

Advanced Encryption Standard

Brian Carter, Ari Kassin, and Tanja Magoc

October 30, 2007

1 Introduction

The Advanced Encryption Standard (AES), which implements the Rijndael cipher, is a symmetric block cipher that was developed as a result of a call by the United States National Institute of Standards and Technology in 1997 for a secure cryptosystem to replace the then standard Data Encryption Standard algorithm, which had become vulnerable to brute-force attacks. Among several proposals, the Rijndael cipher was chosen and was renamed AES in 2001. The cryptosystem is based on a strong mathematical background of field theory, the basics of which we describe in the following section. Next, we present the encryption algorithm, with a note that the decryption, as in all symmetric cryptosystems, uses the same key and the inverse transformations and multiplicative inverses of the polynomials used for encryption. The remainder of the paper discusses the real-life implementations of the algorithm and concluding remarks [2].

2 Mathematical Background

To clearly understand the principles that form the basis of AES, we review some properties of fields as described in [3]. We start with the general definition of a field and proceed, step by step, to more complex structures used in AES.

Definition 1. A field is a set F together with two operations, $+$, addition, and \cdot , multiplication, satisfying the following properties:

1. Closure property under $+$: $\forall a, b \in F, a + b \in F$.
2. Associativity under $+$: $\forall a, b, c \in F, a + (b + c) = (a + b) + c$.
3. Identity for $+$: $\exists e \in F$, a unique identity element called the zero element, such that $\forall a \in F, e + a = a + e = a$.
4. Inverse for $+$: $\forall a \in F, \exists a^{-1} \in F$, the inverse element of a , such that $a + a^{-1} = a^{-1} + a = e$.
5. The above properties also hold under \cdot over the set F^* , which is the set of all nonzero elements of F . Also, note that under multiplication, the element of F satisfying Property 3 is called the unit element.
6. Distributive axiom: $a \cdot (b + c) = a \cdot b + a \cdot c$.

Next, we define the field $(F_2, +_2, \cdot_2)$, which is a step closer to the structure used in AES.

Definition 2. $(F_2, +_2, \cdot_2)$ is a field, where F_2 is the set of binary numbers (i.e., $F_2 = \{0, 1\}$), and $+_2$ and \cdot_2 are the addition and multiplication operations modulo 2, respectively.

AES utilizes the characteristics of a field of polynomials modulo an irreducible polynomial, so we will provide a few more definitions that will ease the understanding of AES. First, we define a polynomial over a field. Then, we will give the definition of an irreducible polynomial. Finally, we define the field of polynomials modulo a polynomial.

Definition 3. A polynomial over a field $(F, +, \cdot)$ is an expression $\sum_{i=0}^{i=n} a_i \cdot x^i$, where $a_i \in F$ and x is a symbol (not an element of F).

Definition 4. A polynomial f is irreducible over the field F if and only if $f = g \cdot h$ with $g, h \in F$ implies that either f or g is a constant.

Definition 5. A field of polynomials $F[x]$ modulo a polynomial f , given by $F[x]_f = \{h \in F \mid h = g \bmod f \text{ for some } g \in F\}$, is a set of polynomials over the field F of degree smaller than the degree of f .

Given the necessary definitions, we state one result that justifies the choice of the polynomial used in AES.

Theorem 1. If f is irreducible, then $F[x]_f$ is a field.

With the review of the basic mathematical tools necessary for an understanding of AES, we proceed to describe the implications of these elements in the AES algorithm [1].

AES uses the byte-based representation of plaintext, so we first note that a byte can be represented using polynomials over the field F_2 with degree lower than 8; for example, a byte 10001101 can be represented as $x^7 + x^3 + x^2 + 1$. Next, we realize that the following polynomial of degree 8, $m(x) = x^8 + x^4 + x^3 + x + 1$, is an irreducible polynomial over the field F_2 , and therefore, by Theorem 1, $F_2[x]_m$, is a field for this $m(x)$. Note that the choice for the polynomial $m(x)$, besides guaranteeing the existence of a field, has another important property: each byte could be represented as a polynomial modulo $m(x)$. The field $F_2[x]_m$ is often represented as $\text{GF}(2^8)$, so we will use this notation for the remainder of the paper.

