AES Cryptosystem (128-bit)

Michael Levet

December 19, 2014

Introduction to AES
This tutorial will introduce the AES Cryptosystem. Both theory and implementation will be provided. All code is provided on the main tutorial page, with a ZIP file including a working copy. While many versions of AES exist, this tutorial will cover the original 128-bit AES Cryptosystem, also known as Rijndael.

The Rijndael Cryptosytem is a symmetric-key, substitution-permutation network. That is, the same key used for encrypting the plaintext is also used to decrypt the ciphertext. We generate our round keys from the provided key, and use those to modify the plaintext on each iteration. After adding in a given round key, we apply a substitution function to each byte of our modified plaintext, then permute the result. We iterate and repeat this process ten times before returning the ciphertext.
In this tutorial, I assume a certain amount of mathematical familiarity. I expect familiarity with modular arithmetic and the concept of relative primality. From this, I will introduce a cursory amount of abstract algebra necessary to use and follow this tutorial. I would strongly suggest reading through the algebra section, making careful note to pay attention to how the operations of addition and multiplication are defined. If this is your first cryptosystem and you don't have a lot of mathematical familiarity, this is probably not the best cryptosystem to work through. A textbook like Stinson or Boneh's online cryptography course are good resources for getting into cryptography.

As a last note, this cryptosystem implementation is purely for study. You should never use your own implementation of a cryptosystem in practice, for numerous security reasons.

Preliminaries- A Brief Introduction to Algebra
Abstract Algebra is not typically covered within a computer science program. Any algebra covered in the context of computer science usually deals with monoids and posets, as opposed to the standard trinity of groups, rings, and fields. Abstract algebra comes into play with the AES cryptosystem in defining the Rijndael Field. This field is used as the backbone of the cryptosystem. In order to understand what a field is, it is first necessary to understand a more basic structure: a group.

A group is a set of elements with a single binary operation. Let G be a group, and let $*$ be the binary operator, which is defined as $* : G \times G \to G$. That is, the operator is closed, or a function that takes a pair of elements from G and maps them to an element in G. The operator is also associative. That is, for any $a, b, c \in G$, $(a*b)*c = a*(b*c)$. We may also drop the star and simply write ab to denote the operation.

A group also has an identity element, which we denote as e. And every element in the group has an inverse. That is, for any $a \in G$, there exists an a^{-1} such that $a * a^{-1} = a^{-1} * a = e$.

This definition of a group is a bit abstract, so let's consider a couple of examples. The first example is the group $(\mathbb{Z}, +)$. That is, consider the integers over addition. The additive identity element is 0; that is, $x + 0 = x$ for every integer x. Every element has an additive inverse. Take $a \in \mathbb{Z}$. Then $a^{-1} = -a$, when dealing with addition. Associativity is trivial. Note that addition over $\mathbb{Z}$ is commutative. Such

groups where the operation is commutative are called Abelian groups.

Consider $(\mathbb{Z} - \{0\}, *)$, the non-zero integers over multiplication. This set fails to form a group, as only ± 1 are invertible over multiplication. However, $(\mathbb{R} - \{0\}, *)$, the set of non-zero real numbers, forms a group over multiplication. Since we aren't dividing by 0, we simply take $\frac{1}{r}$ as the inverse for any $r \in \mathbb{R}$. And so $r * \frac{1}{r} = 1$, the multiplicative identity element.

We now define a field. A field is a set of elements with two binary operations $(\mathbb{F}, +, *)$. We consider 0 the additive identity and 1 the multiplicative identity. The elements over the addition operation form an Abelian group (a commutative group), and the non-zero elements over multiplication also form an Abelian group. Some common examples of fields are the real numbers ($\mathbb{R}$), the rational numbers ($\mathbb{Q}$), the complex numbers ($\mathbb{C}$), and the integers modulo p $\mathbb{Z}_p$ (for p a prime).

Rijndael Field

The AES cryptosystem relies on the Rijndael Field, which is a polynomial field over the field on two elements, which we denote $\mathbb{Z}_2$ (the integers modulo 2). So we start with the polynomials $\mathbb{Z}_2[x]$. That is, the coefficients of the polynomials are taken modulo 2. We then mod out the polynomials by $x^8 + x^4 + x^3 + x + 1$. We denote this field as: $\mathbb{Z}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. Algebraists will recognize the notation as a quotient ring. Those familiar with number theory can think of $x^8 + x^4 + x^3 + x + 1$ as a prime polynomial. So $Z_2[x]/(x^8 + x^4 + x^3 + x + 1)$ is a field much in the same way that $\mathbb{Z}_p$ (or $\mathbb{Z}/p\mathbb{Z}$) is a field. We note the Rijndael Field has 256 elements with additive identity 0 and multiplicative identity 1. Addition of polynomials is done component-wise (as you are probably used to doing), and then we reduce the coefficients mod 2. Multiplication on the Rijndael Field is defined using polynomial multiplication.

