (*c).ciphertext[b + a];
208                 (*c).plaintext[a + b] = storage[a] ^ (*c).
                        ciphertext[b + a];
209             }
210
211             //step 3: shift ciphertext block into shiftreg
                    before running next step
212             for (int a = 0; a < blockSize; a++)
213             {
214                 temp[a] = (*c).ciphertext[a + b];
215             } //copy ciphertext block into temp array (just so
                    I can use the shiftBytesIn function more easily)
216
217             shiftBytesIn((*c).shiftRegister, shiftRegSize, temp
                    , blockSize);
218             round++;
219         }
220     }
221
222
223     void printArr(unsigned char *arr, int size, char format){
224         if (format == 'c'){
225             for (int a = 0; a < size; a++){
226                 printf("%c", arr[a]);
227             }
228         } else if (format == 'x'){
229             for (int a = 0; a < size; a++){
230                 printf("%x␣", arr[a]);
231             }
232         } else if (format == 'd'){
233             for (int a = 0; a < size; a++){
234                 printf("%d␣", arr[a]);
235             }
236         }
237
238         printf("\n");
239     }
240
241     void readFile(unsigned char * filename, unsigned char *
            fileBuffer){
242         FILE *f;
243         f = fopen(filename, "rb");  //open binary file
244         long int fileSize = 0;
245
246         if (f == NULL){
```

```
247             printf("Error, file not found.\n");
248             exit(0);
249         } else {
250             fileSize = getFileSize(filename);
251 
252             printf("Reading %Ld bytes from file.\n", fileSize);
253             fread(fileBuffer, fileSize + 1, 1, f);
254         }
255 
256         fclose(f);
257 }
258 
259 void saveFile(unsigned char * filename, unsigned char *
       fileBuffer, int fileSize){
260         FILE * f;
261         f = fopen(filename, "wb");
262 
263         int numZeros = 0;    //the number of zeros added to a
            file when using CBC
264         int a = fileSize - 1;
265         while(fileBuffer[a] == 0){
266             a--;
267             numZeros++;
268         }
269 
270         fileSize -= numZeros;    //this stops zeroes that were
            added for padding from being saved upon decryption
271 
272         if (f == NULL) {
273             printf("Error. File could not be opened.\n");
274         } else {
275             fwrite(fileBuffer, fileSize, 1, f);
276         }
277         fclose(f);
278 }
279 
280 long int getFileSize(unsigned char * filename){
281         FILE *f;
282         f = fopen(filename, "rb");  //open binary file
283         long int fileSize = 0;
284 
285         if (f == NULL){
286             printf("Error, file not found.\n");
287         } else {
288             fseek(f, 0L, SEEK_END);
289             fileSize = ftell(f);
290         }
291 
292         fclose(f);
```

```
293         return fileSize;
294     }
```

B. CipherModes.h

```
1  //
2  // Created by fouri on 2020/03/30.
3  //
4
5  #ifndef cipher_modes_H
6  #define cipher_modes_H
7  #include <stdio.h>
8
9  /**
10  * A structure to hold relevant info for doing cipher block
        chaining
11  */
12 struct CBC{
13     int pSize;          //size of plaintext in BYTES = no.
          of array indexes
14     int cSize;          //size of the ciphertext created in
          BYTES
15     int blockSize;      //size of each block in BYTES
16     int keySize;        //size of the key in bits
17     unsigned char* plaintext;   //array of chars
18     unsigned char* key;
19     unsigned char* ciphertext;
20     unsigned char* iv;          //this holds the
          initialization vector
21 };
22
23 /**
24  * The structure for the CFB methods. The shiftregister and
        IV should be initialized as having the same contents,
25  * but not being the same object.
26  */
27 struct CFB{
28     int pSize;           //size of the plaintext message/
          ciphertext
29     int shiftRegSize;    //bytes in the IV/Shift Register
30     int blockSize;       //number of bytes processed per
          round
31     int keySize;         //size of the key in bits
32     unsigned char* plaintext;
33     unsigned char* ciphertext;
34     unsigned char* iv;  //init vector. THis does not get
          changed.
35     unsigned char* shiftRegister;
36     unsigned char* key;
37 };
38
39 /**
```

```
40  * @brief Takes a CBC struct and encrypts the data it
        contains, storing the ciphertext in the struct itself.
