Advanced Encryption Standard

Alexander Shevtsov

May 17, 2016

Contents

1 Overview

Advanced Encryption Standard (AES) — cryptographic standard specifying symmetric block cipher.

It was adopted by NIST after a 5-year open international process. Its original name is Rijndael (it's a play on the names of the two Belgian inventors: Joan Daemen and Vincent Rijmen). In most cases titles "AES" and "Rijndael" are used interchangeably.

De facto today Rijndael is standard for block ciphers. It's used to encrypt secret government data, private information, messages. It's used in specifications of W3C, IETF, ISOC—organizations developing standards for Internet. If you're reading that text using computer, smartphone or any other device, most likely, that processor in your device has internal implementation of AES. All in all, millions of people every day rely on this algorithm.

Rijndael is a symmetric-key block algorithm. What do these words mean?

Cryptographic algorithms are used to keep information in secret, to make impossible for an unauthorized person know it. Most of todays algorithms use secret keys which are known only to authorized people.

Symmetric-key algorithm means that the same key is used both for encryption and decryption, and that key must be known for both parties involved. On the contrary, there are asymmetric algorithms using different keys for encryption and decryption, they are also called public-key algorithms.

Block cipher means that data to encrypt or decrypt is divided into blocks of some size, for AES the size of the block is 128 bits.

2 History

On the second of January, 1997 NIST announced an open competition for the new encryption algorithm that would be used instead of Data Encryption Standard (DES), it is a standard developed in seventies, later adopted by NIST in 1977.

In next nine months 15 candidates for the AES were accepted and NIST requested the assistance of the cryptographic research community in analyzing the candidates. These 15 algorithms were: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, MARS, RC6, Rijndael, SAFER+, Serpent, Twofish.

At a minimum, the algorithm for the Advanced Encryption Standard would have to implement symmetric key cryptography as a block cipher and support a block size of 128 bits and key sizes of 128, 192, and 256 bits.

After two AES Candidate Conferences that were held to discuss the results of the analysis that was conducted by the international cryptographic community NIST had chosen five finalist algorithms: MARS, RC6, Rijndael, Serpent and Twofish.

The evaluation criteria were divided into three major categories:

1. Security.
2. Cost.
3. Algorithm and Implementation Characteristics.

Security was the most important factor in the evaluation and encompassed features such as resistance of the algorithm to cryptanalysis, soundness of its mathematical basis, randomness of the algorithm output, and relative security as compared to other candidates.

Cost was a second important area of evaluation that encompassed licensing requirements, computational efficiency (speed) on various platforms, and memory requirements. Since one of NIST's goals was that the final AES algorithm be available worldwide on a royalty-free basis, public comments were specifically sought on intellectual property claims and any potential conflicts. The speed of the algorithm on a variety of platforms needed to be considered. During Round 1, the focus was primarily on the speed associated with 128-bit keys. During Round 2, hardware implementations and the speeds associated with the 192 and 256-bit key sizes were addressed. Memory requirements and software implementation constraints for software implementations of the candidates were also important considerations.

The third area of evaluation was algorithm and implementation characteristics such

as flexibility, hardware and software suitability, and algorithm simplicity. Flexibility includes the ability of an algorithm:

- To handle key and block sizes beyond the minimum that must be supported,
- To be implemented securely and efficiently in many different types of environments, and
- To be implemented as a stream cipher, hashing algorithm, and to provide additional cryptographic services.

It must be feasible to implement an algorithm in both hardware and software, and efficient firmware implementations were considered advantageous. The relative simplicity of an algorithm's design was also an evaluation factor.

During the latter discussions the question about the number of algorithms chosen for the standard was solved to be the be one. Some statements in the favor of multiple algorithms in standards were announced:

- In terms of resiliency, if one AES algorithm were broken, there would be at least one more AES algorithm available and implemented in products.
- Intellectual property concerns could surface later, calling into question the royalty-free availability of a particular algorithm. An alternative algorithm might provide an immediately available alternative that would not be affected by the envisioned concern.
- A set of AES algorithms could cover a wider range of desirable traits than a single algorithm. In particular, it might be possible to offer both high security and high efficiency to an extent not possible with a single algorithm.

However, the arguments in the favor of the only algorithm used in the standard were more convincing:

- Multiple AES algorithms would increase interoperability complexity and raise costs when multiple algorithms were implemented in products.
- Multiple algorithms could be seen as multiplying the number of potential "intellectual property attacks" against implementers.
- The specification of multiple algorithms might cause the public to question NIST's confidence in the security of any of the algorithms.
- Hardware implementers could make better use of available resources by improving the performance of a single algorithm than by including multiple algorithms.

Later, using various criteria (general security, the efficiency of software and hardware implementations, the usage of space, speed of decryption\encryption, attack vulnerability) the Rijndael was chosen.

For additional details of the selection process, see [1].

3 Mathematical preliminaries

In this section we will discuss mathematical terms and concepts used in Rijndael. The reader confident in his own mathematics knowledge can skip this section and return here, if his confidence vanishes throughout the reading. All information below can be found in any book about abstract algebra, for example [3].

Definition 3.1. *Commutative ring is a non-empty set* $\mathbb{A}$ *endowed with two binary operations:* $+$ *(addition),* $\cdot$ *(multiplication), that obey following laws:*

Ring is an abelian group with respect to addition:

1. $(a+b)+c = a+(b+c) \quad \forall a,b,c \in \mathbb{A}$—*associativity of addition.*
2. $a+b = b+a \quad \forall a,b \in \mathbb{A}$—*commutativity of addition.*
3. *There exists an identity element of* $\mathbb{A}$ *called zero* (0)*, such that addition of any element* a *with zero produces* a*:* $\forall a \in \mathbb{A} \quad a+0 = a$
4. *For any element* a *there exists inverse element:* $\forall a \in \mathbb{A}\ \exists(-a) : a+(-a) = 0$.

