



**Darmstadt University of Applied Sciences**  
– Faculty of Computer Science –

**Cryptography Lab Report 2**

by  
**Lennart Eichhorn**  
Matriculation number: 759253

## SIMPLE AES PROGRAMMING

The supplied simple AES implementation shown [Listing 7](#) was supplied as a reference. However, the inverse S-Box required for decryption is missing. Our first task is to implement the inverse S-Box and then use it to implement the decryption function.

We wanted a known working encryption and decryption implementation to compare our implementation against. We build a simple implementation using the OpenSSL library, as shown in [Listing 8](#). We then verified that the OpenSSL implementation produces the same output as the simple implementation.

### 1.1 INVERTING THE S-BOX

[Listing 1](#) shows our function for calculating the inverted S-Box table. It swaps the indices and values of the S-Box table to create the inverted table.

*Listing 1. Function for inverting the S-Box*

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

CPP

### 1.2 VALIDATING THE S-BOX

The inverted S-Box ([Listing 6](#)) was placed into the AES implementation. We confirmed that the decryption function works correctly by testing 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 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);
```

CPP

```

8
9 // Cipher with our implementation
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. We were supplied with the first 13 bytes of the key and a ciphertext that is known to decrypt only to contain lowercase letters and the `.` character. 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 std::array<unsigned char, 13> partial_key = {
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 std::array<unsigned char, 16> ciphertext = {
8     0xbf, 0x3f, 0xb7, 0x7d, 0x93, 0xdd, 0x6c, 0xfd,
9     0xef, 0xb8, 0x82, 0x2b, 0x82, 0xd0, 0x35, 0x8a,
10 };

```

CPP

## 2.1 ESTIMATING PERFORMANCE

We only know the first 13 bytes of the key, so we need to brute force the remaining 3 bytes. Three bytes are 24 bits, so we must try  $2^{24}$  keys. If we can test  $2^{21}$  keys per second, we will need  $2^3$  seconds to test all keys. This is 8 seconds, a reasonable time frame for a brute-force attack.

## NOTE

The performance of  $2^{21}$  keys per second is a wild guess based on nothing.

## 2.2 IMPLEMENTING THE RECOVERY TOOL

We used the AES functions from the previous exercise to implement the recovery tool. [Listing 4](#) shows that we brute force the remaining 3 bytes of the key and check if the characters of the decrypted text match the requirements. If it does, we print the key and the decrypted text.

*Listing 4. Our key recovery tool*

```

1 int main() {
2     std::array<unsigned char, 16> key;
3     for (int i = 0; i < 13; i++) {
4         key[i] = partial_key[i];
5     }
6     for (unsigned char a = 0; a != 255; ++a) {
7         for (unsigned char b = 0; b != 255; ++b) {
8             for (unsigned char c = 0; c != 255; ++c) {
9                 key[13] = a;

```

CPP

```

10     key[14] = b;
11     key[15] = c;
12     auto decryptedPlaintext = simple_decrypt(ciphertext, key);
13     if (std::ranges::all_of(decryptedPlaintext, [](unsigned char c) {
14         return (c >= 'a' && c <= 'z') || c == '.';
15     })) {
16         std::cout << "Found key: ";
17         for (int i = 0; i < 16; i++) {
18             std::cout << std::hex << (unsigned int)key[i];
19         }
20         std::cout << std::endl;
21         std::cout << "Decrypted text is:" << std::endl;
22         for (int i = 0; i < 16; i++) {
23             std::cout << decryptedPlaintext[i];
24         }
25         std::cout << std::endl;
26         return 0;
27     }
28 }
29 }
30 }
31 std::cout << "Key not found" << std::endl;
32 return 1;
33 }

```

Running the tool takes 4.5 seconds on our machine, which aligns with the expected value. The output of the tool is shown in [Listing 5](#). We found the key to be 81596bfb39c62b716e52db9181dabeef, and the decrypted text is thiswasatriumph..

#### *Listing 5. Output of the key recovery tool*

```

1 lennart@erms ~/hda-cryptography-lab-2> time ./key-recovery
2 Found key: 81596bfb39c62b716e52db9181dabeef
3 Decrypted text is:
4 thiswasatriumph.
5 ./key-recovery  4.49s user 0.00s system 98% cpu 4.556 total

```

CONSOLE

## CONCLUSION

---

We successfully implemented the decryption function for the simple AES implementation and verified it against the OpenSSL implementation. We then used the AES functions to recover a partially lost key. We found the key to be `81596bfb39c62b716e52db9181dabeef` and the decrypted text to be `thiswasatriumph. .`

## LIST OF ABBREVIATIONS

---

### **AES**

Advanced Encryption Standard [!\[\]\(d84e7ea36f695d92cb39ec32c307ac93\_img.jpg\)](#)

### **S-Box**

Substitution box [!\[\]\(9dfdaff1d86ba3c1f8353b4d1b61b8c5\_img.jpg\)](#)

## APPENDIX

*Listing 6. Inverted S-Box*

