CSE539

Applied Cryptography , AES Implementation in C

Introduction

AES stands for Advance Encryption Standard. The report is a documentation of how as a part of the project, we tried to understand the standard and implemented it. We have tried to implement AES as securely as possible so that it can be used in real world applications. The report includes the following:
1. Our understanding of the standard document.
2. A description of our implementation.
3. How the project contributed to our understanding of cryptography.
4. Secure coding practices and how we tried to adapt to those in our implementation.
5. A documentation of the code.

We have referred the standard document FIPS PUB 197. The standard specifies the Rijndael algorithm for a fixed block size (128 bits) and three different key lengths(128,192, and 256 bits). The algorithm is designed in a way as to be able to handle additional block sizes and key lengths. Before we could move to the implementation, we had to thoroughly study the document and understand the definitions, terms, symbols, parameters and various functions and how the algorithm utilizes the functions to generate the cipher and decrypt the cipher.

We have used C language and a standard library (stdint) for implementing the project. We have implemented various functions like AddRoundKey() and MixColumns() in different files and combined them in a AES.c file. Our implementation uses the basic functions and data structure and no other external libraries. AES can be executed with the help of keys of length 128, 192 and 256 bits and our implementation considers all the 3 lengths.

The mathematics that is specified in the standard algorithm includes finite field, affine transformation, etc. We have also taken into consideration some of the attacks that are prevailing like the cache attack and the timing attack in our implementation. More importantly apart from learning about the AES algorithm and how it works, we got an understanding of how implementing an existing algorithm in a way that it is not only a correct implementation but also a secure implementation, can be achieved.

Implementation description

We have decided to implement AES in C due to its simplicity. C language is the building block for many other languages. It has variety of data types and operators which become very handy. For inistance the stat type **uint8_t** is of size 8 bits. There is no need of checking the size of a byte. Our main goal was to minimize the use of dynamic memory. Our implementation is such that we have not used any memory allocation or deallocation functions. That will make the code more static and stable. The C language has a direct support for the bitwise operators. C language is a very portable language which means that it can run on any system without any dependencies.
The implementation consists of multiple modules and they are divided as follows:
• addRoundkey
• mixColumns
• subBytes
• shiftRows
• AES

• Common

Each module has 2 files the header file and the .c file. We have defined all the functions and structures that are used in the .c file in the header file. The AES is the main module that is for running the encryption, decryption and the generator function.

AddroundKey: The round key is added to the state using an XOR operation. This module contains only one function

MixColumns: Module for mixColumns and invMixColumns functions. These are transformation functions in the cipher and in the inverse cipher that takes columns of the state and mixes them.

SubBytes: Module for subBytes and invSubBytes functions. These are also transformation functions in the cipher and inverse cipher that performs linear byte substitution on the state using the sBox and inverse sBox.

ShiftRows: module for shiftRows and invShiftRows functions. These functions shift the last three rows cyclically.

Common: Module for other function and data structure that are being used in other modules. Functions like finite multiplication, coefficient multiplication, rotWord, subWord, and more.

AES: This is the main module from where the main function is called. It contains the encryption, decryption, key-expansion and generator function. This can be used in any mode of operation.

The implementation is done such that all the data structures are defined once and there are no redundancies in the code for variables. Also, we have made sure that there is no function or variable or macro that are not used in the implementation.

Crypto learning

While implementing AES, we understood the problems and difficulties that are faced when writing a crypto function. There are many attacks that are prevalent like the timing attack and the cache attacks. Cache attack: variable time instruction which can leak information about the secret key. Table lookups take different time depending on whether they are in the cache or the main memory. In AES the lookup table directly depend on the secret key.

Several types of attack can be done using this loophole or a pitfall. Example of one such attack is the passive timing attack in Minronov,2006 Countermeasures for such attacks can be taken. Some such measures are:
• Adding variable time dummy functions.
• You can preload some values into the cache and preload all tables in the cache
• Force all the operations or instruction to take constant time.
• Instead of using s-box we can evaluate s-box on the fly.

Since AES side channel attacks target either timing or power measure we tried to make sure all operations completed in the same amount of time, and used volatile variables when possible to try and mitigate optimization effects on time.

