Advanced Encryption Standard and Ring Theory

Clifford Berger

Copyright ©2013 by Clifford Berger. This work is made available under the terms of the Creative Commons Attribution 3.0 Unported License,
http://creativecommons.org/licenses/by/3.0/

The Advanced Encryption Standard was established as the modern standard for data encryption in 2001. It is used to encode everything from classified government documents to the traffic through your wifi router. The previous data encryption standard (DES) was first established in 1977, but with the advance of technology DES became increasingly vulnerable. Eventually the only truly secure way to use DES was to use it three times in encryption, which was far too slow, and needless to say, we began to search for a better encryption standard. AES was chosen through a three year competition. Though sponsored by the U.S. National Institute of Standards and technology (NIST), the competition hosted applicants from numerous countries. The AES we use today is based on a cipher developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen. Together they developed the Rijndael cipher, which was selected as the most suitable encryption method for AES. AES is just a version of the Rijndael cipher with the cipher block size restricted to 128 bits.

The Rijndael algorithm performs a set procedure to encrypt a 128 bit block of code. One important property of AES is that it is what is called a symmetric or private key algorithm, which means that the same key is used for encryption and decryption as opposed to a public key cryptosystem. The Rijndael algorithm takes a 128, 192, or 256 bit key to encrypt each block of code. The actions performed by the Rijndael algorithm utilize several key mathematical concepts from linear algebra and ring theory. As we move through the algorithm we will see some very interesting connections between these subjects. In the context of this article we will to focus most heavily on the application of ring theory in AES. Understanding the mechanics of the Rigndael algorithm requires surprisingly little knowledge of cryptography. The algorithm entails 10 to 14 repetitions of a simple four step process, with minor differences in the first and last repetition or "round". Each step of the algorithm is a simple action with mathematical calculations. We will explain cryptologic terms as they appear.

There are essentially three big ideas in cryptography and of course they apply in the case of AES. The three big ideas are confusion, diffusion, and the use of a key. Confusion is the method of essentially mapping each part of your code to something else, a classic example of a basic confusion technique is the Julius Caesar cryptosystem which shifts each letter in the message down three letters. For example the Caesar cipher would encode the word "hello" as "khoor". Diffusion is the act of permuting your code, which is just changing the order of characters. An example of this would be writing your code in a matrix horizontally from top to bottom, then rewriting your code with each column as a group of letters. The third big idea of cryptography is using a key. If you do not use a key someone can decrypt anything if they know how the cipher works. Today practically all cryptosystems use a key. The methods of most cryptosystems can be Googled instantly, but if you do not have the key then you cannot decrypt anything. The AES cryptosystem implements these simple ideas but uses math to perform strong and efficient encryption. Let us walk through each step of the algorithm and see just how it does this.

For the encryption process the Rijndael algorithm requires a plaintext block and key as input. The very first step of the Rijndael algorithm is to load our code into 4×4 matrices. We call this the state matrix. We load our code into the state matrix by entering it top to bottom from left to right. The Rijndael algorithm will apply the same operations to each 4×4 matrix of code, so we will simply focus on how the cipher acts on one of these matrices. As aforementioned, AES is designed to take $128, 192$, or 256 bit keys, but for simplicity's sake we will consider the case with a 128 bit key. The algorithm will function in the same way with a larger key, but with more repeated steps. We now load our key into another 4×4 matrix, again entering the key from top to bottom left to right. Each column contains 4 bytes (32 bits) representing 4 characters, so for a 192 or 256 bit key we would have a 4×6 or 4×8 matrix as the cipher key. Now for each entry in our plaintext matrix we combine it with the corresponding entry of the key matrix using an operation called XOR.

In order to perform these operations we must first represent each character of our plaintext and key with its corresponding binary code. The XOR operation, also know as the "exclusive or" operation, takes two binary values of equal length and performs integer addition modulo 2 on each pair of digits in the same position. This means that it returns a 0 if both digits are the same and a 1 if the digits are different. For example 10110110 XOR 11111111 = 01001001. After applying XOR to each pair of corresponding matrix entries we have a new 4×4 matrix containing binary codes for entries. This is called the add round key step, but we will discuss that in detail later. Now we have a new state matrix like the one below:

$$\begin{bmatrix} P & N & X & E \\ L & & T & R \\ A & T & & E \\ I & E & H & \end{bmatrix} \text{XOR} \begin{bmatrix} 1 & B & K & H \\ 2 & I & E & E \\ 8 & T & Y & R \\ & & & E \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,2} & a_{2,3} & a_{2,4} & a_{2,1} \\ a_{3,3} & a_{3,4} & a_{3,1} & a_{3,2} \\ a_{4,4} & a_{4,1} & a_{4,2} & a_{4,3} \end{bmatrix}$$

Where $a_{i,j}$ is some binary code, and keep in mind that the letters here are actually binary codes as well.