Ring is a monoid with respect to multiplication:

1. $(a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \forall a,b,c \in \mathbb{A}$—*associativity of addition.*
2. $a \cdot b = b \cdot a \quad \forall a,b \in \mathbb{A}$—*commutativity of multiplication.*
3. *There exists an identity element of* $\mathbb{A}$ *called unit* (1)*, such that multiplication of any element* a *with unit produces* a*:* $\forall a \in \mathbb{A} \quad a \cdot 1 = a$

Multiplication is distributive with respect to addition:

$$a \cdot (b+c) = (b+c) \cdot a = (a \cdot b) + (a \cdot c) \quad \forall a,b,c \in \mathbb{A}.$$

Binary operations

$$+ : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$$
$$\cdot : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$$

are just functions that take two elements of $\mathbb{A}$ and produce an element of $\mathbb{A}$. They are usually written in infix form $a + b$, instead of $+(a, b)$ that is used for almost all functions.

Further we will use word "ring" for the term "commutative ring". We will also omit symbol of multiplication and write ab instead of $a \cdot b$.

Examples of rings: integer numbers $\mathbb{Z}$, rational numbers $\mathbb{Q}$, real numbers $\mathbb{R}$, rings of integers modulo n: $\mathbb{Z}_n$, polynomial rings $\mathbb{K}[x]$, formal power series $\mathbb{K}[[x]]$, etc.

You should understand, that $-a$ is a whole symbol, not a result of applying "minus" operation to element a. On the contrary, the concept of subtraction is defined in terms of inverse elements. To subtract b from a means to add $(-b)$ to a: $a - b = a + (-b)$.

Definition 3.2. *An element a of ring $\mathbb{A}$ is called invertible, if it has multiplicative inverse: $\exists b \in \mathbb{A} : a \cdot b = 1$. Multiplicative inverse of a is denoted as a^{-1}.*

For example, in ring $\mathbb{Z}$ there are two invertible elements: 1 and -1.

What are the invertible elements of real numbers $\mathbb{R}$?

Definition 3.3. *Field is a commutative ring where $0 \neq 1$ and every nonzero element is invertible.*

Examples of fields: rational numbers $\mathbb{Q}$, real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$. Integer numbers do not form a field.

Roughly speaking, elements of fields behave like usual numbers, like rationals or reals. For example these facts are true:

1. $a \cdot 0 = 0$ for any element a.
2. Fields have no zero divisors: from $ab = 0$ it follows, that $a = 0$ or $b = 0$.

Theorem 3.1. *Let a, b be elements of a field $\mathbb{F}$. If $ab = 0$, then $a = 0$ or $b = 0$ (or both).*

Proof. If both a and b are not invertible, then both of them are equal to zero. Suppose one of them is invertible, for example a. Then multiply identity $ab = 0$ by a^{-1}:

$$\begin{aligned} a^{-1} \cdot ab &= a^{-1} \cdot 0 \\ (a^{-1}a)b &= 0 \\ 1 \cdot b &= 0 \\ b &= 0 \end{aligned}$$

□

Exercise 1

Prove the first statement from above.

Answer of exercise 1

Using properties we get: $0a = (0 + 0)a = 0a + 0a$, therefore $0a - 0a = 0a$, that is equivalent to $0a = 0$.

Fields are distinguished by the number of elements, if a field has finite number of elements it's called finite or Galois[1] field, if a field has infinite number of elements then it's called (surprisingly) an infinite field. We are going to work only with finite fields.

There are fields, that behave in not so usual way, for example, $1 + 1 + 1 = 0$ in a field $\mathbb{GF}(3)$.

Definition 3.4. *Characteristic of a field is the smallest positive number n, such that*

$$\underbrace{1 + 1 + \ldots 1}_{n \text{ times}} = 0.$$

If there is no such number n, then characteristic is defined to be zero.

Theorem 3.2. *If a field has finite characteristic n, then n is necessarily a prime number.*

You can try to prove that theorem yourself.

Any finite field has number of elements equal to p^n, where p is a prime number and n is some integer greater than zero. Such field is denoted as $\mathbb{GF}(p^n)$. That field has

[1] Évariste Galois, 1811—1832, French mathematician.

characteristic p. Finite fields with the same number of elements are isomorphic, they form the same algebraic structure, differing only in their representation.

You should be familiar with modular arithmetic of $\mathbb{Z}_n$. Elements of that ring are "remainders"[2] of division of integers by n. If n is prime, then it's a field that is interchangeably denoted as $\mathbb{Z}_n$, $\mathbb{GF}(n)$, $\mathbb{F}_n$.

You can define a structure of a field on some set $\mathbb{A}$ by specifying its multiplication and addition tables, for example $\mathbb{GF}(3)$ has three elements we denote by $a,\ b,\ c$:

$+$	a	b	c
a	a	b	c
b	b	c	a
c	c	a	b

$\cdot$	a	b	c
a	a	a	a
b	a	b	c
c	a	c	b

You can ensure, that the field laws are satisfied, so $\mathbb{A}$ with these operations is indeed a field. As it's been already said, that field is isomorphic to any other field with 3 elements. Actually instead of symbols $a,\ b,\ c$ we could use $0, 1, 2$ and $\mathbb{GF}(3)$ is just a field $\mathbb{Z}_3$ of integers modulo 3 where the tables look like:

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Definition 3.5. *Polynomial over field $\mathbb{F}$ is an expression $b_0 + b_1x + b_2x^2 + \cdots + b_kx^k$, where all coefficients $b_i \in \mathbb{F}$. Addition and multiplication of polynomials are defined as usual, you expand the parentheses and reduce the common terms.*

For instance, $(2x + 3x^2) \cdot (1 + x) = 2x + 3x^2 + 2x^2 + 3x^3 = 2x + 5x^2 + 3x^3$.

Theorem 3.3. *Polynomials over a field form a ring.*

The degree of a polynomial is the highest degree of its nonzero terms. The degree of $x^2 + x^3$ is 3, the degree of $3 + 0x^2$ is 0. The degree of a polynomial f is denoted as $\deg f$

[2]To be precise, that ring is not a ring of remainders, rather a ring of classes of equivalence of remainders, that's why this term will be used in quotes.

