A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems

James S. Plank*

Technical Report CS-96-332

Department of Computer Science

University of Tennessee

IMPORTANT

The information dispersal matrix A given in this paper does not have the desired properties. Please see Technical Report CS-03-504 for a correction to this problem. This technical report is available at
`http://www.cs.utk.edu/~plank/papers/CS-03-504.html`.

---

* `plank@cs.utk.edu`. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9409496, and by the ORAU Junior Faculty Enhancement Award.

Abstract

It is well-known that Reed-Solomon codes may be used to provide error correction for multiple failures in RAID-like systems. The coding technique itself, however, is not as well-known. To the coding theorist, this technique is a straightforward extension to a basic coding paradigm and needs no special mention. However, to the systems programmer with no training in coding theory, the technique may be a mystery. Currently, there are no references that describe how to perform this coding that do not assume that the reader is already well-versed in algebra and coding theory.

This paper is intended for the systems programmer. It presents a complete specification of the coding algorithm plus details on how it may be implemented. This specification assumes no prior knowledge of algebra or coding theory. The goal of this paper is for a systems programmer to be able to implement Reed-Solomon coding for reliability in RAID-like systems without needing to consult any external references.

Problem Specification

> Let there be n storage devices, $D_1, D_2, \ldots, D_n$, each of which holds k bytes. These are called the *"Data Devices."* Let there be m more storage devices $C_1, C_2, \ldots, C_m$, each of which also holds k bytes. These are called the "*Checksum Devices.*" The contents of each checksum device will be calculated from the contents of the data devices. The goal is to define the calculation of each C_i such that if any m of $D_1, D_2, \ldots, D_n, C_1, C_2, \ldots, C_m$ fail, then the contents of the failed devices can be reconstructed from the non-failed devices.

Introduction

Error-correcting codes have been around for decades [1, 2, 3]. However, the technique of distributing data among multiple storage devices to achieve high-bandwidth input and output, and using one or more error-correcting devices for failure recovery is relatively new. It came to the fore with "Redundant Arrays of Inexpensive Disks" (*RAID*) where batteries of small, inexpensive disks combine high storage capacity, bandwidth, and reliability all at a low cost [4, 5, 6]. Since then, the technique has been used to design multicomputer and network file systems with high reliability and bandwidth [7, 8], and to design fast distributed checkpointing systems [9, 10, 11, 12]. We call all such systems *"RAID-like"* systems.

The above problem is central to all RAID-like systems. When storage is distributed among n devices, the chances of one of these devices failing becomes significant. To be specific, if the mean time before failure of one device is F, then the mean time to failure of a system of n devices is $\frac{F}{n}$. Thus in such systems, fault-tolerance must be taken into account.

For small values of n and reasonably reliable devices, one checksum device is often sufficient for fault-tolerance. This is the "RAID Level 5" configuration, and the coding technique is called *"n+1-parity."* [4, 5, 6]. With n+1-parity, the i-th byte of the checksum device is calculated to be the bitwise exclusive-or (`XOR`) of the i-th byte of each data device. If any one of the n+1 devices fails, it can be reconstructed as the `XOR` of the remaining n devices. N+1-parity is attractive because of its simplicity. It requires one extra storage device, and one extra write operation per write to any single device. Its main disadvantage is that it cannot recover from more than one simultaneous failure.

As n grows, the ability to tolerate multiple failures becomes important [13]. Several techniques have been developed for this [13, 14, 15, 16], the concentration being small values of m. The most general technique for tolerating m simultaneous failures with exactly m checksum devices is a technique based on Reed-Solomon coding. This fact is cited in almost all papers on RAID-like systems. However, the technique itself is harder to come by.

The technique has an interesting history. It was first presented in terms of secret sharing by Karnin [17], and then by Rabin [18] in terms of information dispersal. Preparata [19] then showed the relationship between Rabin's method and Reed-Solomon codes, hence the labeling of the technique as Reed-Solomon coding. The technique has recently been discussed in varying levels of detail by Gibson [5], Schwarz [20] and Burkhard [13], with citations of standard texts on error correcting codes [1, 2, 3, 21, 22] for completeness.

There is one problem with all the above discussions of this technique — they require the reader to have a thorough knowledge of algebra and coding theory. Any programmer with a bachelor's degree in computer science has the skills to implement this technique, however few such programmers have the background in algebra and coding theory to understand the presentations in these papers and books.

The goal of this paper is to provide a presentation that can be understood by any systems programmer. No background in algebra or coding theory is assumed. We give a complete specification of the technique plus implementation details. A programmer should need no other references besides this paper to implement Reed-Solomon coding for reliability from multiple device failures in RAID-like systems.

General Strategy

Formally, our failure model is that of an *erasure*. When a device fails, it shuts down, and the system recognizes this shutting down. This is as opposed to an *error*, in which a device failure is manifested by storing and retrieving incorrect values that can only be recognized by sort of embedded coding [2, 23].

The calculation of the contents of each checksum device C_i requires a function F_i applied to all the data devices. Figure 1 shows an example configuration using this technique (which we henceforth call *"RS-Raid"*) for $n = 8$ and $m = 2$. The contents of checksum devices C_1 and C_2 are computed by applying functions F_1 and F_2 respectively.

The RS-Raid coding method breaks up each storage device into *words*. The size of each word is w bits, w being chosen by the programmer (subject to some constraints). Thus, the storage devices contain $l = (k \text{ bytes}) \left(\frac{8 \text{ bits}}{\text{byte}}\right) \left(\frac{1 \text{ word}}{w \text{ bits}}\right) = \frac{8k}{w}$ words each. The coding functions F_i operate on a word-by-word basis, as in Figure 2, where $x_{i,j}$ represents the j-th word of device X_i.

To make the notation simpler, we assume that each device holds just one word and drop the extra subscript. Thus we view our problem as consisting of n data words $d_1, \ldots, d_n$ and m checksum words $c_1, \ldots, c_m$ which are computed from the data words in such a way that the loss of any m words can be tolerated.

