# An Overview on Reed-Solomon Error-Correcting Code and Its Implementation for File Recovery

Noel Christoffel Simbolon - 13521096[1]
*Computer Science Study Program*
*School of Electrical Engineering and Informatics*
*Bandung Institute of Technology, Ganesha Street, no. 10, Bandung, 40132, Indonesia*
*[1]13521096@std.stei.itb.ac.id*

***Abstract*— Error-correction codes have been around for a while. Although they did not have much immediate application since it was first discovered, in this day and age, they are much utilized in many fields of study and the commercial sector. One of the categories of error-correction code implementation is the Reed-Solomon codes which are popular and widely used in modern society. They are used in many forms of consumer technologies and have many applications. One of which is data integrity. In this paper, I will overview an implementation of the Reed-Solomon codes in the form of software that is used to encode and decode files.**

***Keywords*—error correction, file, Galois field, Reed-Solomon**

## I. Introduction

This world is constantly moving to a world where computing data is ubiquitous and everyday. This has sparked more and more interaction between humans and data [1]. For example, we use online banking to manage our economical assets, or we utilize word-processing software to accomplish university assignments. Each of these actions requires storing data in a medium after then we confidently continue our activities without ever worrying about our data's safety. However, in theory, there are possibilities that those data undergo corruption, degradation, or erasure. There are many factors on why and how such a phenomenon can happen. Therefore, with the current widespread of human-data interaction, and how they have been deeply ingrained into our everyday lives, data integrity is of the foremost importance now than ever.

There have been constant efforts to develop methods on how we preserve the integrity of data. Most of which are based on theoretical discoveries in mathematics. One such is the discovery of error-correction coding. The discovery was pioneered by an American mathematician, Richard Hamming, when he first invented the first error-correcting code in 1950 which was used for controlling and correcting errors in data over unreliable mediums [2]. Another curious discovery invented by an early 19th century mathematician, Evariste Galois, is an abstract theory of the Galois field which essentially is a theory about fields that contain a finite number of elements, hence its other term: finite field [3]. However, for well over a hundred years, mathematicians looked upon Galois fields as elegant mathematics but of no practical value. In 1959, Irving S. Reed and Gustave Solomon, who were staff members of the MIT Lincoln Laboratory, wrote a five-page paper titled "Polynomial Codes over Certain Finite Fields" which was published in June 1960. This paper was a result of Reed's idea of using the elements of a finite field as an alphabet to use symbols rather than bits, e.g., half-bytes or bytes for the symbols. However, for years after its publication, the Reed-Solomon code was viewed as interesting mathematics and little else. It simply did not appear to be practical with the computing capability of the day. Nowadays, the Reed-Solomon code has been widely used in industrial and consumer electronic devices [4].

In this paper, I shall overview the Reed-Solomon error-correcting code implementation in the form of a library written in the Java programming language that is used to encode and decode files. Such encoding can improve the files' integrity by allowing them to self-correct in case of erasure up to a certain value.

## II. Theoretical Foundation

### *A. Fundamentals of Information Representation in the Computer*

At the smallest scale in the computer, information is stored as *bits*. A bit is just a 0 or 1. In the computer, the value of these bits is represented by transistors by controlling the amount of electricity flowing through it, depending on its threshold voltage. However, rather than accessing individual bits in memory, most computers use blocks of 8 bits, or *bytes*, as the smallest addressable unit of memory [5]. That means, every byte consists of eight zeros or ones.

In the binary numeral system, or base-2 numeral system, we represent each value with 0 or 1. Thus in binary notation, a byte's value ranges from $00000000_2$ to $11111111_2$.

To convert a binary notation into a decimal number, we need to represent a decimal number in terms of sums of $a_n 2^n$. That is, if $x$ is the said decimal number then we wish to have

$$x = \sum_{n \in \mathbb{N}} a_n 2^n$$

The coefficients $a_n$ is then written in descending order of n and all leading zeros then omitted. The final result becomes the binary representation of the decimal $x$ [3]. As such, when viewed as a decimal integer, a byte's value ranges from $0_{10}$ to

$255_{10}$. However, neither binary nor decimal notation is very convenient for describing bit patterns. Binary notation is too verbose, while with decimal notation it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or hexadecimal numbers. Hexadecimal (or simply "hex") uses digits '0' through '9' along with characters 'A' through 'F' to represent 16 possible values. Written in hexadecimal, the value of a single byte can range from $00_{16}$ to $FF_{16}$ [5]. The following table shows the decimal and binary values associated with the 16 hexadecimal digits.