So consider the polynomial $x^9 + x$. We consider its equivalence in the Rijndael field by finding y such that $x^9 + x \equiv y \pmod{x^8 + x^4 + x^3 + x + 1}$.
Finding y is quite similar to modular arithmetic in the integers. Consider $8 \equiv z \pmod 3$. To find z, we divide $8/3$ and consider the remainder, which is 2. That is $8 = 2(3) + 2$. So $8 \equiv 2 \pmod 3$. Similarly, we seek to write $x^9 + x = p(x)(x^8 + x^4 + x^3 + x + 1) + y$, where $p(x)$ is a polynomial in $\mathbb{Z}_2[x]$ and y is in the Rijndael field. So we divide $(x^9 + x)/(x^8 + x^4 + x^3 +^x +1)$ and take the remainder to be y.

So $x^9 + x = x(x^8 + x^4 + x^3 + x + 1) + (x^5 + x^4 + x^2)$.

Since addition is XOR (that is, we reduce the coefficients mod 2), adding in $(x^5 + x^4 + x^2)$ cancels out the extraneous terms. So we get:

$$x(x^8 + x^4 + x^3 + x + 1) + (x^5 + x^4 + x^2) = x^9 + 2x^5 + 2x^4 + 2x^2 + x$$

Reducing the coefficients mod 2 leaves us with $x^9 + x$, as desired. So this is exactly like working with integer division and modular arithmetic in a number theory context.

I offer an implementation of an element in the Rijndael Field, with the class RijndaelPolynomial. The code is included on the main tutorial page. Operations of addition, multiplication, and inversion are used. In order to calculate the multiplicative inverse, we use Lagrange's Theorem or an analogue of Fermat's Little Theorem. By definition of a field, $\mathbb{Z}_2[x]/(x^8 + x^4 + x^3 + x + 1) - \{0\}$ forms an Abelian group over multiplication. Since we exclude the zero element, the group has order (cardinality) 255 elements. Hence, for any element in the multiplicative group, $x^{|G|} = x^{255} = x$ in the group. And so $x^{254} = 1$, which means that $x^{253} = x^{-1}$ over multiplication.

The main takeaway with the Rijndael Field is that it works basically like polynomial addition and multiplication as we are used to it. The coefficients are all 0 or 1, and no element in the field has degree

greater than 7. The Rijndael Field then has the added constraint of working like the integers modulo n. Instead of applying mod n, for n an integer, we mod out by $x^8 + x^4 + x^3 + x + 1$.

Key Expansion

The Key Expansion phase takes the 128-bit key and expands it to get the eleven round keys for each of the ten rounds of the cryptosystem. We apply a round key prior to the start of the first round. Each round key consists of four words, with each word equivalent to four bytes. The procedure is simple enough, but a bit cumbersome.

We consider the keySchedule as an array of 44 words. The first word consists of the first four bytes of the key. The second word consists of the second four bytes. We continue this for the third and fourth words.

To generate the rest of the keySchedule, we use the expandKey() algorithm. The implementation is provided on the main tutorial page. The rotateAndSub() method is not very complicated, and I will provide the RCON constants array later.

So essentially, we take the $i-1$ and $i-4$ Words and XOR them together. The result is the ith Word in the keySchedule. Every fourth word is also run through the rotateAndSub() method, then XOR'd with a given RCON constant.

The rotateAndSub() method begins by moving the front byte to the back, and then applying the subByte() method to each byte. The subByte() method accepts a byte of the form $(a_7, a_6, a_5, ..., a_0)$. This byte as treated as a polynomial in the Rijndael Field. The a_i element is the coefficient of xi in the Rijndael Field. So 01010011 is equivalent to $x^6 + x^4 + x + 1$. The subByte() algorithm can also be viewed as an Affine Transformation, given below:

Sub Bytes Transformation

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \end{bmatrix}.$$

The next step is to calculate the multiplicative inverse of the element in the Rijndael Field, so long as the byte is non-zero. The inverse of $x^6 + x^4 + x + 1$ is $x^7 + x^6 + x^3 + x$, which is equivalent to the binary string 11001010. We then convert the inverse polynomial back to binary and add the vector $c = (01100011)$ to it, with c_7 starting at the far-left.

