Matrix GF2 Algebra
On A5/1 Explained IV

By Juan Chamero, juan.chamero@intaq.org, as of April 2006

Going Backwards
How to invert GF2 Matrices

In this peculiar process where shift registers are feed from their right lsb with Kj unity vectors with its unique nonzero component located at its rightmost bit, the C*K product of any order is reduced to the XORing of last column of C matrices by Kj vectors. The algebra of states transition will take the form:

$$Si(9) = Ci(1)^8*Si(0) + Ci(1)^8*K1 + Ci(1)^7*K2 + \ldots\ldots + Ci(1)^1*K8 + K9$$

$$Si(9) = C(i)^8*Si(0) + XOR\ of\ MiK(0=>9)\ row\ vectors$$
$$Si(9) = MiK(0=>9)XOR$$

Where the matrices MiK(0=>9) (with corresponding K bits diffused in it) , corresponding to traversing from states 0 to 9 are built with as many rows as the path to traverse (9 states ⇔ 9 rows) having the following structure:

Row 1 ⇔ C1(8) ⇔ [0 k1 0 k1]
Row 2 ⇔ C1(7) ⇔ [k2 0 k2 0]
Row 3 ⇔ C1(6) ⇔ [k3 k3 0 k3]
Row 4 ⇔ C1(5) ⇔ [0 k4 k4 0]
Row 5 ⇔ C1(4) ⇔ [0 0 k5 k5]
Row 6 ⇔ C1(3) ⇔ [k6 0 0 k6]
Row 7 ⇔ C1(2) ⇔ [0 k7 0 0]
Row 8 ⇔ C1(1) ⇔ [0 0 k8 0]
Row 9 ⇔ C1(0) ⇔ [0 0 0 k9]

As we see Mi matrices are very easy to generate provided we have pre computed all the Ci's transition matrices. As we are going to generalize soon this procedure could be applied to non zero initial states and for any traverse.

For register R1 the nine steps Mi is as follows:

0 1 0 1
0 0 0 0
1 1 0 1
0 1 1 0
0 0 0 0
1 0 0 1
0 1 0 0
0 0 0 0
0 0 0 1
0 0 1 0

And XOR is even simpler because key bit equal to 0 eliminate the corresponding row. The XOR of the six non zero binary vectors give us the resulting state Si(9) = **[0 0 1 0]**

Let's see the procedure when initial states are non zero. For instance how to "jump" directly from state S1(4) = [1 0 1 0] to the same S1(9) throughout a direct jump of 5 steps.

$$S1(9) = C1(5)*S1(4) + M1(5=>9)(XOR)$$

M1(4=>9)

0 0 1 1 ⇔ k5=0
1 0 0 1 ⇔ k6=1
0 1 0 0 ⇔ k7=1
0 0 1 0 ⇔ k8=0
0 0 0 1 ⇔ k9=1

1 1 0 0 that XORed with the regular transition (in absence of K) from S1(4) to S1(9) via C1(5)*S1(4) that gives us **1 1 1 0** it reproduces S1(9) = (1 1 0 0) XOR (1 1 1 0) = (0 0 1 0)

Algebraically we may define states as:

$$(1)\quad Si(a) = C1(b-a)*Si(b) + M1(a=>b)XK(a=>b)$$

Where the operation X stands for XOR bitwise between M1(a=>) XORed versus K(a=>b), being K(a=>b) a K bits vector that circularly goes from a=>b. Concerning A5/1 matrices Mi that go from Si(0) to Si(64) have the following dimensions:

M1(0 =>64): 64x19
M2(0 =>64): 64x22
M3(0 =>64): 64x23

And Mi matrices coefficients are formed with Ci matrices last columns from Ci(1) to Ci(64). Our problem is *how we may go backwards algebraically*. Going forward is trivial as we have demonstrated applying (1), in A5/1:

S1(64) = C1(64)*S1(0) + M1(1=>64)XK
S2(64) = C2(64)*S2(0) + M2(1=>64)XK
S3(64) = C3(64)*S3(0) + M3(1=>64)XK

As A5/1 proceed to make all Si(0) equal to zero in this mixing with K, expressions simplify to

$$Si(64) = Mi(1=>64)XK$$

Returning to our example we have:

M1(1=>9)XK

C1(8) last column: 0101 ⇔ K1
C1(7) last column: 1010 ⇔ K2
C1(6) last column: 1101 ⇔ K3
C1(5) last column: 0110 ⇔ K4
C1(4) last column: 0011 ⇔ K5
C1(3) last column: 1001 ⇔ K6
C1(2) last column: 0100 ⇔ K7
C1(1) last column: 0010 ⇔ K8
C1(0) last column: 0001 ⇔ K9

Result: **0010**

That represents the following set of 4 equations:

k2 + k3 + k6 = 0 = S1(9) (3) ⇔ bit 3 msb
k1 + k3 + k4 +k7 = 0 = S1(9) (2) ⇔ bit 2
k2 + k4 + k5 + k8 = 1 = S1(9) (1) ⇔ bit 1
k1 + k3 + k5 + k6 + k9 = 0 = S1(9) (0) ⇔ bit 0

And for next register we get

M2(1=>9)XK

C2(8) last column: 010 ⇔ K1
C2(7) last column: 101 ⇔ K2
C2(6) last column: 011 ⇔ K3
C2(5) last column: 111 ⇔ K4
C2(4) last column: 110 ⇔ K5
C2(3) last column: 100 ⇔ K6
C2(2) last column: 101 ⇔ K7
C2(1) last column: 010 ⇔ K8
C2(0) last column: 001 ⇔ K9

Giving place to the following set of equations:

k2 + k4 + k5 + k6 + k7 = 1 = S2(9) (2) ⇔ bit 2 msb
k1 + k3 + k4 + k5 + k8 = 1 = S2(9) (1) ⇔ bit 1
k2 + k3 + k4 + k7 + k9 = 0 = S2(9) (0) ⇔ bit 0

And for last register R3 we get

M3(1=>9)XK

C3(8) last column: 11 ⇔ K1
C3(7) last column: 10 ⇔ K2
C3(6) last column: 01 ⇔ K3
C3(5) last column: 11 ⇔ K4
C3(4) last column: 10 ⇔ K5
C3(3) last column: 01 ⇔ K6
C3(2) last column: 11 ⇔ K7
C3(1) last column: 10 ⇔ K8
C3(0) last column: 01 ⇔ K9

Giving place to the following set of equations:

k1 + k2 + k4 + k5 + k7 + k8 = 1 = S3(9) (1) ⇔ bit 1 msb
k1 + k3 + k4 + k6 + k7 + k9 = 0 = S3(9) (0) ⇔ bit 0

S1(9) (3) ⇔ 0 1 1 0 0 1 0 0 0 k1
S1(9) (2) ⇔ 1 0 1 1 0 0 1 0 0 k2
S1(9) (1) ⇔ 0 1 0 1 1 0 0 1 0 k3
S1(9) (0) ⇔ 1 0 1 0 1 1 0 0 1 k4

S2(9) (2) ⇔ 0 1 0 1 1 1 1 0 0 k5
S2(9) (1) ⇔ 1 0 1 1 1 0 0 1 0 k6
S2(9) (0) ⇔ 0 1 1 1 0 0 1 0 1 k7

S3(9) (1) ⇔ 1 1 0 1 1 0 1 1 0 k8
S3(9) (0) ⇔ 1 0 1 1 0 1 1 0 1 k9

This is then the final expression that given K generates at the end of the “K mixing process”: M*K = S(9), and in the general case:

M*K = S

For the ciphering run and consequently its inversion:

M´S = K

Retrieve the private keyword from a S(64) state, being M´ the inverse matrix of M.