The field $\text{GF}(2^8)$ is associated with two operations, addition and multiplication of polynomials with binary coefficients modulo the polynomial $m(x)$. The addition can be easily performed as bitwise exclusive-or (XOR) operation, which simplifies the practical implementation of this operation and will be further discussed in the AES implementation section.

The multiplication in the field $\text{GF}(2^8)$ could be split into two parts: the first part is the multiplication of two polynomials over the field $(F_2, +_2, \cdot_2)$ —the coefficients of the resulting polynomial could be calculated simply as a bitwise AND. The second step in the multiplication process in the field $\text{GF}(2^8)$ is the division by the polynomial $m(x)$. The remainder of this division is the final product of two polynomials in the field $\text{GF}(2^8)$. The step-by-step explanation of the division of two polynomials is omitted in this paper but can be found in [4].

2.1 Justification of the Theory Behind the AES Algorithm

As mentioned earlier, AES is a symmetric cryptosystem. Therefore, it uses the same key for encryption and decryption. To allow for this phenomena, all operations used in the encryption process must possess an inverse operation to exactly undo the transformations applied to the plaintext in order to recover the plaintext message from the ciphertext at the time of decryption.

A field obviously contains the additive and the multiplicative inverse of each of its elements and guarantees the existence of the inverse operation of any transformation (within the scope of operations available in the field) applied to its elements. Thus, a field satisfies the requirements that will allow use of the same key in the encryption and the decryption processes.

However, not every field would be suitable for algorithms like AES. The field $\text{GF}(2^8)$ makes it simple to represent plaintext as bytes in terms of elements of the field, and the choice of $m(x)$ guarantees that

each byte can be represented by an element of the field. This justifies the choice of the algebraic structure and the particular field used in AES algorithm.

3 AES Algorithm

After reviewing needed mathematical concepts and justifying the theory behind AES, we proceed to an explanation of the algorithm itself. The AES algorithm [1] is relatively simple: like many ciphers, AES is a block cipher. AES specifies three block lengths: 128, 192, or 256 bits. Similarly, AES allows the same choices for a key length. Note that the lengths of a block and a key do not need to be the same for the encryption of a message.

AES uses the notion of a State to maintain each block, whether it is plaintext or ciphertext. The State is a matrix of four rows and Nb columns, where Nb is calculated by dividing the block length by 32. Similarly, the Cipher Key is a matrix of four rows and Nk columns—in this case, Nk is the key length divided by 32. Each entry of each matrix contains exactly one byte, represented as a polynomial, as previously described.

AES is a round-based cipher. The number of rounds required to perform AES encryption or decryption (denoted Nr) is based on the values of Nk and Nb as described in the table that follows:

Nr	$Nb = 4$	$Nb = 6$	$Nb = 8$
$Nk = 4$	10	12	14
$Nk = 6$	12	12	14
$Nk = 8$	14	14	14

The next step is to apply the actual Rijndael algorithm to the State. The process is composed of four steps: `ByteSub`, `ShiftRow`, `MixColumn`, and `AddRoundKey`. These four steps are applied to the State exactly Nr times with one exception: during the last round, the `MixColumn` step is omitted. Thus, in C-like pseudocode, we can simply describe the algorithm in the following way:

```

Round (State, RoundKey) {
    State = ByteSub (State);
    State = ShiftRow (State);
    State = MixColumn (State);
    State = AddRoundKey (State, RoundKey);

    return State;
}

FinalRound (State, RoundKey) {
    State = ByteSub (State);
    State = ShiftRow (State);
    State = AddRoundKey (State, RoundKey);
}

main () {
    for (i=0; i < Nr - 1; i++) {
        State = Round (State, RoundKey);
    }

    State = FinalRound (State, RoundKey);
}

```

Next, we describe each of the four steps of a round.

3.1 ByteSub

We perform `ByteSub` on each entry of the State matrix. First, we need to obtain the multiplicative inverse (in $\text{GF}(2^8)$) of the entry—that is, we solve $a \cdot b \pmod{m(x)} = 1$ for b , where a is the State matrix entry and b is the multiplicative inverse of a . Next, we apply an affine transformation to each entry:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

In this affine transformation representation, y_n is the resulting bit n of the State entry and x_n is the original bit of that entry.