```

1 static const unsigned char INV_SBOX[256] = {
2     0x52, 0x9, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e,
3     0x81, 0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
4     0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32,
5     0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0xb, 0x42, 0xfa, 0xc3, 0x4e,
6     0x8, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49,
7     0x6d, 0x8b, 0xd1, 0x25, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
8     0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50,
9     0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
10    0x90, 0xd8, 0xab, 0x0, 0x8c, 0xbc, 0xd3, 0xa, 0xf7, 0xe4, 0x58, 0x5,
11    0xb8, 0xb3, 0x45, 0x6, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0xf, 0x2,
12    0xc1, 0xaf, 0xbd, 0x3, 0x1, 0x13, 0x8a, 0x6b, 0x3a, 0x91, 0x11, 0x41,
13    0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
14    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
15    0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
16    0x6f, 0xb7, 0x62, 0xe, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b,
17    0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
18    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x7, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59,
19    0x27, 0x80, 0xec, 0x5f, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0xd,
20    0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d,
21    0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
22    0x17, 0x2b, 0x4, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63,
23    0x55, 0x21, 0xc, 0x7d};

```

CPP

*Listing 7. Simple reference AES implementation*

```

1 #include <iostream>
2
3 const unsigned int NUM_ROUNDS = 4 + 6;
4
5 static const unsigned char SBOX[256] = {
6     0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
7     0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
8     0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26,
9     0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
10    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
11    0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
12    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed,
13    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,
15    0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
16    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec,
17    0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
18    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
19    0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
20    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d,

```

CPP



```

21     0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
22     0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
23     0x4b, 0xbd, 0x8b, 0x8a, 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
24     0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11,
25     0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
26     0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
27     0xb0, 0x54, 0xbb, 0x16};
28
29 static const unsigned char INV_SBOX[256] = {0};
30
31 static const unsigned char RCON[255] = {
32     0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
33     0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
34     0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
35     0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
36     0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
37     0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6,
38     0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
39     0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
40     0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
41     0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e,
42     0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
43     0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
44     0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
45     0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
46     0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
47     0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
48     0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb,
49     0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
50     0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
51     0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
52     0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
53     0x74, 0xe8, 0xcb};
54
55 void keyExpansion(const unsigned char *key, unsigned char *roundKey) {
56     unsigned char temp[4], k;
57
58     for (unsigned int i = 0; i < 4; ++i) {
59         roundKey[i * 4 + 0] = key[i * 4 + 0];
60         roundKey[i * 4 + 1] = key[i * 4 + 1];
61         roundKey[i * 4 + 2] = key[i * 4 + 2];
62         roundKey[i * 4 + 3] = key[i * 4 + 3];
63     }
64
65     for (unsigned int i = 4; i < 4 * (NUM_ROUNDS + 1); ++i) {
66         for (unsigned int j = 0; j != 4; ++j)
67             temp[j] = roundKey[(i - 1) * 4 + j];
68         if (i % 4 == 0) {

```

```

69     k = SBOX[temp[0]];
70     temp[0] = SBOX[temp[1]];
71     temp[1] = SBOX[temp[2]];
72     temp[2] = SBOX[temp[3]];
73     temp[3] = k;
74
75     temp[0] = temp[0] ^ RCON[i / 4];
76 }
77 roundKey[i * 4 + 0] = roundKey[(i - 4) * 4 + 0] ^ temp[0];
78 roundKey[i * 4 + 1] = roundKey[(i - 4) * 4 + 1] ^ temp[1];
79 roundKey[i * 4 + 2] = roundKey[(i - 4) * 4 + 2] ^ temp[2];
80 roundKey[i * 4 + 3] = roundKey[(i - 4) * 4 + 3] ^ temp[3];
81 }
82 }
83
84 void addRoundKey(unsigned char *state, const unsigned char *roundKey,
85                 int round) {
86     for (unsigned int i = 0; i != 4; ++i) {
87         for (unsigned int j = 0; j != 4; ++j)
88             state[j * 4 + i] ^= roundKey[round * 4 * 4 + i * 4 + j];
89     }
90 }
91
92 void subBytes(unsigned char *state) {
93     for (unsigned int i = 0; i != 4; ++i) {
94         for (unsigned int j = 0; j != 4; ++j)
95             state[i * 4 + j] = SBOX[state[i * 4 + j]];
96     }
97 }
98
99 void invSubBytes(unsigned char *state) {
100     for (unsigned int i = 0; i != 4; ++i) {
101         for (unsigned int j = 0; j != 4; ++j)
102             state[i * 4 + j] = INV_SBOX[state[i * 4 + j]];
103     }
104 }
105
106 void shiftRows(unsigned char *state) {
107     unsigned char temp;
108
109     // Rotate first row 1 columns to left
110     temp = state[1 * 4 + 0];
111     state[1 * 4 + 0] = state[1 * 4 + 1];
112     state[1 * 4 + 1] = state[1 * 4 + 2];
113     state[1 * 4 + 2] = state[1 * 4 + 3];
114     state[1 * 4 + 3] = temp;
115
116     // Rotate second row 2 columns to left

```