To compute a checksum word c_i for the checksum device C_i, we apply function F_i to the data words:

$$c_i = F_i(d_1, d_2, \ldots, d_n).$$

If a data word on device D_j is updated from d_j to d'_j, then each checksum word c_i is recomputed by applying a func-

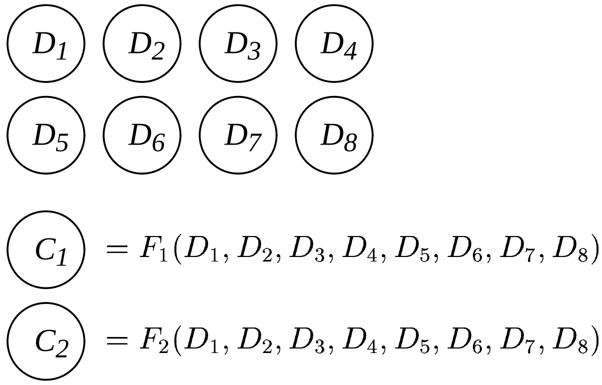


Figure 1: Providing two-site fault tolerance with two checksum devices

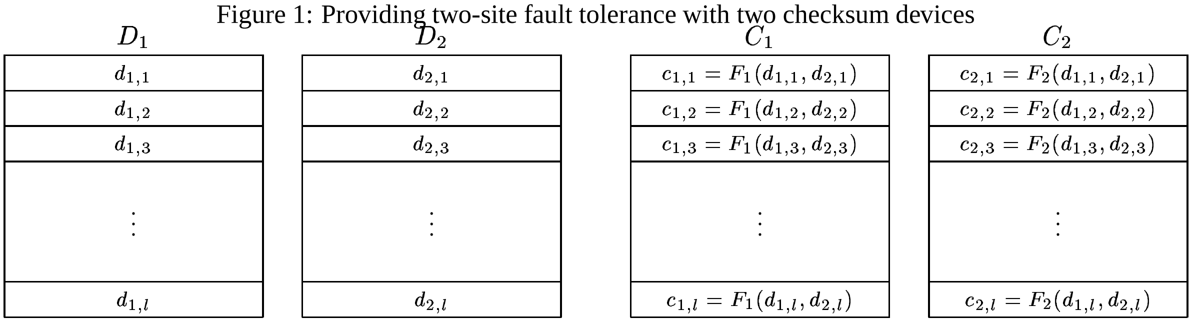


Figure 2: Breaking the storage devices into words ($n = 2$, $m = 2$, $l = \frac{8k}{w}$)

tion $G_{i,j}$ such that:

$$c_i' = G_{i,j}(d_j, d_j', c_i).$$

When up to m devices fail, we reconstruct the system as follows. First, for each failed data device D_j, we construct a function to restore the words in D_j from the words in the non-failed devices. When that is completed, we recompute any failed checksum devices C_i with F_i.

For example, suppose $m = 1$. We can describe n+1-parity in the above terms. There is one checksum device C_1, and words consist of one bit ($w = 1$). To compute each checksum word c_1, we take the parity (XOR) of the data words:

$$c_1 = F_1(d_1, \ldots, d_n) = d_1 \oplus d_2 \oplus \ldots \oplus d_n.$$

If a word on data device D_j changes from d_j to d_j', then c_1 is recalculated from the parity of its old value and the two data words:

$$c_1' = G_{1,j}(d_j, d_j', c_1) = c_1 \oplus d_j \oplus d_j'.$$

If a device D_j fails, then each word may be restored as the parity of the corresponding words on the remaining devices:

$$d_j = d_1 \oplus \ldots \oplus d_{j-1} \oplus d_{j+1} \oplus \ldots \oplus d_n \oplus c_1.$$

In such a way, the system is resilient to any single device failure.

To restate, our problem is defined as follows. We are given n data words $d_1, d_2, \ldots, d_n$ all of size w. We define functions F and G which we use to calculate and maintain the checksum words $c_1, c_2, \ldots, c_m$. We then describe how to reconstruct the words of any lost data device when up to m devices fail. Once the data words are reconstructed, the checksum words can be recomputed from the data words and F. Thus, the entire system is reconstructed.

Overview of the RS-Raid Algorithm

There are three main aspects of the RS-Raid algorithm: using the Vandermonde matrix to calculate and maintain checksum words, using Gaussian Elimination to recover from failures, and using Galois Fields to perform arithmetic. Each is detailed below:

Calculating and Maintaining Checksum Words

We define each function F_i to be a linear combination of the data words:

$$c_i = F_i(d_1, d_2, \ldots, d_n) = \sum_{j=1}^{n} d_j f_{i,j}$$

In other words, if we represent the data and checksum words as the vectors D and C, and the functions F_i as rows of the matrix F, then the state of the system adheres to the following equation:

$$FD = C.$$

We define F to be the $m \times n$ Vandermonde matrix: $f_{i,j} = j^{i-1}$, and thus the above equation becomes:

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \ldots & f_{1,n} \\ f_{2,1} & f_{2,2} & \ldots & f_{2,n} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \ldots & f_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & 2 & 3 & \ldots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \ldots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}.$$

When one of the data words d_j changes to d'_j, then each of the checksum words must be changed as well. This can be effected by subtracting out the portion of the checksum word that corresponds to d_j, and adding the required amount for d'_j. Thus, $G_{i,j}$ is defined as follows:

$$c'_i = G_{i,j}(d_j, d'_j, c_i) = c_i + f_{i,j}(d'_j - d_j).$$

Therefore, the calculation and maintenance of checksum words can be done by simple arithmetic (however, it is a special kind of arithmetic, as explained below).

Recovering From Failures

To explain recovery from errors, we define the matrix A and the vector E as follows: $A = \begin{bmatrix} I \\ F \end{bmatrix}$, and $E = \begin{bmatrix} D \\ C \end{bmatrix}$. Then we have the following equation ($AD = E$):

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \dots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}.$$

We can view each device in the system as having a corresponding row of the matrix A and the vector E. When a device fails, we reflect the failure by deleting the device's row from A and from E. What results a new matrix A' and a new vector E' that adhere to the equation:

$$A'D = E'.$$

Suppose exactly m devices fail. Then A' is a $n \times n$ matrix. Because matrix F is defined to be a Vandermonde matrix, every subset of n rows of matrix A is guaranteed to be linearly independent. Thus, the matrix A' is non-singular, and the values of D may be calculated from $A'D = E'$ using Gaussian Elimination. Hence all data devices can be recovered.