Just like as integers, you can define modulo operation on polynomial ring. For two polynomials f and g there exist unique polynomials q and r, such that:

$$f = gq + r,$$

where $\deg r < \deg g$. Polynomial q is called quotient and r is called remainder.

The same stands for modulo ring, we can define a ring of "remainders" of division by a given polynomial f (it's also called reduction polynomial). That ring consists of polynomials with degree less than degree of f. With an accurate choice of f this ring becomes a field.

Definition 3.6. *Polynomial f over field $\mathbb{F}$ is said to be irreducible, if there are no polynomials g, h over F with degrees greater than zero, such that $f = gh$.*

For example, $x^2 + 1$ is irreducible over $\mathbb{R}$, but it's reducible over $\mathbb{C}$, as we can decompose it as $x^2 + 1 = (x + i)(x - i)$.

Theorem 3.4. *If f is an irreducible polynomial, then the ring of "remainders" of division by f is a field.*

For a finite field $\mathbb{GF}(p^n)$, field of polynomials modulo some polynomial f is considered to be a canonical representation of that field.

Let's consider field $\mathbb{GF}(2^2)$ as a field of polynomials over $\mathbb{GF}(2)$ modulo $f = x^2+x+1$. Writing out all polynomials from that field: $0, 1, x, x + 1$. Tables for that field:

+	0	1	x	x+1
0	0	1	x	x+1
1	1	0	x+1	x
x	x	x+1	0	1
$x+1$	x+1	x	1	0

·	0	1	x	x+1
0	0	0	0	0
1	0	1	x	x+1
x	0	x	x+1	1
$x+1$	0	x+1	1	x

For instance, $(x + 1) \cdot (x + 1) = x^2 + 2x + 1 = x^2 + 1$. (Don't forget, coefficients are from $\mathbb{GF}(2)$, where 2=0). And $x^2 + 1$ equals x modulo $x^2 + x + 1$.

Further we will need to consider polynomials representing field $\mathbb{GF}(2^8)$ modulo $x^8 + x^4+x^3+x+1$. They are just polynomials of degree not greater than 7 with coefficients either 1 or 0.

Exercise 2

What is the inverse of x in $\mathbb{GF}(2^8)$ with reducing polynomial $x^8 + x^4 + x^3 + x + 1$?

Answer of exercise 2

$x^7 + x^3 + x^2 + 1$.

Byte is a collection of 8 bits, therefore every byte represents some polynomial from $\mathbb{GF}(2^8)$. For example, 10011111 represents $x^7 + x^4 + x^3 + x^2 + x + 1$. Note that sum of two polynomials is equal to the XOR (exclusive bitwise "or" operation) of bytes they are represented by. Usually we will denote XOR operation by $\oplus$, or just by ordinary $+$.

4 Description of Rijndael

4.1 Preface

In this section we will describe the specification of Rijndael and implement it in Wolfram Mathematica language.

Mathematica is a high-level language that definitely wasn't created for low-level operations, but Rijndael description mostly uses them. Though, we will encounter some advantages of Mathematica and its built in packages.

Anyway, code from this paper should be not considered as efficient implementation of Rijndael, rather as educational one. Don't copy verbatim code from the code snippets, as some characters from PDF file couldn't be copied right to the Wolfram notebook. We also will use `CamelCase` titles for functions. We will use decimal representation of bytes, for example integer 255 represents byte 11111111_2.

4.2 Rijndael

Rijndael is a key-iterated block cipher: the input for it is divided into blocks of predefined size. To obtain ciphertext an iterative application of transformation (called round) to the block is used.

Unencrypted data is called *plaintext* and the encrypted data is called *ciphertext*. Of course, data needn't to be just text, it could be anything translatable into series of bytes.

The input and output data of Rijndael are one-dimensional arrays of bytes. For encryption the input is a plaintext block and a master key (or cipher key, or just

key, if there is no confusion with "round key"), the output is a ciphertext block. For decryption, the input is a ciphertext block and a master key (decryption key), the output is a plaintext block.

Depending on the key length, number of rounds can be 10 for 128-bit key, 12 for 196-bit key and 14 for 256-bit key. The block length adopted by Advanced Encryption Standard is always 128 bits long, however, in original specification of Rijndael algorithm block size could vary.

4.3 Encryption

Core steps for encryption can be written as:

1. Obtaining the state
2. Key schedule
3. Initial round
   3.1. `AddRoundKey`(state, round key)
4. Ordinary rounds
   4.1. `SubBytes`(state)
   4.2. `ShiftRows`(state)
   4.3. `MixColumns`(state)
   4.4. `AddRoundKey`(state, round key)
5. Final round
   5.1. `SubBytes`(state)
   5.2. `ShiftRows`(state)
   5.3. `AddRoundKey`(state, round key)

We will discuss steps of the algorithm not in the order shown above.

As you could notice, the only difference between "ordinary" round and the final round is the absence of `MixColumn` step.

4.4 Obtaining the state

Rijndael operations are defined to act on a "state" which is 4 by 4 array of bytes.

Let's denote 16 bytes of the plaintext by $b_1 b_2 \dots b_{16}$. These bytes arranged into a matrix (further the word matrix will be used to denote a two-dimensional array) in a following way:

$$A = \begin{bmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ b_4 & b_8 & b_{12} & b_{16} \end{bmatrix}.$$

So, if we as usual denote by a_{ij} an element of A in ith row and jth column, the rule for making matrix from bytes can be written as

$$a_{ij} = b_{i+4j}, \text{ where } 1 \leq i \leq 4 \text{ and } 0 \leq j \leq 3$$

with enumeration of columns and rows beginning from one.

The following Wolfram Mathematica code arranges first 16 elements of a list into a matrix in the needed way.

```
State[list_] := Table[list[[i + 4*j]], {i, 1, 4}, {j, 0, 3}]
```

After encryption is done, i.e. the final round was performed and we got the resulting state a, the cyphertext is constructed from the state by taking bytes in the same order:

$$c_i = a_{i \bmod 4,\ i/4}.$$

4.5 Ordinary rounds

4.5.1 SubBytes

The `SubBytes` step is the only non-linear transformation of the state. For `SubBytes` action we need to treat every byte of the state as a polynomial representation of the finite field GF(2^8) element with $x^8 + x^4 + x^3 + x + 1$ polynomial used for modulo reduction.