But this is just the beginning of the algorithm. It turns out that the Rijndael algorithm actually uses a number of different keys derived from the original key we provided. This is called key expansion, and the algorithm used to calculate the different keys is called the key schedule. The originial key we provided is called the first round key, and the key schedule calculates the new keys for the other rounds (there are 10 rounds with a 128 bit key). Note that we consider adding the first round key as an action performed before the first round. In each round we repeat a set procedure, but we will get to that soon. The key schedule is perhaps the most interesting part of AES, this is the part where we really see the mathematical computations behind the cipher. To get a general idea of what happens we will run through the key schedule and then we can explain what actually happens.

The first thing we do to calculate the next round key is take the last column of the current round key matrix and shift all entries up one slot, moving the top entry to the bottom entry. Now we run the entries of this column through what is called the Rijndael substitution box, or s box for short. S boxes are often used in private key cryptography to provide confusion, the Rijndael s box is just one version. This s box maps every byte in the column to a new byte, providing confusion for the cryptosystem. The Rijndael s box uses key concepts of ring theory to perform this mapping, and we will discuss this in great detail momentarily. Here is a graphical representation, but keep in mind entries are binary codes and not letters.

$$\begin{bmatrix} 1 & B & K & H \\ 2 & I & E & E \\ 8 & T & Y & R \\ & & & E \end{bmatrix} \longrightarrow \begin{bmatrix} H \\ E \\ R \\ E \end{bmatrix} \longrightarrow \begin{bmatrix} E \\ H \\ E \\ R \end{bmatrix} \longrightarrow \begin{bmatrix} \text{S box}(E) \\ \text{S box}(H) \\ \text{S box}(E) \\ \text{S box}(R) \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Where b_i is some binary code

Now we XOR the entries of this new column with what is called a round constant, this is just a column vector that is different for each round. We finally XOR this with the first column of the previous round key. The product of these operations will be the first column of the new round key. The other three columns of our new key are much easier to calculate. For the second, third, and fourth columns of our new key we simply XOR the second, third, and fourth columns of our previous key with the new column we just calculated. To make things clear here is a graphical representation, and please note the round constant entries are represented in hexadecimal for ease of notation.

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \text{XOR} \begin{bmatrix} 01 \\ 00 \\ 00 \\ 00 \end{bmatrix} = \begin{bmatrix} c_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Where c_1 is some binary code. Note that b_1 is the only entry altered by this operation, this holds for the XOR operation on all round constants.

$$\begin{bmatrix} c_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \text{XOR} \begin{bmatrix} 1 \\ 2 \\ 8 \\ \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} \longrightarrow \begin{bmatrix} d_1 & \dots & \dots & \dots \\ d_2 & \dots & \dots & \dots \\ d_3 & \dots & \dots & \dots \\ d_4 & \dots & \dots & \dots \end{bmatrix}$$

This column vector on the right will be the first column of the new key. We XOR the last three columns of the previous round matrix with this vector to get the last three columns of the new key matrix.

We now have the key matrix for the next round, and the key for all future rounds are calculated using this procedure. For a 128 bit key there are a total of 10 rounds (not including when we XOR the state and the initial key) which means this procedure is repeated ten times to produce ten different expanded keys in addition to our first round key. But we have not explained what this S box is or what these round constants are. This is where ring theory really comes into play. Now that we have seen the S box used in AES we will see how it works.

Up to now we have been using a binary code to represent characters. A byte is just an ordered set of eight bits, and each bit is represented by a binary value of 1 or 0. So a byte, or binary code, is just a binary number with eight digits, giving us $2^8 = 256$ different possible bytes. Hexadecimal (base 16 decimal system) notation is frequently used in cryptography because any eight digit binary numer can be represent with two digits in hexadecimal notation. It turns out that we can represent a byte using a degree 7 polynomial with coefficients in $\mathbb{Z}_2$, the ring of integers modulus 2. This means there exists an invertible mapping from binary values to polynomials of degree 7 or less. For example we would map the binary value 10101011 to the polynomial $x^7 + x^5 + x^3 + x + 1$. Since $\mathbb{Z}_2$ is a field we know that $\mathbb{Z}_2[x]$ is an integral domain.