Once the values of D are obtained, the values of any failed C_i may be recomputed from D. It should be obvious that if fewer than m devices fail, the system may be recovered in the same manner, choosing any n rows of A' to perform the Gaussian Elimination. Thus, the system can tolerate any number of device failures up to m.

Arithmetic over Galois Fields

A major concern of the RS-Raid algorithm is that the domain and range of the computation are binary words of a fixed length w. Although the above algebra is guaranteed to be correct when all the elements are infinite precision real numbers, we must make sure that it is correct for these fixed-size words. A common error in dealing with these codes is to perform all arithmetic over the integers modulo 2^w. This *does not work*, as division is not defined for all pairs of elements (for example, $(3 \div 2)$ is undefined modulo 4), rendering the Gaussian Elimination unsolvable in many cases. Instead, we must perform addition and multiplication over a *field* with more than $n + m$ elements [2].

Fields with 2^w elements are called *Galois Fields* (denoted $GF(2^w)$), and are a fundamental topic in algebra (e.g. [3, 21, 24]). This section defines how to perform addition, subtraction, multiplication, and division efficiently over a Galois Field. We give such a description without explaining Galois Fields in general. Appendix A contains a more detailed description of Galois Fields, and provides justification for the arithmetic algorithms in this section.

The elements of $GF(2^w)$ are the integers from zero to $2^w - 1$. Addition and subtraction of elements of $GF(2^w)$ are simple. They are the XOR operation. For example, in $GF(2^4)$:

$$11 + 7 = 1011 \oplus 0111 = 1100 = 12.$$

$$11 - 7 = 1011 \oplus 0111 = 1100 = 12.$$

Multiplication and division are more complex. When w is small (16 or less), we use two logarithm tables, each of length $2^w - 1$, to facilitate multiplication. These tables are `gflog` and `gfilog`:

- `int gflog[]`: This table is defined for the indices 1 to $2^w - 1$, and maps the index to its logarithm in the Galois Field.
- `int gfilog[]`: This table is defined for the indices 0 to $2^w - 2$, and maps the index to its inverse logarithm in the Galois Field. Obviously, `gflog[gfilog[`i`]]` $= i$, and `int gfilog[gflog[`i`]]` $= i$.

With these two tables, we can multiply two elements of $GF(2^w)$ by adding their logs and then taking the inverse log, which yields the product. To divide two numbers, we instead subtract the logs. Figure 3 shows an implementation in C: This implementation makes use of the fact that the inverse log of an integer i is equal to the inverse log of ($i \bmod (2^w - 1)$). This fact is explained in Appendix A. As with regular logarithms, we must treat zero as a special case, as the logarithm of zero is $-\infty$.

```
#define NW (1 << w)   /* In other words, NW equals 2 to the w-th power */

int mult(int a, int b)
{
  int sum_log;

  if (a == 0 || b == 0) return 0;
  sum_log = gflog[a] + gflog[b];
  if (sum_log >= NW-1) sum_log -= NW-1;
  return gfilog[sum_log];
}

int div(int a, int b)
{
  int diff_log;

  if (a == 0) return 0;
  if (b == 0) return -1;            /* Can't divide by 0 */
  diff_log = gflog[a] - gflog[b];
  if (diff_log < 0) diff_log += NW-1;
  return gfilog[diff_log];
}
```

Figure 3: C code for multiplication and division over $GF(2^w)$ (Note: $\texttt{NW} = 2^w$)

Unlike regular logarithms, the log of any non-zero element of a Galois Field is an integer, allowing for exact multiplication and division of Galois Field elements using these logarithm tables.

An important step, therefore, once w is chosen, is generating the logarithm tables for $GF(2^w)$. The algorithm to generate the logarithm and inverse logarithm tables for any w can be found in Appendix A; however the realization of this algorithm in C for $w = 4$, $w = 8$ or $w = 16$ is included here in Figure 4. We include the tables for $GF(2^4)$ as generated by `setup_tables(4)` in Table 1.

```
unsigned int prim_poly_4 = 023;
unsigned int prim_poly_8 = 0435;
unsigned int prim_poly_16 = 0210013;
unsigned short *gflog, *gfilog;

int setup_tables(int w)
{
  unsigned int b, log, x_to_w, prim_poly;

  switch(w) {
    case 4:  prim_poly = prim_poly_4;  break;
    case 8:  prim_poly = prim_poly_8;  break;
    case 16: prim_poly = prim_poly_16; break;
    default: return -1;
  }

  x_to_w = 1 << w;
  gflog  = (unsigned short *) malloc (sizeof(unsigned short) * x_to_w);
  gfilog = (unsigned short *) malloc (sizeof(unsigned short) * x_to_w);

  b = 1;
  for (log = 0; log < x_to_w-1; log++) {
    gflog[b] = (unsigned short) log;
    gfilog[log] = (unsigned short) b;
    b = b << 1;
    if (b & x_to_w) b = b ^ prim_poly;
  }
  return 0;
}
```

Figure 4: C code for generating the logarithm tables of $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
gflog[i]	—	0	1	4	2	8	5	10	3	14	9	7	6	13	11	12
gfilog[i]	1	2	4	8	3	6	12	11	5	10	7	14	15	13	9	—

Table 1: Logarithm tables for $GF(2^4)$

For example, using the values in Table 1 the following is arithmetic in $GF(2^4)$:

$$
\begin{aligned}
3 * 7 &= \texttt{gfilog[gflog[3]+gflog[7]]} = \texttt{gfilog[4+10]} = \texttt{gfilog[14]} = 9 \\
13 * 10 &= \texttt{gfilog[gflog[13]+gflog[10]]} = \texttt{gfilog[13+9]} = \texttt{gfilog[7]} = 11 \\
13 \div 10 &= \texttt{gfilog[gflog[13]-gflog[10]]} = \texttt{gfilog[13-9]} = \texttt{gfilog[4]} = 3 \\
3 \div 7 &= \texttt{gfilog[gflog[3]-gflog[7]]} = \texttt{gfilog[4-10]} = \texttt{gfilog[9]} = 14
\end{aligned}
$$

Therefore, a multiplication or division requires one conditional, three table lookups (two logarithm table lookups and one inverse table lookup), an addition or subtraction, and a modulo operation. For efficiency in Figure 3, we implement the modulo operation as a conditional and a subtraction or addition.