The action of `SubBytes` consists of taking a multiplicative inverse of a polynomial (with zero mapped to the zero) and an affine transformation [3].

So, we take a byte b, treat it as polynomial, find its inverse $c = b^{-1}$ with coefficients $c_0, c_1, \ldots, c_7$, afterwards affine transformation is used as follows:

$$\texttt{SubBytes}(b) = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Wolfram Mathematica has a package used for computations over finite fields, however, it doesn't have the inverse function, so we will find multiplicative inverses by extended Euclidean algorithm.

Wolfram code:

```
InversePolynomial[p_] :=
 If[p == 0, 0,
  PolynomialExtendedGCD[x^8 + x^4 + x^3 + x + 1, p, x, Modulus > 2][[
   2, 2]]]
```

This function computes the multiplicative inverse of the polynomial $p(x)$ modulo chosen polynomial $x^8 + x^4 + x^3 + x + 1$. For instance, the inverse of the polynomial $x^4 + x + 1$ is the polynomial $x^6 + x^3 + x + 1$. One can ensure by multiplying:

$$\begin{aligned}(x^4 + x + 1) \cdot (x^6 + x^3 + x + 1) &= 1 + 2x + x^2 + x^3 + 2x^4 + x^5 + x^6 + 2x^7 + x^{10} = \\ &= 1 + x^2 + x^3 + x^5 + x^6 + x^{10} = 1 \pmod{x^8 + x^4 + x^3 + x + 1}.\end{aligned}$$

List of coefficients of a polynomial can be obtained by using **CoefficientList**.

Note, that **InversePolynomial** function expects the input to be a polynomial of variable x, not a list of coefficients.

[3] Affine transformation F of a vector v (in our case it's just polynomial over finite field) is a composition of a linear transformation with an addition of some vector b: $F(v) = A(v) + b$. In our case that means multiplication of the column-vector v by some matrix A and summing the result with some vector b.

Exercise 1

Create function **PolynomialFromByte** that takes a byte and returns a polynomial representing it.

Answer of exercise 1

```
PolynomialFromByte[byte_] :=
  x^Range[0,7].Reverse[PadLeft[IntegerDigits[byte,2],8]]
```

So, `SubBytes` function looks as follows:

$$\text{SubBytes}[p\_]:=\textbf{Part}[\textbf{FromDigits}[\textbf{Mod}[\begin{pmatrix}1&1&1&1&1&0&0&0\\0&1&1&1&1&1&0&0\\0&0&1&1&1&1&1&0\\0&0&0&1&1&1&1&1\\1&0&0&0&1&1&1&1\\1&1&0&0&0&1&1&1\\1&1&1&0&0&0&1&1\\1&1&1&1&0&0&0&1\end{pmatrix}.\textbf{Reverse}[\textbf{CoefficientList}[\text{InversePolynomial}[p],x,8]]+\begin{pmatrix}0\\1\\1\\0\\0\\0\\1\\1\end{pmatrix},2],2],1]$$

Composing **SubBytes** with **PolynomialFromByte** will give us a function that takes byte and produces its inverse.
For instance, SubBytes[PolynomialFromByte[3]] produces 123.

In further text function **SubBytes** is exactly the composition from above, so it takes bytes and returns bytes.

Instead of writing the inverse and affine transformation functions for every implementation, a precomputed substitution table (called S-box) can be made for all bytes.

Exercise 2

Create such a table.

Answer of exercise 2

```
Table[SubBytes[PolynomialFromByte[i+j]],{i,0,15},{j,0,15}]
```

Usually that table is represented in hexadecimal and it looks like:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

4.5.2 ShiftRows

At the step `ShiftRows` every row of the state is cyclically shifted to the left. The ith row is shifted $i-1$ positions to the left.

Schematically the transformation could be viewed as:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \mapsto \begin{bmatrix} a & b & c & d \\ f & g & h & e \\ k & l & i & j \\ p & m & n & o \end{bmatrix}.$$

Exercise 3

Implement **ShiftRows** function in Wolfram Mathematica.

Answer of exercise 3

```
ShiftRows[state_] := Module[{m = state},
  m[[2]] = RotateLeft[state[[2]],1];
  m[[3]] = RotateLeft[state[[3]],2];
  m[[4]] = RotateLeft[state[[4]],3]; m]
```

4.5.3 MixColumns

The `MixColumns` step operates on the columns of the state. The column consisting of four bytes is treated as a polynomial over $\mathrm{GF}(2^8)$[4]. Polynomial-column is multiplied by a fixed polynomial $c(x) = 0\text{x}03 \cdot x^3 + x^2 + x + 0\text{x}02$. Here the coefficients 0x03 and 0x02 are treated as elements of $\mathrm{GF}(2^8)$. Multiplication is reduced modulo $x^4 + 1$.

Polynomial $c(x)$ is coprime to x^4+1, therefore it's invertible. It ensures that `MixColumns` operation is invertible and ciphertext can be decrypted in the only right way.

Multiplication can be written as matrix multiplication. Let's denote

$$b(x) = c(x) \cdot a(x) \mod (x^4 + 1),$$

then

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0\text{x}02 & 0\text{x}03 & 0\text{x}01 & 0\text{x}01 \\ 0\text{x}01 & 0\text{x}02 & 0\text{x}03 & 0\text{x}01 \\ 0\text{x}01 & 0\text{x}01 & 0\text{x}02 & 0\text{x}03 \\ 0\text{x}03 & 0\text{x}01 & 0\text{x}01 & 0\text{x}02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

For this step Mathematica's package "FiniteFields" fits perfectly. An element of $\mathrm{GF}(2^8)$, for example the polynomial $x^5 + x^3 + x$ in this package is defined as

```
GF[2,{1,1,0,1,1,0,0,0,1}][{0,1,0,1,0,1,0,0}].
```

Where 2 denotes the characteristic of the field, the first list {1,1,0,1,1,0,0,0,1} denotes the coefficients of the reducing polynomial $1+x+x^3+x^4+x^8$ and {0,1,0,1,0,1,0,0} denotes the list of coefficients of considered polynomial.

