Implementation of Galois Field arithmetic and Rijndael Block Encryption in C++

Aven Bross

3/25/2014

Abstract

In this paper I explore implementing the mathematics and operation of Rijndael encryption. I begin with an examination of prime fields and Galois fields. Next, I discuss how each of these concepts can be implemented in C++. Finally I will discuss the Rijndael operations and how I implemented them in the C++ programming language.

1 Introduction

Rijndael is a very common block cipher, accepted as the Advanced Encryption Standard (AES) by the National Institute of Standards and Technology in 2001. The Rijndael has a layer based, non-Feistel structure, that was quite different from other ciphers that were considered in the AES competetion.
Most implementations of Rijndael are heavily optimized for performance, use numerous lookup tables, and completely mask the underlying operations that make Rijndael function. In this paper I will discuss an implementation that builds from the ground up, creating structures for mathematics with finite fields, matrices and finally Rijndael operations.

2 Math

2.1 Prime Fields

The first concept to discuss is the prime field. The prime field, for prime $p \in \mathbb{Z}$, is the set

$$\mathbb{Z}_p = \{0, 1, \ldots, p\}.$$

Addition and multiplication operations are all done $\mod p$.

2.2 Galois Fields

A Galois Fields is defined as

$$gf(p^n) = \{a_{n-1}p^{n-1} + \ldots + a_1 p + a_0 : a_0, \ldots, a_{n-1} \in \mathbb{Z}_p\}.$$

For each field $gf(p^n)$ we must also designate a modulus polynomial $Q(p)$ of degree n.
Addition in Galois Fields simply adds coefficients. For multiplication we simply distribute and multiply as normal for polynomials. If this results in a polynomial of degree greater than $n-1$ then it must be reduced to stay in the field $gf(p^n)$, so we take

$$AB \mod Q.$$

Example:
Let us consider the field $gf(2^4)$ with modulus polynomial $Q = 2^4 + 2 + 1$. Let $A = 2^3 + 2^2 + 1$ and $B = 2^2 + 2$. Then

$$A + B = (2^3 + 2^2 + 1) + (2^2 + 2) = 2^3 + 2 + 1,$$

and

$$\begin{aligned} AB &= (2^3 + 2^2 + 1)(2^2 + 2) \mod 2^4 + 2 + 1 \\ &= (2^5 + 2) \mod 2^4 + 2 + 1 \\ &= 2^2. \end{aligned}$$

We should notice that the prime fields discussed in part I are simply a special case of Galois Fields, $\mathbb{Z}_p \equiv gf(p)$.

3 Classes

This provides an overview of the C++ classes that were implemented to calculate Rijndael.

3.1 Modular Arithmetic

This class is used to perform arithmetic $\mod p$ for $p \in \mathbb{Z}$ prime.
It is a template class *Modular¡T¿* with member variables

```
T _val;
static T _modulus;
```

and constructor

```
Modular(const T& value);
```

3.1.1 Functionality

The *Modular<T>* class defines arithmetic operations in both non-modifying $\{+, -, *, /\}$ and modifying $\{+=, -=, *=, /=\}$ forms. It also provides comparison operators $\{==, <, >, <=, >=\}$. Furthermore it provides the function *addInverse()* and *mulInverse* which compute additive and multiplicative inverses. The function *value()* returns the value as its original type. Finally, the static function *globalSetModulus* allows modification of the member variable *_modulus*.

3.1.2 Algorithms

The Extended Euclidean Algorithm used to find multiplicative inverses.

```
template<typename T>
Modular<T> Modular<T>::mulInverse() const{
    T t = 0;
    T r = _modulus;
    T new_t = 1;
    T new_r = _val;
    while(new_r != 0){
        T quotient = r / new_r;
        T temp_t = t;
        t = new_t;
        new_t = temp_t - quotient * new_t;
        T temp_r = r;
```

```
        r = new_r;
        new_r = temp_r - quotient * new_r;
    }
    if(t<0) t = t+_modulus;
    return Modular<T>(t);
}
```

3.2 Polynomial

This class is used to store a polynomial of the form $a_{n-1}p^{n-1} + \ldots a_1 p + a_0$.

```
vector<Modular<int>> _a;
int _p;
```

and constructors

```
Polynomial(int value = 0, int p = 2, int n = 8);
Polynomial(const vector<Modular<int>> & a, int p = 2);
Polynomial(const Polynomial & other);
```

