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

and

Oliver Hanikel

Matriculation number: 765349

1 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

```cpp
int main() {
  unsigned char invertedSbox[256];
  for (unsigned int i = 0; i != 256; ++i) {
    invertedSbox[SBOX[i]] = i;
  }

  std::printf("static const unsigned char INV_SBOX[256] = {");
  for (auto element : invertedSbox) {
    std::printf("0x%x,", element);
  }
  std::printf("};\n");
}
```

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

```cpp
int main() {
  int devRandom = open("/dev/random", O_RDONLY);

  for (int round = 0; round < 1000000; ++round) {
    std::array<unsigned char, 16> plaintext, key;
    read(devRandom, plaintext.data(), 16);
    read(devRandom, key.data(), 16);
```

```

    // Cipher with our implementation
    auto ciphertext = simple_encrypt(plaintext, key);
    auto decryptedPlaintext = simple_decrypt(ciphertext, key);
    // Cipher with openssl
    auto ciphertextOpenssl = simple_openssl_encrypt(plaintext, key);
    auto decryptedPlaintextOpenssl =
        simple_openssl_decrypt(ciphertextOpenssl, key);

    if (!std::ranges::equal(ciphertext, ciphertextOpenssl)) {
      printf("OpenSSL and simple encryption produced different ciphertexts");
      exit(1);
    }
    if (!std::ranges::equal(plaintext, decryptedPlaintext)) {
      printf("Decryption did not produce the original plaintext");
      exit(1);
    }
    if (!std::ranges::equal(decryptedPlaintext, decryptedPlaintextOpenssl)) {
      printf("OpenSSL and simple decryption produced different results");
      exit(1);
    }
  }
  std::printf("Encryption and decryption seems to work\n");
}
```

2 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

CPP
```cpp
// First 13 bytes of the key
constexpr std::array<unsigned char, 13> partial_key = {
    0x81, 0x59, 0x6b, 0xfb, 0x39, 0xc6, 0x2b,
    0x71, 0x6e, 0x52, 0xdb, 0x91, 0x81,
};
// Ciphertext known to decrypt to only [a-z.] with the correct key
constexpr std::array<unsigned char, 16> ciphertext = {
    0xbf, 0x3f, 0xb7, 0x7d, 0x93, 0xdd, 0x6c, 0xfd,
    0xef, 0xb8, 0x82, 0x2b, 0x82, 0xd0, 0x35, 0x8a,
};
```

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

CPP
```cpp
int main() {
  std::array<unsigned char, 16> key;
  for (int i = 0; i < 13; i++) {
    key[i] = partial_key[i];
  }
  for (unsigned char a = 0; a != 255; ++a) {
    for (unsigned char b = 0; b != 255; ++b) {
      for (unsigned char c = 0; c != 255; ++c) {
        key[13] = a;
```

```
        key[14] = b;
        key[15] = c;
        auto decryptedPlaintext = simple_decrypt(ciphertext, key);
        if (std::ranges::all_of(decryptedPlaintext, [](unsigned char c) {
              return (c >= 'a' && c <= 'z') || c == '.';
            })) {
          std::cout << "Found key: ";
          for (int i = 0; i < 16; i++) {
            std::cout << std::hex << (unsigned int)key[i];
          }
          std::cout << std::endl;
          std::cout << "Decrypted text is:" << std::endl;
          for (int i = 0; i < 16; i++) {
            std::cout << decryptedPlaintext[i];
          }
          std::cout << std::endl;
          return 0;
        }
      }
    }
  }
  std::cout << "Key not found" << std::endl;
  return 1;
}
```

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

```
CONSOLE
lennart@erms ~/hda-cryptography-lab-2> time ./key-recovery
Found key: 81596bfb39c62b716e52db9181dabeef
Decrypted text is:
thiswasatriumph.
./key-recovery  4,49s user 0,00s system 98% cpu 4,556 total
```