We now have a ring of polynomials where all polynomials of degree 7 or less are representations of binary values. Let us consider the set of just these polynomials from $\mathbb{Z}_2[x]$ and $0 \in \mathbb{Z}_2[x]$. We can easily see that these elements form an additive group since the sum of any two elements will also be a polynomial of degree 7 or less, but the set of nonzero elements in $\mathbb{Z}_2[x]$ are not closed under multiplication. In most cases the product of two polynomials will give us a new polynomial with degree greater than 7, which cannot properly represent an eight digit binary value. What if we were to build a ring, or better yet a field, containing only these polynomials representing binary values. This is easily accomplished by building a principal ideal with an irreducible polynomial and forming the quotient ring of $\mathbb{Z}_2[x]$ over this polynomial. This ring must be field since we use an irreducible polynomial, and this field is exactly what the Rijndael cipher uses for calculations.

The Rijndael cipher uses the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1 \in \mathbb{Z}_2[x]$ to form the ideal and quotient ring. The Rijndael algorithm requires a structure for polynomials of maximum degree 7, $m(x)$ was chosen because it's quotient ring in $\mathbb{Z}_2$ restricts polynomials to degree 7 and it's irreducibility makes this structure a field. This field, $\frac{\mathbb{Z}_2[x]}{m(x)}$ is commonly called the Rijndael Finite Field. Every element in this field is of the form $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 + < m(x) >$ with $0 \le a_i \le 1$ so there are a total of $2^8 = 256$ elements in the field and all of them represent a distinct binary value. Since this field contains 2^8 elements it is the Galois Field GF(2^8). We can now say we have an invertible mapping from bytes to a finite field.

Now lets take this opportunity to look at addition in this field and how it relates to the actions we have performed in the Rijndael algorithm. We previously introduced the XOR operation for bytes, it turns out that this operation corresponds to polynomial addition in our field. If we add two polynomials together, we are just adding the coefficients of each term. But the coefficients are over $\mathbb{Z}_2$, so the sum of the pair of coefficients of each term will be 0 if both coefficients are the same (1 or 0), or 1 if the coeffiecients are different. This is exactly what the XOR operation does with bytes. To see this in action let us look at the example of XOR we used earlier and it us corresponding polynomial addition in the field. 10110110 XOR 11111111 = 01001001 corresponds to $(x^7 + x^5 + x^4 + x^2 + x + <>) + (x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 + <>) = x^6 + x^3 + 1 + <>$.

The fact that we are in a field tells us that every nonzero element of the field has a multiplicative inverse. The Rijndael algorithm uses this property as a foundation for the cipher. Now that we have familiarized ourselves with the Rijndael Finite Field we can go back and explain the substitution box from our key schedule. The S box is a mapping that takes one byte value to another. Given a byte value, the S box function first maps the byte to a polynomial in the field, then it calculates the multiplicative inverse and places the coefficients in a vector, if the input byte is 0 it simply returns the zero vector. The algorithm then multiplies this vector by a matrix and adds a constant vector to this matrix vector product. It might be easier to think about the S box as a composition of functions, one that provides the inverse of an element and one that plugs the coefficients of the polynomial into the S box formula. The S box function is defined as follows:

$$\text{S box}(g) = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Where a_i is the coefficient of x^i in g^{-1}. Note that S box returns a vector, the entries of this vector form the output byte.

In the context of cryptography the S box can be considered as a sort of lookup table. The S box maps every byte to a certain output regardless of the first round key or the plaintext to be encrypted, so the input/output data can be loaded in a table. It is much more time efficient for a computer to lookup the output for a certain input than to actually run the S box calculations every time. We should also note that since we are working in a finite field a computer can easily brute force the inverse of a polynomial instead of trying to use an algorithm to calculate the inverse.