3.2.1 Functionality

The *Polynomial* class can be constructed either from an integer value, an array of coefficients or via a copy constructor. It also defines arithmetic operations in both non-modifying $\{+,-,*,/,\%\}$ and modifying $\{+=,-=,*=,/=,\%=\}$ forms. The *reduce()* private function is used throughout the operations to eliminate leading 0 coefficients. In this way *size()* of the polynomial, which is *size()* of the coefficient array, can be used to represent the degree n of the polynomial.

3.2.2 Algorithms

Multiplication is calculated by multiplying coefficients and adding their inices, to represent adding exponents.

```
Polynomial & Polynomial::operator*=(const Polynomial & other){
    if(_p != other._p) throw runtime_error("Mismatched prime.");

    vector<Modular<int>> a;

    for(int i=0; i<other.size()+this->size(); i++){
        a.push_back(0);
    }

    Modular<int>::globalSetModulus(_p);
    for(int i=0; i<this->size(); i++){
        for(int j=0; j<other.size(); j++){
            a[i+j] += _a[i]*other[j];
        }
    }

    swap(_a,a);

    reduce();

    return *this;
}
```

Division p/q is calculated by finding what multiple of q must be subtracted to eliminate the highest degree coefficent of p and repeating until degree of $p <$ degree of q.

```
Polynomial & Polynomial::operator/=(const Polynomial & other){
    if(_p != other._p) throw runtime_error("Mismatched prime.");

    Polynomial quot(0,_p,1);

    Modular<int>::globalSetModulus(_p);
    while(this->size() >= other.size()){
        Modular<int> c =  other._a.back() / _a.back();
        int diff = this->size()-other.size();
        vector<Modular<int>> v;
        for(int i=0; i< diff; i++)
            v.push_back(0);
        v.push_back(c);
        Polynomial p(v,_p);
        quot += p;
        (*this) -= (other * p);
    }

    swap(_a,quot._a);

    return *this;
}
```

Modulus is calculated the same way. Simply not tracking quotient and not swapping gives you the remainder.

3.3 Galois Polynomial

This class is used to store a polynomial in a Galois Field $gf(p^n)$. The class *GaloisPolynomial* has the member variables

```
Polynomial _polynomial;
static Polynomial _modulus;
```

and constructors

```
GaloisPolynomial(int value = 0, int p = 2, int n = 8);
GaloisPolynomial(const vector<Modular<int>> & v, int p = 2);
GaloisPolynomial(const Polynomial & p);
```

3.3.1 Functionality

Almost all of the functionality is borrowed from the Polynomial class, all that is done to make it work as a Galois Polynomial is track a *_modulus* and ensure that every operation is done mod _modulus.

The static function *globalSetModulus(const Polynomial & modulus)* allows modification of *_modulus*.

All arithmetic operations are the same as Polynomial, but a $\%=$ *_modulus* is performed afterwards. Furthermore, in this class the function *inverse()* allows calculation of multiplicative inverse so division operations, $\{/, /=\}$, are performed as multiplying by inverse instead of by the polynomial operations.

3.3.2 Algorithms

Multiplicative inverses are calculated via the Extended Euclidean algorithm, repeated division tracking an auxiliary.

```
GaloisPolynomial GaloisPolynomial::inverse() const{
    // Return 0 on 0
    if(_polynomial.size()<1)
        return GaloisPolynomial(0,_polynomial.getPrime(),1);
    Polynomial p1 = _modulus;
    Polynomial p2 = _polynomial;
    vector<Polynomial> aux;
    aux.push_back(Polynomial(0,_polynomial.getPrime(),1));
    aux.push_back(Polynomial(1,_polynomial.getPrime(),1));

    // Divide until we hit 1
    while(p1.size()>1 && p2.size()>1){
        Polynomial quot = p1/p2;
        Polynomial rem = p1%p2;
        Polynomial a_n = aux.back();
        a_n*=quot;
        a_n+=aux[aux.size()-2];
        aux.push_back(a_n);
        p1 = p2;
        p2 = rem;
    }

    return GaloisPolynomial(aux.back());
}
```

