



Darmstadt University of Applied Sciences
– Faculty of Computer Science –

cryptography lab report 2

by
Lennart Eichhorn
Matriculation number: 759253

Part I
REPORT

INTRODUCTION

1.1 SIMPLE AES PROGRAMMING

The supplied simple AES implementation shown [Listing 4](#) was supplied as a reference. However it is missing the inverse S-Box required for decryption. Our first task is to implement the inverse S-Box and then use it to implement the decryption function.

We wanted to have a known working encryption and decryption implementation to compare our implementation against. For this we build simple implementation using the openssl library, as shown in [Listing 5](#). We then verified that the openssl implementation produces the same output as the simple implementation.

[Listing 1](#) shows our function for calculation the inverted SBOX table. It swaps the indices and values of the SBOX table, to create the inverted table. We added the inverted SBOX into the simple AES implementation. We confirmed that the decryption function now works correctly by fuzzing the implementation against 1000000 random blocks and keys. As shown in [Listing 2](#), we validated the result for each block by comparing it with the openssl implementation.

Listing 1. openssl based AES tool

```

1  int main() {
2      unsigned char inversedSbox[256];
3      for (unsigned int i = 0; i != 256; ++i) {
4          inversedSbox[SBOX[i]] = i;
5      }
6
7      std::printf("static const unsigned char INVERTED_SBOX[256] = {");
8      for (auto element : inversedSbox) {
9          std::printf("0x%x,", element);
10     }
11     std::printf("};\n");
12 }
```

CPP

Listing 2. Setup for comparing our implementation against openssl

```

1  int main() {
2      int devRandom = open("/dev/random", O_RDONLY);
3
4      for (int round = 0; round < 1000000; ++round) {
5          std::array<unsigned char, 16> plaintext, key;
6          read(devRandom, plaintext.data(), 16);
7          read(devRandom, key.data(), 16);
8
9          // Cipher with our implementation
```

CPP

```

10  auto ciphertext = simple_encrypt(plaintext, key);
11  auto decryptedPlaintext = simple_decrypt(ciphertext, key);
12  // Cipher with openssl
13  auto ciphertextOpenssl = simple_openssl_encrypt(plaintext, key);
14  auto decryptedPlaintextOpenssl =
15      simple_openssl_decrypt(ciphertextOpenssl, key);
16
17  if (!std::ranges::equal(ciphertext, ciphertextOpenssl)) {
18      printf("OpenSSL and simple encryption produced different ciphertexts");
19      exit(1);
20  }
21  if (!std::ranges::equal(plaintext, decryptedPlaintext)) {
22      printf("Decryption did not produce the original plaintext");
23      exit(1);
24  }
25  if (!std::ranges::equal(decryptedPlaintext, decryptedPlaintextOpenssl)) {
26      printf("OpenSSL and simple decryption produced different results");
27      exit(1);
28  }
29  }
30  std::printf("Encryption and decryption seems to work\n");
31 }

```

SIMPLE AES CRACKING

For the second exercise we were tasked with recovering a partially lost 128-bit (16-byte) AES key (shown in [\[partially-known-key\]](#)). We were supplied with the first 13 bytes of the key and a ciphertext that is known to decrypt to only ASCII characters matching the regex `[a-z.]`. The partial key and the ciphertext are shown in [Listing 3](#).

Listing 3. Supplied data for the second exercise

```
1 // First 13 bytes of the key
2 constexpr unsigned char partial_key[13] = {
3     0x81, 0x59, 0x6b, 0xfb, 0x39, 0xc6, 0x2b,
4     0x71, 0x6e, 0x52, 0xdb, 0x91, 0x81,
5 };
6 // Ciphertext known to decrypt to only [a-z.] with the correct key
7 constexpr unsigned char ciphertext[16] = {
8     0xbf, 0x3f, 0xb7, 0x7d, 0x93, 0xdd, 0x6c, 0xfd,
9     0xef, 0xb8, 0x82, 0x2b, 0x82, 0xd0, 0x35, 0x8a,
10 };
```

