The mathematics of RAID-6

H. Peter Anvin <*hpa@zytor.com*>

29 December 2003 DRAFT

RAID-6 supports losing any two drives. The way this is done is by computing two syndromes, generally referred **P** and **Q**.

1 A quick summary of Galois field algebra

The algebra used for this is the algebra of a Galois field, $\mathbf{GF}(2^8)$. A smaller or larger field could also be used, however, a smaller field would limit the number of drives possible, and a larger field would require extremely large tables.

$\mathbf{GF}(2^8)$ allows for a maximum of $2^8 = 256$ drives, 254 of which can be data drives; the reason for this is shown below.

The *representation* of $\mathbf{GF}(2^8)$ used is the same one as used by the Rijndael (AES) cryptosystem. It has the following properties; this is not, however, an exhaustive list nor a formal derivation of these properties; for more in-depth coverage see any textbook on group and ring theory.

Note: A number in {} is a Galois field element (i.e. a byte) in hexadecimal representation; a number without {} is a conventional integer.

1. The *addition* field operator (+) is represented by bitwise XOR.
2. As a result, addition and subtraction are the same operation: $A + B = A - B$.
3. The additive identity element (0) is represented by {00}.
4. Thus, $A + A = A - A = \{00\}$.
5. *Multiplication* ($\cdot$) by {02} is implemented by the following bitwise relations:

$$
\begin{aligned}
(x \cdot \{\texttt{02}\})_7 &= x_6 \\
(x \cdot \{\texttt{02}\})_6 &= x_5 \\
(x \cdot \{\texttt{02}\})_5 &= x_4 \\
(x \cdot \{\texttt{02}\})_4 &= x_3 + x_7 \\
(x \cdot \{\texttt{02}\})_3 &= x_2 + x_7 \\
(x \cdot \{\texttt{02}\})_2 &= x_1 + x_7 \\
(x \cdot \{\texttt{02}\})_1 &= x_0 \\
(x \cdot \{\texttt{02}\})_0 &= x_7
\end{aligned}
$$

Hardware engineers will recognize as a linear feedback shift register (LFSR), and matematicians as boolean polynomial multiplication modulo the irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$.

6. The multiplicative identity element (1) is represented by {01}.

$$A \cdot \{\texttt{01}\} = \{\texttt{01}\} \cdot A = A$$

7. The following basic rules of algebra apply:

Addition is commutative:	$A + B$	$=$	$B + A$
Addition is associative:	$(A + B) + C$	$=$	$A + (B + C)$
Multiplication is commutative:	$A \cdot B$	$=$	$B \cdot A$
Multiplication is associative:	$(A \cdot B) \cdot C$	$=$	$A \cdot (B \cdot C)$
Distributive law:	$(A + B) \cdot C$	$=$	$A \cdot C + B \cdot C$

8. Any nonzero element can uniquely divide an element:

   If $A \cdot B = C$ then $C/B = A$ for any $B \neq \{\texttt{00}\}$.

   In particular, $A/A = \{\texttt{01}\}$ for any $A \neq \{\texttt{00}\}$.

9. Multiplying by zero is zero:

$$A \cdot \{\texttt{00}\} = \{\texttt{00}\}$$

10. Any value can be multiplied by observing that bits decompose the same as in ordinary arithmetic, and applying the distributive law:

$$
\begin{aligned}
\{\texttt{02}\}^2 &= \{\texttt{02}\} \cdot \{\texttt{02}\} &= \{\texttt{04}\} \\
\{\texttt{02}\}^3 &= \{\texttt{04}\} \cdot \{\texttt{02}\} &= \{\texttt{08}\} \\
\{\texttt{02}\}^4 &= \{\texttt{08}\} \cdot \{\texttt{02}\} &= \{\texttt{10}\} \\
\{\texttt{02}\}^5 &= \{\texttt{10}\} \cdot \{\texttt{02}\} &= \{\texttt{20}\} \\
\{\texttt{02}\}^6 &= \{\texttt{20}\} \cdot \{\texttt{02}\} &= \{\texttt{40}\} \\
\{\texttt{02}\}^7 &= \{\texttt{40}\} \cdot \{\texttt{02}\} &= \{\texttt{80}\}
\end{aligned}
$$

(Note, however: $\{\texttt{02}\}^8 = \{\texttt{1d}\}$.)

