Introduction to the new AES Standard: Rijndael

Paul Donis

This paper will explain how the Rijndael Cipher Reference Code in C works.

Rijndael is a block cipher that encrypts and decrypts 128, 192, and 256 bit blocks, using 128, 192, and 256 byte keys in any combination. The block is considered to be structured as 4, 6, or 8 columns of 4 bytes, depending on block size.

The basic operations applied to the block are:

1) KeyAddittion: XORing each byte with a round key (done before the first round for “whitening,” and again at the end of each round),

2) Substitution: Applying an S-box (substituting each byte with another, based on an equation in GF(2^8)),

3) ShiftRow: Shifting the rows in a circular way, the amount of shift (0, 1, 2, 3, or 4 bytes) depending on the position from the top and on the block size, and

4) MixColumn: Mixing the 4, 6, or 8 columns vertically by taking invertible linear combinations (in GF(2^8) of the elements in each column.

In terms of the above operations, the top level structure of Rijndael is:

KeyAddition

Substitution
ShiftRow
MixColumn
KeyAddition

Substitution
ShiftRow
MixColumn
KeyAddition

.
.
.

Substitution
ShiftRow
MixColumn
KeyAddition

Substitution
ShiftRow
KeyAddition

The number of rounds depends on the block and key sizes. For Block and Key sizes both of 128 bits, there are 10 rounds, nine regular rounds, and one short round at the end without MixColumn.

If either the Block or Key size is 192, but not 256, bits, there are 12 rounds. If either the Block or Key is 256 bits, there are 14 rounds.

The reason there is no MixColumn in the last round is to make the structure, when starting at the bottom and working up, similar the structure going down. This makes it possible the use the same code or circuitry, with relatively minor changes, for both encryption and decryption.

To understand the operations used above, we need some mathematical background, which assumes only that the reader knows the basic concept of integers (mod n).

The F in GF(2^8) stands for Field. Examples of fields are complex numbers and integers mod p (p prime). A field has a set of objects which can be combined by either of two operations (addition and multiplication) to produce other objects in the set. There is an additive identity (when added to a number changes nothing, i.e. ‘zero’), there is a multiplicative identity i.e. ‘one’. There is an additive inverse (minus the number), and there is a multiplicative inverse (x^-1) for all numbers except the additive identity i.e. except for ‘zero’. There is a distributive law: a*(c + b) = a*c + a*b. We know that the numbers mod 26 = 2 * 13 do not have a multiplicative inverse for two (two times anything is even and will never give a remainder of one when divided the even 26). So the requirement that p be prime is essential.

The G stands of Evariste Galois, who, before dying in a duel at age 20, proved that all fields, which are finite, have a limited set of possible structures. Whatever the details of how we represent them, for a set of numbers p (prime) and n, all GF(p^n) work the same (are isomorphic).

For us, p = 2, and n = 8. That means that the elements of our GF are represented as polynomials of degree less than eight in numbers (mod 2). Since numbers (mod 2) are either 0 or 1, there is a direct correlation between the sequence of polynomial coefficients and the bits in a byte. Here is an example:

x^6 + x^4 + x^2 + x + 1 = ‘57’ in Hex
=01010111 in Binary

Just as with integers, we can have modulo arithmetic in these polynomials. In integer modulo arithmetic, the numbers 123, 13, and 20003 are all = 3 (mod 10). To

discover that all the “large” numbers were equal to 3 (mod 10), we divided by 10 (the modulus) and kept the remainder. In integer modulo arithmetic, the modulus has to be a prime for us to have multiplicative inverses of all non-zero numbers.

The same is true for polynomials; we need the equivalent of a prime for polynomials. The creators of Rijndael have selected one for us:

m(x) = x^8 + x^4 + x^3 + x + 1 = ‘11B’ Hex =100011011 in Binary

This polynomial m(x) cannot be factored (is irreducible), and makes multiplicative inverses possible, because it acts like a prime did in (mod p) integer arithmetic.

HERE IS THE BIGGIE:

We define a new arithmetic on bytes by seeing them as coefficients of polynomials mod the above polynomial m(x). Just as in (mod p) arithmetic with integers, where we divide to get a remainder smaller than the modulus, if we have a polynomial of degree more than seven, we divide by m(x) to get a polynomial of degree less than eight. Since m(x)is of degree eight, the remainder will be of degree seven or less.

So how does addition work?