CPP

Listing 4. Simple reference AES implementation

```

1 #include <algorithm>
2 #include <cassert>
3 #include <exception>
4 #include <fcntl.h>
5 #include <iostream>
6 #include <iterator>
7 #include <map>
8 #include <openssl/aes.h>
9 #include <openssl/evp.h>
10 #include <random>
11 #include <ranges>
12 #include <unistd.h>
13
14 const unsigned int NUM_ROUNDS = 4 + 6;
15
16 static const unsigned char SBOX[256] = {
17     0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
18     0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
19     0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26,
20     0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
21     0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
22     0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
23     0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed,
24     0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
25     0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
26     0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
27     0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec,
28     0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
29     0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
30     0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
31     0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d,
32     0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
33     0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
34     0x4b, 0xbd, 0x8b, 0x8a, 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
35     0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11,
36     0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
37     0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
38     0xb0, 0x54, 0xbb, 0x16};
39
40 static const unsigned char INV_SBOX[256] = {
41     0x52, 0x9, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e,
42     0x81, 0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
43     0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32,
44     0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0xb, 0x42, 0xfa, 0xc3, 0x4e,
45     0x8, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49,
46     0x6d, 0x8b, 0xd1, 0x25, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
47     0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50,

```

```

48 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
49 0x90, 0xd8, 0xab, 0x0, 0x8c, 0xbc, 0xd3, 0xa, 0xf7, 0xe4, 0x58, 0x5,
50 0xb8, 0xb3, 0x45, 0x6, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0xf, 0x2,
51 0xc1, 0xaf, 0xbd, 0x3, 0x1, 0x13, 0x8a, 0x6b, 0x3a, 0x91, 0x11, 0x41,
52 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
53 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
54 0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
55 0x6f, 0xb7, 0x62, 0xe, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b,
56 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
57 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x7, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59,
58 0x27, 0x80, 0xec, 0x5f, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0xd,
59 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d,
60 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
61 0x17, 0x2b, 0x4, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63,
62 0x55, 0x21, 0xc, 0x7d};
63
64 static const unsigned char RCON[255] = {
65 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
66 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
67 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
68 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
69 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
70 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6,
71 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
72 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
73 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
74 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e,
75 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
76 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
77 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
78 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
79 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
80 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
81 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb,
82 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
83 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
84 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
85 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
86 0x74, 0xe8, 0xcb};
87
88 void keyExpansion(const unsigned char *key, unsigned char *roundKey) {
89     unsigned char temp[4], k;
90
91     for (unsigned int i = 0; i < 4; ++i) {
92         roundKey[i * 4 + 0] = key[i * 4 + 0];
93         roundKey[i * 4 + 1] = key[i * 4 + 1];
94         roundKey[i * 4 + 2] = key[i * 4 + 2];
95         roundKey[i * 4 + 3] = key[i * 4 + 3];

```

```

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

```



```

144 state[1 * 4 + 0] = state[1 * 4 + 1];
145 state[1 * 4 + 1] = state[1 * 4 + 2];
146 state[1 * 4 + 2] = state[1 * 4 + 3];
147 state[1 * 4 + 3] = temp;
148
149 // Rotate second row 2 columns to left
150 temp = state[2 * 4 + 0];
151 state[2 * 4 + 0] = state[2 * 4 + 2];
152 state[2 * 4 + 2] = temp;
153
154 temp = state[2 * 4 + 1];
155 state[2 * 4 + 1] = state[2 * 4 + 3];
156 state[2 * 4 + 3] = temp;
157
158 // Rotate third row 3 columns to left
159 temp = state[3 * 4 + 0];
160 state[3 * 4 + 0] = state[3 * 4 + 3];
161 state[3 * 4 + 3] = state[3 * 4 + 2];
162 state[3 * 4 + 2] = state[3 * 4 + 1];
163 state[3 * 4 + 1] = temp;
164 }
165
166 void invShiftRows(unsigned char *state) {
167     unsigned char temp;
168
169     // Rotate first row 1 columns to right
170     temp = state[1 * 4 + 3];
171     state[1 * 4 + 3] = state[1 * 4 + 2];
172     state[1 * 4 + 2] = state[1 * 4 + 1];
173     state[1 * 4 + 1] = state[1 * 4 + 0];
174     state[1 * 4 + 0] = temp;
175
176     // Rotate second row 2 columns to right
177     temp = state[2 * 4 + 0];
178     state[2 * 4 + 0] = state[2 * 4 + 2];
179     state[2 * 4 + 2] = temp;
180
181     temp = state[2 * 4 + 1];
182     state[2 * 4 + 1] = state[2 * 4 + 3];
183     state[2 * 4 + 3] = temp;
184
185     // Rotate third row 3 columns to right
186     temp = state[3 * 4 + 0];
187     state[3 * 4 + 0] = state[3 * 4 + 1];
188     state[3 * 4 + 1] = state[3 * 4 + 2];
189     state[3 * 4 + 2] = state[3 * 4 + 3];
190     state[3 * 4 + 3] = temp;
191 }