The `ByteSub` step of the algorithm could be represented by the following pseudocode:

```
ByteSub (State) {
    for (each entry x of State) {
        obtain the multiplicative inverse, x', of x in GF(2^8);
        apply the affine transformation to x';
    }

    return State;
}
```

3.2 ShiftRow

The `ShiftRow` process is very simple. We circularly shift to the left the bottom three rows of the State matrix: the second row is shifted by $C1$, the third row by $C2$, and the bottom row by $C3$. These shift amounts are based on the block length Nb :

Nb	$C1$	$C2$	$C3$
4	1	2	3
6	1	2	3
8	1	3	4

We present the pseudocode for the `ShiftRow` operation together with the subroutine `ShiftRowLeft` that shifts each Row circularly to the left C units:

```
ShiftRow (State) {
    Row2 = ShiftRowLeft (Row2, C1);
    Row3 = ShiftRowLeft (Row3, C2);
    Row4 = ShiftRowLeft (Row4, C3);

    return State;
}
```

```
ShiftRowLeft (Row, C) {
    for (i = 0; i < C; i++) {
```

```

    temp[i] = Row[i];
}

for (i = 0; i < Nb - C; i++) {
    row[i] = row[i + C];
}

for (i = 0; i < C; i++) {
    row[Nb - C + i] = temp[i];
}

Return Row;
}

```

3.3 MixColumn

In the `MixColumn` process, we consider each column of the State matrix to be a polynomial over $\text{GF}(2^8)$. Next, we multiply each column by the polynomial $c(x) = '03'x^3 + '01'x^2 + '01'x + '02'$. This multiplication is done modulo $(x^4 + 1)$. The multiplication can be simplified by using matrix multiplication (for which all multiplications are modulo $x^4 + 1$)—that is, we calculate $b(x) = (c(x) \cdot a(x)) \bmod (x^4 + 1)$ for each column of the State. In this matrix multiplication, we let vector b be the resulting column and vector a be the original column.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Again, it is important to note that the `MixColumn` process is not performed during the final round of AES.

A simple description of the `MixColumn` process follows:

```

MixColumn (State) {
    for (i = 0; i < Nb; i++) {
        multiply column i by the constant matrix c;
    }

    return State;
}

```

3.4 RoundKey

The final process of AES requires only the use of XOR operations. We first need to obtain a Round Key such that the Round Key matrix contains the same number of entries as the State matrix. We start with the cipher key matrix and add new columns as follows: the first new column is the XOR of the last and the first columns of the original key representation. The next column is produced by applying the XOR operation to the last existing column and the second column of the matrix. We continue this procedure of creating column i by XOR-ing the last existing column (column $i - 1$) with column $i - Nk$, where Nk is the number of columns of the original key matrix. In this way, we produce a large matrix of four rows and $Nb \cdot Nr$ columns.

The process of generating the matrix of extended key can be simplified as follows:

```

KeyGeneration (CipherKey) {
    for (i = 0; i < Nk; i++) {
        let column i of ExtendedKey equal column i of CipherKey;
    }

    for (i = Nk; i < Nb * Nr; i++) {
let column i of ExtendedKey equal
    [column (i - 1) of ExtendedKey] XOR [column (i - Nk) of ExtendedKey];
    }

    return ExtendedKey;
}

```

For each round of the algorithm, we use a different Round Key. This key is extracted from the above constructed `ExtendedKey` matrix by taking the first Nb columns of the matrix in the first round of the algorithm, the second Nb columns for the second round, and so on. Continuing in this fashion, the i^{th} round of the algorithm uses the part of the matrix from column $Nb \cdot (i - 1)$ to the column $Nb \cdot i - 1$. The key for the given Round is produced as follows:

```

KeySelection (ExtendedKey, Round) {
    initialize RoundKey to an empty matrix of size (4 x Nb);

    for (i = 0; i < Nb; i++) {
        replace column i of RoundKey with column [Nb * (Round - 1) + i] of
            ExtendedKey;
    }

    return RoundKey;
}

```

Finally, to perform the `AddRoundKey` process, we simply XOR each entry of the `RoundKey` matrix with the corresponding entry of the `State` matrix.

```

AddRoundKey (State, RoundKey) {
    for (each entry x of State) {
        State[x] = State[x] XOR (corresponding entry of RoundKey);
    }

    return State;
}

```

4 AES Implementations

The AES algorithm is often presented in a scientific manner, trying to show all the steps and operations needed to complete each phase of the algorithm. According to Daemen and Rijmen [1], there are only four processes in AES, namely `ByteSub`, `ShiftRow`, `MixColumn`, and `AddRoundKey`, each executed one after another iteratively. From the implementation perspective, this is not the best way of applying the Rijndael algorithm, and some work could be done to optimize parts of the algorithm.