k2 + k3 + k6 = 0
k1 + k3 + k4 +k7 = 0
k2 + k4 + k5 + k8 = 1
k1 + k3 + k5 + k6 + k9 = 0
k2 + k4 + k5 + k6 + k7 = 1
k1 + k3 + k4 + k5 + k8 = 1
k2 + k3 + k4 + k7 + k9 = 0
9k1 + k2 + k4 + k5 + k7 + k8 = 1
k1 + k3 + k4 + k6 + k7 + k9 = 0

Eq 9: k9= k1 + k3 + k4 + k6 + k7

k2 + k3 + k6 = 0
k1 + k3 + k4 +k7 = 0
k2 + k4 + k5 + k8 = 1
k1 + k3 + k5 + k6 + 0 + k1 + k3 + k4 + k6 + k7 = 0 = k4 + k5 + k7
k2 + k4 + k5 + k6 + k7 = 1
k1 + k3 + k4 + k5 + k8 = 1
k2 + k3 + k4 + k7 + 0 + k1 + k3 + k4 + k6 + k7 = 0 = k1 + k2 + k6
k1 + k2 + k4 + k5 + k7 + k8 = 1

k2 + k3 + k6 = 0
k1 + k3 + k4 +k7 = 0
k2 + k4 + k5 + k8 = 12
k4 + k5 + k7 = 0
k2 + k4 + k5 + k6 + k7 = 1
k1 + k3 + k4 + k5 + k8 = 1
k1 + k2 + k6 = 0
k1 + k2 + k4 + k5 + k7 + k8 = 1

Eq. 8: k8 = 1 + k1 + k2 + k4 + k5 + k7

k2 + k3 + k6 = 0
k1 + k3 + k4 +k7 = 0
k2 + k4 + k5 + 1 + k1 + k2 + k4 + k5 + k7 = 1 => k1 + k7 = 0
k4 + k5 + k7 = 0
k2 + k4 + k5 + k6 + k7 = 1
k1 + k3 + k4 + k5 + 1 + k1 + k2 + k4 + k5 + k7 = 1 => k2 + k3 + k7 = 0
k1 + k2 + k6 = 0

k2 + k3 + k6 = 0
k1 + k3 + k4 +k7 = 0
k1 + k7 = 0
k4 + k5 + k7 = 0
k2 + k4 + k5 + k6 + k7 = 1
k2 + k3 + k7 = 0
k1 + k2 + k6 = 0

Eq. 7: k7 = k2 + k3

k2+ k3 + k6 = 0
k1 + k3 + k4 + k2 + k3 = 0 => k1 + k2 + k4 = 0
k1 + k2 + k3 = 0 => k1 + k2 + k3
k4 + k5 + k2 + k3 = 0 => k2 + k3 + k4 + k5 = 0
k2 + k4 + k5 + k6 + k2 + k3 = 1 => k2 + k3 + k4 + k5 + k6= 1
k1 + k2 + k6 = 0

k2+ k3 + k6 = 0
k1 + k2 + k4 = 0
k1 + k2 + k3 = 0
k2 + k3 + k4 + k5 = 0
k2 + k3 + k4 + k5 + k6= 1
k1 + k2 + k6 = 0

Eq. 6: k6 = k1 + k2

k2+ k3 + k1 + k2 = 0 => k1 + k3 = 0
k1 + k2 + k4 = 0
k1 + k2 + k3 = 0
k2 + k3 + k4 + k5 = 0
k2 + k3 + k4 + k5 + k1 + k2= 1 => k1 + k3 + k4 + k5 = 1

k1 + k3 = 0
k1 + k2 + k4 = 0
k1 + k2 + k3 = 0
k2 + k3 + k4 + k5 = 0
k1 + k3 + k4 + k5 = 1

