The Laws of Cryptography: *The Finite Field GF(2^8)*

by Neal R. Wagner

NOTE: This site is obsolete. See book draft (in PDF):

The Laws of Cryptography with Java Code.

Finite Fields.

A *field* is an algebraic object with two operations: addition and multiplication, represented by **+** and *****, although they will not necessarily be ordinary addition and multiplication. Using **+**, all the elements of the field must form a commutative group, with identity denoted by **0** and the inverse of **a** denoted by **-a**. Using *****, all the elements of the field except **0** must form another commutative group with identity denoted **1** and inverse of **a** denoted by $\mathbf{a^{-1}}$. (The element **0** has no inverse under *****.) Finally, the *distributive identity* must hold: **a*(b + c) = (a*b) + (a*c)**, for all field elements **a**, **b**, and **c**.

There are a number of different infinite fields, including the rational numbers (fractions), the real numbers (all decimal expansions), and the complex numbers. Cryptography focuses on *finite* fields. It turns out that for any prime integer **p** and any integer **n** greater than or equal to **1**, there is a unique field with $\mathbf{p^n}$ elements in it, denoted $\mathbf{GF(p^n)}$. (The ``GF'' stands for ``Galois Field'', named after the brilliant young French mathematician who discovered them.) Here ``unique'' means that any two fields with the same number of elements must be essentially the same, except perhaps for giving the elements of the field different names.

In case **n** is equal to **1**, the field is just the *integers mod* ***p***, in which addition and multiplication are just the ordinary versions followed by taking the remainder on division by **p**. The only difficult part of this field is finding the multiplicative inverse of an element, that is, given a non-zero element **a** in $\mathbf{Z_p}$, finding $\mathbf{a^{-1}}$. This is the same as finding a **b** such that **a*b % p = 1**. This calculation can be done with the extended Euclidean algorithm, as is explained elsewhere in these notes.

The Finite Field GF(2^8).

The case in which **n** is greater than one is much more difficult to describe. In cryptography, one almost always takes **p** to be **2** in this case. This section just treats the special case of **p = 2** and **n = 8**, that is. $\mathbf{GF(2^8)}$, because this is the field used by the new U.S. *Advanced Encryption Standard* (AES).

The AES works primarily with *bytes* (8 bits), represented from the right as:

$\mathbf{b_7b_6b_5b_4b_3b_2b_1b_0}$.

The 8-bit elements of the field are regarded as *polynomials* with coefficients in the field $\mathbf{Z_2}$:

$$\mathbf{b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0}.$$

The field elements will be denoted by their sequence of bits, using two hex digits.

Addition in GF(2^8).

To add two field elements, just add the corresponding polynomial coefficients using addition in $\mathbf{Z_2}$. Here addition is modulo **2**, so that **1 + 1 = 0**, and addition, subtraction and exclusive-or are all the same. The identity element is just zero: **00000000** (in bits) or **0x00** (hex).

Multiplication in GF(2^8).

Multiplication is this field is much more difficult and harder to understand, but it can be implemented very efficiently in hardware and software. The first step in mutiplying two field elements is to multiply their corresponding polynomials just as in beginning algebra (except that the coefficients are only **0** or **1**, and **1 + 1 = 0** makes the calculation easier, since many terms just drop out). The result would be up to a degree **14** polynomial -- too big to fit into one byte. A finite field now makes use of a fixed degree eight irreducible polynomial (a polynomial that cannot be factored into the product of two simpler polynomials). For the AES the polynomial used is the following (other polynomials could have been used):

$\mathbf{m(x) = x^8 + x^4 + x^3 + x + 1 = 0x11b}$ (hex).

The intermediate product of the two polynomials must be divided by **m(x)**. The remainder from this division is the desired product.

This sounds hard, but is easier to do by hand than it might seem (though error-prone). To make it easier to write the polynomials down, adopt the convension that instead of $\mathbf{x^8 + x^4 + x^3 + x + 1}$ just write the exponents of each non-zero term. (Remember that terms are either zero or have a **1** as coefficient.) So write the following for **m(x)**: **(8 4 3 1 0)**.

Now try to take the product **(7 5 4 2 1) * (6 4 1 0)** (which is the same as **0xb6 * 0x53** in hexadecimal. First do the multiplication, remembering that in the sum below only an odd number of like powered terms results in a final term:

```
(7 5 4 2 1) * (6 4 1 0)      gives (one term at a time)

(7 5 4 2 1) * (6)    =   (13    11 10 9 8 7)
(7 5 4 2 1) * (4)    =         (11    9 8   6 5)
(7 5 4 2 1) * (1)    =                (8   6 5   3 2)
(7 5 4 2 1) * (0)    =    +              7   5 4   2 1)
                         ---------------------------------
                         (13       10 9 8     5 4 3   1)
```

The final answer requires the remainder on division by **m(x)**, or **(8 4 3 1)**. (This is like ordinary polynomial division, though easier because of the simpler arithmetic.

```
                                (13       10 9 8     5 4 3   1)
(8 4 3 1 0) * (5)     =         (13          9 8   6 5)
                                -------------------------------
                                          (10       6   4 3   1)
(8 4 3 1 0) * (2) =                       (10       6 5   3 2)
                                -------------------------------
                                                     (5 4   2 1)
                                                  (the remainder)
```

Thus the final result says that **0xb6 * 0x53 = 0x36** in the field. (When I did the calculations above, I made two separate mistakes, but checked my work with techniques below.)