For example:

$$\begin{aligned}\{\texttt{8d}\} &= \{\texttt{80}\} + \{\texttt{08}\} + \{\texttt{04}\} + \{\texttt{01}\} \\ &= \{\texttt{02}\}^7 + \{\texttt{02}\}^3 + \{\texttt{02}\}^2 + \{\texttt{01}\}\end{aligned}$$

Thus:

$$A \cdot \{\texttt{8d}\} = A \cdot \{\texttt{02}\}^7 + A \cdot \{\texttt{02}\}^3 + A \cdot \{\texttt{02}\}^2 + A$$

or, equivalently,

$$A \cdot \{\texttt{8d}\} = (((A \cdot \{\texttt{02}\}^4) + A) \cdot \{\texttt{02}\} + A) \cdot \{\texttt{02}\}^2 + A$$

11. *Raising to a power* (repeated multiplication with the same value) is congruent mod 255 (cardinality of all elements except $\{\texttt{00}\}$). Also note that the exponent is an *ordinary integer*[1] as opposed to a Galois field element.

$$\left.\begin{array}{lllllll} A^{256} & = & \{\texttt{01}\} \cdot A & = & A & & \\ A^{255} & = & \{\texttt{01}\} & & & & \\ A^{254} & = & A^{255}/A & = & \{\texttt{01}\}/A & = & A^{-1} \end{array}\right\} A \neq \{\texttt{00}\}$$

A^{-1} is called the *inverse* (or reciprocal) of A. $\{\texttt{01}\}$ is its own inverse, $\{\texttt{00}\}$ lacks inverse, for all other elements $A^{-1} \neq A$.

For any A, any $B \neq \{\texttt{00}\}$, $A/B = A \cdot B^{-1}$. Accordingly, $A/A = A \cdot A^{-1} = \{\texttt{01}\}$ for any $A \neq \{\texttt{00}\}$.

12. There are elements (g), called generators, of the field such that g^n doesn't repeat until they have exhausted all elements of the field except $\{\texttt{00}\}$. For the AES field representation, $\{\texttt{02}\}$ is such a generator.

13. Accordingly, any generator g defines a function from the nonzero elements in $\mathbf{GF}(2^8)$ to the elements in $\mathbb{Z}_{255}$ (i.e. the integers 0-254, modulo 255) called the *logarithm with base g* and written $\log_g$. For example, $\{\texttt{02}\}^4 = \{\texttt{10}\}$, so $\log_{\{\texttt{02}\}} \{\texttt{10}\} = 4$.

For any nonzero Galois field elements A and B:

$$A \cdot B = C \iff \log_g A \oplus \log_g B = \log_g C$$

... where $\oplus$ represents conventional integer addition modulo 255. Therefore:

$$C = g^{(\log_g A \oplus \log_g B)}$$

[1] Formally, an element in $\mathbb{Z}_{255}$.

2 Application to RAID-6

We treat each disk block as a vector of bytes, and will perform the same calculations on each byte in the vector. Symbols in **boldface** represent vectors (where each byte has a different value); constants, or symbols in *italics* represent scalars (same value across every data byte.)

In order to be able to suffer the loss of any two disks, we need to compute two *syndromes*, here referred to as **P** and **Q**.

For n data disks $\mathbf{D}_0$, $\mathbf{D}_1$, $\mathbf{D}_2$, ... $\mathbf{D}_{n-1}$ ($n \leq 254$) compute:

$$\mathbf{P} = \mathbf{D}_0 + \mathbf{D}_1 + \mathbf{D}_2 + ... + \mathbf{D}_{n-1} \quad (1)$$

$$\mathbf{Q} = g^0 \cdot \mathbf{D}_0 + g^1 \cdot \mathbf{D}_1 + g^2 \cdot \mathbf{D}_2 + ... + g^{n-1} \cdot \mathbf{D}_{n-1} \quad (2)$$

where g is any generator of the field (we use $g = \{\texttt{02}\}$.)