4.1 Software Implementation

One way to speed up execution and, at the same time, prevent timing attacks, is by combining the `ByteSub` and `ShiftRow` processes with the `MixColumn` step by transforming them into a sequence of table lookups. This method requires four 256-entry 32-bit tables, each utilizing one kilobyte of memory, for a total memory utilization of the order of four kilobytes. With this approach, a round of the AES algorithm could be applied with sixteen table lookups and twelve 32-bit XOR operations followed by four 32-bit XOR operations in the `AddRoundKey` step [5]. To allow this speedup method to work, we need to be sure to use a byte-oriented approach that allows us to combine the `SubByte`, `ShiftRow`, and `MixColumn` steps into a single round operation.

There are many implemented AES algorithms, written in different programming languages, including combinations of high-level languages and assembly code. Perhaps the clearest and easiest version is written in C# [6]. This version is roughly 440 lines of code and does not utilize any assembly code. The implementation is very good if there is a need for a fast implementation that can be maintained and debugged using C#.

Another implementation worth mentioning is the implementation initiated by Rijmen, one of the authors of AES, and its subsequent improvements [8], named `Crypto++`. `Crypto++` is a highly optimized implementation of AES written in C++, which uses some assembly code as well to gain better performance. This implementation also uses lookup tables instead of on-the-spot calculations, and it assumes the tables are uploaded in the first-level cache for fast access and algorithm speedup.

With many different implementations existing, a good method to test the correctness of an implementation is to encrypt a plaintext n times (such that $n > 1$), then decrypt it $n - 1$ times, and finally verify that the expected results were obtained [7]. For example, we can encrypt the plaintext, and then keep encrypting each new obtained ciphertext (the new “plaintext”) 999 more times. Next, we decrypt the ciphertext repeatedly 999 times and compare it against the expected ciphertext after encrypting the original plaintext only once. This is a good practice to make sure that the algorithm is giving expected results rather than losing small pieces of information, such as decimal values, and resulting in an approximation, which would not be acceptable in real world applications.

4.2 Real-world Applications

Every day, we face applications which use the AES algorithm. This algorithm has been used in different electronic devices in real life. To mention just a few applications, the algorithm has found use in PCs, PDAs, ATMs, mobile phones, wireless devices, and DVDs, just to name a few [9]. The impressive facts about the AES implementations come from its use on 8-bit processors used in many embedded systems, which require low-cost and low-power performance. One of the most well-developed and highly used applications of this type of processor is the Wireless Sensor Network used for various monitoring applications [11]. For this particular application, AES requires a code length of only about a kilobyte and RAM of only 52 bytes when using a 256-bit key and even less RAM when using smaller keys [10].

Another important implementation of AES algorithm is in the security of smart cards. It has been tested that Rijndael was the fastest-performing cipher when compared to several other algorithms that were candidates for AES in the late 90s. At the same time, AES was using the least amount of RAM, which adds to the effectiveness of the algorithm. AES demonstrated impressive speed and small memory requirements for both—a low-cost 8-bit smart card, such as the Intel 8051, or a more sophisticated 32-bit processor, such as the ARM card [13].

Further, the AES algorithm is used in larger systems as a method of privacy transform for IP Security and Internet Key Exchange. The Rijndael cipher replaced the previously used Data Encryption Standard (DES) in this role because of its higher security, a result of the larger key size. Also, the variable key lengths do not allow an outsider to crack the security in any faster way than an exhaustive search [15].

The real-world implementations of AES do not end here. We can keep on listing the devices, systems, and situations in which this cipher is used. We will now turn our attention to the security of the algorithm.

4.3 Attacks on AES

Theoretically, there is not a known way to crack the AES cryptosystem other than the use of brute-force attacks, but the actually implemented algorithm uses parts of the hardware that leave the system open to attacks [5]. A side-channel attack could happen because each part of the implemented algorithm has its own runtime, so in theory, if the attacker can correlate runtimes with knowledge of the implementation, he could extract information about the key. It is only fair to say that these attacks are very difficult to attempt and practically impossible in real life, as there are many circumstances and knowledge of implementation details is required. Also, there are practices that can be used to defend against these attacks, like manipulating the kernel behavior and trying to send information along specific CPU pipelines, but as stated earlier, the attack, by itself, is only theoretical and very unlikely to occur in the real world.