Table I. Hexadecimal notation

| Hex digit | Decimal value | Binary value |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

In Java, numeric constants starting with the prefix `0x` are interpreted as being in hexadecimal, while numeric constants starting with the prefix `0b` are interpreted as being in binary. The characters 'A' through 'F' may be written in either upper or lowercase. For example, we could write the number $A7BC3DF_{16}$ as `0xA7BC3DF`, as `0xa7bc3df`, or even mixing upper- and lowercase (e.g., `0xA7bC3dF`).

To understand this better, suppose you are given a hexadecimal number `0x69A20B`. You can convert this to binary format by expanding each hexadecimal digit, as follows:

| Hex | 6 | 9 | A | 2 | 0 | B |
|---|---|---|---|---|---|---|
| Binary | 0110 | 1001 | 1010 | 0010 | 0000 | 1011 |

Thus, the binary representation is of the hex `0x69A20B` is `011010011010001000001011`.

## B. Boolean Algebra and Bitwise Operations

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole (1815–1864) around 1850 and thus is known as Boolean algebra. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the basic principles of logical reasoning. The simplest Boolean algebra is defined over the two-element set {0, 1}.

There are several operations defined in this algebra. The Boolean operation ~ corresponds to the logical operation NOT, denoted by the symbol $\neg$. That is, we say that $\neg P$ is true when $P$ is not true, and vice versa. Correspondingly, $\sim p$ equals 1 when $p$ equals 0, and vice versa. Boolean operation & corresponds to the logical operation AND, denoted by the symbol $\wedge$. We say that $P \wedge Q$ holds when both $P$ is true, and $Q$ is true. Correspondingly, $p$ & $q$ equals 1 only when $p = 1$ and $q = 1$. Boolean operation | corresponds to the logical operation OR, denoted by the symbol $\vee$. We say that $P \vee Q$ holds when either $P$ is true, or $Q$ is true. Correspondingly, $p \mid q$ equals 1 when either $p = 1$ or $q = 1$. Boolean operation ^ corresponds to the logical operation EXCLUSIVE-OR, denoted by the symbol $\oplus$. We say that $P \oplus Q$ holds when either $P$ is true or $Q$ is true, but not both. Correspondingly, $p$ ^ $q$ equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$ [5].

The summary of logical operations and their corresponding bitwise operator in Java are showed in the table below.

Table II. Bitwise operators in Java

| Logical operation | Java bitwise operator |
|---|---|
| NOT | ~ |
| AND | & |
| OR | \| |
| EXCLUSIVE-OR | ^ |

To understand this better in the context of Java, take a look at the examples shown below.

| *a* | *b* | Operation | Result |
|---|---|---|---|
| 0b0110 | 0b0110 | ~a & b | 0b0000 |
| 0x69 | 0x55 | a & b | 0x41 |
| 0b0110 | 0b0101 | a \| b | 0b0111 |
| 0xAB | 0xAB | a ^ b | 0x00 |

## C. Matrices, and Matrix Operations

Most of information stored in the computer is organized into arrays. A two-dimensional form of arrays is called "matrices" (plural of "matrix) which has rows and columns. For its applications, it is desirable to develop an "arithmetic of matrices" in which matrices can be added, subtracted, and multiplied in a useful way.

**Equality.** — Two matrices are defined to be *equal* if they have the same size and their corresponding entries are equal. Consider the matrices

$$A = \begin{bmatrix} 2 & 1 \\ 3 & x \end{bmatrix}, B = \begin{bmatrix} 2 & 1 \\ 3 & 5 \end{bmatrix}, C = \begin{bmatrix} 2 & 1 & 0 \\ 3 & 4 & 0 \end{bmatrix}$$

If $x =$ , then $A = B$, but for all other values of $x$ the matrices $A$ and $B$ are not equal, since not all of their corresponding entries are the same. There is no value of $x$ for which $A = C$ since $A$ and $C$ have different sizes.

**Addition and Subtraction.** — If $A$ and $B$ are matrices of the same size, then the *sum* $A + B$ is the matrix obtained by adding the entries of $B$ to the corresponding entries of $A$, and the *difference* $A - B$ is the matrix obtained by subtracting the entries of $B$ from the corresponding entries of $A$. Matrices of different sizes cannot be added or subtracted.