Improved Multiplication in GF(2^8).

The above calculations could be converted to a program, but there is a better way. One does the calculations working from the low order terms, and repeatedly multiplying by **(1)**. If the result is of degree **8**, just add (the same as subtract) **m(x)** to get degree **7**. Again this can be illustrated using the above notation and the same example operands: **r * s = (7 5 4 2 1) * (6 4 1 0)**

```
i    powers of  r: r * (i)     Simplified Result          Final Sum
=============================================================================
0                               (7   5 4   2 1  )      (7   5 4   2 1  )
-----------------------------------------------------------------------------
1   (7   5 4   2 1  )*(1)  =  (8   6 5   3 2)
                             +(8       4 3   1 0)
                             ---------------------
                                  (6 5 4   2 1 0)     +  (6 5 4   2 1 0)
                                                         -----------------
                                                          (7 6           0)
-----------------------------------------------------------------------------
2     (6 5 4   2 1 0)*(1)  =    (7 6 5   3 2 1)
-----------------------------------------------------------------------------
3   (7 6 5   3 2 1)  *(1)  =  (8 7 6   4 3 2)
                             +(8       4 3   1 0)
                             ---------------------
                                (7 6       2 1 0)
-----------------------------------------------------------------------------
4   (7 6       2 1 0)*(1)  =  (8 7       3 2 1)
                             +(8       4 3   1 0)
                             ---------------------     (7 6           0)
                                (7     4   2   0)    + (7     4   2   0)
```

```
                                                                  ------------------
                                                                     (6   4   2)
-------------------------------------------------------------------------------------
5   (7     4   2   0)*(1)  =  (8     5   3   1)
                             +(8       4 3   1 0)
                              --------------------
                                     (5 4       0)
                                                                     (6   4   2)
6       (5 4       0)*(1)  =      (6 5       1)       +   (6 5       1)
                                                                  ----------------
                                                                       (5 4   2 1)
```

The final answer is the same as before. Here is an algorithm (almost) in Java that realizes the above calculations:

```
public byte FFMul(unsigned byte a, unsigned byte b) {
   unsigned byte aa = a, bb = b, r = 0, t;
   while (aa != 0) {
      if ((aa & 1) != 0)
         r = r ^ bb;
      t = bb & 0x80;
      bb = bb << 1;
      if (t != 0)
         bb = bb ^ 0x1b;
      aa = aa >> 1;
   }
   return r;
}
```

Unfortunately, Java has no **`unsigned byte`** type, and the logical operations produce a 32-bit integer. Finally, one ought to be able to use Java's ``right shift with zero fill'' operator **`>>>`**, but it doesn't work as it is supposed to. See Unsigned bytes in Java for a discussion of the problems encountered in converting the above ``Java'' program to actual Java. Later examples below show the code for this function.

Using Logarithms to Multiply.

When I was young (a *long* time ago) there were no pocket calculators. We had to do without modern conveniences like Gatorade. Mathematical calculations, if not done with a slide rule, were carried out by hand. Often we used printed tables of logarithms to turn multiplications into easier additions. (In these ``elder'' days, believe it or not, the printed tables included tables of the logarithms of trig functions of angles, so that you got the log directly for further calculations.)

As a simple example, suppose one wanted the area of a circle of radius **`23.427`** cm. Use the famous formula **`pi r`**$^{\mathbf{2}}$ (``pie are square, cake are round''), so one needs to calculate **`23.427 * 23.427 *`**

3.1416. We would look up the logarithm (base 10) of each number in the printed table: **log(23.427) = 1.369716** and **log(3.1416) = .497156**. Now add two copies of the first number and one of the second: **1.369716 + 1.369716 + .497156 = 3.236588**. Finally, take the ``anti-log'' (that is, take **10** to the power **3.236588**) to get the final answer: **1724.2** cm^2. This works because **log(area) = log(pi*r^2) = log(pi) + log(r) + log(r)**. (The actual use of log tables was much more horrible than the above might indicate. In case you want to find out how it *really* worked, look here, but have an air sickness bag handy.)

In a similar way, in finite fields one can replace the harder multiplication by the easier addition, at the cost of looking up ``logarithms'' and ``anti-logarithms.''

Generators in Fields.