41  * @param c The CBC structure used.
42  */
43 void iterativeEncryptCBC(struct CBC * c);
44
45
46 /**
47  * @brief Takes a CBC struct and and decrypts the data it
        contains iteratively rather than recursively.
48  * @param c The CBC structure used.
49  */
50 void iterativeDecryptCBC(struct CBC * c);
51
52 /**
53  * @brief Acts as a shift register, shifting new data into
        an existing array.
54  * @param shiftReg The register that is accepting new data
55  * @param regSize The size of the accepting register
56  * @param newData The array containing the new data
57  * @param newDataSize The size of the new data array.
58  */
59 void shiftBytesIn(unsigned char* shiftReg, int regSize,
    unsigned char* newData, int newDataSize);
60
61 /**
62  * @brief Uses the cipher feedback mode to encrypt a
        message. Internally, this function uses the AES
63  * encryption algorithm to encrypt the IV/Shift Register
64  * @param c The CFB struct containing the plaintext and
        other relevant information
65  */
66 void iterativeEncryptCFB(struct CFB *c);
67
68 /**
69  * @brief Uses the cipher feedback mode to decrypt a
        message. Internally, this function uses the AES
70  * decryption algorithm to decrypt the IV/Shift Register
71  * @param c The CFB struct containing the plaintext and
        other relevant information
72  */
73 void iterativeDecryptCFB(struct CFB *c);
74
75
76 /**
77  * @brief Prints the contents of an array separated by
        spaces, with a newline at the end
78  * @param arr The array containing the chars
79  * @param size The size of the array
```

```
80   * @param format The char that indicates to the printf
         function how it should display the data in the array
81   */
82  void printArr(unsigned char *arr, int size, char format);
83
84
85  /**
86   * @brief Accepts a filename and opens the file. The
         function assumes that the file is located in
87   * the root directory of the program. (Same folder as main.
         c)
88   * @param filename The name of the file that must be
         encrypted.
89   */
90  void readFile(unsigned char * filename, unsigned char *
       fileBuffer);
91
92  /**
93   * @brief Saves the given array as a file, designated by
         the filename.
94   * @param filename Name of the file, including the
         extension.
95   * @param fileBuffer Array containing data.
96   * @param fileSize Number of elements in fileBuffer.
97   */
98  void saveFile(unsigned char * filename, unsigned char *
       fileBuffer, int fileSize);
99
100 /**
101  * @brief Returns the size of the file in bytes.
102  * @param filename The filename
103  * @return File size in bytes
104  */
105 long int getFileSize(unsigned char * filename);
106 #endif //ARMANDT_ARMANDT_H
```

III. Main and makefile

A. *main.c*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <printf.h>
5  #include "string.h"
6  #include "CipherModes.h"
7  #include "AES.h"
8
9  int main(int argc, char *argv[])
10 {
11
12     bool encrypt = false;
13     bool decrypt = false;
14     bool cbc = false;
15     bool cfb = false;
16     bool textIn = false;
17     bool fileIn = false;
18     unsigned char inputText[1000];
19     unsigned char key[33];
20     unsigned char iv[101];
21     unsigned char *inFileName = NULL;
22     unsigned char outFileName[100];
23     unsigned char tempOutFileName[200];
24     unsigned char newFileBuffer[1000000]; //approx 1mb file
25     int keyLength = 0;  //the length of the key given by
          the user
26     int streamLen = 0;
27     int argLength = 0;  //the length of arguments like text
          , keys, etc.
28     int plaintextLength = 0;
29
30     struct CBC cbcStruct;
31     struct CFB cfbStruct;
32
33     for (int a = 1; a < argc; a++)
34     {
35         if (strcmp(argv[a], "-e") == 0)
36         {
37             encrypt = true;
38             if (!((strcmp(argv[a + 1], "-cbc") == 0) || (
                  strcmp(argv[a + 1], "-cfb") == 0))){
39                 keyLength = atoi(argv[a+1]);
40             }   //if neither cbc nor cfb are used, then the
                    keylength must still be entered.