In matrix notation, if $A = [a_{ij}]$ and $B = [b_{ij}]$ have the same size, then

$$(A + B)_{ij} = (A)_{ij} + (B)_{ij} = a_{ij} + b_{ij}$$

and

$$(A - B)_{ij} = (A)_{ij} - (B)_{ij} = a_{ij} - b_{ij}$$

Consider the matrices

$$A = \begin{bmatrix} 2 & 1 & 0 & 3 \\ -1 & 0 & 2 & 4 \\ 4 & -2 & 7 & 0 \end{bmatrix},$$
$$B = \begin{bmatrix} -4 & 3 & 5 & 1 \\ 2 & 2 & 0 & -1 \\ 3 & 2 & -4 & 5 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Then

$$A + B = \begin{bmatrix} -2 & 4 & 5 & 4 \\ 1 & 2 & 2 & 3 \\ 7 & 0 & 3 & 5 \end{bmatrix}$$

and

$$A - B = \begin{bmatrix} 6 & -2 & -5 & 2 \\ -3 & -2 & 2 & 5 \\ 1 & -4 & 11 & -5 \end{bmatrix}$$

The expressions $A + C$, $B + C$, $A - C$, and $B - C$ are undefined.

**Scalar multiplication.** — If $A$ is any matrix and $c$ is any scalar, then the *product* c$A$ is the matrix obtained by multiplying each entry of the matrix $A$ by $c$. The matrix $cA$ is said to be a *scalar multiple* of $A$.

In matrix notation, if $A = [a_{ij}]$, then

$$(cA)_{ij} = c(A)_{ij} = ca_{ij}$$

For the matrices

$$A = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 3 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 & 2 & 7 \\ -1 & 3 & -5 \end{bmatrix}, C = \begin{bmatrix} 9 & -6 & 3 \\ 3 & 0 & 12 \end{bmatrix}$$

we have

$$2A = \begin{bmatrix} 4 & 6 & 8 \\ 2 & 6 & 2 \end{bmatrix}, (-1)B = \begin{bmatrix} 0 & -2 & -7 \\ 1 & -3 & 5 \end{bmatrix},$$
$$\frac{1}{3}C = \begin{bmatrix} 3 & -2 & 1 \\ 1 & 0 & 4 \end{bmatrix}$$

It is common practice to denote $(-1)B$ by $-B$.

**Matrix multiplication.** — If $A$ is an $m \times r$ matrix and $B$ is an $r \times n$ matrix, then the product $AB$ is the $m \times n$ matrix whose entries are determined as follows: To find the entry in row $i$ and column $j$ of $AB$, single out row $i$ from the matrix $A$ and column $j$ from the matrix $B$. Multiply the corresponding entries from the row and column together, and then add the resulting products.

Consider the matrices

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 6 & 0 \end{bmatrix}, B = \begin{bmatrix} 4 & 1 & 4 & 3 \\ 0 & -1 & 3 & 1 \\ 2 & 7 & 5 & 2 \end{bmatrix}$$

Since $A$ is a $2 \times 3$ matrix and $B$ is a $3 \times 4$ matrix, the product AB is a $2 \times 4$ matrix. To determine, for example, the entry in row 2 and column 3 of $AB$, we single out row 2 from $A$ and column 3 from $B$. Then, we multiply corresponding entries together and add up these products.

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 6 & 0 \end{bmatrix} \begin{bmatrix} 4 & 1 & 4 & 3 \\ 0 & -1 & 3 & 1 \\ 2 & 7 & 5 & 2 \end{bmatrix} = \begin{bmatrix} \dots & \dots & \dots & \dots \\ \dots & \dots & 26 & \dots \end{bmatrix}$$

$$(2 \cdot 4) + (6 \cdot 3) + (0 \cdot 5) = 26$$

The entry in row 1 and column 4 of $AB$ is computed as follows:

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 6 & 0 \end{bmatrix} \begin{bmatrix} 4 & 1 & 4 & 3 \\ 0 & -1 & 3 & 1 \\ 2 & 7 & 5 & 2 \end{bmatrix} = \begin{bmatrix} \dots & \dots & \dots & 13 \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$(1 \cdot 3) + (2 \cdot 1) + (4 \cdot 2) = 13$$