```

```

192
193 // XTIME is a macro that finds the product of {02} and the argument to XTIME
194 // modulo {1b}
195 #define XTIME(x) (((x) << 1) ^ (((x) >> 7) & 1) * 0x1b))
196
197 // Multiplty is a macro used to multiply numbers in the field GF(2^8)
198 #define MULTIPLY(x, y) \
199     (((y)&1) * (x)) ^ (((y) >> 1 & 1) * XTIME(x)) ^ \
200     (((y) >> 2 & 1) * XTIME(XTIME(x))) ^ \
201     (((y) >> 3 & 1) * XTIME(XTIME(XTIME(x)))) ^ \
202     (((y) >> 4 & 1) * XTIME(XTIME(XTIME(XTIME(x))))))
203
204 void mixColumns(unsigned char *state) {
205     unsigned char Tmp, t;
206     for (unsigned int i = 0; i != 4; ++i) {
207         t = state[0 * 4 + i];
208         Tmp = state[0 * 4 + i] ^ state[1 * 4 + i] ^ state[2 * 4 + i] ^
209             state[3 * 4 + i];
210         state[0 * 4 + i] ^= XTIME(state[0 * 4 + i] ^ state[1 * 4 + i]) ^ Tmp;
211         state[1 * 4 + i] ^= XTIME(state[1 * 4 + i] ^ state[2 * 4 + i]) ^ Tmp;
212         state[2 * 4 + i] ^= XTIME(state[2 * 4 + i] ^ state[3 * 4 + i]) ^ Tmp;
213         state[3 * 4 + i] ^= XTIME(state[3 * 4 + i] ^ t) ^ Tmp;
214     }
215 }
216
217 void invMixColumns(unsigned char *state) {
218     unsigned char a, b, c, d;
219     for (unsigned int i = 0; i != 4; ++i) {
220         a = state[0 * 4 + i];
221         b = state[1 * 4 + i];
222         c = state[2 * 4 + i];
223         d = state[3 * 4 + i];
224
225         state[0 * 4 + i] = MULTIPLY(a, 0x0e) ^ MULTIPLY(b, 0x0b) ^
226             MULTIPLY(c, 0x0d) ^ MULTIPLY(d, 0x09);
227         state[1 * 4 + i] = MULTIPLY(a, 0x09) ^ MULTIPLY(b, 0x0e) ^
228             MULTIPLY(c, 0x0b) ^ MULTIPLY(d, 0x0d);
229         state[2 * 4 + i] = MULTIPLY(a, 0x0d) ^ MULTIPLY(b, 0x09) ^
230             MULTIPLY(c, 0x0e) ^ MULTIPLY(d, 0x0b);
231         state[3 * 4 + i] = MULTIPLY(a, 0x0b) ^ MULTIPLY(b, 0x0d) ^
232             MULTIPLY(c, 0x09) ^ MULTIPLY(d, 0x0e);
233     }
234 }
235
236 void cipher(const unsigned char *in, const unsigned char *roundKey,
237             unsigned char *out) {
238     unsigned char state[4 * 4];
239

```

```

240     for (unsigned int i = 0; i != 4; ++i) {
241         for (unsigned int j = 0; j != 4; ++j)
242             state[j * 4 + i] = in[i * 4 + j];
243     }
244
245     addRoundKey(state, roundKey, 0);
246     for (unsigned int round = 1; round < NUM_ROUNDS; ++round) {
247         subBytes(state);
248         shiftRows(state);
249         mixColumns(state);
250         addRoundKey(state, roundKey, round);
251     }
252     subBytes(state);
253     shiftRows(state);
254     addRoundKey(state, roundKey, NUM_ROUNDS);
255
256     for (unsigned int i = 0; i != 4; ++i) {
257         for (unsigned int j = 0; j != 4; ++j)
258             out[i * 4 + j] = state[j * 4 + i];
259     }
260 }
261
262 void decipher(const unsigned char *in, const unsigned char *roundKey,
263             unsigned char *out) {
264     unsigned char state[4 * 4];
265
266     for (unsigned int i = 0; i != 4; ++i) {
267         for (unsigned int j = 0; j != 4; ++j)
268             state[j * 4 + i] = in[i * 4 + j];
269     }
270
271     addRoundKey(state, roundKey, NUM_ROUNDS);
272     for (unsigned int round = NUM_ROUNDS - 1; round > 0; --round) {
273         invShiftRows(state);
274         invSubBytes(state);
275         addRoundKey(state, roundKey, round);
276         invMixColumns(state);
277     }
278     invShiftRows(state);
279     invSubBytes(state);
280     addRoundKey(state, roundKey, 0);
281
282     for (unsigned int i = 0; i != 4; ++i) {
283         for (unsigned int j = 0; j != 4; ++j)
284             out[i * 4 + j] = state[j * 4 + i];
285     }
286 }
287