41         }
42         else if (strcmp(argv[a], "-d") == 0)
```

```
43          {
44              decrypt = true;
45          }
46          else if (strcmp(argv[a], "-cbc") == 0)
47          {
48              cbc = true;
49              keyLength = atoi(argv[a + 1]);
50              if (!((keyLength == 128) || (keyLength == 192)
                    || (keyLength == 256)))
51              {
52                  printf("Incorrect␣key␣size␣entered.␣Closing
                        .\n");
53                  return 0;
54              }
55              cbcStruct.keySize = keyLength;
56              cbcStruct.blockSize = 16;
57          }
58          else if (strcmp(argv[a], "-cfb") == 0)
59          {
60              cfb = true;
61              keyLength = atoi(argv[a + 1]);
62              if (!((keyLength == 128) || (keyLength == 192)
                    || (keyLength == 256)))
63              {
64                  printf("Incorrect␣key␣size␣entered.␣Closing
                        .\n");
65                  return 0;
66              }
67              cfbStruct.keySize = keyLength;
68          }
69          else if (strcmp(argv[a], "-t") == 0)
70          {
71              textIn = true;
72              argLength = strlen(argv[a + 1]);
73              unsigned char t[argLength + 1];
74              for (int b = 0; b < argLength; b++){
75                  inputText[b] = argv[a+1][b];
76              }
77              inputText[argLength] = '\0';
78  //            plaintext = inputText;
79              plaintextLength = argLength;
80              if (cbc){
81                  if (encrypt){
82                      cbcStruct.plaintext = inputText;
83                      cbcStruct.pSize = plaintextLength;
84                      cbcStruct.cSize = (cbcStruct.pSize / 16
                            + 1) * 16;
85                  } else {
86                      cbcStruct.ciphertext = inputText;
```

```
87                      cbcStruct.cSize = plaintextLength;
88                      cbcStruct.pSize = cbcStruct.cSize;
89                  }
90              } else if (cfb){
91                  if (encrypt){
92                      cfbStruct.pSize = plaintextLength;
93                      cfbStruct.plaintext = inputText;
94                  } else {
95                      cfbStruct.pSize = plaintextLength;
96                      cfbStruct.ciphertext = inputText;
97                  }
98              }
99              a++;
100         }
101         else if (strcmp(argv[a], "-key") == 0)
102         {
103             argLength = strlen(argv[a + 1]);
104             for (int b = 0; b < argLength; b++){
105                 key[b] = argv[a+1][b];
106             }
107             key[argLength] = '\0';
108 //            key = key;
109
110             if (cbc){
111                 cbcStruct.key = key;
112             } else if (cfb) {
113                 cfbStruct.key = key;
114                 cfbStruct.shiftRegSize = 32;
115             }
116             a++;
117         }
118         else if (strcmp(argv[a], "-iv") == 0)
119         {
120             argLength = strlen(argv[a + 1]);
121 //            unsigned char iv[argLength + 1];
122             for (int b = 0; b < argLength; b++){
123                 iv[b] = argv[a+1][b];
124             }
125             iv[argLength] = '\0';
126
127             if (cbc) {
128                 cbcStruct.iv = iv;
129                 cbcStruct.blockSize = 16;
130             } else if (cfb) {
131                 cfbStruct.iv = iv;
132                 cfbStruct.shiftRegSize = argLength;
133             }
134             a++;
135         }
```

```
136         else if (strcmp(argv[a], "-fi") == 0)
137         {
138             fileIn = true;
139             argLength = strlen(argv[a + 1]);
140             unsigned char temp[argLength + 1];
141             for (int b = 0; b < argLength; b++){
142                 temp[b] = argv[a+1][b];
143                 tempOutFileName[b] = temp[b];
144             }
145             temp[argLength] = '\0';
146             inFileName = temp;
147             plaintextLength = getFileSize(inFileName);
148             readFile(inFileName, newFileBuffer);
149 
150             for (int b = 0; b < argLength; b++){
151                 inFileName[b] = 0;
152             }
153 
154             if (cbc) {
155                 if (encrypt){
156                     cbcStruct.plaintext = newFileBuffer;
157                     cbcStruct.pSize = plaintextLength;
158                     cbcStruct.cSize = (cbcStruct.pSize / 16
                            + 1) * 16;
159                 } else {
160                     cbcStruct.ciphertext = newFileBuffer;
161                     cbcStruct.cSize = plaintextLength;
162                     cbcStruct.pSize = plaintextLength;
163                 }
164             } else if (cfb) {
165                 if (encrypt){
166                     cfbStruct.plaintext = newFileBuffer;
167                     cfbStruct.pSize = plaintextLength;
168                 } else {
169                     cfbStruct.ciphertext = newFileBuffer;
170                     cfbStruct.pSize = plaintextLength;
171                 }
172             }
173             a++;
174         }
175         else if (strcmp(argv[a], "-fo") == 0)
176         {
177             argLength = strlen(argv[a + 1]);