```

117     temp = state[2 * 4 + 0];
118     state[2 * 4 + 0] = state[2 * 4 + 2];
119     state[2 * 4 + 2] = temp;
120
121     temp = state[2 * 4 + 1];
122     state[2 * 4 + 1] = state[2 * 4 + 3];
123     state[2 * 4 + 3] = temp;
124
125     // Rotate third row 3 columns to left
126     temp = state[3 * 4 + 0];
127     state[3 * 4 + 0] = state[3 * 4 + 3];
128     state[3 * 4 + 3] = state[3 * 4 + 2];
129     state[3 * 4 + 2] = state[3 * 4 + 1];
130     state[3 * 4 + 1] = temp;
131 }
132
133 void invShiftRows(unsigned char *state) {
134     unsigned char temp;
135
136     // Rotate first row 1 columns to right
137     temp = state[1 * 4 + 3];
138     state[1 * 4 + 3] = state[1 * 4 + 2];
139     state[1 * 4 + 2] = state[1 * 4 + 1];
140     state[1 * 4 + 1] = state[1 * 4 + 0];
141     state[1 * 4 + 0] = temp;
142
143     // Rotate second row 2 columns to right
144     temp = state[2 * 4 + 0];
145     state[2 * 4 + 0] = state[2 * 4 + 2];
146     state[2 * 4 + 2] = temp;
147
148     temp = state[2 * 4 + 1];
149     state[2 * 4 + 1] = state[2 * 4 + 3];
150     state[2 * 4 + 3] = temp;
151
152     // Rotate third row 3 columns to right
153     temp = state[3 * 4 + 0];
154     state[3 * 4 + 0] = state[3 * 4 + 1];
155     state[3 * 4 + 1] = state[3 * 4 + 2];
156     state[3 * 4 + 2] = state[3 * 4 + 3];
157     state[3 * 4 + 3] = temp;
158 }
159
160 // XTIME is a macro that finds the product of {02} and the argument to XTIME
161 // modulo {1b}
162 #define XTIME(x) (((x) << 1) ^ (((x) >> 7) & 1) * 0x1b))
163
164 // Multiplty is a macro used to multiply numbers in the field GF(2^8)

```