First must come the concept of a *generator* of a finite field. Generators also play a role is certain simple but common random number generators, as is detailed in another section. A generator is an element whose successive powers take on every element except the zero. For example, in the field $\mathbf{Z_{13}}$, try successive powers of several elements, looking for a generator:

> Try powers of **5**, taken modulo **13**: **5^1 = 5**, **5^2%13 = 25%13 = 12**, **5^3%13 = (12*5)%13 = 8**, **5^4%13 = (8*5)%13 = 1**, so succesive powers of **5** just take on the values **5**, **12**, **8**, **1**, and repeat, so that **5** is not a generator.
>
> Now try powers of **4**, taken modulo **13**: **4^1 = 4**, **4^2%13 = 16%13 = 3**, **4^3%13 = (3*4)%13 = 12**, **4^4%13 = (12*4)%13 = 9**, **4^5%13 = (9*4)%13 = 10**, **4^6%13 = (10*4)%13 = 1**, so successive powers make a longer cycle, but still not all elements: **4**, **3**, **12**, **9**, **10**, **1**, and repeat, so **4** is also not a generator.
>
> Finally try successive powers of **2**, taken modulo **13**: **2^1 = 2**, **2^2%13 = 4%13 = 4**, **2^3%13 = 8%13 = 8**, **2^4%13 = (8*2)%13 = 3**, **2^5%13 = (3*2)%13 = 6**, **2^6%13 = (6*2)%13 = 12**, **2^7%13 = (12*2)%13 = 11**, **2^8%13 = (11*2)%13 = 9**, **2^9%13 = (9*2)%13 = 5**, **2^9%13 = (5*2)%13 = 10**, **2^9%13 = (10*2)%13 = 7**, **3^6%13 = (7*2)%13 = 1**, so successive powers take on all non-zero elements: **2**, **4**, **8**, **3**, **6**, **12**, **11**, **9**, **5**, **10**, **7**, **1**, and repeat, so **2** is a generator. (Wheh!)

The above random search shows that generators are hard to discover and are not intuitive. It turns out that **0x03**, which is the same as **x + 1** as a polynomial, is the simplest generator for **GF(2^8)**. Its powers take on all **255** non-zero values of the field. In fact the table below of ``exponentials'' or ``anti-logs'' gives each possible power. (The table is really just a simple linear table, not really 2-dimensional, but it has been arranged so that the two hex digits are on different axes.) Here **E(rs)** is the field element given by **03^{rs}**, where these are hex numbers, and the initial ``**0x**'' is left off for simplicity.

Table of ``exponentials'': E(rs) = 03^rs

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

Similarly, here is a table of ``logarithms'', where the entry **`L(rs)`** is the field element that satisfies **`rs = 03`$^{\texttt{L(rs)}}$**, where these are hex numbers, and again the initial ``**`0x`**'' is left off.

Table of ``logarithms'': rs = 03^L(rs)																	
L(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**		00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
	7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7

b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

These tables were created using the multiply function in the previous subsection. Here is a Java program that directly outputs the HTML source to make the tables: Generate Multiply Tables.

As an example, suppose one wants the product **b6 * 53** (the same product as in the examples above, leaving off the ``**0x**''). Use the **L** table above to look up **b6** and **53**: **L(b6) = b1** and **L(53) = 30**. This means that

$$(b6)*(53) = (03)^{(b1)} * (03)^{(30)} = (03)^{(b1 + 30)} = (03)^{(e1)}.$$

If the sum above gets bigger than **ff**, just subtract **255** as shown. This works because the powers of **03** repeat after **255** iterations. Now use the **E** table to look up $(03)^{(e1)}$, which is the answer: **(36)**.

Thus these tables give a much simpler and faster algorithm for multiplication:

```
public byte FFMulFast(unsigned byte a, unsigned byte b){
   int t = 0;;
   if (a == 0 || b == 0) return 0;
   t = L[a] + L[b];
   if (t > 255) t = t - 255;
   return E[t];
}
```

As before, this is Java as if it had an *unsigned* **byte** type, which it doesn't. The actual Java code requires some short, ugly additions. (See Unsigned bytes in Java to convert the above ``Java'' program to actual Java.)

This section has presented two algorithms for multiplying field elements, a slow one and a fast one. As a check, here is a program that compares the results of all 65536 possible products to see that the two methods agree (which they do): Compare multiplications.

The Multiplicative Inverse of Each Field Element.

Later work with the AES will also require the multiplicative inverse of each field element except **0**, which has no inverse. This inverse is easy to calculate, given the tables above. If **g** is the generator **03** (leaving off

the ``**0x**''), then the inverse of $g^{rs}$ is $g^{ff - rs}$, so that for example, to find the inverse of **6b**, look up in the ``log'' table to see that $6b = g^{54}$, so the inverse of **6b** is $g^{ff - 54} = g^{ab}$, and from the "exponential" table, this is **df**. The following table shows the result of carrying out the above procedure for each non-zero field element.

Table of multiplicative inverses: rs*inv(rs) = 01

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

See table generating program for code that will calculate and print the HTML source for the above table.

Revision date: **2001-12-23**. (Please use ISO 8601, the International Standard.)