Eq. 5: k5 = 1 + k1 + k3 + k4

0k1 + k3 = 0
k1 + k2 + k4 = 0
k1 + k2 + k3 = 0
k2 + k3 + k4 + 1 + k1 + k3 + k4 = 0 => k1 + k2 = 1

k1 + k3 = 0
k1 + k2 + k4 = 0
k1 + k2 + k3 = 0
k1 + k2 = 1

Eq. 4: k4 = k1 + k2

k1 + k3 = 0
k1 + k2 + k3 = 0
k1 + k2 = 1

Eq. 3: k3 = k1

k1 + k2 + k1 = 0 => k2 = 0
k1 + k2 = 1

k2 = 0
k1 + k2 = 1

Eq. 2: k2 = 0

k1 = 1
Eq. 1: k1 = 1

Now going backwards

k1 = **1**
k2 = **0**
k3 = k1 = **1**
k4 = k1 + k2 = 1 + 0 = **1**
k5 = 1 + k1 + k3 + k4 = 1 + 1 + 1 + 1 = **0**
k6 = k1 + k2 = 1 + 0 = **1**
k7 = k2 + k3 = 0 + 1 = **1**
k8 = 1 + k1 + k2 + k4 + k5 + k7 = 1 + 1 + 0 + 1 + 0 + 1 = **0**
k9 = 0 + k1 + k3 + k4 + k6 + k7 = 0 + 1 + 1 + 1 + 1 + 1 = **1**

K(9) = M´*S(9) = **[1 0 1 1 0 1 1 0 1]**

The procedure is straightforward like in the conventional Gauss Jordan method, by eliminating a variable at a time and once we get k1 going “forward“ we may proceed “backwards to solve k2 by using the (n-1) equation between k1 and k2, and then k3 using the (n-2) equation between k1, k2 and k3, and so on and so forth to get kn. To have a general procedure we may compute n times this replacement algorithm for a matrix US of n “Unity States”:

US1 = [1 0 0 ……0]
US2 = [0 1 0 ……0]
US3 = [0 0 1 ……0]
……
USn = [0 0 0 ……1]

Pseudo Gauss-Jordan XOR Algorithm => US => M´

Appendices

A) something about the XOR matrix logic

Let’s see how we should deal with the classical algebra operations aOb where the Operator symbol O stands for product (.), division (/), addition (+), subtraction (-) respectively.

a + b = c has the following outcomes:

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

a – b = c bound to a = b + c, using the same table

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

That property simplifies transformations to leave variable alone equations, for example:

a + b + c + d = 1
a = 1 + b + c + d,

Because subtracting b, c, or d is performed by adding these terms at each side of the equation: equals cancel each other: a + (b + b) + (c + c) + (d + d) = 1 + b + c + d => a + 0 + 0 + 0 = 1 + b + c + d.

Multiplication does not present any problem. Normally operations are performed via the regular binary arithmetic operator (.):

.	0	1
0	0	0
1	0	1

Let's see what happens with / operation defined as follows:

a/b = c, bound to a = b.c

a	b	c
0	0	0
0	1	0
1	0	?
1	1	1

We may appreciate here the ambiguity in 0/0 and 0/1 that gives the same result and the undefined 1/0 operation.

B) A5/1 Characteristic Matrices Generation

Now that we have solved the whole problem: how to get the Private Keyword once arrived to the state S(-22) via matrices we have to generate the first powers of the basic transition matrix for the three registers R1, R2 and R3. These basic matrices are:

C1(1)
[0100000000000000000]
[0010000000000000000]
[0001000000000000000]
[0000100000000000000]
.....
[0000000000000000010]
[0000000000000000001]
[1110010000000000000]

C2(1)
[01000000000000000000]
[00100000000000000000]
[00010000000000000000]
[00001000000000000000]
.....
[00000000000000000010]
[00000000000000000001]
[11000000000000000000]

C3(1)

[01000000000000000000000]
[00100000000000000000000]
[00010000000000000000000]
[00001000000000000000000]
……
[00000000000000000000010]
[00000000000000000000001]
[11100000000000010000000]

And for these basic matrices we have to perform their 64 powers, from Ci(1) to Ci(1)^64. With their last columns we proceed to build the M matrices that have to be inverted.

