An Advanced Encryption Standard implementation written in The Ruby Programming Language

Jurriaan Pruis

June 9, 2015

Abstract

This article involves the implementation of the 128 bit key/block length Advanced Encryption Standard (Rijndael). The purpose of this implementation was to get a better understanding of encryption and codes.

Introduction

The Advanced Encryption Standard is based on the Rijndael cipher developed by Joan Daemen and Vincent Rijmen. It's currently being used in almost every bit of technology you can find. The AES standard supports different key lengths, but the current implementation only deals with a 16 byte (128 bit) block and key length.

The AES consists of a number of operations that are done on different levels. These levels are: Bytes, Words and States. In this article we are going to start at the lowest level, the byte level.

Bytes

The space of all possible bytes is defined as $\mathbb{F}_2^8$. To make life easier, most of the operations on the bytes are defined as table lookups.

Addition and subtraction

Addition and subtraction in $\mathbb{F}_2^8$ is implemented as a bitwise exclusive or operation:

Listing 1: Addition and subtraction

```
class Byte
  def initialize(value)
    @value = value.to_i & 0xFF # Force byte to be a byte
  end

  # Addition is the same as XORing the values
  def +(other)
    other = Byte.new(other)
    Byte.new(to_i ^ other.to_i)
  end

  # Addition is the same as subtraction
  alias_method :-, :+

  def to_i
    @value
  end
end
```

The λ affine map

For the affine map the space of all bytes is identified with $R_8 := \mathbb{F}/(x^8+1)$. This means that multiplying with x is equivalent to a left rotate of all bits in $\mathbb{F}_2^8$.

The map λ is defined as:

$$\lambda : \mathbb{F}_2^8 \to \mathbb{F}_2^8, f \mapsto (x^4 + x^4 + x^2 + x + 1)f + x^6 + x^5 + x + 1 \bmod (x^8 + 1)$$

The Ruby implementation is as follows:

Listing 2: λ affine map in Ruby

```
s = x = byte # 1·byte
4.times do
  s = ((s << 1) | (s >> 7)) & 0xFF # Rotate byte (s = s·x)
  x ^= s # x = x + s
end
x ^ 0x63 # x^6 + x^5 + x + 1
```

Multiplication

While the affine map is computed mod (x^8+1) (like most cyclic codes of size 8) all other operations will be done mod $m = x^8 + x^4 + x^3 + x + 1$. The space of all bytes $\mathbb{F}_2^8$ is then identified with the field $\mathbb{F}_{256} = \mathbb{F}_2[x]/m$.

In this implementation logarithm tables are used to do multiplication and division in mod m, so that it is possible to use table lookups to do multiplications.

$$b_0 \cdot b_1 = \log^{-1}(\log b_0 + \log b_1)$$

With some numbers in the $\mathbb{F}_{256}$ field it is possible to traverse all possible values in this field by using exponentiation. One of these numbers is $x+1$.

Multiplication with $x+1$ is done in the implementation by doing a left bit shift and adding the original value to it $(b*(x+1) = b*x+b)$:

Listing 3: Computing the logarithm tables

```
@alog = []
@log = []
value = 1 # Start with the 0th power
256.times do |log| # For all exponents 0-255
  @alog[log] = value
  @log[@alog[log]] = log

  value <<= 1 # multiply by x
  if value > 0xFF # If the value overflows
    value ^= 0x11B # m = x^8 + x^4 + x^3 + x^1 + 1
  end
  value ^= @alog[log] # Add it's original value to itself
end

# Set some sensible defaults as log is only valid from 1-255
# and antilog only from 0-254
@log[0] = 0
@alog[255] = @alog[0]
```

This produces the following tables:

Table 1: The logarithm table

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

Table 2: The anti-logarithm table

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

Computation of the inverse

By using the logarithm tables it's possible to compute the inverse of a byte like this:

$$b^{-1} = \frac{1}{b} = \log^{-1}(\log 1 - \log b)$$

This is implemented like this:

Listing 4: Computation of the inverse tables

```
@inverse = [0] # the inverse of 0
1.upto(255) do |byte|
  @inverse[byte] = alog[log[1] - log[byte]]
end
```

Table 3: The resulting inverse table

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

The Rijndael S-box

The substitution box σ is a bijective function which maps $\mathbb{F}_2^8$ to $\mathbb{F}_2^8$. The sbox used is used to obscure the relationship between the key and encrypted data (this makes it harder to compute the key if you have both the encrypted and decrypted data, which is possible if you use XOR/Caesar encryption for example).