The computations for the remaining entries are

$$\begin{gathered} (1 \cdot 4) + (2 \cdot 0) + (4 \cdot 2) = 12 \\ (1 \cdot 1) + (2 \cdot 1) + (4 \cdot 7) = 27 \\ (1 \cdot 4) + (2 \cdot 3) + (4 \cdot 5) = 30 \\ (2 \cdot 4) + (6 \cdot 0) + (0 \cdot 2) = 8 \\ (2 \cdot 1) + (6 \cdot 1) + (0 \cdot 7) = -4 \\ (2 \cdot 3) + (6 \cdot 1) + (0 \cdot 2) = 12 \end{gathered}$$

$$AB = \begin{bmatrix} 12 & 27 & 30 & 13 \\ 8 & -4 & 26 & 12 \end{bmatrix}$$

The definition of matrix multiplication requires that the number of columns of the first factor $A$ be the same as the number of rows of the second factor $B$ in order to form the product $AB$. If this condition is not satisfied, the product is undefined [6].

**Transpose of a matrix.** — If $A$ is any $m \times n$ matrix, then the transpose of$A$, denoted by $A^T$, is defined to be the $n \times m$ matrix that results by interchanging the rows and columns of $A$; that is, the first column of $A^T$ is the first row of $A$, the second column of $A^T$ is the second row of $A$, and so forth.

The following are some examples of matrices and their transposes

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}, B = \begin{bmatrix} 2 & 3 \\ 1 & 4 \\ 5 & 6 \end{bmatrix},$$

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \\ a_{14} & a_{24} & a_{34} \end{bmatrix}, B = \begin{bmatrix} 2 & 1 & 5 \\ 3 & 4 & 6 \end{bmatrix}$$

## *D. Identity Matrices and the Inverse of a Matrix*

A square matrix with one's on the main diagonal and zeros elsewhere is called an *identity matrix*. An identity matrix is denoted by the letter *I*. Some examples are

$$[1], \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If $A$ is a square matrix, and if there exists a matrix $B$ of the same size for which $AB = BA = I$, then $A$ is said to be invertible (or nonsingular) and $B$ is called an inverse of $A$. If no such matrix $B$ exists, then $A$ is said to be singular. An invertible matrix has exactly one inverse. If $A$ is invertible, then its inverse is denoted by the symbol $A^{-1}$. Thus,

$$AA^{-1} = I \text{ and } A^{-1}A = I$$

## *E. Vandermonde Matrices*

Named after Alexandre-Théophile Vandermonde, a Vandermonde matrix is a matrix with the terms of a geometric progression in each row. A Vandermonde matrix $V$, which is an element of the set of all $n$-by-$n$ matrices over a field $F$, denoted $V \in M_n(F)$, has the form

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}$$

in which $x_1, \dots, x_n \in F$; that is, $V = [v_{ij}]$ with $v_{ij} = x_i^{j-1}$ [7].

## *F. Galois Field*

A Galois field is a field that contains a finite number of elements. As with any field, a finite field is a set on which the operations of multiplication, addition, subtraction, and division are defined and satisfy certain basic rules. A finite field has the property that arithmetic operations on field elements always have a result in the field.

The elements of Galois Field $GF(p^n)$ is defined as

$$\begin{aligned} &GF(p^n) = (0, 1, 2, \dots, p-1) \cup \\ &(p, p+1, p+2, \dots, p+p-1) \cup \\ &(p^2, p^2+1, p^2+2, \dots, p^2+p-1) \cup \dots \cup \\ &(p^{n-1}, p^{n-1}+1, p^{n-1}+2, \dots, p^{n-1}+p-1) \end{aligned}$$

where $p \in \mathbb{P}$ and $n \in \mathbb{Z}^+$. The order of the field is given by $p^n$while $p$ is called the characteristic of the field.

## *G. Reed-Solomon Error-Correcting Code*

The Reed-Solomon error-correcting code is a group of error-correcting codes that operate on a block of data treated as a set of the Galois field called symbols. The Reed-Solomon error-correcting code is characterized by three parameters: an alphabet size $q$, a block length $n$, and a message length $k$, with $k < n \leq q$. The set of alphabet symbols is interpreted as the finite field of order $q$, and thus, $q$ must be a prime power.