Warning: We need then to program two routines, namely: Matrix Multiplication between them and with a vector and Matrix Inversion.

C) Binary Matrix Inversion Sample check

The algorithm resembles the classical Gauss Jordan algorithm adapted to GF2. It deals with matrices of order m that are expanded within a mx2m order array where the first subspace of mxm order is occupied by the matrix to be inverted and the second of same order is virtually occupied by a Unit Matrix.

Iter 1 Before => Alter																										
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
0	0	0	1	0			0	0	1	0	0			0	0	0	1	0			0	0	1	0	0
0	0	0	0	1			0	0	0	1	0			0	0	0	0	1			0	0	0	1	0
1	1	0	0	0			0	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 2 Before => Alter																										
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
0	0	0	1	0			0	0	1	0	0			0	0	0	1	0			0	0	1	0	0
0	0	0	0	1			0	0	0	1	0			0	0	0	0	1			0	0	0	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 3 Before => Alter																										
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
0	0	0	1	0			0	0	1	0	0			0	0	0	1	0			0	0	1	0	0
0	0	0	0	1			0	0	0	1	0			0	0	0	0	1			0	0	0	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 4 Before => After																										
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
0	0	0	1	0			0	0	1	0	0			0	0	0	1	0			0	0	1	0	0
0	0	0	0	1			0	0	0	1	0			0	0	0	0	1			0	0	0	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 5 Before => After																										
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
0	0	0	1	0			0	0	1	0	0			0	0	0	1	0			0	0	1	0	0
0	0	0	0	1			0	0	0	1	0			0	0	0	0	1			0	0	0	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1

Pivotal Sequence: 23451
Inverse:

1	0	0	0	1
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0

Iter 1 Before => Alter																										
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0

0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
1	1	0	1	0			0	0	1	0	0			1	1	0	1	0			1	0	1	0	0
0	0	0	1	1			0	0	0	1	0			0	0	0	1	1			0	0	0	1	0
1	1	0	0	0			0	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 2 Before => After																								
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
1	1	0	1	0			1	0	1	0	0			1	1	0	1	0			1	0	1	0	0
0	0	0	1	1			0	0	0	1	0			0	0	0	1	1			0	0	0	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 3 Before => After																								
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
1	1	0	1	0			1	0	1	0	0			1	1	0	1	0			1	0	1	0	0
0	0	0	1	1			0	0	0	1	0			1	1	0	1	1			1	0	1	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 4 Before => After																								
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
1	1	0	1	0			1	0	1	0	0			1	1	0	1	0			1	0	1	0	0
1	1	0	1	1			1	0	1	1	0			1	1	0	1	1			1	0	1	1	0
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1
Iter 5 Before => After																								
0	1	0	0	0			1	0	0	0	0			0	1	0	0	0			1	0	0	0	0
0	0	1	0	0			0	1	0	0	0			0	0	1	0	0			0	1	0	0	0
1	1	0	1	0			1	0	1	0	0			1	0	0	1	0			0	0	1	0	1
1	1	0	1	1			1	0	1	1	0			1	0	0	0	1			0	0	1	1	1
1	1	0	0	0			1	0	0	0	1			1	1	0	0	0			1	0	0	0	1

Pivotal Sequence: 23451
Inverse:

1	0	0	0	1
1	0	0	0	0
0	1	0	0	0
0	0	1	0	1
0	0	1	1	1

As appreciated the procedure is very simple. Once the pivotal column is selected only rows with their corresponding elements to that column are non zero are processed. At their turn in each row are only processed those elements whose headers at pivotal row level are also non zero.

Warning: Most of these matrices are sparse and accordingly we may design fast scripts pointing from non zero to non zero elements once mapped.