178             if (argLength > 100){
179                 printf("The␣output␣file␣name␣entered␣is␣too
                        ␣long.␣Please␣use␣fewer␣than␣100␣
                        characters.\n");
180                 return 0;
181             }
```

```
182
183             for (int b = 0; b < argLength - 2; b++){
184                 outFileName[b] = argv[a+1][b + 2];
185             }
186             outFileName[argLength] = '\0';
187
188             for (int b = 199; b > 0; b--){
189                 if (tempOutFileName[b] == '/'){
190                     if (cbc){
191                         tempOutFileName[b + 1] = 'C';
192                         tempOutFileName[b + 2] = 'B';
193                         tempOutFileName[b + 3] = 'C';
194                     } else {
195                         tempOutFileName[b + 1] = 'C';
196                         tempOutFileName[b + 2] = 'F';
197                         tempOutFileName[b + 3] = 'B';
198                     }
199                     tempOutFileName[b + 4] = '␣';
200                     tempOutFileName[b + 5] = 'O';
201                     tempOutFileName[b + 6] = 'u';
202                     tempOutFileName[b + 7] = 't';
203                     tempOutFileName[b + 8] = 'p';
204                     tempOutFileName[b + 9] = 'u';
205                     tempOutFileName[b + 10] = 't';
206                     tempOutFileName[b + 11] = '/';
207
208                     for (int c = 0; c < argLength - 2; c++)
                        {
209                         tempOutFileName[b + c + 11] =
                            outFileName[c];
210                     }// add the part of the path for the
                        folders and the name for the output
                        file
211                     break;
212                 }
213             }
214             a++;
215         }
216         else if (strcmp(argv[a], "-streamlen") == 0)
217         {
218             streamLen = atoi(argv[a + 1]);
219
220
221             if (cfb) {
222                 if (!((streamLen == 8) || (streamLen == 64)
                        || (streamLen == 128))){
223                     printf("Please␣try␣again␣and␣enter␣a␣
                        valid␣streamlength.\n");
224                     return 0;
```

```
225                     }
226                     cfbStruct.blockSize = atoi(argv[a + 1]);
227                 } else if (cbc){
228                     printf("Streamlen is not allowed for CBC
                        operation. Please try again.\n");
229                     return 0;
230                 }
231                 a++;
232             }
233             else if (strcmp(argv[a], "-h") == 0)
234             {
235                 printf("\n
                    ==========================================================
                    n");
236                 printf("The following commands are available: \
                    n");
237                 printf("-e: \t\t\t\t Encryption\n");
238                 printf("-d: \t\t\t\t Decryption\n");
239                 printf("-cbc <len >:\t\t\t CBC Encryption/
                    Decryption\n");
240                 printf("-cfb <len >:\t\t\t CFB Encryption/
                    Decryption\n");
241                 printf("<len >: \t\t\t\t Key length: either 128,
                    192 or 256\n");
242                 printf("-t <text >:\t\t\t Enter the text to
                    encrypt after this tag, surrounded by
                    quotation marks.\n");
243                 printf("-key <password >: \t Enter the password
                    after this tag.\n");
244                 printf("-iv <init vect >: \t Enter the
                    initialisation vector after this tag.\n");
245                 printf("-fi <input file >:\t Enter the name of
                    the input file.\n");
246                 printf("-fo <output file >:\t Enter the name of
                    the output file.\n");
247                 printf("-streamlen <len >: \t Enter the
                    streamlength after this tag.\n");
248                 printf("-h: \t\t\t\t Enter this tag to display
                    this message.\n");
249                 printf("
                    ==========================================================
                    n");
250             }
251         }
252
253         int s = 0;
254         if (cbc){
255             if (encrypt){
256                 s = cbcStruct.cSize;  //make an array to hold
```

```
                the ciphertext
257         } else {
258             s = cbcStruct.pSize;  //plaintext array will be
                    same size as ciphertext
259         }
260     } else if (cfb){
261         s = cfbStruct.pSize;    //both arrays are the same
               size for cfb
262     } else {
263         s = (plaintextLength / 16 + 1) * 16;
264     }
265
266     unsigned char newArray[s + 1];  //this will store
           cipher or plaintext
267
268     if (cbc) {
269         if (encrypt){
270             cbcStruct.ciphertext = newArray;
271             printf("Encryption has started.\n");
272             iterativeEncryptCBC(&cbcStruct);
273             printf("Done encrypting.\n");
274
275
276             if (fileIn){
277                 saveFile(tempOutFileName, cbcStruct.