3.4 QSMatrix

This is a simple template class to represent an nXn matrix. It has member variables

```
vector<vector<T> > _mat;
int _rows;
int _cols;
```

and constructors

```
QSMatrix(int rows, int cols, const T& _initial);
QSMatrix(int rows, int cols, const vector<T> & _initial);
QSMatrix(const QSMatrix<T>& rhs);
```

3.4.1 Functionality

This class allows addition and subtraction of same sized matrices, multiplication of matrices that can be multiplied, and finally multiplication of matrices with acceptable vectors. It also has accessors to grab elements by (row,column).

4 Rijndael Operations

4.1 Galois Field

The Rijndael cipher uses the Galois Field $gf(2^8)$ with modulus polynomial $2^8+2^4+2^3+2^1+1$.
The way $gf(2^8)$ represents a number happens to be the same way a byte is represented in binary data,

which is exactly how the fields will be utilized in Rijndael.

4.2 Data Representation

Rijndael is a block cipher that operates on 128bit blocks of data, or 16bytes. The 16 bytes of data are represented as 16 polynomials $a_{i,j} \in gf(2^8)$ in a 4x4 matrix. This matrix is known as the state S.

$$S = \begin{pmatrix} a_{(0,0)} & a_{(0,1)} & a_{(0,2)} & a_{(0,3)} \\ a_{(1,0)} & a_{(1,1)} & a_{(1,2)} & a_{(1,3)} \\ a_{(2,0)} & a_{(2,1)} & a_{(2,2)} & a_{(2,3)} \\ a_{(3,0)} & a_{(3,1)} & a_{(3,2)} & a_{(3,3)} \end{pmatrix}$$

4.3 S-Box

The core operation in Rijndael encryption is the S-Box. The S-Box acts as a non-linear mapping $s : gf(2^8) \to gf(2^8)$. This process is defined as $s(p) = A \cdot p^{-1} + b$. First p is treated as a polynomial in $gf(2^8)$ and we take it's multiplicative inverse, p^{-1}. Then we treat the coefficients of $p = \{a_0, \ldots, a_7\}$ as a vector and perform the operation

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

The S-Box can be inverted by performing the operations

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

and then taking the multiplicative inverse of the result.
The C++ implementation follows, with the rijndael_A, rijndael_b being a vector representation of the A matrix and the b vector described above.

```
GaloisPolynomial & sBox(GaloisPolynomial & p){
    // Invert each element
    p = p.inverse();

    // Calculate affine transformation
    vector<Modular<int>> coef;
    for(int r=0; r<8; r++){
        Modular<int> sum(0);
        for(int c=0; c<8; c++){
```

```
            sum += p[c] * rijndael_A[r*8+c];
        }
        coef.push_back(sum);
    }
    p = GaloisPolynomial(coef);

    // Add polynomial b
    p += rijndael_b;

    return p;
}
```

4.4 Sub Bytes

In the Sub Bytes operation, the S-Box is applied to every polynomial in the state S.

$$S' = \begin{pmatrix} s(a_{(0,0)}) & s(a_{(0,1)}) & s(a_{(0,2)}) & s(a_{(0,3)}) \\ s(a_{(1,0)}) & s(a_{(1,1)}) & s(a_{(1,2)}) & s(a_{(1,3)}) \\ s(a_{(2,0)}) & s(a_{(2,1)}) & s(a_{(2,2)}) & s(a_{(2,3)}) \\ s(a_{(3,0)}) & s(a_{(3,1)}) & s(a_{(3,2)}) & s(a_{(3,3)}) \end{pmatrix}$$

This operation is inverted by performing the inverse S-Box on each element.