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 2 = 0 (mod 2)

1 – 1 = 0 = 1 + 1 (mod 2)

It turns out that addition and subtraction are the same, and equivalent to bit-wise XOR. Since, in polynomial

addition, coefficients of corresponding power are added, adding two bytes is the same as byte-wise XOR.

To do products, we need to do polynomial products, using the normal rules of polynomial algebra, and reduce the result mod m(x). For example, 5 * 3 is:

(x^2 + 1)*(x^1 + 1) = [00000101]*[00000011]
= x^3 + x^2 + x^1 + x^0 = [00001111]

Nine times two is:

(x^3 + 1)*(x^1) = [00001001]*[00000010]
= x^4 + x^1 = [00010010]

So a product of a “small” number by two is a left shift of one. How convenient! “Small” means that the shift does not overflow to x^8; if it overflows, m(x) must be subtracted.

To see what that mod m(x) is all about, let us look at the powers of two.

These are powers of two mod 100011011 in GF(2^8).

00000001 00000010 00000100 00001000
00010000 00100000 01000000 10000000
00011011 00110110 01101100 11011000
01111011 11110110 11110111 11110101
11110001 11111001 11101001 11001001...

Notice what happens in the third row. Pretend there is a one bit to the left, and compare with m(x). The start of the fourth row is the last element of the third row shifted left and added (XORed) to m(x).

In practice, if space permits the use of tables, we can use tables of logs to do products. That is done in this

implementation of Rijndael. Because m(x) acts like a prime, we have an element (generator) for which the powers from 0 to 255 run through all possible values of byes. Thus, logs are possible. Let us look at the tables of logs and anti-logs (exponentials) used and see if we can figure out the chosen base.

```
word8 Logtable[256] = {
  0,   0,  25,   1,  50,   2,  26, 198,
 75, 199,  27, 104,  51, 238, 223,   3,
100,   4, 224,  14,  52, 141, 129, 239,
 76, 113,   8, 200, 248, 105,  28, 193,
125, 194,  29, 181, 249, 185,  39, 106,
```

```
word8 Alogtable[256] = {
  1,   3,   5,  15,  17,  51,  85, 255,
 26,  46, 114, 150, 161, 248,  19,  53,
 95, 225,  56,  72, 216, 115, 149, 164,
247,   2,   6,  10,  30,  34, 102, 170,
229,  52,  92, 228,  55,  89, 235,  38,
```

The first row of the log table is the logs of:
0, 1, 2, 3, 4, 5, 6, etc. Zero is arbitrarily given a log of zero, and we check for zero before using the log tables. Log 1 = 0, OK. We see a log of 1 at the 3 position. So the base to power one is three, i.e. the base is three. We see that base squared is 5. It looks funny, but in polynomial arithmetic, it is:

```
 [00000011]  for the one bit in three
[00000011]   for the two bit in three (remember the shift)
[000000101]  mod M changes nothing for power less than 8
 [00000101]  this is five, and the table is correct.
```

In the anti-log table, the third position (start with 0) is 15, and we see a three in the 15^{th} (start with 0) position in the log table.

This is the start of the Substitution Block for encryption.

```
word8 S[256] = {
 99, 124, 119, 123, 242, 107, 111, 197,
 48,   1, 103,  43, 254, 215, 171, 118,
202, 130, 201, 125, 250,  89,  71, 240,
173, 212, 162, 175, 156, 164, 114, 192,
183, 253, 147,  38,  54,  63, 247, 204,
```

Let's look at the code. This is reference code for ease of understanding and for checking the validity of implementation of optimized code. The optimized code combines some steps.

Notice that in the first block the numbers add to 0 (mod 4) in each row, to 0 (mod 6) in the next block, and to 0 (mod 8) in the last block. These numbers are the shift row (encryption) and reverse shift row (decryption) tables. The first column is used for encryption, and the second for decryption.

My comments are in capital letters.

THESE TABLES DETERMINE THE AMOUNT OF SHIFT IN EACH ROW.

```
static word8 shifts[3][4][2] = {
   0, 0,
   1, 3,
   2, 2,
   3, 1,

   0, 0,
   1, 5,
   2, 4,
   3, 3,

   0, 0,
   1, 7,
   3, 5,
   4, 4
};
```

THIS IS USED TO DO PRODUCTS WITH LOG AND ANTILOG TABLES.

```
word8 mul(word8 a, word8 b) {
   /* multiply two elements of GF(2^m)
    * needed for MixColumn and InvMixColumn
    */
     if (a && b) return Alogtable[(Logtable[a] +
Logtable[b])%255];
     else return 0;
}
```