It's defined as follows:

$$\sigma : \mathbb{F}_2^8 \to \mathbb{F}_2^8, f \mapsto \begin{cases} \lambda(0) = x^6 + x^5 + x + 1 & \text{if } f = 0; \\ \lambda((f \bmod (m))^{-}1) & \text{else} \end{cases}$$

The implementation is basically the same as the λ affine map implementation. The only difference is that the inverse of the byte is used as input.

Listing 5: Computation of the sbox tables

```
@sbox = []
@inverse_sbox = []
256.times do |byte|
  s = x = inverse[byte] # This works even for 0 since inverse[0] = 0
  4.times do
    s = ((s << 1) | (s >> 7)) & 0xFF # Rotate byte
    x ^= s
  end
  @sbox[byte] = x ^ 0x63 # x^6 + x^5 + x + 1
  @inverse_sbox[@sbox[byte]] = byte
end
```

Table 4: The S-box

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

The rest of the implementation

In the implementation the Byte object uses the generated tables to perform all operations.

Listing 6: Byte implementation

```
class Byte
  # Perform exponentation
  def **(exponent)
    Tables.alog[(log * exponent) % 0xFF]
  end

  # Perform a multiplication using the log table
  def *(other)
    other = Byte.new(other)
    return Byte.new(0) if @value == 0 || other.value == 0
    Byte.new(Tables.alog[(log + other.log) % 0xFF])
  end

  # Perform devision by multiplying by the inverse
  def /(other)
    self * other.inverse
  end

  def sbox
    Byte.new(Tables.sbox[value])
  end

  def inverse_sbox
    Byte.new(Tables.inverse_sbox[value])
  end

  def inverse
    Byte.new(Tables.inverse[value])
  end

  # The log method returns a Fixnum, since it's value is not in the Rijndael
  # field
  def log
    Tables.log[value]
  end
end
```

Words

A word ($\mathbb{F}_{256}^4$) is a row of bytes defined as $w = (b_0, b_1, b_2, b_3)$ with $b \in \mathbb{F}_{256}$.

Listing 7: Word implementation

```
class Word
  attr_accessor :bytes
  def initialize(*bytes)
    # Make sure the values are all valid bytes
    @bytes = bytes.flatten.map! { |byte| Byte.new(byte) }
  end
end
```

S-box and the ξ map

The previously defined S-box on w is defined as follows:

$$\sigma(w) = \sigma(b_0, b_1, b_2, b_3) := (\sigma(b_0), \sigma(b_1), \sigma(b_2), \sigma(b_3)).$$

The implementation

Listing 8: S-box on a Word

```
class Word
  def sbox!
    bytes.map!(&:sbox) # Apply S-box to all elements
    self
  end
end
```

A second operation called ξ which applies σ and rotates the bytes one step is defined as follows:

$$\xi(w) = \xi(b_0, b_1, b_2, b_3) := (\sigma(b_1), \sigma(b_2), \sigma(b_3), \sigma(b_0)).$$

Listing 9: ξ on a Word

```
class Word
  def xi!
    sbox! # Apply S-box on all elements
    bytes.rotate! # Rotate the bytes
    self
  end
end
```

Mix columns (or the μ/ν maps)

A way to interpret the μ and ν maps is to see them as the following matrices:

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

$$\nu = \mu^3 = \mu^{-1} = \begin{pmatrix} x^3+x^2+x & x^3+x+1 & x^3+x^2+1 & x^3+1 \\ x^3+1 & x^3+x^2+x & x^3+x+1 & x^3+x^2+1 \\ x^3+x^2+1 & x^3+1 & x^3+x^2+x & x^3+x+1 \\ x^3+x+1 & x^3+x^2+1 & x^3+1 & x^3+x^2+x \end{pmatrix}$$