C++ implementation:

```
QSMatrix<GaloisPolynomial> & subBytes(QSMatrix<GaloisPolynomial> & state){
    Modular<int>::globalSetModulus(2);
    for(int i=0; i<state.getRows(); i++){
        for(int j=0; j<state.getCols(); j++){
            sBox(state(i,j));
        }
    }

    return state;
}
```

4.5 Shift Rows

In the Shift Rows operation, each row in the state is shifted over by its index. The purpose of this state is to shift polynomials between columns which will later be mixed together.

$$S' = \begin{pmatrix} a_{(0,0)} & a_{(0,1)} & a_{(0,2)} & a_{(0,3)} \\ a_{(1,1)} & a_{(1,2)} & a_{(1,3)} & a_{(1,0)} \\ a_{(2,2)} & a_{(2,3)} & a_{(2,0)} & a_{(2,1)} \\ a_{(3,3)} & a_{(3,0)} & a_{(3,1)} & a_{(3,2)} \end{pmatrix}$$

This operation is easily inverted by simply right shifting instead of left shifting the rows.
C++ implementation:

```
QSMatrix<GaloisPolynomial> & shiftRows(QSMatrix<GaloisPolynomial> & state){
    for(int i=0; i<state.getRows(); i++){
        vector<GaloisPolynomial> temp;
        for(int j=0; j<state.getCols(); j++){
            int c = (j+i)%state.getCols();
```

```
            temp.push_back(state(i,c));
        }
        for(int j=0; j<state.getCols(); j++){
            state(i,j) = temp[j];
        }
    }

    return state;
}
```

4.6 Mix Columns

In this operation, the matrix as a whole is multiplied by a mixing matrix. The purpose of this operation is to mix together the polynomials in each column.

$$S' = \begin{pmatrix} 2 & 3 & 1 & 1) \\ 1 & 2 & 3 & 1) \\ 1 & 1 & 2 & 3) \\ 3 & 1 & 1 & 2) \end{pmatrix} S$$

This operation can be inverted by multiplying by the inverse matrix

$$S = \begin{pmatrix} 14 & 11 & 13 & 9) \\ 9 & 14 & 11 & 13) \\ 13 & 9 & 14 & 11) \\ 11 & 13 & 9 & 14) \end{pmatrix} S'$$

C++ implementation follows, with rijndael_M bein the matrix described above.

```
QSMatrix<GaloisPolynomial> & mixColumns(QSMatrix<GaloisPolynomial> & state){
    state = rijndael_M * state;
    return state;
}
```

4.7 Add Key

In this operation we simply add together they state S and the current round key K_i. Round keys are generated via the Rijndael key schedule, discussed later.

$$S' = S + K_i$$

C++ implementation:

```
QSMatrix<GaloisPolynomial> & addRoundKey(QSMatrix<GaloisPolynomial> & state,
                                         const QSMatrix<GaloisPolynomial> & key){
    state += key;
    return state;
}
```

4.8 Encryption Process

To encrypt we begin by adding the initial key. Then we loop through rounds going through Sub Bytes, Shift Rows, Mix Columns and Add Key each round until the final round, where the Mix Columns step is ommitted.

C++ implementation:

```
addRoundKey(state, keyMatrices[0]);
for(int i=1; i<rounds; i++){
    subBytes(state);
    shiftRows(state);
    mixColumns(state);
    addRoundKey(state, keyMatrices[i]);
}
subBytes(state);
shiftRows(state);
addRoundKey(state, keyMatrices[rounds]);
```

Decryption is done via a reverse process:

```
addRoundKey(state, keyMatrices[rounds]);
shiftRows_inverse(state);
subBytes_inverse(state);
for(int i=rounds-1; i>0; i--){
    addRoundKey(state, keyMatrices[i]);
    mixColumns_inverse(state);
    shiftRows_inverse(state);
    subBytes_inverse(state);
}
addRoundKey(state, keyMatrices[0]);
```

5 Key Schedule

5.1 Key Size

The version of Rijndael I examined utilizes a key of 128bits, but it can also use keys of 192bits or 256bits. This 128bit key needs to be expanded to a number of 128bit keys equal to the number of rounds plus one initial key. The standard for Rijndael is 10 rounds and thus 11 keys.

5.2 Key Schedule Core

This core operation is utilized many times throughout the key schedule's operation. The takes a word of 4 bytes and an iteration number i.

- Rotate each the each byte in the word to the left by one byte.
- Apply S-Box to each byte.
- Add in Galois Field (XOR) the first byte in the word with 2^{i-1}.