So, for implementing this step we need to translate hexadecimal values 0x02, 0x03, 0x01 to Mathematica's finite field elements and do the same with values a_0, a_1, a_2, a_3. After such a translation ordinary matrix multiplication would produce the needed result.

[4]Don't mix up with polynomials over GF(2), coefficients of a polynomial over $\mathrm{GF}(2^8)$ can be themselves treated as polynomials over GF(2).

Exercise 4

Implement **ShiftRows** function.
Hint: try to use **ToElementCode** and **FromElementCode** functions from package FiniteFields.

Answer of exercise 4

```
SetFieldFormat[GF[2,{1,1,0,1,1,0,0,0,1}], FormatType > FullForm];
one = GF[2,{1,1,0,1,1,0,0,0,1}][{1}];
two = GF[2,{1,1,0,1,1,0,0,0,1}][{0,1}];
three = GF[2,{1,1,0,1,1,0,0,0,1}][{1,1}];
MixColumn[column_] := ( {
    {two, three, one, one},
    {one, two, three, one},
    {one, one, two, three},
    {three, one, one, two}})
    .(FromElementCode[GF[2, {1,1,0,1,1,0,0,0,1}],#]&/@column)
MixColumns[state_] := Module[{m = state},
  m[[All, 1]] = ToElementCode[#] & /@ MixColumn[state[[All, 1]]];
  m[[All, 2]] = ToElementCode[#] & /@ MixColumn[state[[All, 2]]];
  m[[All, 3]] = ToElementCode[#] & /@ MixColumn[state[[All, 3]]];
  m[[All, 4]] = ToElementCode[#] & /@ MixColumn[state[[All, 4]]]; m]
```

Exercise 5

Ensure that multiplying of a polynomials modulo $x^4 + 1$ can be written as a matrix multiplication.
Hint: expand $(c_0 + c_1x + c_2x^2 + c_3x^3) \cdot (a_0 + a_1x + a_2x^2 + a_3x^3)$ and reduce the result modulo $x^4 + 1$. Try to emulate the result by matrix multiplication.

Answer of exercise 5

Expanding gives us following:

$$\begin{aligned}&(c_0 + c_1x + c_2x^2 + c_3x^3) \cdot (a_0 + a_1x + a_2x^2 + a_3x^3) = \\ &= a_3c_3x^6 + x^5(a_3c_2 + a_2c_3) + x^4(a_3c_1 + a_2c_2 + a_1c_3) + x^3(a_3c_0 + a_2c_1 + a_1c_2 + a_0c_3) + \\ &\qquad + x^2(a_2c_0 + a_1c_1 + a_0c_2) + x(a_1c_0 + a_0c_1) + a_0c_0.\end{aligned}$$

Modulo $x^4 + 1$ it's the same as:

$$\begin{aligned}&x^3(a_3c_0 + a_2c_1 + a_1c_2 + a_0c_3) + x^2(a_2c_0 + a_1c_1 + a_0c_2 - a_3c_3) + \\ &\qquad + x(a_1c_0 + a_0c_1 - a_3c_2 - a_2c_3) + a_0c_0 - a_3c_1 - a_2c_2 - a_1c_3.\end{aligned}$$

Don't forget, that in the field of characteristic 2 addition and subtraction are the same operations. Finally we get:

$$x^3\left(a_3c_0+a_2c_1+a_1c_2+a_0c_3\right)+x^2\left(a_2c_0+a_1c_1+a_0c_2+a_3c_3\right)+ \\ +x\left(a_1c_0+a_0c_1+a_3c_2+a_2c_3\right)+a_0c_0+a_3c_1+a_2c_2+a_1c_3.$$

On the other hand, let's compute the result of matrix multiplication:

$$\begin{bmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} a_0c_0+a_3c_1+a_2c_2+a_1c_3 \\ a_1c_0+a_0c_1+a_3c_2+a_2c_3 \\ a_2c_0+a_1c_1+a_0c_2+a_3c_3 \\ a_3c_0+a_2c_1+a_1c_2+a_0c_3 \end{bmatrix}.$$

You can observe, that coefficients of the resulting polynomial are exactly the elements of the result of matrix multiplication.

4.5.4 AddRoundKey

Perhaps, this step is the simplest one. For every round there is a round key obtained from the master key during key schedule process that would be described in next section.

For now, assume we got 128-bit round key, the action of `AddRoundKey` is just the state combined with round key using bitwise XOR operation. If we have the state as matrix B with entries b_{ij} and the round key is matrix R with entries r_{ij}, the resulting state is matrix A with entries $a_{ij} = b_{ij} \oplus r_{ij}$, where $\oplus$ denotes XOR operation.

$$\mathbf{AddRoundKey}(B,R) = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \oplus \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{41} & r_{42} & r_{43} & r_{44} \end{bmatrix} = B \oplus R.$$

In Wolfram language that operation is named **BitXor**.

4.6 Key schedule

For simplicity, we will describe the case of 128-bit key (so the number of rounds is 10), the other cases are similar to that one with slight changes. For the encryption

of one block it's needed to have 11 round keys: 1 for initial round, 9 for "ordinary" rounds and 1 for the final round. Each round key has length of 128 bits.

The master key bytes $z_1 z_2 \dots z_{16}$ should be arranged into a square matrix in the same way as bytes of the plaintext form the initial state:

$$Z = \begin{bmatrix} z_1 & z_5 & z_9 & z_{13} \\ z_2 & z_6 & z_{10} & z_{14} \\ z_3 & z_7 & z_{11} & z_{15} \\ z_4 & z_8 & z_{12} & z_{16} \end{bmatrix}.$$

During key schedule additional columns would be added to that matrix, so the resulting matrix has 44 columns. Columns to be added are recursively defined in terms of previously defined columns. Each column depends on the previous column, on the column 4 positions earlier and on the round constants. We will denote ith column as c_i and start counting columns from zero.