Reed–Solomon codes are able to detect and correct multiple symbol errors. By adding $t = n - k$ check symbols to the data, a Reed–Solomon code can detect (but not correct) any combination of up to $t$ erroneous symbols or locate and correct up to $\lfloor t/2 \rfloor$ erroneous symbols at unknown locations. As an erasure code, it can correct up to $t$ erasures at locations that are known and provided to the algorithm, or it can detect and correct combinations of errors and erasures. Reed–Solomon codes are also suitable as multiple-burst bit-error correcting codes since a sequence of $b + 1$ consecutive bit errors can affect at most two symbols of size $b$. The choice of $t$ is up to the designer of the code and may be selected within wide limits [8].

In the original view of Reed & Solomon (1960), every codeword of the Reed–Solomon code is a sequence of function values of a polynomial of degree less than $k$. In order to obtain a codeword of the Reed–Solomon code, the message symbols (each within the $q$-sized alphabet) are treated as the coefficients of a polynomial $p$ of degree less than $k$, over the finite field $F$ with $q$ elements. In turn, the polynomial $p$ is evaluated at $n \leq q$ distinct points $a_1, \dots, a_n$ of the field $F$, and the sequence of values is the corresponding codeword. Common choices for a set of evaluation points include $\{0, 1, 2, \dots, n-1\}$, $\{0, 1, \alpha, \alpha 2, \dots, \alpha n - 2\}$, or for $n < q$, $\{1, \alpha, \alpha 2, \dots, \alpha n - 1\}$, ... , where $\alpha$ is a primitive element of $F$.

The set $C$ of codewords of the Reed–Solomon code is defined as follows:

$$C = \{(p(a_1), p(a_2), \dots, p(a_n)) \mid \\ p \text{ is a polynomial over } F \text{ of degree} < k\}$$

While the number of different polynomials of degree less than $k$ and the number of different messages is both equal to $q^k$, and thus every message can be uniquely mapped to such a polynomial, there are different ways of doing this encoding. The original construction of Reed & Solomon (1960) interprets the message $x$ as the coefficients of the polynomial $p$, whereas subsequent constructions interpret the message as the values of the polynomial at the first $k$ points $a_1, \dots, a_k$ and obtain the polynomial $p$ by interpolating these values with a polynomial of degree less than $k$.

In the original construction of Reed & Solomon (1960), the message $x = (x_1, \dots, x_k) \in F^k$ is mapped to the polynomial $p_x$ with

$$p_x(a) = \sum_{i=1}^{k} x_i a^{i-1}$$

The codeword of $x$ is obtained by evaluating $p_x$ at $n$ different points $a_1, \dots, a_n$ of the field $F$. Thus, the classical encoding function $C: F^k \to F^n$ for the Reed–Solomon code is defined as follows:

$$C(x) = (p_x(a_1), \dots, p_x(a_n))$$

This function $C$ is a linear mapping, that is, it satisfies $C(x) = x^T \cdot A$ for the following $(k \times n)$-matrix $A$ with elements from

$F$:

$$A = \begin{bmatrix} 1 & \cdots & 1 & \cdots & 1 \\ a_1 & \cdots & a_k & \cdots & a_n \\ a_1^2 & \cdots & a_k^2 & \cdots & a_n^2 \\ \vdots & & \vdots & & \vdots \\ a_1^{k-1} & \cdots & a_k^{k-1} & \cdots & a_n^{k-1} \end{bmatrix}$$

This matrix is the transpose of a Vandermonde matrix over $F$. In other words, the Reed-Solomon code is a linear code, and in the classical encoding procedure, its generator matrix is $A$ [9].

## III. Reed-Solomon Error-Correction Code Implementation

Throughout this section, I will give a general review of the Reed-Solomon code implementation written in the Java programming language. The source code of the library can be found in the remote Git repository linked here.

In this implementation, the failure model is that of a file *erasure*. This is as opposed to a file *error*, in which an *error* is manifested by storing and retrieving incorrect values [10]. To address this problem, we encode a file using the Reed-Solomon code and then calculate the parity of the file. The result of this encoding is then broken up into shards of files. In the case of shard erasure, the original file can still be recovered as long as there are still enough shards to recalculate the original contents of the file.

We will take a close look at the `SampleEncoder` class, which is the class that does the encoding. This class has the variables `DATA_SHARDS`, `PARITY_SHARDS`, `TOTAL_SHARDS`, and `BYTES_IN_INT`. By default, the `SampleEncoder` class encodes a file with Reed-Solomon 4+2. It means the original file will be broken into four shards, then the program will calculate parity from it and add two more shards. From six shards, if there is a total of four shards after encoding and file erasure, regardless of the order of encoding, the contents of the original file can still be reconstructed. In short, the maximum number of shards we can lose to still be able to construct the original file is the same as the number of parity shards.