C++ implementation:

```
vector<GaloisPolynomial>& keyExpandCore(vector<GaloisPolynomial> & word,
                                        int iteration){
    // Do a simple left rotate of bytes
    GaloisPolynomial temp(word[0]);
    word[0] = word[1];
    word[1] = word[2];
    word[2] = word[3];
    word[3] = temp;

    // Apply S-Box to each byte
    for(int i=0; i<4; i++){
        sBox(word[i]);
    }

    // Apply Rcon operation (add 2^(iteration-1) to word[0])
    vector<Modular<int>> v;
    for(int i=0; i<iteration-2; i++){
        v.push_back(0);
    }
    v.push_back(1);
    word[0] += GaloisPolynomial(v);

    return word;
}
```

5.3 Key Schedule

This will describe the process to retrieve $128 \cdot 11$ bits of key.

The first 128bits are simply the encryption key. Initialize the round counter $i = 1$.
Next, the following process loops, generating 16bytes each time, until we have the desired number of bits in our key.

- 1. Set temporary word k equal to the previous 4 bytes.
- 2. Perform key schedule core on k with i, $i+ = 1$.
- 3. Add in Galois Field (XOR) k with the the word 128bits earlier in the key.
- 4. Append this word to the key.
- 5. Perform the previous steps 3 times, but skip step 2 each time.

C++ implementation:

```
// Just copy in key for first 16 bytes
for(int i=0; i<16; i++){
    keyBytes.push_back(extractPoly(key));
}

int iteration = 1;

// Continue until we have a 128 bit key for each round + an initial key
while(keyBytes.size()<16*(rounds+1)){
    for(int w=0; w<4; w++){
        // Copy last 4 bytes
        vector<GaloisPolynomial> word;
        for(int i=0; i<4; i++){
```

```
            word.push_back(keyBytes[keyBytes.size()-5+i]);
        }

        // Apply core operations if first word
        if(w==0) keyExpandCore(word, iteration);

        // Add each byte with that 16 bytes back
        for(int i=0; i<4; i++){
            word[i] += keyBytes[keyBytes.size()-16+i];
        }

        // Append
        keyBytes.insert(keyBytes.end(), word.begin(), word.end());
    }

    iteration += 1;
}
```

6 Results

6.1 Galois Inverse Calculations

The most import part of the *GaloisPolynomial* class was the multiplicative inverse calculation. It uses almost all of the operations in the *Polynomial* and is core to the functionality of the Rijndael S-Box. Below is a test printout from the multiplicative inverse calculation.

```
Finding inverse of 110
```

Remainder	Quotient	Auxiliary
1*2^8+1*2^7+1*2^5+1*2^4+1*2^0		
1*2^6+1*2^5+1*2^3+1*2^2+1*2^1		1*2^0
1*2^3+1*2^0	1*2^2	1*2^2
1*2^1	1*2^3+1*2^2	1*2^5+1*2^4+1*2^0
1*2^0	1*2^2	1*2^7+1*2^6

```
= c0
```