Column c_i with number not multiple of 4 is obtained by XOR'ing columns c_{i-1} and c_{i-4}. For example,

$$c_9 = c_8 \oplus c_5.$$

Column c_{4j} with number being the multiple of 4 is obtained by XOR'ing the column $c_{4(j-1)}$ and the modified previous column $f(c_{4j-1})$. Function f at first modifies all bytes of the column by `SubBytes` function:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \mapsto \begin{bmatrix} \texttt{SubBytes}(a) \\ \texttt{SubBytes}(b) \\ \texttt{SubBytes}(c) \\ \texttt{SubBytes}(d) \end{bmatrix}.$$

Afterwards the column is cyclically rotated (of course, vertically) by one position down:

$$\begin{bmatrix} \texttt{SubBytes}(a) \\ \texttt{SubBytes}(b) \\ \texttt{SubBytes}(c) \\ \texttt{SubBytes}(d) \end{bmatrix} \mapsto \begin{bmatrix} \texttt{SubBytes}(b) \\ \texttt{SubBytes}(c) \\ \texttt{SubBytes}(d) \\ \texttt{SubBytes}(a) \end{bmatrix}.$$

After rotation the resulting column is XOR'ed with round constant column (RCC),

its entries are:

$$\begin{bmatrix} \mathrm{RC}(j) \\ \text{0x00} \\ \text{0x00} \\ \text{0x00} \end{bmatrix}.$$

So, the first byte is the only non-zero one in the column. The value RC(j) is defined recursively:

$$\begin{aligned} \mathrm{RC}(1) &= \text{0x01} \\ \mathrm{RC}(2) &= \text{0x02} \\ \mathrm{RC}(j) &= \text{0x02} \cdot \mathrm{RC}(j-1). \end{aligned}$$

Summing up all the modifications we can write:

$$c_{4j} = c_{4(j-1)} \oplus \texttt{Rotate}(\texttt{SubBytes}(c_{4j-1})) \oplus \mathrm{RCC}(j).$$

The round key for the ith round is obtained by taking $c_{4i}, c_{4i+1}, c_{4i+2}, c_{4i+3}$ columns. The index of the initial round is 0, the index of a final round is 10.

Let's create the function that makes the expanded keys consisting of 44 columns. We will need a function to insert columns to a matrix:

```
InsertColumn[matrix_, position_, column_] :=
 Transpose[Insert[Transpose[matrix], column, position]]
```

Next we need RCC function.

Exercise 6

Create RCC function.

Answer of exercise 6

```
RC[i_] :=
 If[i == 1, GF[2,{1,1,0,1,1,0,0,0,1}][{1}],
  If[i == 2, GF[2,{1,1,0,1,1,0,0,0,1}][{0,1}], RC[2]*RC[i-1]]]
RCC[i_] := {ToElementCode[RC[i]], 0, 0, 0}
```

After that is done, we will create function **ModifyKeystate** to insert a single column into expanded key. It may be done as this:

```
ModifyKeystate[keystate_, i_] :=
 InsertColumn[keystate, i,
  If[Mod[i-1,4] != 0,
   BitXor[keystate[[All, i-1]], keystate[[All, i-4]]],
   Flatten[
    BitXor[keystate[[All, i-4]],
     RotateLeft[
      Map[SubBytes, Map[PolynomialFromByte, keystate[[All, i-1]]]]],
      RCC[Quotient[i-1, 4]]]]]]
```

Finally, we get **ExpandKey** function:

```
ExpandKey[keystate_]:=Fold[ModifyKeystate, keystate, Range[5, 44]]
```

Round keys are successively taken 4 columns of the expanded key.

4.7 Final round

As it was stated earlier, the only difference between final and "ordinary" round is the absence of `MixColumn` step.

4.8 Encryption

Now we got all prerequisites for the encryption. Let's encrypt one of the test values provided by authors of algorithm.

- Key (in hexadecimal): "ffffffffffffffffffffffffffffffff".
- Data to encrypt: "00000000000000000000000000000000"

Exercise 7

Encrypt data with given key.

Answer of exercise 7

At first, let's define some helper functions:

```
DoRound[state_, roundkey_] :=
 BitXor[MixColumns[ShiftRows[Map[SubBytes, state, {2}]]], roundkey]
```

```
FinalRound[state_,roundkey_] :=
 BitXor[ShiftRows[Map[SubBytes,state,{2}]],roundkey]
```

Input data and key:

```
key = Interpreter["HexInteger"] /@ Table["ff", {i, 16}];
data = Interpreter["HexInteger"] /@ Table["00", {i, 16}];
keystate = State[key];
datastate = State[data];
```

The process of encryption:

```
expanded = ExpandKey[keystate];
initstate = BitXor[datastate, keystate];
state = Fold[DoRound, initstate,
   Table[expanded[[All,4*i+1;;4*i+4]],{i,1,9}]];
finalkey =
  Table[expanded[[All,4*i+1;;4*i+4]],{i,0,10}][[11]];
  BaseForm[FinalRound[state, finalkey] // MatrixForm, 16]
```

That gives us output (displayed in decimal):

$$\begin{pmatrix} 161 & 135 & 137 & 56 \\ 246 & 125 & 100 & 191 \\ 37 & 95 & 72 & 201 \\ 140 & 205 & 69 & 44 \end{pmatrix}.$$

It's exactly what we wanted to get!

4.9 Decryption

A short overview of decryption process. Decryption is done by similar steps in reverse order:

1. Key schedule
2. Inverse final round
   - 2.1. `InvAddRoundKey`(state, round key)
   - 2.2. `InvShiftRows`(state)
   - 2.3. `InvSubBytes`(state)

3. Inverse ordinary rounds
    3.1. `InvAddRoundKey`(state, round key)
    3.2. `InvMixColumns`(state)
    3.3. `InvShiftRows`(state)
    3.4. `InvSubBytes`(state)
4. Inverse initial round
    4.1. `InvAddRoundKey`(state, round key)

4.9.1 Key schedule

As the algorithm is symmetric, same round keys are used for decryption, just in reversed order. That means, inverse initial round that is done last uses the master key.