We have now covered all aspects of the key expansion process except for the round constants. The round constant for any round is calculated from the previous round constant. Earlier we used the first round constant, which is just a vector of size four with the binary value 1 in entry one and 0 for the other three entries. The next round constant is calculated by taking this first entry of the current round constant and converting it to a polynomial in the field. Then we multiply this polynomial by x, convert it back to a byte, and finally place it in the first slot of a size four vector with the 0 byte in the last three entries. All round keys have 0 in the last three entries of the vector. This process is repeated until we have calculated all ten round constants (there can be up to 14 for a 256 bit key).

Now that we have covered key expansion and the Rijndael substitution box, we can continue through the Rijndael algorithm and see the procedure for one round of AES encryption. The last step we performed in the was performing the XOR operation on our plaintext bytes and our first round key, the resulting matrix is our current state matrix. We can now perform the next step, called substituting keys. This is considered the first step of the first round, adding the first round key is done before the first round actually starts. Now that we understand the S box, this is very straightforward. We apply the S box function to every entry of the state matrix, mapping every byte in the matrix to something new. We now have a new state matrix and have completed this step of the algorithm. This step provides the cryptologic confusion we discussed earlier, so in every round we blur the relationship of each byte.

The next step in this round is called the shift rows step. This step is just permuting the state matrix in a certain way. The first row of the matrix is left unaltered in this step. In the second row each entry is moved one entry to the left with the leftmost entry wrapping around to the rightmost entry. In the third row each entry is moved two to the left, again with the 2 leftmost entries wrapping around to the right side. As you might expect in the third row we simply shift each entry three entries to the left, wrapping around yet again. We have now completed this step and have a new state matrix with the same bytes in different places. The purpose of this step is to provide the big idea of diffusion we discussed.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,2} & a_{2,3} & a_{2,4} & a_{2,1} \\ a_{3,3} & a_{3,4} & a_{3,1} & a_{3,2} \\ a_{4,4} & a_{4,1} & a_{4,2} & a_{4,3} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

The next step is almost as interesting as the key schedule. This step is called the mix columns step, and it is so interesting because it can be performed either by using matrix multiplication or polynomial multiplication. Here we get to see how the two relate in this context. Since it is easier to understand we will start with the matrix multiplication method and then see how it corresponds to polynomial multiplication. The matrix multiplication method takes each column of the state matrix as a vector and runs it through an invertible linear transformation before returning it to the state matrix. The new column is calculated by:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Where vector $\mathbf{x}$ is the column vector from the old state matrix and $\mathbf{y}$ is the new column. Note that the mix columns matrix is nonsingular and therefore invertible.

The other method using polynomials is much more interesting. This method takes each column of the state matrix and converts it to a polynomial of degree 3 or less with coefficients in $GF(2^8)$. This means that instead of having a binary value of 1 or 0 we have byte values as coefficients. We now take this polynomial representing the column and multiply it by a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$ modulo $x^4 + 1$. $c(x)$ and $x^4 + 1$ are coprime so $c(x)$ is an invertible polynomial. The coefficients of the resulting polynomials are now the new entries of that column of the state matrix. To see how this relates to the matrix multiplication method we will expand this equation.

$$\begin{aligned} b(x) &= (3x^3 + x^2 + x + 2)(ax^3 + bx^2 + cx + d) && \mod x^4 + 1 \\ &= 3ax^6 + ax^5 + ax^4 + 2ax^3 + 3bx^5 + bx^4 + bx^3 + 2bx^2 + 3cx^4 + cx^3 + cx^2 + 2cx \\ &+ 3dx^3 + dx^2 + dx + 2d \\ &= (2a + 3b + c + d)x^3 + (a + 2b + 3c + d)x^2 + (a + b + 2c + 3d)x + 3a + b + c \\ &+ 2d && \mod x^4 + 1 \end{aligned}$$

Where $b(x)$ is the polynomial with coefficients for the new column and a, b, c, and d are the entries of this column from the state matrix.

The coefficients of this polynomial provide an explicit formula for the entries of the new column of the state matrix, and this formula is exactly the same as formula given by matrix multiplication. We now see how the linear transformation matrix can be derived completely from this operation of polynomial multiplication. It turns out that modular multiplication with a fixed polynomial can be written as matrix multiplication. This step also provides diffusion for the cipher. After applying this process to every column in the state matrix we have completed the mix columns step and have now nearly completed the round. We now proceed to the final step of the round, adding the round key.