The Algorithm Summarized

Given n data devices and m checksum devices, the RS-Raid algorithm for making them fault-tolerant to up to m failures is as follows.

1. Choose a value of w such that $2^w > n + m$. It is easiest to choose $w = 8$ or $w = 16$, as words then fall directly on byte boundaries. Note that with $w = 16$, $n + m$ can be as large as $65,535$.

2. Set up the tables `gflog` and `gfilog` as described in Appendix A and implemented in Figure 4.

3. Set up the matrix F to be the $m \times n$ Vandermonde matrix: $f_{i,j} = j^{i-1}$ (for $1 \le i \le m, 1 \le j \le n$) where multiplication is performed over $GF(2^w)$.

4. Use the matrix F to calculate and maintain each word of the checksum devices from the words of the data devices. Again, all addition and multiplication is performed over $GF(2^w)$.

5. If any number of devices up to m fail, then they can be restored in the following manner. Choose any n of the remaining devices, and construct the matrix A' and vector E' as defined previously. Then solve for D in $A'D = E'$. This enables the data devices to be restored. Once the data devices are restored, the failed checksum devices may be recalculated using the matrix F.

An Example

As an example, suppose we have three data devices and three checksum devices, each of which holds one megabyte. Then $n = 3$, and $m = 3$. We choose w to be four, since $2^w > n + m$, and since we can use the logarithm tables in Table 1 to illustrate multiplication.

Next, we set up `gflog` and `gfilog` to be as in Table 1. We construct F to be a 3×3 Vandermonde matrix, defined over $GF(2^4)$:

$$F = \begin{bmatrix} 1^0 & 2^0 & 3^0 \\ 1^1 & 2^1 & 3^1 \\ 1^2 & 2^2 & 3^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix}$$

Now, we can calculate each word of each checksum device using $FD = C$. For example, suppose the first word of D_1 is 3, the first word of D_2 is 13, and the first word of D_3 is 9. Then we use F to calculate the first words of C_1, C_2, and C_3:

$$\begin{aligned} C_1 &= (1)(3) \oplus (1)(13) \oplus (1)(9) \\ &= 3 \oplus 13 \oplus 9 \\ &= 0011 \oplus 1101 \oplus 1001 = 0111 = 7 \\ C_2 &= (1)(3) \oplus (2)(13) \oplus (3)(9) \\ &= 3 \oplus 9 \oplus 8 \\ &= 0011 \oplus 1001 \oplus 1000 = 0010 = 2 \\ C_3 &= (1)(3) \oplus (4)(13) \oplus (5)(9) \\ &= 3 \oplus 1 \oplus 11 \\ &= 0011 \oplus 0001 \oplus 1011 = 1001 = 9 \end{aligned}$$

Suppose we change D_2 to be 1. Then D_2 sends the value $(1 - 13) = (0001 \oplus 1101) = 12$ to each checksum device, which uses this value to recompute its checksum:

$$
\begin{aligned}
C_1 &= 7 \oplus (1)(12) = 0111 \oplus 1100 = 11 \\
C_2 &= 2 \oplus (2)(12) = 2 \oplus 11 = 0010 \oplus 1011 = 9 \\
C_3 &= 9 \oplus (4)(12) = 9 \oplus 5 = 1001 \oplus 0101 = 12
\end{aligned}
$$

Suppose now that devices D_2, D_3, and C_3 are lost. Then we delete the rows of A and E corresponding to D_1, D_2, and C_3 to get $A'D = E'$:

$$
\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} D = \begin{bmatrix} 3 \\ 11 \\ 9 \end{bmatrix}
$$

By applying Gaussian elimination, we can invert A' to yield the following equation: $D = (A')^{-1}E'$, or:

$$
D = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 11 \\ 9 \end{bmatrix}.
$$

From this, we get:

$$
D_2 = (2)(3) \oplus (3)(11) \oplus (1)(9) = 6 \oplus 14 \oplus 9 = 1
$$

$$
D_3 = (3)(3) \oplus (2)(11) \oplus (1)(9) = 5 \oplus 5 \oplus 9 = 9
$$

And then:

$$
C_3 = (1)(3) \oplus (4)(1) \oplus (5)(9) = 3 \oplus 4 \oplus 11 = 12
$$

Thus, the system is recovered.

Implementation and Performance Details

We examine some implementation and performance details of RS-Raid coding on two applications: a RAID controller, and a distributed checkpointing system. Both are pictured in Figure 5. In a RAID controller, there is one central processing location that controls the multiple devices. A distributed checkpointing system is more decentralized. Each device is controlled by a distinct processing unit, and the processing units communicate by sending messages over a communication network.

RAID Controllers

In RAID systems, a basic file system operation is when a process writes an entire stripe's worth of data to a file. The file system must break up this data into n blocks, one for each data device, calculate m blocks worth of encoding information, and then write one block to each of the n+m devices. The overhead of calculating c_1 is

$$
S_{Block}(n-1)\left(\frac{1}{R_{\mathsf{XOR}}}\right),
$$

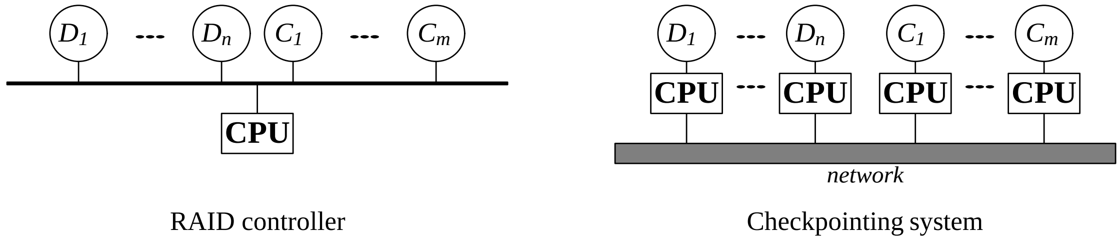


Figure 5: RAID-like configurations

where S_{Block} is the size of a block and R_{XOR} is the rate of performing XOR. This is because the first row of F is all ones and therefore there are no Galois Field multiplications in the calculation of c_1. The overhead of calculating c_i where $i > 1$ is:

$$S_{Block}(n-1)\left(\frac{1}{R_{\mathsf{XOR}}} + \frac{1}{R_{GFmult}}\right),$$

where R_{GFmult} is the rate of performing Galois Field multiplications. This is because n-1 of the n data blocks must be multiplied by some $f_{i,j} \neq 1$ before being XOR'd together. Thus the overhead of calculating the m checksum blocks is

$$S_{Block}(n-1)\left(\frac{m}{R_{\mathsf{XOR}}} + \frac{(m-1)}{R_{GFmult}}\right).$$

The cost of writing an entire parity stripe is therefore the above figure plus the time to write one block to each of the $n+m$ disks.[1]

A second basic file system operation is overwriting a small number of bytes of a file. This updates the information stored on one disk, and necessitates a recalculation of the encoding on each checksum disk. To be specific, for each word of disk D_j that is changed from d_j to d'_j, the appropriate word of each checksum disk C_i is changed from c_i to $c_i + f_{i,j}(d'_j - d_j)$, where arithmetic is performed over the Galois Field.

The cost of computing $(d'_j - d_j)$ is one XOR operation. This needs to be performed just one time. The cost of multiplying $(d'_j - d_j)$ by $f_{i,j}$ is zero if $i = 1$ or $j = 1$, and one Galois Field multiplication if $i > 1$ and $j > 1$. Finally, the cost of adding $f_{i,j}(d'_j - d'_j)$ to c_i is one XOR operation for each value of i. Thus, the total cost of changing a word from d_j to d'_j is:

$$\text{The cost of writing one word to } m+1 \text{ disks} + \begin{cases} \left(\frac{m+1}{R_{\mathsf{XOR}}}\right) & \text{if } j = 1 \\ \left(\frac{m+1}{R_{\mathsf{XOR}}}\right) + \left(\frac{m-1}{R_{GFmult}}\right) & \text{otherwise.} \end{cases}$$

The dominant portion of this cost is the cost of writing to the disks. For this reason, Gibson defines the *update penalty* of an encoding strategy to be the number of disks that must be updated per word update [14]. For RS-Raid coding, the update penalty is m disks, which is the minimum value for tolerating m failures. As in all RAID systems, the encoding information may be distributed among the $n+m$ disks to avoid having the checksum disks become hot spots [5, 26].

The final operation of concern is recovery. Here, we assume that $y \leq m$ failures have occurred and the system must recover the contents of the y disks. In the RS-Raid algorithm, recovery consists of performing Gaussian Elimination of an equation $A'D = E'$ so that $(A')^{-1}$ is determined. Then, the contents of all the failed disks may be calculated as a linear combination of the disks in E'. Thus, recovery has two parts: the Gaussian Elimination and the recalculation.

[1] We do not include any equations for the time to perform disk reads/writes because the complexity of disk operation precludes a simple encapsulation [25].

Since at least $n - y$ rows of A' are identity rows, the Gaussian Elimination takes $O(y^2 n)$ steps. As y is likely to be small this should be very fast (i.e. milliseconds). The subsequent recalculation of the failed disks can be broken into parity stripes. For each parity stripe, one block is read from each of the n non-failed disks. One block is then calculated for each of the failed disks, and then written to the proper replacement disk. The cost of recovering one block is therefore:

$$\left(\begin{array}{c} \text{The cost of reading one} \\ \text{block from each of } n \text{ disks} \end{array} \right) + \left(\frac{(y) S_{Block}(n-1)}{R_{\mathsf{XOR}}} \right) + \left(\frac{(y) S_{Block}(n)}{R_{GFmult}} \right) + \left(\begin{array}{c} \text{The costs of writing one} \\ \text{block to each of } y \text{ disks} \end{array} \right).$$

Note that the $\left(\frac{(y) S_{Block}(n)}{R_{GFmult}} \right)$ term accounts for the fact that all the elements of $(A')^{-1}$ may be greater than one. For more detailed information on other parameters that influence the performance of recovery in RAID systems, see Reference [26].

Checkpointing Systems

In distributed checkpointing systems, the usage of RS-Raid encoding is slightly different from its usage in the RAID controller. Here, there are two main operations, checkpointing and recovery. With checkpointing, we assume that the data devices hold data, but that the checksum devices are uninitialized. There are two basic approaches that can be taken to initializing the checksum devices:

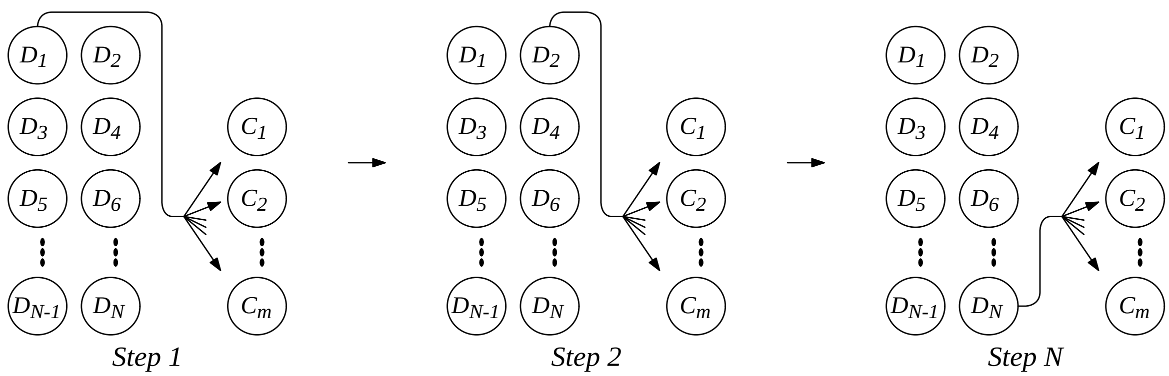


Figure 6: The broadcast algorithm

The Broadcast Algorithm (Figure 6): Each checksum device C_i initializes its data to zero. Then each data device D_j broadcasts its contents to every checksum device C_i. Upon receiving D_j's data, C_i multiplies it by $f_{i,j}$ and XOR's it into its data space. When this is done, all the checksum devices are initialized. The time complexity of this method is

$$n S_{device} \left(\frac{1}{R_{broadcast}} + \frac{1}{R_{GFmult}} + \frac{1}{R_{\mathsf{XOR}}} \right),$$