                       ciphertext, cbcStruct.cSize);
278                 printf("File saved in the CBC folder.\n");
279             } else if (textIn){
280                 printArr(cbcStruct.ciphertext, cbcStruct.
                       cSize, 'x');
281             }
282
283 //              iterativeDecryptCBC(&cbcStruct);
284 //              printArr(cbcStruct.plaintext, cbcStruct.pSize
        , 'c');
285
286         } else {
287             cbcStruct.plaintext = newArray;
288             printf("Decryption has started.\n");
289             iterativeDecryptCBC(&cbcStruct);
290             printf("Done decrypting. \n");
291 //              printArr(cbcStruct.plaintext, cbcStruct.pSize
        , 'c');
292
293             if (fileIn){
294                 saveFile(tempOutFileName, cbcStruct.
                       plaintext, cbcStruct.pSize);
295                 printf("File saved in the CBC folder.\n");
296             }
```

```
297
298 //              printArr(cbcStruct.plaintext, cbcStruct.pSize
        , 'c');
299             }
300         } else if (cfb) {
301             unsigned char newShiftReg[cfbStruct.shiftRegSize];
302             cfbStruct.shiftRegister = newShiftReg;
303             for (int a = 0; a < cfbStruct.shiftRegSize; a++){
304                 newShiftReg[a] = iv[a];
305             }
306
307             if (encrypt){
308                 cfbStruct.ciphertext = newArray;
309                 printf("Encryption has started.\n");
310                 iterativeEncryptCFB(&cfbStruct);
311                 printf("Done encrypting.\n");
312                 // printArr(cfbStruct.ciphertext, cfbStruct.
                       pSize, 'x');
313
314                 if (fileIn){
315                     saveFile(tempOutFileName, cfbStruct.
                           ciphertext, cfbStruct.pSize);
316                     printf("File saved in the CFB folder.\n");
317                 }
318
319 //                iterativeDecryptCFB(&cfbStruct);
320 //                printArr(cfbStruct.plaintext, cfbStruct.pSize
        , 'c');
321             } else {
322                 cfbStruct.plaintext = newArray;
323                 printf("Decryption has started.\n");
324                 iterativeDecryptCFB(&cfbStruct);
325                 printf("Done decrypting.\n");
326
327                 if (fileIn){
328                     saveFile(tempOutFileName, cfbStruct.
                           plaintext, cfbStruct.pSize);
329                     printf("File saved in the CFB folder.\n");
330                 }
331             }
332         } else {
333             if (encrypt){
334                 set_key_length(keyLength);
335                 test_functionality(inputText, keyLength, key);
336 //                pad_and_encrypt(inputText, newArray,
        plaintextLength, keyLength, key);
337             } else {
338                 general_decrypt(inputText, plaintextLength,
                       keyLength, key);
```

```
339             }
340
341         }    //neither cbc nor cfb
342
343         return 0;
344     }
```

B. makefile

```
1  main: main.o CipherModes.o AES.o
2          gcc -static main.o CipherModes.o AES.o -o main
3
4  main.o: main.c CipherModes.h AES.h
5          gcc -c main.c
6
7  CipherModes.o: CipherModes.c CipherModes.h
8          gcc -c CipherModes.c
9
10 AES.o: AES.c AES.h
11         gcc -c AES.c
12
13 run:
14         ./main
15
16 clean:
17         rm -f main CipherModes.o AES.o main.o
```