3 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 🔗

S-Box
: Substitution box 🔗

4 APPENDIX

Listing 6. Inverted S-Box

CPP

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

Listing 7. Simple reference AES implementation

CPP

```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, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed,
    0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    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, 0xcd, 0x0c, 0x13, 0xec,
    0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
    0xde, 0x5e, 0x0b, 0xdb, 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, 0xe1, 0xf8, 0x98, 0x11,
    0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    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, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
    0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
    0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
    0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6,
    0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
    0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
    0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e,
    0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
    0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
    0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
    0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
    0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
    0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
    0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb,
    0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
    0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
    0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    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]];
```

```c
      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];
  }
}

void subBytes(unsigned char *state) {
  for (unsigned int i = 0; i != 4; ++i) {
    for (unsigned int j = 0; j != 4; ++j)
      state[i * 4 + j] = SBOX[state[i * 4 + j]];
  }
}

void invSubBytes(unsigned char *state) {
  for (unsigned int i = 0; i != 4; ++i) {
    for (unsigned int j = 0; j != 4; ++j)
      state[i * 4 + j] = INV_SBOX[state[i * 4 + j]];
  }
}

void shiftRows(unsigned char *state) {
  unsigned char temp;

  // Rotate first row 1 columns to left
  temp = state[1 * 4 + 0];
  state[1 * 4 + 0] = state[1 * 4 + 1];
  state[1 * 4 + 1] = state[1 * 4 + 2];
  state[1 * 4 + 2] = state[1 * 4 + 3];
  state[1 * 4 + 3] = temp;

  // Rotate second row 2 columns to left
  temp = state[2 * 4 + 0];
  state[2 * 4 + 0] = state[2 * 4 + 2];
```

```
  state[2 * 4 + 2] = temp;

  temp = state[2 * 4 + 1];
  state[2 * 4 + 1] = state[2 * 4 + 3];
  state[2 * 4 + 3] = temp;

  // Rotate third row 3 columns to left
  temp = state[3 * 4 + 0];
  state[3 * 4 + 0] = state[3 * 4 + 3];
  state[3 * 4 + 3] = state[3 * 4 + 2];
  state[3 * 4 + 2] = state[3 * 4 + 1];
  state[3 * 4 + 1] = temp;
}

void invShiftRows(unsigned char *state) {
  unsigned char temp;

  // Rotate first row 1 columns to right
  temp = state[1 * 4 + 3];
  state[1 * 4 + 3] = state[1 * 4 + 2];
  state[1 * 4 + 2] = state[1 * 4 + 1];
  state[1 * 4 + 1] = state[1 * 4 + 0];
  state[1 * 4 + 0] = temp;

  // Rotate second row 2 columns to right
  temp = state[2 * 4 + 0];
  state[2 * 4 + 0] = state[2 * 4 + 2];
  state[2 * 4 + 2] = temp;

  temp = state[2 * 4 + 1];
  state[2 * 4 + 1] = state[2 * 4 + 3];
  state[2 * 4 + 3] = temp;

  // Rotate third row 3 columns to right
  temp = state[3 * 4 + 0];
  state[3 * 4 + 0] = state[3 * 4 + 1];
  state[3 * 4 + 1] = state[3 * 4 + 2];
  state[3 * 4 + 2] = state[3 * 4 + 3];
  state[3 * 4 + 3] = temp;
}

// XTIME is a macro that finds the product of {02} and the argument to XTIME
// modulo {1b}
#define XTIME(x) (((x) << 1) ^ ((((x) >> 7) & 1) * 0x1b))

// Multiplty is a macro used to multiply numbers in the field GF(2^8)
#define MULTIPLY(x, y)                                                   \
  ((((y) & 1) * (x)) ^ (((y) >> 1 & 1) * XTIME(x)) ^                     \
   (((y) >> 2 & 1) * XTIME(XTIME(x))) ^                                  \
```

```
  (((y) >> 3 & 1) * XTIME(XTIME(XTIME(x)))) ^                                 \
  (((y) >> 4 & 1) * XTIME(XTIME(XTIME(XTIME(x))))))

void mixColumns(unsigned char *state) {
  unsigned char Tmp, t;
  for (unsigned int i = 0; i != 4; ++i) {
    t = state[0 * 4 + i];
    Tmp = state[0 * 4 + i] ^ state[1 * 4 + i] ^ state[2 * 4 + i] ^
          state[3 * 4 + i];