THIS DOES KEY ADDITION. FOR 128 BITS, THIS COULD BE DONE AS FOUR 32 BIT XOR'S. "BC" IS THE NUMBER OF COLUMNS IN A BLOCK, EITHER 4, 6, OR 8. "MAXBC" IS SET TO 8; IF YOU WANTED TO ALLOW BLOCK SIZES OF 512, YOU WOULD SET IT TO 16.

```
void KeyAddition(word8 a[4][MAXBC], word8 rk[4][MAXBC],
word8 BC) {
     /* Exor corresponding text input and round key input
bytes
      */
     int i, j;

     for(i = 0; i < 4; i++)
          for(j = 0; j < BC; j++) a[i][j] ^= rk[i][j];
}
```

THIS DOES ROW SHIFT. NOTICE THAT d PICKS THE DIRECTION IN THE SHIFTS MATRIX.

```
void ShiftRow(word8 a[4][MAXBC], word8 d, word8 BC) {
     /* Row 0 remains unchanged
      * The other three rows are shifted a variable amount
      */
     word8 tmp[MAXBC];
     int i, j;

     for(i = 1; i < 4; i++) {
          for(j = 0; j < BC; j++) tmp[j] = a[i][(j +
shifts[SC][i][d]) % BC];
          for(j = 0; j < BC; j++) a[i][j] = tmp[j];
     }
```

```
}

THIS IS A TABLE LOOKUP TO APPLY THE S-BOXES.

void Substitution(word8 a[4][MAXBC], word8 box[256], word8
BC) {
     /* Replace every byte of the input by the byte at that
place
      * in the nonlinear S-box
      */
     int i, j;

     for(i = 0; i < 4; i++)
          for(j = 0; j < BC; j++) a[i][j] = box[a[i][j]] ;
}


NOTE THE MULITPLICATIONS BY 2 AND 3.  THEY RELATE TO THE
MIXING MATRIX BELOW.  HERE THE COLUMN ELEMENTS ARE TREATED
AS COEFFICIENTS OF POLYNOMIALS OF DEGREE LESS THAN FOUR,
MOD  M(X) = X^4 + 1.


void MixColumn(word8 a[4][MAXBC], word8 BC) {
        /* Mix the four bytes of every column in a linear
way
      */
     word8 b[4][MAXBC];
     int i, j;

     for(j = 0; j < BC; j++)
          for(i = 0; i < 4; i++)
               b[i][j] = mul(2,a[i][j])
                    ^ mul(3,a[(i + 1) % 4][j])
                    ^ a[(i + 2) % 4][j]
                    ^ a[(i + 3) % 4][j];
     for(i = 0; i < 4; i++)
          for(j = 0; j < BC; j++) a[i][j] = b[i][j];
}
```

This is the Mixing Matrix for the Mix Columns step.

2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2

It is derived from this Mixing Polynomial:

‘03’x^3 + ‘01’x^2 + ‘01’x + ‘02’

This Mixing Polynomial is relatively co-prime with a new Modulus Polynomial:

M(X) = ‘01’X^4 + ‘01’ = X^4 + 1.

Notice that the coefficients of the Mixing Polynomial are ‘03’, ‘01’, and ‘02’. They are eight bits in length. They obey the rules of arithmetic set out at the start. M(X) is used as a modulus for arithmetic of polynomials of degree less than four in elements of the previously defined GF.

Notice that the previously defined lower case m(x) is of degree 8, and x is 0 or 1.

The new upper case M(X) is of degree four, and X is an element of the previously defined GF. This X is a byte in a column that is to be mixed.

The polynomial M(X) = X^4 + 1 is used for the modulus, because it gives rise to the property that:

x * (a*x^3 + b*x^2 + c*x + d)
= b*x^3 + c*x^2 + d*x + a (mod (x^4 +1))

It is a circular left shift of the coefficients. To see this, perform the product and "subtract" a * (x^4 + 1) to reduce the degree to less than four. Remember that a = -a (add and subtract are the same in the GF defined earlier). This shifting property is the reason that the rows of the Mixing Matrix shifted copies of each other.

M(X) is not irreducible (X + 1 is a factor), but the Mixing Polynomial is relatively prime to it. Just as in integer arithmetic, the non-zero numbers relatively prime to 2 and 13 have inverses (mod 26), the Mixing Polynomial has an inverse mod M(X). Because the polynomial has an inverse, the corresponding Mixing Matrix has an inverse. Thus, encryption can be reversed, and decryption is possible. Clever!