Secure Coding

Secure coding is a practice of developing computer software to prevent from any future vulnarabilities or insecurity. Bugs, flaws and defects are the main cause of software vulnarabilities. These defects, flaws are occured due to some common software programming errors. We are going to enumerate where we have used secure coding complying the SERT document for secure coding.

Before enumerating the code, there are many standards which we have followed through out our implementation. These rules or recommendations are:
- DCL31-C - Declare identifiers before using them
- DCL39-C - Avoid information leakage when passing a structure across trust boundaries
- DCL40-C - Do not create incompatible declarations of the same object or function
- ARR30 C - Do not form or use out of bounds pointers or array subscript
- MSC30 C - Do not use rand() function for generating pseudo random generators
- MSC37 C - Ensure that control never reaches the end of a non-void function
- DCL07-C - Include the appropriate type information in function declarators
- INT07-C - Use only explicitly signed or unsigned char type for numeric values

Below are some of the secure coding rules and recommendation that we have used in our code.

AES.c

```
#define getrandom(buf, size, flags) syscall(SYS_getrandom, buf, size, flags)
```

MSC30 C - Do not use rand() function for generating pseudo random generators

---

common.c

```
uint8_t c2 = a & 0xF; //second 4 bits of a
    uint8_t c1 = a >> 4;  //first 4 of a
    static const uint8_t s[16][16] = ①
    {
        {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},
    .
    .
    }
```

① EXP 40-C - Donot modify const objects

---

common.c

```
uint8_t xtime(uint8_t value)
{
     uint8_t temp = value << 1; ①
     if(GETBIT(value, 7) == 1){
          temp ^= 0x1b;
     }
     return temp;
}

uint8_t finite_mul(uint8_t a, uint8_t b) {
     volatile uint8_t gf[OCTET];
     volatile int i;
     volatile uint8_t result = 0;

     gf[0] = a;
     for(i=1;i<OCTET;i++){
          gf[i] = xtime(gf[i-1]); ②
     }

     for (i=0;i<OCTET;i++){
          if(GETBIT(b,i) == 1) {
               result ^= gf[i];
          }
     }

     return result;
}
```

① INT34 C - Do not shift an expression by a negative number of bits that exists in the operand

② EXP33 C - Donot read uninitialized memory. Here the function Xtime is called only after assigning any value to it.

AES.c

```
void arrayXor (uint8_t *a, uint8_t *b,uint8_t *out,int l){  ①
    int i;
    for (i=0;i<l;i++){
        out[i]= a[i] ^ b[i];
    }
}
```

① ARR32 C - Ensure size arguments for variable length arrays are in valid range

---

AES.c

```
int Cipher(block_t *in, block_t *out, key_t *key) {
     state_t state;
     key_size_t key_size = key->key_size;
     block_t key_schedule;
     uint8_t *w;
     int Nb;
     int Nk;
     int Nr;
     int r;
    int c;
.
.
.
```

DCL04-C - Do not declare more than one variable per declaration DCL19-C - Minimize the scope of variables and functions

---

common.h

```
typedef struct {
     uint8_t array[BLOCK_SIZE];
     int size;
} block_t;

typedef struct {
     uint8_t array[STATE_ROWS][NB_SIZE];
     int rows;
     int columns;
} state_t;
.
.
.
```

DCL05-C - Use typedefs of non-pointer types only

AES.c

```
state.array[r][c] = in->array[(r*STATE_ROWS)+c];
```

```
copySubArray(w, key_schedule.array,(r*Nb), (r+1)*Nb-1);
```

```
out->array[(r*STATE_ROWS)+c] = state.array[r][c];
```

EXP00-C - Use parentheses for precedence of operation

ShiftRows.c

```
void shiftRows(state_t *state) {
     int r;
    int c;
     volatile uint8_t row[STATE_ROWS];

     for(r=1;r<STATE_ROWS;r++) {
          for(c=0;c<NB_SIZE;c++) {
               row[c] = state->array[r][(c+SHIFT(r,NB_SIZE) % NB_SIZE)];   ①
          }
          for(c=0;c<NB_SIZE;c++) {
               state->array[r][c] = row[c];                              ①
          }
     }
}
```

EXP19-C - Use braces for the body of an if, for, or while statement

---

AES.c

```
void copySubArray(uint8_t *in, uint8_t *out, int from, int to){
.
```

ARR02-C - Explicitly specify array bounds, even if implicitly defined by an initializer