The word is then interpreted as a column vector:

$$\mu \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad \nu \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Implementation:

Listing 10: Mix/inverse mix columns implementation

```
class Word
  def mix! # μ
    do_mix(0x02, 0x03, 0x01, 0x01)
  end

  def demix! # ν
    do_mix(0x0E, 0x0B, 0x0D, 0x09)
  end

  private

  # Applies the μν/ transformation
  def do_mix(*vector)
    @bytes = 4.times.each_with_object([]) do |i, newbytes|
      newbytes[i] = 4.times.reduce(0) do |a, index|
        bytes[index] * vector[(index - i) % 4] + a
      end
    end
    self
  end
end
```

In this implementation the matrix computation is done row by row. The supplied vector is the first row of the μ/ν matrix. This vector is then multiplied with the word and then shifted to the left for 4 times (for each row of the matrix).

States

A state ($(\mathbb{F}_{256}^4)^4$) consists of 4 words (w_j with bytes a_j, b_j, c_j, d_j) that are treated like column vectors:

$$w_0 = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}, \; w_1 = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}, \; w_2 = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}, \; w_3 = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix}.$$

By defining the state like this we can interpret the state S as a 4×4 matrix of bytes.

All previously defined operations are the responsibility of the Word and Byte classes in this implementation:

Listing 11: State implementation

```
class State
  attr_accessor :words

  def initialize(*words)
    @words = words.flatten
  end

  def sbox!
    words.map!(&:sbox!)
    self
  end

  def mix!
    words.map!(&:mix!)
    self
  end

  def demix!
    words.map!(&:demix!)
    self
  end

  def inverse_sbox!
    words.map!(&:inverse_sbox!)
    self
  end
end
```

Blinding/XOR

It's possible to blind/xor states with another state. This is the same as byte by byte addition.

$$\tau_s(x) = x + s$$

Listing 12: Blinding

```
class Word
  # Blinds the Word
  def <<(other)
    4.times { |index| bytes[index] += other[index] }
    self
  end

  # Blinds a copy of the Word
  def +(other)
    Word.new(4.times.map { |index| bytes[index] + other[index] })
  end
end

class State
  # Used to blind the state
  def <<(other)
    4.times { |index| @words[index] << other[index] }
    self
  end

  # Blinds the state but returns a copy
  def +(other)
    State.new(4.times.map { |index| @words[index] + other[index] })
  end
end
```

ShiftRows (ρ)

Using the previously defined state S we define:

$$\rho(s) = \rho\left(\begin{pmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{pmatrix}\right) := \begin{pmatrix} a_0 & b_0 & c_0 & d_0 \\ b_1 & c_1 & d_1 & a_1 \\ c_2 & d_2 & a_2 & b_2 \\ d_3 & a_3 & b_3 & c_3 \end{pmatrix}$$

ρ is implemented on a word by word basis, it builds new columns for word i in the following manner (byte index between brackets):

$$w_i^{'} = \begin{pmatrix} w_{i+0}(0) \\ w_{i+1}(1) \\ w_{i+3}(2) \\ w_{i+4}(3) \end{pmatrix}$$

Listing 13: Shift rows implementation

```
class State
  # Shifts the rows (ρ)
  def shift_rows!
    row_shift
  end

  # Reverses shift_rows! (ρ^-1 = ρ^4)
  def unshift_rows!
    row_shift(-1) # Should be the same as row_shift(4)
  end

  private

  def row_shift(direction = 1)
    @words = 4.times.map do |i|
      Word.new(4.times.map { |byte| words[(i + direction * byte) % 4][byte] })
    end
    self
  end
end
```

Keys

The key is used to encrypt states. In this implementation the key is the same thing as a state as the key space equals the state space (in this specific case where the key length equals the block length).