P is the ordinary XOR parity, since "addition" is XOR. **Q** is referred to as a Reed-Solomon code.

If we lose one data drive, we can use the normal XOR parity to recover the failed drive data, just as we would do for RAID-5. If we lose a non-data drive, i.e. **P** or **Q**, then we can just recompute.

If we lose one data drive plus the **Q** drive, we can recalculate the data drive using the XOR parity, and then recompute the **Q** drive.

If we lose one data drive plus the **P** drive, we can recompute the lost data drive ($\mathbf{D}_x$) from the **Q** drive by computing $\mathbf{Q}_x$ as if $\mathbf{D}_x = \{\texttt{00}\}$, and observing:

$$\mathbf{Q}_x + g^x \cdot \mathbf{D}_x = \mathbf{Q} \quad (3)$$

Here, x, $\mathbf{Q}$ and $\mathbf{Q}_x$ are known. Since addition and subtraction is the same:

$$g^x \cdot \mathbf{D}_x = \mathbf{Q} + \mathbf{Q}_x \quad (4)$$

$$\mathbf{D}_x = (\mathbf{Q} + \mathbf{Q}_x)/g^x = (\mathbf{Q} + \mathbf{Q}_x) \cdot g^{-x} \quad (5)$$

where, per the algebra rules, $g^{-x} = g^{255-x}$.

If we lose two data drives, $\mathbf{D}_x$ and $\mathbf{D}_y$, but still have the **P** and **Q** values, we compute $\mathbf{P}_{xy}$ and $\mathbf{Q}_{xy}$ by setting the missing drives to $\{\texttt{00}\}$, and we get:

$$\mathbf{P}_{xy} + \mathbf{D}_x + \mathbf{D}_y = \mathbf{P} \quad (6)$$

$$\mathbf{Q}_{xy} + g^x \cdot \mathbf{D}_x + g^y \cdot \mathbf{D}_y = \mathbf{Q} \quad (7)$$

x, y, $\mathbf{P}$, $\mathbf{P}_{xy}$, $\mathbf{Q}$ and $\mathbf{Q}_{xy}$ are known.

Divide the second equation by g^x:

$$g^{-x} \cdot \mathbf{Q}_{xy} + \mathbf{D}_x + g^{y-x} \cdot \mathbf{D}_y = g^{-x} \cdot \mathbf{Q} \quad (8)$$

Remembering that addition equals subtraction in this algebra:

$$\mathbf{D}_x + g^{y-x} \cdot \mathbf{D}_y = g^{-x} \cdot \mathbf{Q} + g^{-x} \cdot \mathbf{Q}_{xy} \tag{9}$$

$$\mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot \mathbf{D}_y \tag{10}$$

Substitute into the first equation, solve for $\mathbf{D}_y$:

$$\mathbf{D}_y = \mathbf{P} + \mathbf{P}_{xy} + \mathbf{D}_x \tag{11}$$

$$\mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy} + \mathbf{D}_x) \tag{12}$$

$$\mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x}(\mathbf{P} + \mathbf{P}_{xy}) + g^{y-x} \cdot \mathbf{D}_x \tag{13}$$

$$\mathbf{D}_x + g^{y-x} \cdot \mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy}) \tag{14}$$

$$(g^{y-x} + \{\texttt{01}\}) \cdot \mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy}) \tag{15}$$

If $g^{y-x} + \{\texttt{01}\} \neq \{\texttt{00}\}$, we can divide by it. This requires $g^{y-x} \neq \{\texttt{01}\}$; this will be true as long as $y \neq x$, mod 255. Since we can have no more than 254 data disks, $0 \leq x, y \leq n-1 \leq 254$, this implies the only constraint is $y \neq x$, which is true by assumption. Thus, we can divide:

$$\mathbf{D}_x = \frac{g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy})}{g^{y-x} + \{\texttt{01}\}} \tag{16}$$