The subByte() method, KeyExpansion class, and Word class are all included on the main tutorial page.

Encryption Algorithm

Now it is time to examine the encryption algorithm. We accept a 128-bit key along with a 128-bit plaintext message to encrypt. Let the plaintext message $x = (x_0, x_1, ..., x_{15})$ be the byte representation, where each x_i is a byte of the plaintext message.

From the byte representation of x, we construct a 4×4 array called State as follows:

$$\text{State} := \begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix}$$

The idea now is that we add each round key to State, where addition is the XOR operation. Then each byte of State is permuted using the SubBytes algorithm, and then we rearrange the rows and columns of State. The algorithm exactly, is as follows:

```
encrypt(plaintext, key)
    keyExpansion := expand(key) //get the array of 44 Words
    state := convertToMatrix(plaintex) //construct 4x4 array

    RoundKey := array(keyExpansion[0], ..., keyExpansion[3])
    state := state XOR RoundKey

    for i = 1 to 9
        subBytes(state)
        shiftRows(state)
        mixColumns(state)

        RoundKey := array(keyExpansion[4i], ..., keyExpansion[4i+3])
        state := state XOR RoundKey
    end for

    subBytes(state)
    shiftRows(state)

    RoundKey := array(keyExpansion[40], ..., keyExpansion[43])
    state := state XOR RoundKey

    return format(state)
```

The final ciphertext is assembled from the 4x4 array state in the same way that it is constructed. The first column consists of the first four bytes; the second column the second four bytes; etc. So:

$$ciphertext := (state_{0,0}, state_{1,0}, ..., state_{3,0}, state_{0,1}, ..., state_{3,1}, ..., state_{3,3})$$

Now let's step through each portion of the algorithm. We have already covered key expansion. So the first important step is to add the round key. Each round key is formed from four consecutive Words, which are column vectors. We then add the two matrices component-wise using an XOR operation.

The next step is the shiftRows() step. It takes the current state matrix and shifts the elements in the ith row i elements to the left. So row 0 is unchanged, row 1 has its elements shifted to the left by one slot, etc. The implementation of shiftRows(), contained in the State class, is on the main tutorial page.

Finally, consider the mixColumns() operation. We use the following matrix below and set $State := M \cdot State$.

$$M = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix}$$

The mixColumn() operation appears in the Word class, on the main tutorial page.

Decryption
The AES cryptosystem is a symmetric-key cryptosystem. That means in the decryption algorithm works by running the encryption algorithm essentially in reverse. Each of the operations addRoundKey(), mixColumns(), shiftRows(), and subBytes() needs to be inverted. The addRoundKey() method operates using the XOR operation, which is its own inverse. The shiftRowsInverse() step is just as trivial. We simply rotate each row to the right to undo the left-shift from the encryption algorithm.

So the general algorithm for decryption is:

```
decrypt(ciphertext, key)
    keyExpansion := expand(key) //get the array of 44 Words
    state := convertToMatrix(ciphertex) //construct 4x4 array

    RoundKey := array(keyExpansion[40], ..., keyExpansion[43])
    state := state XOR RoundKey
    shiftRowsInverse(state)
    subBytesInverse(state)

    for i = 9 to 1
       RoundKey := array(keyExpansion[4i], ..., keyExpansion[4i + 3])
       state := state XOR RoundKey

       mixColumnsInverse(state)
       shiftRowsInverse(state)
       subBytesInverse(state)

    end for


    RoundKey := array(keyExpansion[0], ..., keyExpansion[3])
    state := state XOR RoundKey

    return format(state)
```

Let's now examine each of the inverse operations:

Sub Byte Inverse:

Much like the original subByte() algorithm, the subByteInverse() is an affine transformation, given by the following matrix equation:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

I offer an implementation (in the Word class) on the main tutorial page, which evaluates this affine transformation. The constant array used is provided in the Word class as well, and corresponds to the vector being XOR'd with the linear transformation.

Mix Columns Inverse

The mixColumnsInverse() algorithm is given by the following linear transformation. Note that each number in the square matrix corresponds to an element in the Rijndael field. We convert each number in the square matrix to binary, and the binary representation corresponds to a polynomial in the Rijndael field. So the vectors are over the Rijndael field (addition is XOR and multiplication is polynomial multiplication where the coefficients are reduced mod 2).

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

I provide an implementation (in the Word class) on the main tutorial page.