Where S_{device} is the size of the device and $R_{broadcast}$ is the rate of message broadcasting. This assumes that message-sending bandwidth dominates latency, and that the checksum devices do not overlap computation and communication significantly.

The Fan-in Algorithm (Figure 7): This algorithm proceeds in m steps — one for each C_i. In step i, each data device D_j multiplies its data by $f_{i,j}$, and then the data devices perform a fan-in XOR of their data, sending the final result to C_i. The time complexity of this method is

$$m S_{device} \left(\frac{\log n}{R_{\mathsf{XOR}}} + \frac{\log n + 1}{R_{network}} \right) + \left(\frac{(m-1) S_{device}}{R_{GFmult}} \right),$$

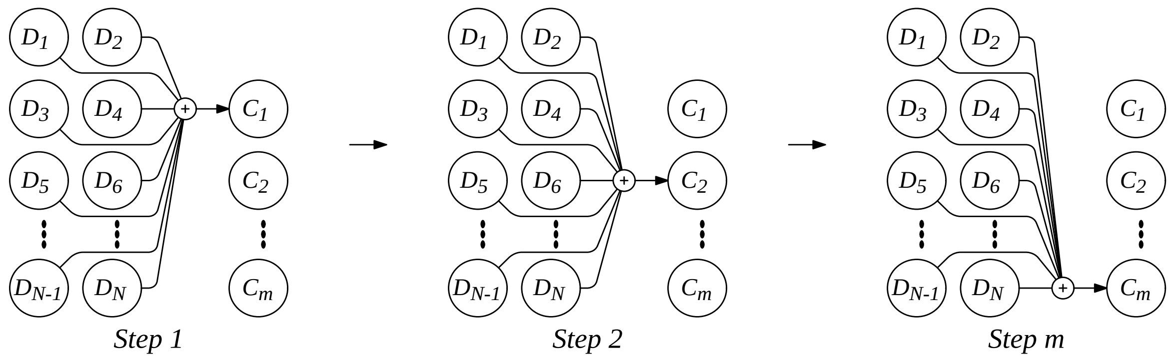


Figure 7: The Fan-in algorithm

where $R_{network}$ is the network bandwidth. This takes into account the fact that no Galois Field multiplications are necessary to compute C_1. Moreover, this equation assumes that there is no contention for the network during the fan-in. On a broadcast network like an Ethernet, where two sets of processors cannot exchange messages simultaneously, the $\log n$ terms become $n-1$.

Obviously, the choice of algorithm is dictated by the characteristics of the network.

Recovery from failure is straightforward. Since the Gaussian Elimination is fast, it should be performed redundantly in the CPU's of each device (as opposed to performing the Gaussian Elimination with some sort of distributed algorithm).

The recalculation of the failed devices can then be performed using either the broadcast or fan-in algorithm as described above. The cost of recovery should thus be slightly greater than the cost of computing the checksum devices.

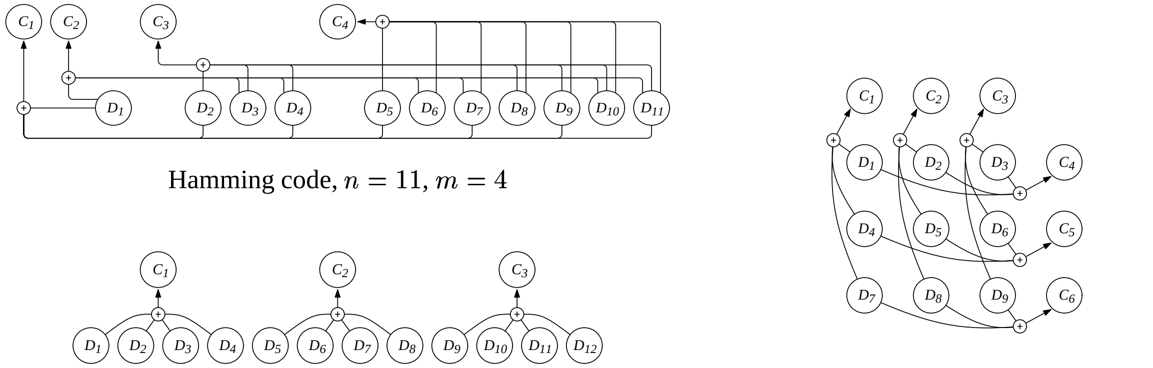
Other Coding Methods

There are other coding methods that can be used for fault-tolerance in RAID-like systems. Most are based on parity encodings, where each checksum device is computed to be the bitwise exclusive-or of some subset of the data devices:

$$c_i = a_{i,1}d_1 \oplus a_{i,2}d_2 \oplus \ldots \oplus a_{i,n}d_n, \text{ where } a_{i,j} \in \{0,1\}.$$

Although these methods can tolerate up to m failures (for example, all the checksum devices can fail), they do not tolerate *all* combinations of m failures. For example, the well-known Hamming code can be adapted for RAID-like systems [5]. With Hamming codes, $m = \lceil \log(m+n-1) \rceil$ checksum devices are employed, and all two-device failures may be tolerated. One-dimensional parity [14] is another parity-based method that can tolerate certain classes of multiple-device failures. With one-dimensional parity, the data devices are partitioned into m groups, $g_1 \ldots g_m$, and each checksum device c_i is computed to be the parity of the data devices in g_i. With one-dimensional parity, the system can tolerate one failure per group. Note that when $m=1$, this is simply n+1-parity, and when $m=n$, this is equivalent to device mirroring.

