h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences

– Faculty of Computer Science –

cryptography lab report 2

by
Lennart Eichhorn
Matriculation number: 759253

Part I

REPORT

1 INTRODUCTION

Text goes here

2 APPENDIX

Listing 1. Simple reference AES implementation

```cpp
#include <iostream>


const unsigned int NUM_ROUNDS = 4 + 6;

static const unsigned char SBOX[256] =
{
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
```

```
 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
```

```
  0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
```

```
  0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
```

```
  0x27, 0xb2, 0x75,
12   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
```

```
  0xe3, 0x2f, 0x84,
13   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
```

```
  0x4c, 0x58, 0xcf,
14   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
```

```
  0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
```

```
  0xff, 0xf3, 0xd2,
16   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
```

```
  0x5d, 0x19, 0x73,
17   0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
```

```
  0x5e, 0x0b, 0xdb,
18   0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
```

```
  0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
```

```
  0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
```

```
  0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
```

```
  0xc1, 0x1d, 0x9e,
22   0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
```

```
  0x55, 0x28, 0xdf,
23   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
```

```
  0x54, 0xbb, 0x16
};

static const unsigned char INV_SBOX[256] = {0};

static const unsigned char RCON[255] =
{
  0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
```

```
  0xab, 0x4d, 0x9a,
31   0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
```

```
  0xc5, 0x91, 0x39,
32   0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
```

```
  0x83, 0x1d, 0x3a,
33   0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
```

```
  0x36, 0x6c, 0xd8,
34   0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
```

```
  0x7d, 0xfa, 0xef,
35   0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
```

```
  0x33, 0x66, 0xcc,
36   0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
```

```
  0x40, 0x80, 0x1b,
37   0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
```

```
  0x6a, 0xd4, 0xb3,
38   0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
```

```
  0x25, 0x4a, 0x94,
39   0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
```

```
  0x08, 0x10, 0x20,
40   0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
```

```
  0xc6, 0x97, 0x35,
41   0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
```

```
  0x61, 0xc2, 0x9f,
42   0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
```

```
  0x01, 0x02, 0x04,
43   0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
```

```
  0x5e, 0xbc, 0x63,
44   0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
```

```
  0xe4, 0xd3, 0xbd,
45   0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74,
```

```
  0xe8, 0xcb
};


void keyExpansion(const unsigned char* key, unsigned char* roundKey)
{
    unsigned char temp[4], k;

    for (unsigned int i = 0; i < 4; ++i)
    {
        roundKey[i * 4 + 0] = key[i * 4 + 0];
        roundKey[i * 4 + 1] = key[i * 4 + 1];
        roundKey[i * 4 + 2] = key[i * 4 + 2];
        roundKey[i * 4 + 3] = key[i * 4 + 3];
    }

    for (unsigned int i = 4; i < 4 * (NUM_ROUNDS + 1); ++i)
    {
        for (unsigned int j = 0; j != 4; ++j)
            temp[j] = roundKey[(i - 1) * 4 + j];
        if (i % 4 == 0)
        {
            k = SBOX[temp[0]];
            temp[0] = SBOX[temp[1]];
            temp[1] = SBOX[temp[2]];
            temp[2] = SBOX[temp[3]];
            temp[3] = k;

            temp[0] =  temp[0] ^ RCON[i / 4];
        }
        roundKey[i * 4 + 0] = roundKey[(i - 4) * 4 + 0] ^ temp[0];
        roundKey[i * 4 + 1] = roundKey[(i - 4) * 4 + 1] ^ temp[1];
        roundKey[i * 4 + 2] = roundKey[(i - 4) * 4 + 2] ^ temp[2];
        roundKey[i * 4 + 3] = roundKey[(i - 4) * 4 + 3] ^ temp[3];
    }
}


void addRoundKey(unsigned char* state, const unsigned char* roundKey, int round)
{
  for (unsigned int i = 0; i != 4; ++i)
  {
    for (unsigned int j = 0; j != 4; ++j)
      state[j * 4 + i] ^= roundKey[round * 4 * 4 + i * 4 + j];
  }
}