For any particular data reconstruction, we can simplify this by precomputing a few multiplication tables:

$$A = \frac{g^{y-x}}{g^{y-x} + \{\texttt{01}\}} = g^{y-x} \cdot (g^{y-x} + \{\texttt{01}\})^{-1} \tag{17}$$

$$B = \frac{g^{-x}}{g^{y-x} + \{\texttt{01}\}} = g^{-x} \cdot (g^{y-x} + \{\texttt{01}\})^{-1} \tag{18}$$

... which only depend on x and y as opposed to on the data bytes.
The expression then becomes:

$$\mathbf{D}_x = A \cdot (\mathbf{P} + \mathbf{P}_{xy}) + B \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) \tag{19}$$

We can then get $\mathbf{D}_y$ from the previous expression:

$$\mathbf{D}_y = (\mathbf{P} + \mathbf{P}_{xy}) + \mathbf{D}_x \tag{20}$$

3 Making it go fast

The biggest problem with RAID-6 has historically been the high CPU cost of computing the **Q** syndrome. The biggest cost is related to the cost of Galois field multiplication, which doesn't map conveniently onto standard CPU hardware, and therefore has typically been done by table lookup.

Table lookups, however, are inherently serializing; it would be desirable to make use of the wide datapaths of current CPUs.

In order to do this, we factor equation 2 as such:

$$\mathbf{Q} = ((...\mathbf{D}_{n-1}...) \cdot g + \mathbf{D}_2) \cdot g + \mathbf{D}_1) \cdot g + \mathbf{D}_0 \quad (21)$$

The only operations in this is addition, i.e. XOR, and multiplication by $g = \{02\}$. Thus, we only need an efficient way to implement multiplication by $\{02\}$ in order to compute **Q** quickly, not arbitrary multiplication.

Multiplication by $\{02\}$ for a single byte can be implemeted using the C code:

```
uint8_t c, cc;
cc = (c << 1) ^ ((c & 0x80) ? 0x1d : 0);
```

Now, we want to do this on multiple bytes in parallel. Assume for the moment we are on a 32-bit machine (the extension to 64 bits should be obvious), and separate these into two parts:

```
uint32_t v, vv;

vv  = (v << 1) & 0xfefefefe;
vv ^= ((v & 0x00000080) ? 0x0000001d : 0) +
      ((v & 0x00008000) ? 0x00001d00 : 0) +
      ((v & 0x00800000) ? 0x001d0000 : 0) +
      ((v & 0x80000000) ? 0x1d000000 : 0);
```

The `0xfefefefe` of the first statement masks any bits that get shifted into the next byte. The second statement is clearly too complex to be efficiently executed, however. If we can produce a mask based on the top bit in each byte, we could just do:

```
uint32_t v, vv;

vv  = (v << 1) & 0xfefefefe;
vv ^= MASK(v) & 0x1d1d1d1d;
```

In standard portable C, one implemenation of this `MASK()` function looks like:

```
uint32_t MASK(uint32_t v)
{
  v &= 0x80808080;              /* Extract the top bits */
  return (v << 1) - (v >> 7);   /* Overflow on the top bit is OK */
}
```

The result is `0x00` for any byte with the top bit clear, `0xff` for any byte with the top bit set. This is the algorithm used in the file `raid6int.uc`.

For additional speed improvements, it is desirable to use any integer vector instruction set that happens to be available on the machine, such as MMX or SSE-2 on x86, AltiVec on PowerPC, etc. These instruction sets typically have quirks that may make them easier or harder to use than the integer implementation. In particular, the MMX/SSE-2 instruction `PCMPGTB` conveniently implements the `MASK()` function when comparing against zero, and the `PADDB` instruction implements the shift and mask in the first line of the operations on `vv` when added with itself.

Note that none of this will avoid the arbitrary multiplications of equations 5 and 19. Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case.

4 Beyond RAID-6

Reed-Solomon coding can be exploited further to allow for any combination of n data disks plus m redundancy disks allowing for any m failures to be recovered. However, with increasing amount of redundancy, the higher the overhead both in CPU time and I/O. The Linux RAID-6 work has been focused on handling the case of $m = 2$ efficiently in order for it to be practically useful.

An excellent paper on implementing arbitrarily complex recovery sets using Reed-Solomon coding can be found at:

`http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html`