Moving on, now we will focus on the main method of the `SampleEncoder` class. In the first part of the main method, the program parses the command line argument, which is the path of the file to be encoded. If the file does not exist in the path, it terminates the program.

Next, it stores the file size of the file to be encoded. Then, it calculates the size of each shard because each shard must have the same size Afterwards, it creates a buffer holding the size of the input file (represented as 4-byte integer), reads from the file input stream, and verifies if the number of bytes read is equal to the size of the input file. After that, it creates a matrix of bytes and copies the shards that have not been encoded into it. Next, the program creates a Reed-Solomon coding matrix and encodes the parity of the data matrix using the coding matrix.

Finally, after the encoding is finished, the program writes the encoded data matrix into `TOTAL_SHARDS` different files (in this case, it is 6), each having the same size.

Next, we will focus on the process of decoding. which is done by the `SampleDecoder` class. As with the `SampleEncoder` class, the `SampleDecoder` class also has the same variables, and its value should also be the same as the variables in the `SampleEncoder` class for it to be able to reconstruct the file. Now, we will take a closer look at the main method of the `SampleEncoder` class.

The main method parses a command line argument, which is the path of the file to be decoded. If the number argument is not equal to 1, the program terminates. Next, it reads in any of the shards that are present in the directory passed in the argument. The decoding process needs at least `DATA_SHARDS` shards to be able to reconstruct the file (in this case, it is 4). The program terminates if the number of shards present is not enough. Otherwise, the program continues to make empty buffers for missing shards and uses the Reed-Solomon code to fill it (if any). Then, the program combines the data shards into one buffer. Although it is not efficient, it is convenient. Next, it extracts the file size (represented as a 4-byte integer) that was written at the beginning of the file in the encoding process. The extracted file size is useful for the next step, which is writing the reconstructed file.

The core encoding and decoding process is oversimplified in [11, Fig. 1] below.

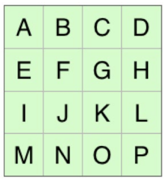

Fig. 1(a) The original data represented as matrix. Each data shard is represented as a matrix row.

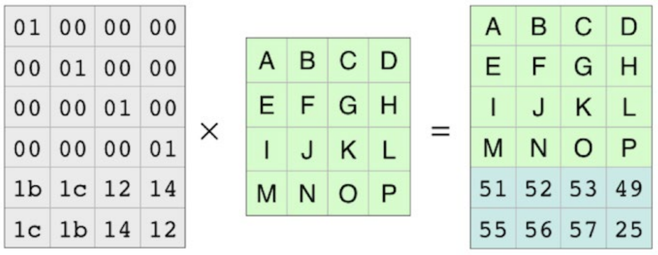

Fig. 1(b) The coding matrix (left) generated by the Reed-Solomon code is multiplied with the original data, computing the parity (represented as the last two rows of the matrix on the right).

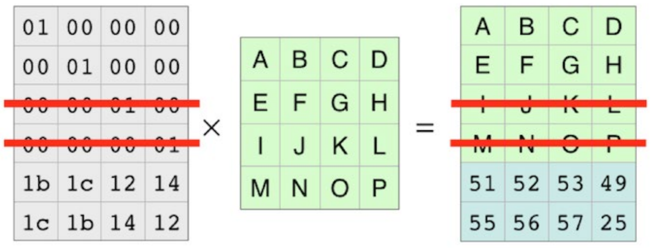

Fig. 1(c) Data erasure. Two of the six rows are lost.

$$\begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 1b & 1c & 12 & 14 \\ 1c & 1b & 14 & 12 \end{bmatrix} \times \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ 51 & 52 & 53 & 49 \\ 55 & 56 & 57 & 25 \end{bmatrix}$$

Fig. 1(d) The matrix equation still holds without the two erased rows.

$$\begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 8d & f6 & 7b & 01 \\ f6 & 8d & 01 & 7b \end{bmatrix} \times \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 1b & 1c & 12 & 14 \\ 1c & 1b & 14 & 12 \end{bmatrix} \times \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} = \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 8d & f6 & 7b & 01 \\ f6 & 8d & 01 & 7b \end{bmatrix} \times \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ 51 & 52 & 53 & 49 \\ 55 & 56 & 57 & 25 \end{bmatrix}$$

