



# Reed-Solomon Error Correcting Codes

Ohad Rodeh



# Introduction

---

- Reed-Solomon codes are used for error correction
- The code was invented in 1960 by Irving S. Reed and Gustave Solomon, at the MIT Lincoln Laboratory
- This lecture describes a particular application to storage
- It is based on: **A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like systems**, James S. Plank, University of Tennessee, 1996.

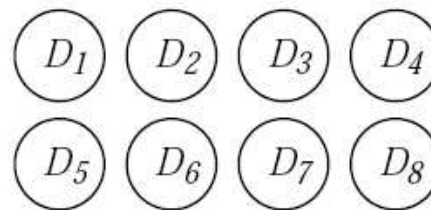


# Problem definition

- $n$  storage devices holding data:  $D_1, \dots, D_n$ . Each holds  $k$  bytes.
- $m$  storage devices holding checksums:  $C_1, \dots, C_m$ . Each holds  $k$  bytes.
- The checksums are computed from the data.
- The goal: if any  $m$  devices fail, they can be reconstructed from the surviving devices.

# Constructing the checksums

- The value of the checksum devices is computed using a function  $F$



$$C_1 = F_1(D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8)$$

$$C_2 = F_2(D_1, D_2, D_3, D_4 | D_5, D_6, D_7, D_8)$$

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

# Words

- The RS-RAID scheme breaks the data into words of size  $w$  bits
- The coding scheme works on stripes whose width is  $w$

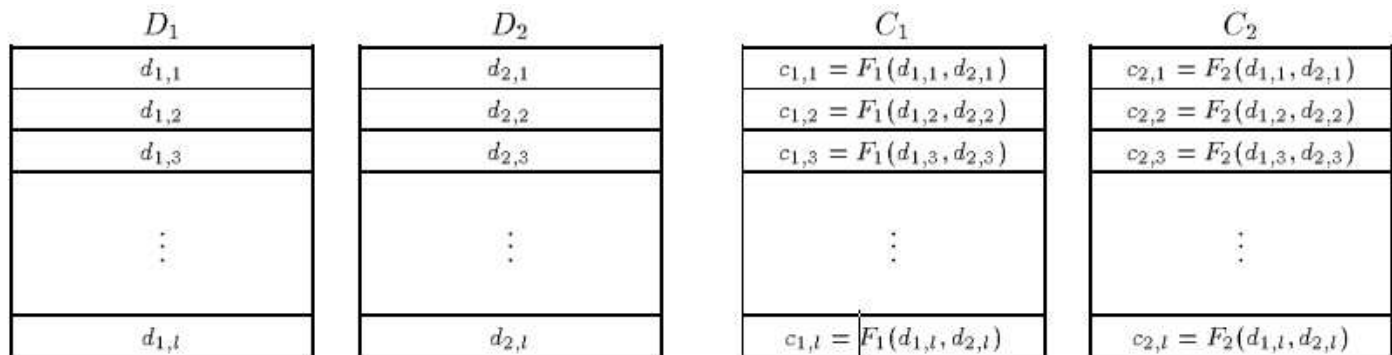


Figure 2: Breaking the storage devices into words ( $n = 2$ ,  $m = 2$ ,  $l = \frac{8k}{w}$ )

# Simplifying the problem

- From here we focus on a single stripe
- The data words are  $d_1, d_2, \dots, d_n$
- The checksum words are  $c_1, c_2, \dots, c_m$

$$c_i = F(d_1, d_2, \dots, d_n)$$

- If  $d_j$  changes to  $d'_j$  compute:

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

- When data devices fail we re-compute the data from the available devices and then re-compute the failed checksum devices from the data devices

# Example, RAID-4 as RS-RAID

- Set  $m = 1$
- Describe  $n + 1$  parity:

$$c_i = F(d_1, d_2, \dots, d_n) = d_1 \oplus d_2 \oplus \dots \oplus d_n$$

- If device  $j$  fails then:

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

# Restating the problem

- There are  $n$  data words  $d_1, \dots, d_n$  all of size  $w$
- We will
  1. Define functions  $F$  and  $G$  used to calculate the parity words  $c_1, \dots, c_m$
  2. Describe how to recover after losing up to  $m$  devices
- Three parts to the solution:
  1. Using Vandermonde matrix to compute checksums
  2. Using Gaussian elimination to recover from failures
  3. Use of Galois fields to perform arithmetic