6.2 Encryption Run

Here is the test printout from a short encrypt and decrypt run, each polynomial/byte displayed in hex. Encrypting goes top to bottom on the right, decrypting bottom to top on the right.

```
Adding Round Key
|5  13 3d f2| |60 cb 1e f7| |65 d8 23 5 |
|59 75 2b 7e| |37 cb 7c 68| |6e be 57 16|
|a3 70 66 2c|+|e9 2d 17 86|=|4a 5d 71 aa|
|8f cb a5 32| |1  e4 14 6d| |8e 2f b1 5f|

Sub Bytes
|65 d8 23 5 | |e8 5d 62 b6|
|6e be 57 16| |2b 67 29 db|
|4a 5d 71 aa|=|a1 15 dc 75|
|8e 2f b1 5f| |60 a3 45 6b|

Shift Rows
|e8 5d 62 b6| |e8 5d 62 b6|
|2b 67 29 db| |67 29 db 2b|
|a1 15 dc 75|=|dc 75 a1 15|
|60 a3 45 6b| |6b 60 a3 45|

Mix Columns
|2  3  1  1 | |e8 5d 62 b6| |7f d4 1a f0|
|1  2  3  1 | |67 29 db 2b| |98 f0 94 9a|
|1  1  2  3 |*|dc 75 a1 15|=|3b 3e 1e 78|
|3  1  1  2 | |6b 60 a3 45| |e4 7b 2b df|

Adding Round Key
|7f d4 1a f0| |9c 54 8  6e| |e3 80 12 9e|
|98 f0 94 9a| |5a 57 28 60| |c2 a7 bc fa|
|3b 3e 1e 78|+|87 77 40 ae|=|bc 49 5e d6|
|e4 7b 2b df| |61 63 63 2d| |85 18 48 f2|

Sub Bytes
|e3 80 12 9e| |e  b2 6a 20|
|c2 a7 bc fa| |ce df fa 8d|
|bc 49 5e d6|=|fa 4a cd a0|
|85 18 48 f2| |ba ac 21 3b|

Shift Rows
|e  b2 6a 20| |e  b2 6a 20|
|ce df fa 8d| |df fa 8d ce|
|fa 4a cd a0|=|cd a0 fa 4a|
|ba ac 21 3b| |3b ba ac 21|

Adding Round Key
|e  b2 6a 20| |15 85 d9 28| |1b 37 b3 8 |
|df fa 8d ce| |77 42 ad b9| |a8 b8 20 77|
|cd a0 fa 4a|+|af 0  2  3 |=|62 a0 f8 49|
|3b ba ac 21| |d8 cc 63 2f| |e3 76 cf e |

Adding Round Key
|65 d8 23 5 | |60 cb 1e f7| |5  13 3d f2|
|6e be 57 16| |37 cb 7c 68| |59 75 2b 7e|
|4a 5d 71 aa|+|e9 2d 17 86|=|a3 70 66 2c|
|8e 2f b1 5f| |1  e4 14 6d| |8f cb a5 32|

Inverting Sub Bytes
|e8 5d 62 b6| |65 d8 23 5 |
|2b 67 29 db| |6e be 57 16|
|a1 15 dc 75|=|4a 5d 71 aa|
|60 a3 45 6b| |8e 2f b1 5f|

Inverting Shift Rows
|e8 5d 62 b6| |e8 5d 62 b6|
|67 29 db 2b| |2b 67 29 db|
|dc 75 a1 15|=|a1 15 dc 75|
|6b 60 a3 45| |60 a3 45 6b|

Inverting Mix Columns
|e  b  d  9 | |7f d4 1a f0| |e8 5d 62 b6|
|9  e  b  d | |98 f0 94 9a| |67 29 db 2b|
|d  9  e  b |*|3b 3e 1e 78|=|dc 75 a1 15|
|b  d  9  e | |e4 7b 2b df| |6b 60 a3 45|

Adding Round Key
|e3 80 12 9e| |9c 54 8  6e| |7f d4 1a f0|
|c2 a7 bc fa| |5a 57 28 60| |98 f0 94 9a|
|bc 49 5e d6|+|87 77 40 ae|=|3b 3e 1e 78|
|85 18 48 f2| |61 63 63 2d| |e4 7b 2b df|

Inverting Sub Bytes
|e  b2 6a 20| |e3 80 12 9e|
|ce df fa 8d| |c2 a7 bc fa|
|fa 4a cd a0|=|bc 49 5e d6|
|ba ac 21 3b| |85 18 48 f2|

Inverting Shift Rows
|e  b2 6a 20| |e  b2 6a 20|
|df fa 8d ce| |ce df fa 8d|
|cd a0 fa 4a|=|fa 4a cd a0|
|3b ba ac 21| |ba ac 21 3b|

Adding Round Key
|1b 37 b3 8 | |15 85 d9 28| |e  b2 6a 20|
|a8 b8 20 77| |77 42 ad b9| |df fa 8d ce|
|62 a0 f8 49|+|af 0  2  3 |=|cd a0 fa 4a|
|e3 76 cf e | |d8 cc 63 2f| |3b ba ac 21|
```

References

[1] Nover, H. (2005). Algebraic Cryptanalysis of AES: An Overview. University of Wisconsin, USA.

[2] Benvenuto, C. (2012). Galois Field in Cryptography. University of Washington.

[3] Daemen, Joan, and Vincent Rijmen. "The Rijndael block cipher." AES Proposal http://csrc. nist. gov/encryption/aes/rijndael/Rijndael. pdf (1999).

[4] Gallian, J. (2009). Contemporary abstract algebra. Cengage Learning.