```

165 #define MULTIPLY(x, y) \
166     (((y) & 1) * (x)) ^ (((y) >> 1 & 1) * XTIME(x)) ^ \
167     (((y) >> 2 & 1) * XTIME(XTIME(x))) ^ \
168     (((y) >> 3 & 1) * XTIME(XTIME(XTIME(x)))) ^ \
169     (((y) >> 4 & 1) * XTIME(XTIME(XTIME(XTIME(x))))))
170
171 void mixColumns(unsigned char *state) {
172     unsigned char Tmp, t;
173     for (unsigned int i = 0; i != 4; ++i) {
174         t = state[0 * 4 + i];
175         Tmp = state[0 * 4 + i] ^ state[1 * 4 + i] ^ state[2 * 4 + i] ^
176             state[3 * 4 + i];
177         state[0 * 4 + i] ^= XTIME(state[0 * 4 + i] ^ state[1 * 4 + i]) ^ Tmp;
178         state[1 * 4 + i] ^= XTIME(state[1 * 4 + i] ^ state[2 * 4 + i]) ^ Tmp;
179         state[2 * 4 + i] ^= XTIME(state[2 * 4 + i] ^ state[3 * 4 + i]) ^ Tmp;
180         state[3 * 4 + i] ^= XTIME(state[3 * 4 + i] ^ t) ^ Tmp;
181     }
182 }
183
184 void invMixColumns(unsigned char *state) {
185     unsigned char a, b, c, d;
186     for (unsigned int i = 0; i != 4; ++i) {
187         a = state[0 * 4 + i];
188         b = state[1 * 4 + i];
189         c = state[2 * 4 + i];
190         d = state[3 * 4 + i];
191
192         state[0 * 4 + i] = MULTIPLY(a, 0x0e) ^ MULTIPLY(b, 0x0b) ^
193             MULTIPLY(c, 0x0d) ^ MULTIPLY(d, 0x09);
194         state[1 * 4 + i] = MULTIPLY(a, 0x09) ^ MULTIPLY(b, 0x0e) ^
195             MULTIPLY(c, 0x0b) ^ MULTIPLY(d, 0x0d);
196         state[2 * 4 + i] = MULTIPLY(a, 0x0d) ^ MULTIPLY(b, 0x09) ^
197             MULTIPLY(c, 0x0e) ^ MULTIPLY(d, 0x0b);
198         state[3 * 4 + i] = MULTIPLY(a, 0x0b) ^ MULTIPLY(b, 0x0d) ^
199             MULTIPLY(c, 0x09) ^ MULTIPLY(d, 0x0e);
200     }
201 }
202
203 void cipher(const unsigned char *in, const unsigned char *roundKey,
204             unsigned char *out) {
205     unsigned char state[4 * 4];
206
207     for (unsigned int i = 0; i != 4; ++i) {
208         for (unsigned int j = 0; j != 4; ++j)
209             state[j * 4 + i] = in[i * 4 + j];
210     }
211
212     addRoundKey(state, roundKey, 0);

```

```

213 for (unsigned int round = 1; round < NUM_ROUNDS; ++round) {
214     subBytes(state);
215     shiftRows(state);
216     mixColumns(state);
217     addRoundKey(state, roundKey, round);
218 }
219 subBytes(state);
220 shiftRows(state);
221 addRoundKey(state, roundKey, NUM_ROUNDS);
222
223 for (unsigned int i = 0; i != 4; ++i) {
224     for (unsigned int j = 0; j != 4; ++j)
225         out[i * 4 + j] = state[j * 4 + i];
226 }
227 }
228
229 void decipher(const unsigned char *in, const unsigned char *roundKey,
230              unsigned char *out) {
231     unsigned char state[4 * 4];
232
233     for (unsigned int i = 0; i != 4; ++i) {
234         for (unsigned int j = 0; j != 4; ++j)
235             state[j * 4 + i] = in[i * 4 + j];
236     }
237
238     addRoundKey(state, roundKey, NUM_ROUNDS);
239     for (unsigned int round = NUM_ROUNDS - 1; round > 0; --round) {
240         invShiftRows(state);
241         invSubBytes(state);
242         addRoundKey(state, roundKey, round);
243         invMixColumns(state);
244     }
245     invShiftRows(state);
246     invSubBytes(state);
247     addRoundKey(state, roundKey, 0);
248
249     for (unsigned int i = 0; i != 4; ++i) {
250         for (unsigned int j = 0; j != 4; ++j)
251             out[i * 4 + j] = state[j * 4 + i];
252     }
253 }
254
255 int main(int argc, char *argv[]) {
256     unsigned char roundKey[240];
257     unsigned char out[16];
258
259     // Sample
260     {

```

```

261  const unsigned char in[16] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
262                                'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'};
263  const unsigned char key[16] = {0xa3, 0x28, 0x4e, 0x09, 0xc6, 0xfe,
264                                0x53, 0x29, 0x97, 0xef, 0x6d, 0x10,
265                                0x74, 0xc3, 0xde, 0xad};
266
267  std::cout << std::endl
268            << "Text before encryption:" << std::hex << std::endl;
269  for (unsigned int i = 0; i != 4 * 4; ++i)
270      std::cout << "0x" << (unsigned int)in[i] << ", ";
271  std::cout << std::endl;
272
273  keyExpansion(key, roundKey);
274  cipher(in, roundKey, out);
275
276  std::cout << std::endl << "Text after encryption:" << std::hex << std::endl;
277  for (unsigned int i = 0; i != 4 * 4; ++i)
278      std::cout << "0x" << (unsigned int)out[i] << ", ";
279  std::cout << std::endl;
280 }
281
282 return 0;
283 }

```

*Listing 8. 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
26     std::cout << std::endl << "Text after encryption:" << std::hex << std::endl;

```

CPP

```
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 }
```