```

```

288 std::array<unsigned char, 16>
289 simple_encrypt(std::array<unsigned char, 16> plaintext,
290               std::array<unsigned char, 16> key) {
291     unsigned char roundKey[240];
292     std::array<unsigned char, 16> ciphertext;
293     keyExpansion(key.data(), roundKey);
294     cipher(plaintext.data(), roundKey, ciphertext.data());
295     return ciphertext;
296 }
297
298 std::array<unsigned char, 16>
299 simple_decrypt(std::array<unsigned char, 16> ciphertext,
300               std::array<unsigned char, 16> key) {
301     unsigned char roundKey[240];
302     std::array<unsigned char, 16> plaintext;
303     keyExpansion(key.data(), roundKey);
304     decipher(ciphertext.data(), roundKey, plaintext.data());
305     return plaintext;
306 }
307
308 std::array<unsigned char, 16>
309 simple_openssl_encrypt(std::array<unsigned char, 16> plaintext,
310                       std::array<unsigned char, 16> key) {
311     AES_KEY aes_key;
312     AES_set_encrypt_key(key.data(), 128, &aes_key);
313     std::array<unsigned char, 16> ciphertext;
314     AES_encrypt(plaintext.data(), ciphertext.data(), &aes_key);
315     return ciphertext;
316 }
317
318 std::array<unsigned char, 16>
319 simple_openssl_decrypt(std::array<unsigned char, 16> ciphertext,
320                       std::array<unsigned char, 16> key) {
321     AES_KEY aes_key;
322     AES_set_decrypt_key(key.data(), 128, &aes_key);
323     std::array<unsigned char, 16> plaintext;
324     AES_decrypt(ciphertext.data(), plaintext.data(), &aes_key);
325     return plaintext;
326 }
327
328 // tag::main[]
329 int main() {
330     int devRandom = open("/dev/random", O_RDONLY);
331
332     for (int round = 0; round < 1000000; ++round) {
333         std::array<unsigned char, 16> plaintext, key;
334         read(devRandom, plaintext.data(), 16);
335         read(devRandom, key.data(), 16);

```

```

336
337 // Cipher with our implementation
338 auto ciphertext = simple_encrypt(plaintext, key);
339 auto decryptedPlaintext = simple_decrypt(ciphertext, key);
340 // Cipher with openssl
341 auto ciphertextOpenssl = simple_openssl_encrypt(plaintext, key);
342 auto decryptedPlaintextOpenssl =
343     simple_openssl_decrypt(ciphertextOpenssl, key);
344
345 if (!std::ranges::equal(ciphertext, ciphertextOpenssl)) {
346     printf("OpenSSL and simple encryption produced different ciphertexts");
347     exit(1);
348 }
349 if (!std::ranges::equal(plaintext, decryptedPlaintext)) {
350     printf("Decryption did not produce the original plaintext");
351     exit(1);
352 }
353 if (!std::ranges::equal(decryptedPlaintext, decryptedPlaintextOpenssl)) {
354     printf("OpenSSL and simple decryption produced different results");
355     exit(1);
356 }
357 }
358 std::printf("Encryption and decryption seems to work\n");
359 }
360 // end::main[]

```

Listing 5. openssl based AES tool

```

1 #include <iostream>
2 #include <openssl/aes.h>
3 #include <openssl/evp.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[]) {
7
8     const unsigned char in[16] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
9                                     'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'};
10    const unsigned char key[16] = {0xa3, 0x28, 0x4e, 0x09, 0xc6, 0xfe,
11                                    0x53, 0x29, 0x97, 0xef, 0x6d, 0x10,
12                                    0x74, 0xc3, 0xde, 0xad};
13
14    std::cout << std::endl << "Text before encryption:" << std::hex << std::endl;
15    for (unsigned int i = 0; i != 4 * 4; ++i)
16        std::cout << "0x" << (unsigned int)in[i] << ", ";
17    std::cout << std::endl;
18
19    unsigned char out[16] = {
20        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
21    };
22    AES_KEY aes_key;
23    AES_set_encrypt_key(key, 128, &aes_key);
24    AES_encrypt(in, out, &aes_key);
25

```

CPP

```
26  std::cout << std::endl << "Text after encryption:" << std::hex << std::endl;
27  for (unsigned int i = 0; i != 4 * 4; ++i)
28      std::cout << "0x" << (unsigned int)out[i] << ", ";
29  std::cout << std::endl;
30
31  return 0;
32 }
```