Another planned attack to AES algorithm was the square attack, which was successful in breaking Rijndael's predecessor, a block cipher called Square. The square attack exploits the byte-oriented structure of the algorithm [14] to extract information about the cipher key. This attack has been proven successful if the AES algorithm would have no more than seven rounds for the 128-bit key and no more than nine rounds for the 192- and 256-bit keys. However, with the current number of rounds for each possible key length, the square attack does not seem to threaten the security of AES unless we are able to reach the level of power necessary to break Rijndael cipher.

Attacks on AES have had more success when implementing AES onto small-memory devices such as DVDs. Hackers have been able to find the keys used for encryption by using a debugger to inspect the memory space of running HD-DVD and Blu-ray player software programs [12].

Another relatively successful attack against small devices such as smart cards is power analysis [13]. The idea of this attack is to measure the power consumption of a smart card while writing the subkey bytes into RAM. This cryptanalysis method does not rely on specific plaintext or implementation details, and therefore, is an even larger cause for concern as a possible attack on the AES cryptosystem.

However, even with a few devices vulnerable to attacks, AES still remains one of the most secure cryptosystems, especially in applications with a larger memory space, which allows use of longer keys and additional masking in the key generation part of the algorithm.

5 Conclusion

In conclusion, AES is a symmetric block cipher that relies heavily on a background of field theory, yet the algorithm itself is easily understood and implemented in different programming languages with a need for only about 440 lines of code in C#. The Rijndael cipher basically consists of four invertible transformations iterated a predetermined number of times. However, the effectiveness of the algorithm comes from the possible optimizations, usually incorporating lookup tables instead of calculations in the field $GF(2^8)$.

AES exploits different block and key sizes to allow for variety of real-life applications. It has found successful applications in small embedded systems such as mobile phones and wireless sensor networks, as well as in larger desktop computers. With this in mind, even with a few possible attacks on AES, Rijndael remains one of the most secure ciphers known today that allows encryption and decryption (with key knowledge) in a reasonable amount of time, and at the same time, cannot be broken (without key knowledge) in a period of time that could endanger information security.

References

- [1] Daemen, J. and Rijmen, V., *AES Proposal: Rijndael*. 1999.
- [2] “Advanced encryption standard process”,
http://en.wikipedia.org/wiki/Advanced_Encryption_Standard_process
- [3] Mao, W. *Modern Cryptography: Theory & Practice*. Upper Saddle River, NJ: Prentice Hall PTR, 2004.
- [4] “Polynomial long division”,
<http://www.sosmath.com/algebra/factor/fac01/fac01.html>
- [5] “Advanced encryption standard”,
http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- [6] McCaffrey, J., “Keep Your Data Secure with the New Advanced Encryption Standard”,
<http://msdn.microsoft.com/msdnmag/issues/03/11/AES/>
- [7] Denis, T. St., *Cryptography for Developers*. Syngress Publishing, 2007.
- [8] “Crypto++ library 5.5.2”,
<http://www.cryptopp.com>
- [9] “VIA PadLock Security Engine”,
<http://www.via.com.tw/en/initiatives/padlock/hardware.jsp#aes>
- [10] “Advanced encryption standard released today”,
<http://www.seifried.org/security/cryptography/20001002-aes.html>
- [11] “Review of Hardware Architectures for Advanced Encryption Standard Implementations Considering Wireless Sensor Networks”,
<http://www.springerlink.com/content/m578n4022434n735/>
- [12] “Advanced Access Content System”,
http://en.wikipedia.org/wiki/Advanced_Access_Content_System
- [13] “Second advanced encryption standard candidate conference”,
<http://nvl.nist.gov/pub/nistpubs/jres/104/4/html/j44ce-dwo.htm>
- [14] Lucks, S., “Attacking seven rounds of Rijndael under 192-bit and 256-bit keys”. *Theoretical Informatik*, Germany.
- [15] “Cisco documentation”,
http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newft/122t/122t13/ft_aes.htm