4.9.2 InvAddRoundKey

This step is identical to `AddRoundKey`, as XOR operation is inverse to itself.

4.9.3 InvShiftRows

At that step rows shifting is reversed:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \mapsto \begin{bmatrix} a & b & c & d \\ h & e & f & g \\ k & l & i & j \\ n & o & p & m \end{bmatrix}.$$

4.9.4 InvSubBytes

For this step we need to do the inverse affine transformation and to take a multiplicative inverse. Usually it's done by using precomputed inverse S-Box:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1x	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2x	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3x	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4x	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5x	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6x	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7x	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8x	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9x	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
ax	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
bx	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
cx	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
dx	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
ex	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
fx	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

4.9.5 InvMixColumns

This step is done just like the direct `MixColumns` with the multiplication matrix changed to this:

$$\begin{bmatrix} 0x0e & 0x0b & 0x0d & 0x09 \\ 0x09 & 0x0e & 0x0b & 0x0d \\ 0x0d & 0x09 & 0x0e & 0x0b \\ 0x0b & 0x0d & 0x09 & 0x0e \end{bmatrix}.$$

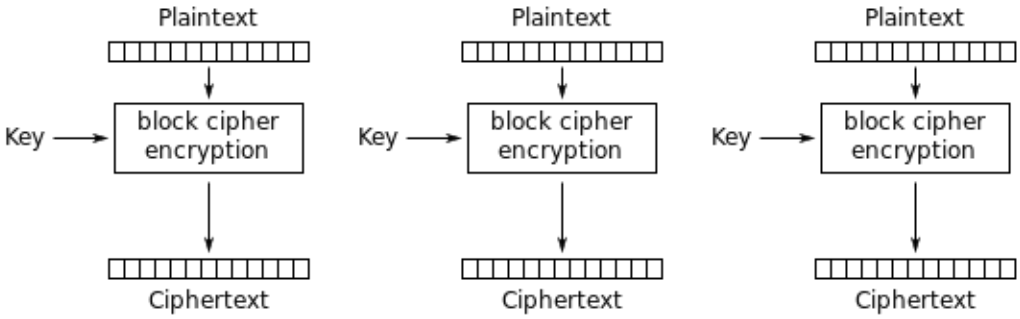
5 Security

5.1 Modes of block ciphers.

Block ciphers are defined to encipher blocks, but usually data the the user wants to encrypt is not equal to the size of the block, so there are different methods of using block ciphers to encrypt data that is longer than one block. These methods are called modes of operations. We will have a short overview on two of them here.

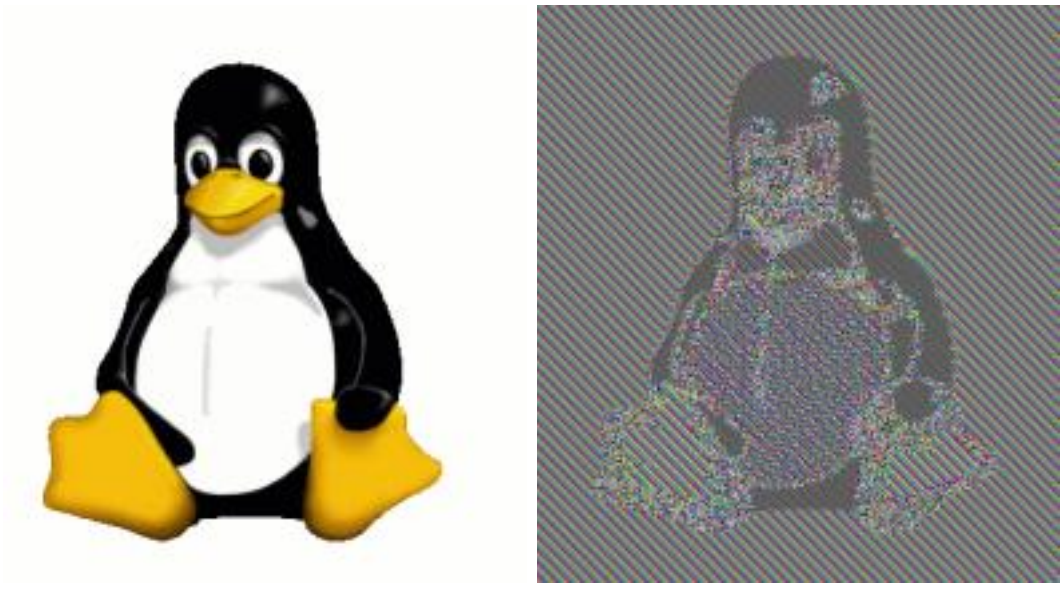
5.1.1 Electronic Codebook (ECB)

It's the simplest one, the data is divided into blocks and each of them is encrypted with the same key.

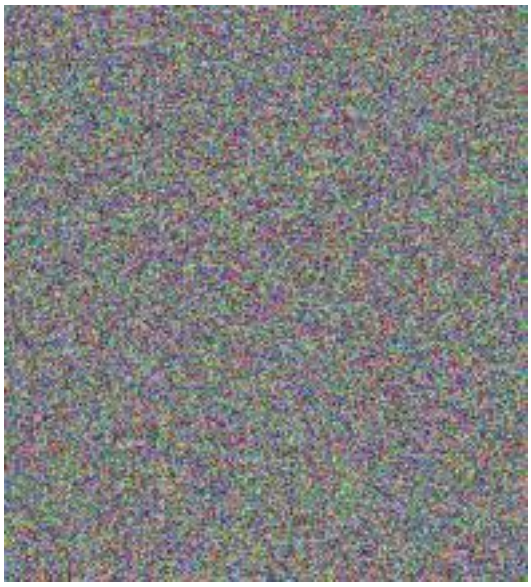


Electronic Codebook (ECB) mode encryption

This mode could not be considered secure, as after encryption blocks of the same plaintext have the same ciphertext. To illustrate this idea, one can encrypt bytes of the image:

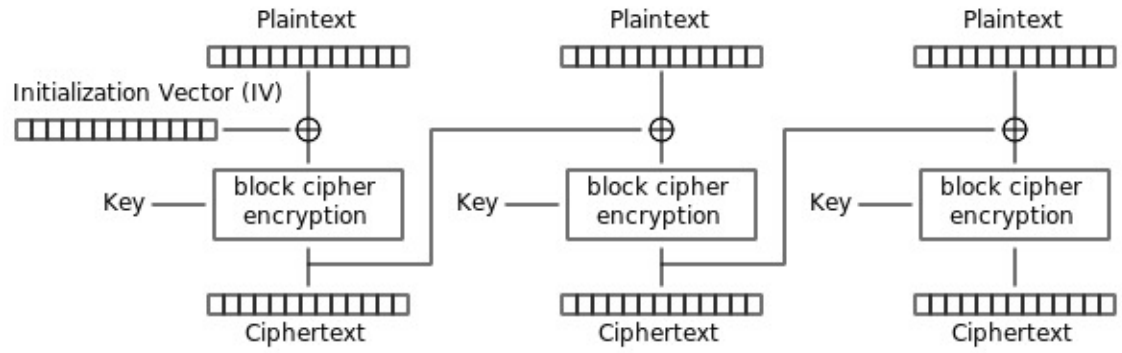


Despite you can't decipher a single block to obtain a plaintext, you can see the pattern of blocks of the original plaintext. Compare it to the same image encrypted using another mode (CBC):



5.1.2 Cipher Block Chaining (CBC)

In this mode each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point.



Cipher Block Chaining (CBC) mode encryption

This mode is the most common one used for encryption. It's slower than ECB and could not be done in parallel.

With these words been said about modes, it's clear, that security of ciphertext is not only about the security of the used algorithm, but of course, at first attention is paid

to it.

5.2 Security of Rijndael

Cryptographic attacks are made to have some sort of solution for obtaining plaintext having only ciphertext. A cipher is considered "broken" if there is a method that is sufficiently faster than brute force.

Brute force is a method of successively checking all possible variants for the plaintext. In AES key length is 128 bits, that means there are 2^{128} ($\approx 10^{38}$) different keys. Modern supercomputers can check approximately 10^{18} blocks per second, so breaking a single block with by brute force will consume more time than the age of the universe ($\approx 10^{17}$ seconds).

There are more sophisticated methods of breaking ciphers. Two of them will be shortly discussed here. Both of them could not be used for AES (and not any other publicly known method).

5.2.1 Linear cryptoanalysis

It was discovered by Mitsuru Matsui in 1992 and was aimed to crack DES and FEAL algorithms.

The idea consists of two steps, first one is to construct linear equations involving plaintext, key and ciphertext. These equations should have high probability of being true.

The second step is about using this equations along with known plaintext-ciphertext pairs of to find the key.

A condition for applying linear cryptanalysis to a block cipher is to find "effective" linear expressions. Let $A(i_1, i_2, \ldots, i_a)$ be the bitwise sum of the bits of A with indices in a selection pattern $i_1, i_2, \ldots, i_a$:

$$A(i_1, i_2, \ldots, i_a) = A(i_1) \oplus A(i_2) \oplus \ldots \oplus A(i_a)$$

Let P, C and K denote the plaintext, the ciphertext and the key, respectively.

The aim is to find linear experssions of the following type:

$$P(i_1, i_2, \ldots, i_a) \oplus C(j_1, j_2, \ldots, j_a) = K(k_1, k_2, \ldots, k_a)$$

with indices i, j, k being fixed bit locations.

The effectiveness, or deviation, of such a linear expression in linear cryptoanalysis is given by $|p - 1/2|$ where p is the probability that the expression holds. By checking the value of the left-hand side of for a large number of plaintext-ciphertext pairs, the right-hand side can be guessed by taking the value that occurs most often. In principle, this gives a single bit of information about the key. It's known that the probability of making a wrong guess is very small if the number of plaintext-ciphertext pairs is larger than $|p - \frac{1}{2}|^{-2}$.

As the only non-linear step of AES is the use of S-box, analysis is concentrated on it.

Further information can be found in [4].

5.2.2 Differential cryptoanalysis

Differential cryptanalysis is a chosen-plaintext (difference) attack in which a large number of plaintext-ciphertext pairs are used to determine the value of key bits. Usually difference of plaintext is defined as result A' of XOR. After encrypting plaintext with known difference, the attacker then computes the differences of the corresponding ciphertexts, hoping to detect statistical patterns in their distribution. The work factor of the attack depends critically on the largest probability Prob($B'|A'$) with B' being a difference at some fixed intermediate stage of the block cipher, e.g. at the input of the last round.

In the basic form of the attack, key information is extracted from the output pairs in the following way. For each pair it is assumed that the intermediate difference is equal to B'. The absolute values of the output pair and the (assumed) intermediate difference B' impose restrictions upon a number l of key bits of the last round key. A pair is said to suggest the subkey values that are compatible with these restrictions. While for some pairs many keys are suggested, no keys are found for other pairs , implying that the output values are incompatible with B'. For each suggested subkey value, a corresponding entry in a frequency table is incremented.

The attack is successful if the correct value of the subkey is suggested significantly more often than any other value.

Further reading: [5].

References

[1] *Report on the Development of the Advanced Encryption Standard* `http://csrc.nist.gov/archive/aes/`

[2] Joan Daemen and Vincent Rijmen. *The Design of Rijndael.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[3] Serge Lang. *Algebra.* Springer-Verlag, New York, 2002.

[4] Mitsuru Matsui. *Linear Cryptanalysis Method for DES Cipher.* In Workshop on the theory and application of cryptographic techniques on Advances in cryptology (EUROCRYPT '93), Tor Helleseth (Ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 386-397.

[5] Biham, E. and A. Shamir. *Differential Cryptanalysis of DES-like Cryptosystems. Advances in Cryptology — CRYPTO'90.* Springer-Verlag.