Two-dimensional parity [14] is an extension of one-dimensional parity that tolerates any two device failures. With two-dimensional parity, m must be greater than or equal to $2\sqrt{n}$, which can result in too much cost if devices are expensive. Other strategies for parity-based encodings that tolerate two and three device failures are discussed in Reference [14]. Since



Hamming code, $n = 11$, $m = 4$

One-dimensional parity, $n = 12$, $m = 3$

Two-dimensional parity, $n = 9$, $m = 6$

Figure 8: Parity-based encodings

all of these schemes are based on parity, they show better performance than RS-Raid coding for equivalent values of m. However, unlike RS-Raid coding, these schemes do not have *minimal device overhead*. In other words, there are some combinations of $k \leq m$ device failures that the system cannot tolerate.

An important coding technique for two device failures is EVENODD coding [15]. This technique tolerates all two device failures with just two checksum devices, and all coding operations are XOR's. Thus, it too is faster than RS-Raid coding. To the author's knowledge, there is no parity-based scheme that tolerates three or more device failures with minimal device overhead.

Conclusion

This paper has presented a complete specification for implementing Reed-Solomon coding for RAID-like systems. With this coding, one can add m checksum devices to n data devices, and tolerate the failure of any m devices. This has application in disk arrays, network file systems and distributed checkpointing systems.

This paper does not claim that RS-Raid coding is the best method for all applications in this domain. For example, in the case where $m = 2$, EVENODD coding [15] solves the problem with better performance, and one-dimensional parity [14] solves a similar problem with even better performance. However, RS-Raid coding is the only general solution for all values of n and m.

The table-driven approach for multiplication and division over a Galois Field is just one way of performing these actions. For values where $n + m < 65,536$, this is an efficient software solution that is easy to implement and does not consume much physical memory. For larger values of $n + m$, other approaches (hardware or software) may be necessary. See References [2], [27] and [28] for examples of other approaches.

Acknowledgements

The author thanks Joel Friedman, Kai Li, Michael Puening, Norman Ramsey, Brad Vander Zanden and Michael Vose for their valuable comments and discussion concerning this paper.

References

[1] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, 1968.

[2] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes, Second Edition*. The MIT Press, Cambridge, Massachusetts, 1972.

[3] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[4] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *1988 ACM Conference on Management of Data*, pages 109–116, June 1988.

[5] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. The MIT Press, Cambridge, Massachusetts, 1992.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 1994.

[7] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *Operating Systems Review – 14th ACM Symposium on Operating System Principles*, 27(5):29–43, December 1993.

[8] P. Cao, S. B. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems*, 12(3), 1994.

[9] J. S. Plank and K. Li. Faster checkpointing with $N + 1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

[10] J. S. Plank, Y. Kim, and J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *25th International Symposium on Fault-Tolerant Computing*, pages 351–360, Pasadena, CA, June 1995.

[11] T. Chiueh and P. Deng. Efficient checkpoint mechanisms for massively parallel machines. In *26th International Symposium on Fault-Tolerant Computing*, Sendai, June 1996.

[12] J. S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *15th Symposium on Reliable Distributed Systems*, October 1996.

[13] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *23rd International Symposium on Fault-Tolerant Computing*, pages 432–441, Toulouse, France, June 1993.

[14] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, April 1989.

[15] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *21st Annual International Symposium on Computer Architecture*, pages 245—254, Chicago, IL, April 1994.

[16] C-I. Park. Efficient placement of parity and data to tolerate two disk failures in disk array systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1177–1184, November 1995.

[17] E. D. Karnin, J. W. Greene, and M. E. Hellman. On secret sharing systems. *IEEE Transactions on Information Theory*, IT-29(1):35–41, January 1983.

[18] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[19] F. P. Preparata. Holographic dispersal and recovery of information. *IEEE Transactions on Information Theory*, 35(5):1123–1124, September 1989.

[20] T. J. E. Schwarz and W. A. Burkhard. RAID organization and performance. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 318–325, Yokohama, June 1992.

[21] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.

[22] S. B. Wicker and V. K. Bhargava. *Reed-Solomon Codes and Their Applications*. IEEE Press, New York, 1994.

[23] D. Wiggert. *Codes for Error Control and Synchronization*. Artech House, Inc., Norwood, Massachusetts, 1988.

[24] I. N. Herstein. *Topis in Algebra, Second Edition*. Xerox College Publishing, Lexington, Massachusetts, 1975.

[25] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[26] M. Holland, G. A. Gibson, and D. P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *23rd International Symposium on Fault-Tolerant Computing*, pages 422–423, Toulouse, France, June 1993.

[27] A. Z. Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II*. Springer-Verlag, New York, 1991.

[28] D. W. Clark and L-J. Weng. Maximal and near-maximal shift register sequences: Efficient event counters and easy discrete logarithms. *IEEE Transactions on Computers*, 43(5):560–568, 1994.

Appendix A: Galois Fields, as applied to this algorithm

Galois Fields are a fundamental topic of algebra, and are given a full treatment in a number of texts [24, 3, 21]). This Appendix does not attempt to define and prove all the properties of Galois Fields necessary for this algorithm. Instead, our goal is to give enough information about Galois Fields that anyone desiring to implement this algorithm will have a good intuition concerning the underlying theory.

A *field* $GF(n)$ is a set of n elements closed under addition and multiplication, for which every element has an additive and multiplicative inverse (except for the 0 element which has no multiplicative inverse). For example, the field $GF(2)$ can be represented as the set $\{0, 1\}$, where addition and multiplication are both performed modulo 2 (i.e. addition is XOR, and multiplication is the bit operator AND). Similarly, if n is a prime number, then we can represent the field $GF(n)$ to be the set $\{0, 1, \ldots, n-1\}$ where addition and multiplication are both performed modulo n.