# Vandermonde matrix

- Each checksum word  $c_i$  is a linear combination of the data words
- The matrix contains linearly independent rows

$$FD = C$$

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

# Changing a single word

- If data in device  $j$  changes from  $d_j$  to  $d'_j$  then

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

- This makes changing a single word reasonably efficient

# Recovering from failures

- In the matrix below, even after removing  $m$  rows, the matrix remains invertible

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



# Galois fields

- We want to use bytes as our data and checksum elements. Therefore, we need a field with 256 elements.
- A field  $GF(n)$  is a set of  $n$  elements closed under addition and multiplication
- Every element has an inverse for addition and multiplication



# Primes

- If  $n = p$  is a prime then  $Z_p = \{0, \dots, n - 1\}$  is a field with addition and multiplication module  $p$ .
- For example,  $GF(2) = \{0, 1\}$  with addition/multiplication modulo 2.
- If  $n$  is not a prime then addition/multiplication modulo  $n$  over the set  $\{0, 1, \dots, n - 1\}$  are not a field
  1. For example, if  $n = 4$ , and the elements are  $\{0, 1, 2, 3\}$
  2. Element 2 does not have a multiplicative inverse



# $GF(2^w)$

- Therefore, we cannot simply use elements  $\{0, \dots, 2^n - 1\}$  with addition/multiplication module  $2^n$ , we need to use Galois fields
- We use the Galois field  $GF(2^w)$
- The elements of  $GF(2^w)$  are polynomials with coefficients that are zero or one.
- Arithmetic in  $GF(2^w)$  is like polynomial arithmetic module a primitive polynomial of degree  $w$
- A primitive polynomial is one that is cannot be factored

# $GF(4)$

- $GF(4)$  contains elements:  $\{0, 1, x, x + 1\}$
- The primitive polynomial is  $q(x) = x^2 + x + 1$
- Arithmetic:

$$x + (x + 1) = 1$$

$$x \times x = x^2 \pmod{(x^2 + x + 1)} = x + 1$$

$$x \times (x + 1) = x^2 + x \pmod{(x^2 + x + 1)} = 1$$



# Efficient arithmetic

- It is important to implement arithmetic efficiently over  $GF(2^w)$
- Elements in  $GF(2^n)$  are mapped onto the integers  $0, \dots, 2^n - 1$
- Mapping of  $r(x)$  onto word of size  $w$
- $i$ -th bit is equal to the coefficient of  $x_i$  in  $r(x)$
- Addition is performed with XOR
- How do we perform efficient multiplication? Polynomial multiplication is too slow.

# Primitive polynomial

- A primitive polynomial  $q(x)$  is one that cannot be factored
- If  $q(x)$  is primitive then  $x$  generates all the elements in  $GF(2^n)$

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



# Multiplication

- Multiplication (and division) is done using logarithms
- $a \times b = x^{\log_x(a) + \log_x(b)}$
- Build two tables
  1. gflog[] : maps an integer to its logarithm
  2. gfilog[] : maps an integer to its inverse logarithm
- In order to multiply two integers,  $a$  and  $b$ 
  1. Compute their logarithms
  2. Add
  3. Compute the inverse logarithm
- For division: subtract instead of add

# $GF(16)$

- Example tables for  $GF(16)$
- Example computations:

$i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$gflog[i]$	$-\infty$	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	1

Table 1: Logarithm tables for  $GF(16)$

For example, in  $GF(16)$ :

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

# $GF(16)$ cont.

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  $GF(16)$



# Summary

---

- Choose a convenient value for  $w$ ,  $w = 8$  is a good choice
- Setup the tables  $gflog$  and  $gfilog$
- Setup the Vandermonde matrix, arithmetic is over  $GF(2^w)$
- Use  $F$  to maintain the checksums
- If any device fails, use  $F$  to reconstruct the data



# Conclusions

---

- We presented Reed-Solomon error correction codes
- Advantages:
  1. Conceptually simple
  2. Can work with any  $n$  and  $m$
- Disadvantage: Computationally more expensive than XOR based codes