```

```
93  void subBytes(unsigned char* state)
94  {
95    for (unsigned int i = 0; i != 4; ++i)
96    {
97      for (unsigned int j = 0; j != 4; ++j)
98        state[i * 4 + j] = SBOX[state[i * 4 + j]];
99    }
100 }
101
102
103 void invSubBytes(unsigned char* state)
104 {
105   for (unsigned int i = 0; i != 4; ++i)
106   {
107     for (unsigned int j = 0; j != 4; ++j)
108       state[i * 4 + j] = INV_SBOX[state[i * 4 + j]];
109   }
110 }
111
112
113 void shiftRows(unsigned char* state)
114 {
115   unsigned char temp;
116
117   // Rotate first row 1 columns to left
118   temp = state[1 * 4 + 0];
119   state[1 * 4 + 0] = state[1 * 4 + 1];
120   state[1 * 4 + 1] = state[1 * 4 + 2];
121   state[1 * 4 + 2] = state[1 * 4 + 3];
122   state[1 * 4 + 3] = temp;
123
124   // Rotate second row 2 columns to left
125   temp = state[2 * 4 + 0];
126   state[2 * 4 + 0] = state[2 * 4 + 2];
127   state[2 * 4 + 2] = temp;
128
129   temp = state[2 * 4 + 1];
130   state[2 * 4 + 1] = state[2 * 4 + 3];
131   state[2 * 4 + 3] = temp;
132
133   // Rotate third row 3 columns to left
134   temp = state[3 * 4 + 0];
135   state[3 * 4 + 0] = state[3 * 4 + 3];
136   state[3 * 4 + 3] = state[3 * 4 + 2];
137   state[3 * 4 + 2] = state[3 * 4 + 1];
138   state[3 * 4 + 1] = temp;
139 }
140
141
142 void invShiftRows(unsigned char* state)
143 {
144   unsigned char temp;
```

```
145
146   // Rotate first row 1 columns to right
147   temp = state[1 * 4 + 3];
148   state[1 * 4 + 3] = state[1 * 4 + 2];
149   state[1 * 4 + 2] = state[1 * 4 + 1];
150   state[1 * 4 + 1] = state[1 * 4 + 0];
151   state[1 * 4 + 0] = temp;
152
153   // Rotate second row 2 columns to right
154   temp = state[2 * 4 + 0];
155   state[2 * 4 + 0] = state[2 * 4 + 2];
156   state[2 * 4 + 2] = temp;
157
158   temp = state[2 * 4 + 1];
159   state[2 * 4 + 1] = state[2 * 4 + 3];
160   state[2 * 4 + 3] = temp;
161
162   // Rotate third row 3 columns to right
163   temp = state[3 * 4 + 0];
164   state[3 * 4 + 0] = state[3 * 4 + 1];
165   state[3 * 4 + 1] = state[3 * 4 + 2];
166   state[3 * 4 + 2] = state[3 * 4 + 3];
167   state[3 * 4 + 3] = temp;
168 }
169
170
171 // XTIME is a macro that finds the product of {02} and the argument to XTIME modulo
    {1b}
172 #define XTIME(x) (((x) << 1) ^ ((((x) >> 7) & 1) * 0x1b))
173
174 // Multiplty is a macro used to multiply numbers in the field GF(2^8)
175 #define MULTIPLY(x, y) \
176 ( \
177   (((y) & 1) * (x)) ^ \
178   (((y) >> 1 & 1) * XTIME(x)) ^ \
179   (((y) >> 2 & 1) * XTIME(XTIME(x))) ^ \
180   (((y) >> 3 & 1) * XTIME(XTIME(XTIME(x)))) ^ \
181   (((y) >> 4 & 1) * XTIME(XTIME(XTIME(XTIME(x))))) \
182 )
183
184
185 void mixColumns(unsigned char* state)
186 {
187   unsigned char Tmp, t;
188   for (unsigned int i = 0; i != 4; ++i)
189   {
190     t = state[0 * 4 + i];
191     Tmp = state[0 * 4 + i] ^ state[1 * 4 + i] ^ state[2 * 4 + i] ^ state[3 * 4 + i];
```

```
192     state[0 * 4 + i] ^= XTIME(state[0 * 4 + i] ^ state[1 * 4 + i]) ^ Tmp;
193     state[1 * 4 + i] ^= XTIME(state[1 * 4 + i] ^ state[2 * 4 + i]) ^ Tmp;
194     state[2 * 4 + i] ^= XTIME(state[2 * 4 + i] ^ state[3 * 4 + i]) ^ Tmp;
195     state[3 * 4 + i] ^= XTIME(state[3 * 4 + i] ^ t) ^ Tmp;
196   }
197 }
198
199
200 void invMixColumns(unsigned char* state)
201 {
202   unsigned char a, b, c, d;
203   for (unsigned int i = 0; i != 4; ++i)
204   {
205     a = state[0 * 4 + i];
206     b = state[1 * 4 + i];
207     c = state[2 * 4 + i];