Rijndael key schedule

The encryption uses 10 'derivatives' of the key to encrypt a state. These derived subkeys are computed word by word. This goes as follows:

$$k_l := (w_{4l}, w_{4l+1}, w_{4l+2}, w_{4l+3})$$

$$w_j := \begin{cases} \xi(w_{j-1}) + w_{j-4} + (x^{(j/4)-1}, 0, 0, 0) & \text{if } j = 0 \bmod 4; \\ w_{j-1} + w_{j-4} & \text{else} \end{cases}$$

So for a 4 word key you need 40 of these derived words.

Listing 14: Rijndael key schedule implementation

```
class Key < State
  # Get k_l
  def expanded_key(l)
    4.times.map { |i| key[l * 4 + i] }
  end

  # Dynamically derive the words using the Rijndael key schedule
  # Use a memoized Hash for storage
  def key
    @key ||= Hash.new do |hash, j|
      hash[j] = hash[j - 4] +
        if j % 4 == 0
          hash[j - 1].xi + Word.new(Byte.new(2) ** ((j / 4) - 1), 0, 0, 0)
        else
          hash[j - 1]
        end
    end.tap do |hash| # initializes key[0..3]
      @words.each_with_index do |word, index|
        hash[index] = word
      end
    end
  end
end
```

Encryption and decryption

Encryption and decryption is done using the following operations:

$$\epsilon_k = \tau_{k_{10}} \rho \sigma \tau_{k_9} \mu \rho \sigma \tau_{k_8} \mu \rho \sigma \tau_{k_7} \mu \rho \sigma \tau_{k_6} \mu \rho \sigma \tau_{k_5} \mu \rho \sigma \tau_{k_4} \mu \rho \sigma \tau_{k_3} \mu \rho \sigma \tau_{k_2} \mu \rho \sigma \tau_{k_1} \mu \rho \sigma \tau_{k_0}$$

$$\begin{aligned} \delta_k = {} & \tau_{k_0} \sigma^{-1} \rho^{-1} \nu \tau_{k_1} \sigma^{-1} \rho^{-1} \nu \tau_{k_2} \sigma^{-1} \rho^{-1} \nu \tau_{k_3} \sigma^{-1} \rho^{-1} \nu \tau_{k_4} \sigma^{-1} \rho^{-1} \nu \circ \\ & \circ \tau_{k_5} \sigma^{-1} \rho^{-1} \nu \tau_{k_6} \sigma^{-1} \rho^{-1} \nu \tau_{k_7} \sigma^{-1} \rho^{-1} \nu \tau_{k_8} \sigma^{-1} \rho^{-1} \nu \tau_{k_9} \sigma^{-1} \rho^{-1} \tau k_{10} \end{aligned}$$

You can see that it's mostly a repeated applications of the blinding, sbox, shift_rows and mix columns steps (with the latest round missing the mix step). This is implemented directly (albeit more clearly than in the above formulas):

Listing 15: The encrypt/decrypt methods

```
class Key < State
  def encrypt(state)
    state += expanded_key(0)
    1.upto(9) do |index|
      state.sbox!
      state.shift_rows!
      state.mix!
      state << expanded_key(index)
    end
    state.sbox!
    state.shift_rows!
    state << expanded_key(10)
  end

  def decrypt(state)
    state += expanded_key(10)
    state.unshift_rows!
    state.inverse_sbox!
    9.downto(1) do |index|
      state << expanded_key(index)
      state.demix!
      state.unshift_rows!
      state.inverse_sbox!
    end
    state << expanded_key(0)
  end
end
```

Testing procedure

Every part of the AES implementation was tested separately. The key scheduling part was tested using test vectors from http://samiam.org/key-schedule.html. The resulting program is tested using test vectors from NIST (http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf).

Full source

The full source of the AES implementation is available at https://github.com/jurriaan/rubyAES

References

[1] H.W. Lenstra. "Rijndael for algebraists". In: (2002).
[2] Jaap Top. *Beveiliging en Codes*. 2014.
[3] Sam Trenholme. *The AES encryption algorithm*. 2005.