However, suppose $n > 1$ is not a prime. Then the set $\{0, 1, \ldots, n-1\}$ where addition and multiplication are both performed modulo n is *not a field*. For example, let n be four. Then the set $\{0, 1, 2, 3\}$ is indeed closed under addition and multiplication modulo 4, however, the element 2 has no multiplicative inverse (there is no $a \in \{0, 1, 2, 3\}$ such that $2a \equiv 1 \pmod 4$). Thus, we *cannot* perform our coding with binary words of size $w > 1$ using addition and multiplication modulo 2^w. Instead, we need to use Galois Fields.

To explain Galois Fields, we work with polynomials of x whose coefficients are in $GF(2)$. This means, for example, that if $r(x) = x + 1$, and $s(x) = x$, then $r(x) + s(x) = 1$. This is because

$$x + x = (1 + 1)x = 0x = 0.$$

Moreover, we take such polynomials modulo other polynomials, using the following identity:

> If $r(x) \bmod q(x) = s(x)$, then $s(x)$ is a polynomial with a degree less than $q(x)$, and $r(x) = q(x)t(x) + s(x)$, where $t(x)$ is any polynomial of x.

Thus, for example, if $r(x) = x^2 + x$, and $q(x) = x^2 + 1$, then $r(x) \bmod q(x) = x + 1$.

Let $q(x)$ be a *primitive* polynomial of degree w whose coefficients are in $GF(2)$. This means that $q(x)$ cannot be factored, and that the polynomial x can be considered a *generator* of $GF(2^w)$. To see how x generates $GF(2^w)$, we start with the elements 0, 1, and x, and then continue to enumerate the elements by multiplying the last element by x and taking the result modulo $q(x)$ if it has a degree $\geq w$. This enumeration ends at 2^w elements – the last element multiplied by $x \bmod q(x)$ equals 1.

For example, suppose w = 2, and $q(x) = x^2 + x + 1$. To enumerate $GF(4)$ we start with the three elements 0, 1, and x, then then continue with $x^2 \bmod q(x) = x + 1$. Thus we have four elements: $\{0, 1, x, x + 1\}$. If we continue, we see that $(x + 1)x \bmod q(x) = x^2 + x \bmod q(x) = 1$, thus ending the enumeration.

The field $GF(2^w)$ is constructed by finding a primitive polynomial $q(x)$ of degree w over $GF(2)$, and then enumerating the elements (which are polynomials) with the generator x. Addition in this field is performed using polynomial addition, and multiplication is performed using polynomial multiplication and taking the result modulo $q(x)$. Such a field is typically written $GF(2^w) = GF(2)[x]/q(x)$.

Now, to use $GF(2^w)$ in the RS-Raid algorithm, we need to map the elements of $GF(2^w)$ to binary words of size w. Let $r(x)$ be a polynomial in $GF(2^w)$. Then we can map $r(x)$ to a binary word b of size w by setting the ith bit of b to the coefficient of x^i in $r(x)$. For example, in $GF(4) = GF(2)[x]/x^2 + x + 1$, we get the following table:

Generated Element of $GF(4)$	Polynomial Element of $GF(4)$	Binary Element b of $GF(4)$	Decimal Representation of b
0	0	00	0
x^0	1	01	1
x^1	x	10	2
x^2	$x+1$	11	3

Addition of binary elements of $GF(2^w)$ can be performed by bitwise exclusive-or. Multiplication is a little more difficult. One must convert the binary numbers to their polynomial elements, multiply the polynomials modulo $q(x)$, and then convert the answer back to binary. This can be implemented, in a simple fashion, by using the two logarithm tables described earlier: one that maps from a binary element b to power j such that x^j is equivalent to b (this is the `gflog` table, and is referred to in the literature as a "discrete logarithm"), and one that maps from a power j to its binary element b. Each table has $2^w - 1$ elements (there is no j such that $x^j = 0$). Multiplication then consists of converting each binary element to its discrete logarithm, then adding the logarithms modulo $2^w - 1$ (this is equivalent to multiplying the polynomials modulo $q(x)$) and converting the result back to a binary element. Division is performed in the same manner, except the logarithms are subtracted instead of added. Obviously, elements where $b = 0$ must be treated as special cases. Therefore, multiplication and division of two binary elements takes three table lookups and a modular addition.

Thus, to implement multiplication over $GF(2^w)$, we must first set up the tables `gflog` and `gfilog`. To do this, we first need a primitive polynomial $q(x)$ of degree w over $GF(2^w)$. Such polynomials can be found in texts on error correcting codes [1, 2]. We list examples for powers of two up to 64 below:

$$\begin{aligned} w = 4: &\quad x^4 + x + 1 \\ w = 8: &\quad x^8 + x^4 + x^3 + x^2 + 1 \\ w = 16: &\quad x^{16} + x^{12} + x^3 + x + 1 \\ w = 32: &\quad x^{32} + x^{22} + x^2 + x + 1 \\ w = 64: &\quad x^{64} + x^4 + x^3 + x + 1 \end{aligned}$$

We then start with the element $x^0 = 1$, and enumerate all non-zero polynomials over $GF(2^w)$ by multiplying the last element by x, and taking the result modulo $q(x)$. This is done in Table 2 below for $GF(2^4)$, where $q(x) = x^4 + x + 1$.

It should be clear now how the C code in Figure 4 generates the `gflog` and `gfilog` arrays for $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$.

Generated Element	Polynomial Element	Binary Element	Decimal Element
0	0	0000	0
x^0	1	0001	1
x^1	x	0010	2
x^2	x^2	0100	4
x^3	x^3	1000	8
x^4	$x+1$	0011	3
x^5	x^2+x	0110	6
x^6	x^3+x^2	1100	12
x^7	x^3+x+1	1011	11
x^8	x^2+1	0101	5
x^9	x^3+x	1010	10
x^{10}	x^2+x+1	0111	7
x^{11}	x^3+x^2+x	1110	14
x^{12}	x^3+x^2+x+1	1111	15
x^{13}	x^3+x^2+1	1101	13
x^{14}	x^3+1	1001	9
x^{15}	1	0001	1

Table 2: Enumeration of the elements of $GF(16)$