208     d = state[3 * 4 + i];
209
210     state[0 * 4 + i] = MULTIPLY(a, 0x0e) ^ MULTIPLY(b, 0x0b) ^ MULTIPLY(c, 0x0d
```

```
    ) ^ MULTIPLY(b, 0x0b) ^ MULTIPLY(c, 0x0d) ^ MULTIPLY(d, 0x09);
211     state[1 * 4 + i] = MULTIPLY(a, 0x09) ^ MULTIPLY(b, 0x0e) ^ MULTIPLY(c, 0x0b
```

```
    ) ^ MULTIPLY(b, 0x0e) ^ MULTIPLY(c, 0x0b) ^ MULTIPLY(d, 0x0d);
212     state[2 * 4 + i] = MULTIPLY(a, 0x0d) ^ MULTIPLY(b, 0x09) ^ MULTIPLY(c, 0x0e
```

```
    ) ^ MULTIPLY(b, 0x09) ^ MULTIPLY(c, 0x0e) ^ MULTIPLY(d, 0x0b);
213     state[3 * 4 + i] = MULTIPLY(a, 0x0b) ^ MULTIPLY(b, 0x0d) ^ MULTIPLY(c, 0x09
```

```
  ) ^ MULTIPLY(b, 0x0d) ^ MULTIPLY(c, 0x09) ^ MULTIPLY(d, 0x0e);
214   }
215 }
216
217
218 void cipher(const unsigned char* in, const unsigned char* roundKey, unsigned char* o
    ut)
219 {
220   unsigned char state[4 * 4];
221
222   for (unsigned int i = 0; i != 4; ++i)
223   {
224     for (unsigned int j = 0; j != 4; ++j)
225       state[j * 4 + i] = in[i * 4 + j];
226   }
227
228   addRoundKey(state, roundKey, 0);
229   for (unsigned int round = 1; round < NUM_ROUNDS; ++round)
230   {
231     subBytes(state);
232     shiftRows(state);
233     mixColumns(state);
234     addRoundKey(state, roundKey, round);
235   }
236   subBytes(state);
237   shiftRows(state);
238   addRoundKey(state, roundKey, NUM_ROUNDS);
239
240   for (unsigned int i = 0; i != 4; ++i)
241   {
242     for (unsigned int j = 0; j != 4; ++j)
243       out[i * 4 + j] = state[j * 4 + i];
244   }
245 }
246
247
248 void decipher(const unsigned char* in, const unsigned char* roundKey, unsigned char*
    out)
249 {
250   unsigned char state[4 * 4];
251
252   for (unsigned int i = 0; i != 4; ++i)
253   {
254     for (unsigned int j = 0; j != 4; ++j)
255       state[j * 4 + i] = in[i * 4 + j];
256   }
257
258   addRoundKey(state, roundKey, NUM_ROUNDS);
259   for (unsigned int round = NUM_ROUNDS - 1; round > 0; --round)
260   {
261     invShiftRows(state);
262     invSubBytes(state);
263     addRoundKey(state, roundKey, round);
264     invMixColumns(state);
```

```
265   }
266   invShiftRows(state);
267   invSubBytes(state);
268   addRoundKey(state, roundKey, 0);
269 
270   for (unsigned int i = 0; i != 4; ++i)
271   {
272     for (unsigned int j = 0; j != 4; ++j)
273       out[i * 4 + j] = state[j * 4 + i];
274   }
275 }
276 
277 
278 int main(int argc, char* argv[])
279 {
280   unsigned char roundKey[240];
281   unsigned char out[16];
282 
283   // Sample
284   {
285     const unsigned char in[16] =
286     {
287       'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
288       'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'
289     };
290     const unsigned char key[16] =
291     {
292       0xa3, 0x28, 0x4e, 0x09, 0xc6, 0xfe, 0x53, 0x29,
293       0x97, 0xef, 0x6d, 0x10, 0x74, 0xc3, 0xde, 0xad
294     };
295 
296     std::cout << std::endl << "Text before encryption:" << std::hex << std::endl;
297     for (unsigned int i = 0; i != 4 * 4; ++i)
298       std::cout << "0x" << (unsigned int)in[i] << ", ";
299     std::cout << std::endl;
300 
301     keyExpansion(key, roundKey);
302     cipher(in, roundKey, out);
303 
304     std::cout << std::endl << "Text after encryption:" << std::hex << std::endl;
305     for (unsigned int i = 0; i != 4 * 4; ++i)
306       std::cout << "0x" << (unsigned int)out[i] << ", ";
307     std::cout << std::endl;
308   }
309 
310   return 0;
311 }
```