Earlier we discussed how to perform the key expansion step, now we finally get to implement the newly calculated round key in the algorithm. After familiarizing ourselves with the other parts of the algorithm, I think most would agree this is the easiest step of all. All we do now is take our state matrix and XOR it with the current round key we calculated. This is exactly what we did in the very first step after loading our key and plaintext bytes into matrices. But now we can think of these as matrices of polynomials in the Rijndael Finite Field representing the binary values, and the XOR operation is equivalent to just adding the matrices together. After we have done this we have finally completed a round of AES.

Now all the algorithm is going to do is repeat the steps of substituting bytes, shifting rows, mixing columns, and adding the round key until it reaches the final round. This final round is exactly the same as all the other except we cut out the process of mixing columns. In the final round the diffusion provided by the mixing columns step will not be passed on to the next round, so performing this step would not increase security and would only slow things down. Now we have completed the entire encoding process for AES. Like most any other cryptosystem AES is just a method of obscuring the relationship between the plaintext and cyphertext, diffusing this data, and making ensuring the cryptosystem cannot be cracked without a key.

The decryption process is almost exactly the same as the encryption process, just in opposite order and using the inverse of the transformations we used in the encryption algorithm. Since this time we know we will need to calculate the keys for every round of the algorithm, we will start with the key expansion procedure. This is exactly the same as the procedure we described in the encryption algorithm, but we will be starting with the tenth key and will work our way back.

Going in reverse order of the encryption algorithm, the first step we execute is the add round key step. For our first round we will use the eleventh round key (since there are 10 round keys in addition to the first round key), and will use the $(12 - i)$th round key for round i of the decryption process. Since the add round key step uses the XOR operation, applying the same XOR operation will invert the add round key process.

Now we want to apply the inverse of the shift rows step. To do this we apply the shift row step just like in the encryption process, but instead of shifting to the left we will shift to the right. Now we are ready to perform the inverse of the substitute bytes step. Recall that for this step we ran every byte in the state matrix through the Rijndael substitution box, where the substitution box was a mapping to a new byte. The substitution box is a bijective mapping and therefore must be invertible. The inverse of S box is:

$$\text{S box}^{-1}(g) = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Where a_i is the coefficient of x^i in g^{-1}

So the substitute keys step is performed by running all entries of the state matrix through Sbox^{-1}. We have now completed the first round of decryption. The following rounds repeat this process, but we now must use the inverse of the mix columns step after we add the round key and before we perform the shift rows step.

Earlier we established that the mix columns matrix was invertible and so the linear transformation must be invertible. So to perform the inverse of this step we simply run each column of the state matrix through the inverse linear transformation (multiplying the column by the inverse of the matrix). We could also perform this step with polynomials like we did in the encryption process, but we multiply our polynomial representation of the column by $c^{-1}(x) = 11x^3 + 13x^2 + 9x + 14 \bmod (x^4 + 1)$ instead of $c(x)$. This will invert the mix columns step from the encryption process.

So the decryption process for every round after the first is to add the round key, inverse mix columns, inverse shift rows, and inverse substitute bytes. When the algorithm reaches the final round it finishes by XORing our original key with the state matrix, and the resulting matrix contains the bytes of the plaintext originally encoded.

We have now covered all important aspects of how the Rijndael algorithm is used for encryption and decryption in the AES. Note that this article focused on using a 128 bit key, but if we were to use a 192 or 256 bit key the only major difference is that we would have to do 12 or 14 rounds, respectively. This in turn would also modify the key expansion step. Executing more rounds of the algorithm always trades performance for security. We can conclude from this investigation of AES that ring theory has some great applications in the real world, and the algebraic structure it supplies makes computing much more efficient. We can now think of a letter as being represented by a distinct polynomial, and we also made an interesting connection between matrix multiplication and modular multiplication with a fixed polynomial. Overall the AES standard makes great use of mathematical concepts to provide great security in encryption while also offering great performance.

Bibliography

Bhargav, Shrivathsa, Larry Chen, Abhinandan Majumdar, and Shiva Ramudit. “128-bit AES decryption.” Columbia University. N.p., n.d. Web. 25 Apr 2013. http://www.cs.columbia.edu/ sedwards/classes/2008/4840/reports/AES.pdf

Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001.

Moser, Jeff. “A Stick Figure Guide to the AES.” Moserware. N.p., 22 Sep 2009. Web. 25 Apr 2013. http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html.

Rijmen, Vincent, and Joan Daemen. The Design of Rijndael: AES. Springer, 2002.