    state[0 * 4 + i] ^= XTIME(state[0 * 4 + i] ^ state[1 * 4 + i]) ^ Tmp;
    state[1 * 4 + i] ^= XTIME(state[1 * 4 + i] ^ state[2 * 4 + i]) ^ Tmp;
    state[2 * 4 + i] ^= XTIME(state[2 * 4 + i] ^ state[3 * 4 + i]) ^ Tmp;
    state[3 * 4 + i] ^= XTIME(state[3 * 4 + i] ^ t) ^ Tmp;
  }
}

void invMixColumns(unsigned char *state) {
  unsigned char a, b, c, d;
  for (unsigned int i = 0; i != 4; ++i) {
    a = state[0 * 4 + i];
    b = state[1 * 4 + i];
    c = state[2 * 4 + i];
    d = state[3 * 4 + i];

    state[0 * 4 + i] = MULTIPLY(a, 0x0e) ^ MULTIPLY(b, 0x0b) ^
                       MULTIPLY(c, 0x0d) ^ MULTIPLY(d, 0x09);
    state[1 * 4 + i] = MULTIPLY(a, 0x09) ^ MULTIPLY(b, 0x0e) ^
                       MULTIPLY(c, 0x0b) ^ MULTIPLY(d, 0x0d);
    state[2 * 4 + i] = MULTIPLY(a, 0x0d) ^ MULTIPLY(b, 0x09) ^
                       MULTIPLY(c, 0x0e) ^ MULTIPLY(d, 0x0b);
    state[3 * 4 + i] = MULTIPLY(a, 0x0b) ^ MULTIPLY(b, 0x0d) ^
                       MULTIPLY(c, 0x09) ^ MULTIPLY(d, 0x0e);
  }
}

void cipher(const unsigned char *in, const unsigned char *roundKey,
            unsigned char *out) {
  unsigned char state[4 * 4];

  for (unsigned int i = 0; i != 4; ++i) {
    for (unsigned int j = 0; j != 4; ++j)
      state[j * 4 + i] = in[i * 4 + j];
  }

  addRoundKey(state, roundKey, 0);
  for (unsigned int round = 1; round < NUM_ROUNDS; ++round) {
    subBytes(state);
    shiftRows(state);
    mixColumns(state);
```

```
    addRoundKey(state, roundKey, round);
  }
  subBytes(state);
  shiftRows(state);
  addRoundKey(state, roundKey, NUM_ROUNDS);

  for (unsigned int i = 0; i != 4; ++i) {
    for (unsigned int j = 0; j != 4; ++j)
      out[i * 4 + j] = state[j * 4 + i];
  }
}

void decipher(const unsigned char *in, const unsigned char *roundKey,
              unsigned char *out) {
  unsigned char state[4 * 4];

  for (unsigned int i = 0; i != 4; ++i) {
    for (unsigned int j = 0; j != 4; ++j)
      state[j * 4 + i] = in[i * 4 + j];
  }

  addRoundKey(state, roundKey, NUM_ROUNDS);
  for (unsigned int round = NUM_ROUNDS - 1; round > 0; --round) {
    invShiftRows(state);
    invSubBytes(state);
    addRoundKey(state, roundKey, round);
    invMixColumns(state);
  }
  invShiftRows(state);
  invSubBytes(state);
  addRoundKey(state, roundKey, 0);

  for (unsigned int i = 0; i != 4; ++i) {
    for (unsigned int j = 0; j != 4; ++j)
      out[i * 4 + j] = state[j * 4 + i];
  }
}

int main(int argc, char *argv[]) {
  unsigned char roundKey[240];
  unsigned char out[16];

  // Sample
  {
    const unsigned char in[16] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
                                  'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'};
    const unsigned char key[16] = {0xa3, 0x28, 0x4e, 0x09, 0xc6, 0xfe,
                                   0x53, 0x29, 0x97, 0xef, 0x6d, 0x10,
                                   0x74, 0xc3, 0xde, 0xad};
```

```

    std::cout << std::endl
              << "Text before encryption:" << std::hex << std::endl;
    for (unsigned int i = 0; i != 4 * 4; ++i)
      std::cout << "0x" << (unsigned int)in[i] << ", ";
    std::cout << std::endl;

    keyExpansion(key, roundKey);
    cipher(in, roundKey, out);

    std::cout << std::endl << "Text after encryption:" << std::hex << std::endl;
    for (unsigned int i = 0; i != 4 * 4; ++i)
      std::cout << "0x" << (unsigned int)out[i] << ", ";
    std::cout << std::endl;
  }

  return 0;
}
```

Listing 8. `openssl` *based AES tool*

CPP

```cpp
#include <iostream>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

  const unsigned char in[16] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
                                'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'};
  const unsigned char key[16] = {0xa3, 0x28, 0x4e, 0x09, 0xc6, 0xfe,
                                 0x53, 0x29, 0x97, 0xef, 0x6d, 0x10,
                                 0x74, 0xc3, 0xde, 0xad};

  std::cout << std::endl << "Text before encryption:" << std::hex << std::endl;
  for (unsigned int i = 0; i != 4 * 4; ++i)
    std::cout << "0x" << (unsigned int)in[i] << ", ";
  std::cout << std::endl;

  unsigned char out[16] = {
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  };
  AES_KEY aes_key;
  AES_set_encrypt_key(key, 128, &aes_key);
  AES_encrypt(in, out, &aes_key);

  std::cout << std::endl << "Text after encryption:" << std::hex << std::endl;
  for (unsigned int i = 0; i != 4 * 4; ++i)
    std::cout << "0x" << (unsigned int)out[i] << ", ";
  std::cout << std::endl;

  return 0;
}
```