Fig. 1(e) The coding matrix is guaranteed to be invertible. Both sides of the equation are multiplied by the inverse of the coding matrix (left-most matrix).

$$\cancel{\begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 8d & f6 & 7b & 01 \\ f6 & 8d & 01 & 7b \end{bmatrix}} \times \cancel{\begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 1b & 1c & 12 & 14 \\ 1c & 1b & 14 & 12 \end{bmatrix}} \times \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} = \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 8d & f6 & 7b & 01 \\ f6 & 8d & 01 & 7b \end{bmatrix} \times \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ 51 & 52 & 53 & 49 \\ 55 & 56 & 57 & 25 \end{bmatrix}$$

Fig. 1(f) The inverse of the coding matrix and the coding matrix cancel out.

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} = \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 8d & f6 & 7b & 01 \\ f6 & 8d & 01 & 7b \end{bmatrix} \times \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ 51 & 52 & 53 & 49 \\ 55 & 56 & 57 & 25 \end{bmatrix}$$

Fig. 1(g) This leaves the equation for reconstructing the original data from the pieces that are available.

## IV. Conclusion

In case of file erasure up to a certain value, file recovery can be done by recalculating the contents of the erased file using an implementation of the Reed-Solomon error-correcting code as overviewed in the contents of this paper. The erasure coding technology is useful in many areas, particularly in local or cloud data storage. Such technology is essential in large data infrastructure to improve its fault tolerance, mitigating data erasures, and as a workaround when doing vault maintenance.

## V. Appendix

The Java implementation of the Reed-Solomon error-correcting code overviewed in this paper is available in the remote Git repository linked here. It was originally written by an engineer at Backblaze, a cloud storage and data backup company based in the USA, for its services. Currently, it is open source under the MIT license.

## VI. Acknowledgment

This paper would not have been possible without the opportunity given by my lecturer, Dr. Nur Ulfa Maulidevi, S.T., M.Sc. Her enthusiasm for educating her students motivated me to understand better and explore the field of discrete mathematics further. I would also like to thank Mr. Evan Su, maintainer of the encryption software Picocrypt, Mr. Vivek Verma, educational content creator on mathematics, and Mr. Brian Beach, engineer at Backblaze. Their extraordinary work has inspired me to pick the Reed-Solomon error-correcting code as the main subject of this paper.

## References

[1] R. Mortier, H. Haddadi, T. Henderson, D. McAuley, and J. Crowcroft, “Human-Data Interaction: The Human Face of the Data-Driven Society.” arXiv, Jan. 06, 2015. Accessed: Dec. 10, 2022. [Online]. Available: http://arxiv.org/abs/1412.6159
[2] R. Hamming, “Error Detecting and Error Correcting Codes,” vol. 29, Apr. 1950.
[3] C. J. Benvenuto, “Galois Field in Cryptography,” p. 11.
[4] I. S. Reed, “A brief history of the development of error correcting codes,” Computers & Mathematics with Applications, vol. 39, no. 11, pp. 89–93, Jun. 2000, doi: 10.1016/S0898-1221(00)00112-7.
[5] R. E. Bryant and D. R. O’Hallaron, Computer systems: a programmer’s perspective, Third edition. Boston: Pearson, 2016.
[6] H. Anton and A. Kaul, “Elementary Linear Algebra, 12th Edition”.
[7] R. A. Horn and C. R. Johnson, Matrix analysis, 2nd ed. Cambridge ; New York: Cambridge University Press, 2012.
[8] M. Riley, “An introduction to Reed-Solomon codes: principles, architecture and implementation.” [Online]. Available: https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html
[9] I. S. Reed and G. Solomon, “Polynomial Codes Over Certain Finite Fields,” Journal of the Society for Industrial and Applied Mathematics, vol. 8, no. 2, pp. 300–304, Jun. 1960, doi: 10.1137/0108018.
[10] J. S. Plank, “A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems,” Softw: Pract. Exper., vol. 27, no. 9, pp. 995–1012, Sep. 1997, doi: 10.1002/(SICI)1097-024X(199709)27:9<995::AID-SPE111>3.0.CO;2-6.
[11] B. Beach, “Backblaze Open-sources Reed-Solomon Erasure Coding Source Code,” Jun. 16, 2015. https://www.backblaze.com/blog/reed-solomon/

## Statement of Originality

I hereby declare that this paper is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, December 12th, 2022

[illegible]

Noel Christoffel Simbolon